

Report

Time series analysis on stock market Nifty - 50 data Using ARIMA and LSTM models

Author: Sandesh Nonavinakere Sunil

Matriculation No.: 7026004

Course of Studies: MBIDA

Author: Nishan Chandrashekhar Poojary

Matriculation No.: 7026796

Course of Studies: MBIDA

First examiner: Prof. Dr. Elmar Wings

Submission date: June 27, 2025

Contents

Contents	i
List of Figures	xii
List of Tables	xiii
List of Listings	xv
Acronyms	xvii
1. Abstract	1
I. Introduction	3
2. Introduction	5
2.1. problem's description	5
2.2. Literature Review	6
2.3. challenges	6
2.4. solution	7
2.5. report's structure	7
II. Domain Knowledge	9
3. Domain Problem	11
3.1. Application	11
3.2. Problem	11
3.3. Data Aquisition	12
3.4. Data Quantity	13
3.5. Data Quality	13
3.6. Data Relevance	14
3.7. Outliers	14
3.8. Anomalies	15
4. Data Mining	17
4.1. Long Short-Term Memory (LSTM)	17

4.2.	AutoRegressive Integrated Moving Average (ARIMA)	17
4.3.	Applications	17
4.4.	Relevance to This Project	19
4.5.	Hyperparameters to Be Considered	19
4.6.	Requirements	21
4.7.	Input	22
4.8.	Output	23
4.9.	Example with a Program	24
4.10.	Example with a Program	26
4.11.	Further Reading	28
III. Domain Tools		29
5. Development Environment		31
5.1.	Python	31
5.1.1.	Version	31
5.1.2.	Installation	31
5.1.3.	Configuration	33
5.1.4.	First Steps	34
5.2.	Python Virtual Environment (<code>venv</code>)	34
5.2.1.	Rationale for Using Virtual Environments	34
5.2.2.	Installation	35
5.2.3.	Creating and Activating the Environment	35
5.2.4.	Configuration: Installing Dependencies	35
5.2.5.	First Steps: Validation and Usage	36
5.2.6.	Benefits for Model Development and Collaboration	36
5.3.	Visual Studio Code (VS Code)	36
5.3.1.	Overview	36
5.3.2.	Installation	37
5.3.3.	System Requirements	38
5.3.4.	Configuration	39
5.3.5.	First Steps	39
5.3.6.	Advantages and Considerations	40
5.4.	GitHub	40
5.4.1.	Repository Overview	40
5.4.2.	Project Management Strategy	41
5.4.3.	Plan	42
IV. Methodology		45
6. Methodology		47
6.1.	Cross-Industry Standard Process for Data Mining	47

6.2.	CRISP-DM process flow for the project	47
6.3.	KDD (Knowledge Discovery in Databases)	49
6.4.	KDD Flow for the project	51
6.5.	Relevance to This Project	53
6.6.	Machine Learning Pipeline	54
6.6.1.	Problem Definition	54
6.6.2.	Data Acquisition	54
6.6.3.	Data Preprocessing	55
6.6.4.	Model Building	55
6.6.5.	Model Evaluation	56
6.6.6.	Visualization and Interpretation	56
6.6.7.	Deployment and Interface	56
6.6.8.	Logging and Reproducibility	56
6.7.	ML Pipeline Diagram	57
V. Application Development		59
7. Data		61
7.1.	Data Structure	61
7.2.	Data Size	61
7.3.	Data Format	62
7.4.	Data Anomalies	62
7.5.	Live data Integration	63
7.6.	Data Origin	64
8. Developer documentation		67
8.1.	Development Idea	67
8.2.	Development Flowchart (Conceptual Overview)	68
8.3.	Machine Learning Pipeline	72
8.3.1.	Input Data Acquisition	72
8.3.2.	Preprocessing	72
8.3.3.	Model Selection	73
8.3.4.	Model Training	73
8.3.5.	Prediction	74
8.3.6.	Evaluation	74
8.3.7.	Deployment and Output	75
8.4.	Program	76
8.4.1.	Structure and Modules	76
8.4.2.	Parameter Handling	78
8.5.	Error Handling	79
8.5.1.	errorHandler.py	80
8.6.	Message Handling	81
8.6.1.	messageHandler.py	81

8.7. Test Automation	83
8.8. Naming Conventions	84
8.8.1. Naming Conventions and Code Organization	84
8.8.2. Directory and Module Names	84
8.8.3. Function and Variable Names	85
8.8.4. Test File Naming	85
9. Database	87
9.1. Data Selection	87
9.1.1. Origin	87
9.1.2. Features	87
9.1.3. Data Types	88
9.1.4. Data Quality	89
9.1.5. Quantity	90
9.1.6. Fairness and Bias	91
9.2. Data Processing	93
9.2.1. One Database	93
9.2.2. Centralized Dataset for Stock Forecasting	94
9.2.3. Database Properties and Structure	96
9.2.4. Original Dataset Properties	97
9.2.5. Summary Statistics of Stock Prices	98
9.2.6. Correlation Analysis	98
9.3. Outlier Detection and Feature Justification	99
9.3.1. Anomaly Detection and Handling Missing Value . .	101
10. Data Transformation	105
10.1. Data Transformation	105
10.1.1. Data Collection	105
10.1.2. Data Loading	105
10.1.3. Data Preprocessing	106
10.1.4. Model Initialization	107
11. Data Mining	109
11.1. Feature Selection	109
11.2. Time Series Analysis	109
11.3. Model Selection	109
11.4. Input	110
11.5. Training	110
11.6. Hyperparameters	111
11.7. Interpretation	112
11.8. Output	113
12. Algorithms	115
12.1. ARIMA	115
12.1.1. Objectives	116

12.1.2. Model Architecture	117
12.1.3. Improvements	120
12.1.4. Model Serialization and Reusability	121
12.1.5. Hyperparameters	122
12.1.6. Configuration	124
12.1.7. Code	126
12.2. Output	129
12.2.1. Input	131
12.2.2. Further Readings	133
12.3. LSTM	135
12.3.1. Introduction	135
12.3.2. Objectives	135
12.3.3. Model Architecture	136
12.3.4. Improvements	138
12.3.5. Walk-Forward Retraining Framework	138
12.3.6. Separation of Models for Each Target Variable . .	138
12.3.7. Dropout Regularization	139
12.3.8. Scalability and Reusability via Modular Design .	139
12.3.9. Consistent Evaluation and Logging	139
12.3.10. Hyperparameters	139
12.3.11. Lookback Window	139
12.3.12. Number of LSTM Units	139
12.3.13. Dropout Rate	140
12.3.14. Batch Size and Epochs	140
12.3.15. Optimizer and Learning Rate	140
12.3.16. Retraining Interval and Rolling Window	140
12.3.17. Configuration	141
12.3.18. Code	142
12.3.19. Data Handler Module	142
12.3.20. Training Loop	143
12.3.21. Input	144
12.3.22. Output	147
12.3.23. Further Readings	151
13. Packages	153
13.1. Pandas	153
13.1.1. Pandas Overview	153
13.1.2. Core Data Structures	153
13.1.3. Installation	153
13.1.4. Typical Use Cases	154
13.1.5. Practical Examples	154
13.1.6. File I/O Reference	157
13.1.7. Further Reading	157

13.2. NumPy	157
13.2.1. NumPy Overview	157
13.2.2. Core Objects & Functionality	157
13.2.3. Installation	158
13.2.4. Typical Use Cases	158
13.2.5. Practical Examples	158
13.2.6. File I/O Reference	160
13.2.7. Further Reading	161
13.3. Joblib	161
13.3.1. Joblib Overview	161
13.3.2. Core Components	161
13.3.3. Installation	162
13.3.4. Typical Use Cases	162
13.3.5. Practical Examples	162
13.3.6. Caching & Parallel Backend Reference	165
13.3.7. Further Reading	165
13.4. Scikit-learn	166
13.4.1. Scikit-learn Overview	166
13.4.2. Core Components	166
13.4.3. Installation	166
13.4.4. Typical Use Cases	167
13.4.5. Practical Examples	167
13.4.6. Model Persistence & Utility Reference	172
13.4.7. Further Reading	172
13.5. Statsmodels	172
13.5.1. Statsmodels Overview	172
13.5.2. Core Components	173
13.5.3. Installation	173
13.5.4. Typical Use Cases	173
13.5.5. Practical Examples	174
13.5.6. Result & I/O Reference	178
13.5.7. Further Reading	178
13.6. TensorFlow	179
13.6.1. TensorFlow Overview	179
13.6.2. Core Building Blocks	179
13.6.3. Installation	179
13.6.4. Typical Use Cases	180
13.6.5. Practical Examples	180
13.6.6. Model I/O & Deployment Reference	184
13.6.7. Further Reading	184
13.7. Streamlit	184
13.7.1. Streamlit Overview	185
13.7.2. Core Building Blocks	185
13.7.3. Installation	185

13.7.4. Typical Use Cases	186
13.7.5. Practical Examples	186
13.7.6. Widget & I/O Reference	188
13.7.7. Further Reading	188
14. Development to Deployment	189
14.1. Data Structure	189
14.2. Tools	191
14.3. Description of the Filetypes	193
14.4. File Structure for Model Exchange	194
14.4.1. Directory Layout	194
14.4.2. Model Saving and Exporting	194
14.4.3. Naming Conventions	195
14.4.4. Model Portability Goals	196
14.4.5. Documentation and Reproducibility	196
14.5. Saving Models	196
14.5.1. Saving ARIMA Models	196
14.5.2. Saving LSTM Models with Keras	197
14.5.3. Saving Preprocessing Scalers	197
14.5.4. Versioning and Portability	198
14.6. Loading Models	198
14.6.1. Loading ARIMA Models	199
14.6.2. Loading LSTM Models	199
14.6.3. Loading Preprocessing Scalers	200
14.6.4. Model Integration	201
15. Evaluation	203
15.1. ARIMA Model Diagnostics	203
15.1.1. Evaluation Concept	203
15.1.2. Application	203
15.1.3. Results	204
15.2. LSTM Model Evaluation	206
15.2.1. Evaluation Concept	206
15.2.2. Application	206
15.2.3. Results	207
15.3. Model Accuracy Evaluation	209
15.3.1. Concept	209
15.3.2. Application	210
15.3.3. Results	211
15.4. Summary:	212
16. Validation	213
16.1. Validation Methodologies in Machine Learning	213
16.1.1. Holdout Validation	213

16.1.2. k-Fold Cross-Validation	214
16.1.3. Stratified k-Fold Cross-Validation	215
16.1.4. Leave-One-Out Cross-Validation (LOOCV)	215
16.1.5. Time Series Split / Rolling Forecast Origin	216
16.2. Validation Approaches Implemented in Our Project	217
16.2.1. Idea 1:	217
16.2.2. Idea 2:	219
16.2.3. Idea 3:	221
17. Monitoring	223
17.1. Idea	223
17.2. Plan/Description	223
17.2.1. Overview	223
17.2.2. Data Acquisition and Preprocessing	223
17.2.3. Model Loading and Inference	223
17.2.4. Evaluation and Visualization	224
17.2.5. User Interface and Localization	224
17.2.6. Model Storage and Portability	224
17.2.7. Logging and Debugging	224
17.2.8. Streamlit Keep-Alive	224
17.3. Getting New Data	226
17.4. Data Updating in the ML Pipeline	227
17.5. Validation and Consistency Checks	228
17.6. Code: Utility Functions	229
17.6.1. Data Preprocessing Functions	229
17.6.2. Feature Engineering Functions	230
17.6.3. Model Training Functions	231
17.6.4. Model Evaluation Functions	233
17.7. Unit Tests	234
17.8. Privacy	239
17.8.1. Secure Data Transmission and Storage	239
17.8.2. Compliance with Data Protection Regulations	239
17.8.3. Access Control and Data Anonymization	240
17.9. Robustness	240
17.9.1. Handling Missing or Corrupted Data	240
17.9.2. Ensuring Adaptability to Market Conditions	241
17.9.3. Regular Testing Under Diverse Scenarios	241
17.10. Process	241
17.10.1. Data Collection and Preprocessing	241
17.10.2. Model Development and Training	242
17.10.3. Model Evaluation and Validation	242
17.10.4. Continuous Monitoring and Maintenance	242
17.10.5. Documentation and Communication	243

VI. Application Deployment	245
18. Deployment	247
18.1. Directory Structure	247
18.1.1. Folder Responsibilities	248
18.1.2. Concept	250
18.1.3. Deployment Flowchart	252
18.2. Machine Learning Pipeline	253
18.3. Parameter Handling	255
18.4. Error Handling	255
18.4.1. Usage Example	255
18.4.2. Function Signature	256
18.4.3. Behavior	257
18.5. Internationalization & Centralized Messaging	257
18.5.1. Retrieval Example	257
18.5.2. MessageHandler API	258
18.5.3. Benefits	259
18.6. CI/CD & Test Automation	259
18.6.1. Workflow Configuration	259
18.6.2. Explanation	260
18.7. Keep-Alive “Ping”	260
18.7.1. Health Check Endpoint	261
18.7.2. External Ping Configuration	261
18.7.3. Implementation Notes	261
18.7.4. Benefits	261
18.7.5. Streamlit Deployment Configuration	262
VII. Results and Conclusion	265
19. Result	267
19.1. Results	267
19.1.1. Forecasting Accuracy Metrics	267
19.1.2. Visual Comparison of Predictions	267
19.1.3. Next-Day Prediction Output	268
19.1.4. Summary	269
20. Conclusion	271
20.1. Key Contributions	271
20.2. Performance Summary	271
20.3. To Do	272
20.4. Unanswered Points	272
20.5. Next Steps	272

VIII Application Appendix	275
A. Bill and List of Materials	277
A.1. Bill of Materials: Software	277
A.2. List of Materials: Hardware	278
B. List of Packages and Tools	281
Bibliography	283

List of Figures

5.1. Python download page	33
5.2. VS Code interface with Python project loaded	37
5.3. VS Code official download page	37
5.4. macOS terminal with VS Code CLI command	38
6.1. CRISP-DM Process Applied to NIFTY 50 Forecasting System	49
6.2. KDD Process Workflow	50
6.3. Knowledge Discovery in Databases (KDD) Process Applied to Stock Forecasting	53
6.4. Machine Learning Pipeline for NIFTY 50 Forecasting	57
8.1. Condensed Development Flowchart for NIFTY 50 Forecasting	71
8.2. Condensed ML Pipeline for NIFTY 50 Forecasting	75
12.1. ARIMA model architecture	118
12.2. Actual vs Predicted Open	131
12.3. LSTM Architecture	137
12.4. Actual vs Predicted Open	150
15.1. ARIMA Residual Diagnostics Open	205
15.2. ARIMA Residual Diagnostics Close	206
15.3. LSTM Residual Diagnostics Close	208
15.4. LSTM Residual Diagnostics Close	209
16.1. Concept of Validation [Tha+04]	213
16.2. Holdout Cross-Validation Workflow [Tur24]	214
16.3. K-Fold Cross-Validation Workflow [Tur24]	214
16.4. Stratified K-Fold Cross-Validation Workflow [Tur24]	215
16.5. Leave-One-Out Cross-Validation Workflow [Tur24]	216
16.6. Time Series Cross-Validation Workflow [Tur24]	216
18.1. Prediction Output	251
18.2. Pipeline flowchart of the NIFTY 50 forecasting system, from data acquisition and preprocessing to model loading, prediction, and interactive visualization	253
18.3. NIFTY 50 Forecasting: End-to-End Machine Learning Pipeline	254

18.4. Streamlit deployment configuration interface for GitHub-linked applications	262
19.1. Actual vs Predicted Close Prices: Comparison of ARIMA and LSTM performance.	268
19.2. Actual vs Predicted Open Prices: ARIMA shows better alignment with true values.	268
19.3. Prediction summary and error metrics output from the deployed system.	269

List of Tables

5.1. System Requirements for VS Code on Windows	38
5.2. System Requirements for VS Code on macOS	39
5.3. System Requirements for VS Code on Linux	39
5.4. Software Stack and Dependency Version	43
6.1. Data Preprocessing Steps for ARIMA and LSTM	55
7.1. Yahoo Finance NIFTY 50 Dataset: Column Descriptions .	61
9.1. Dataset Summary	91
9.2. Descriptive Statistics of NIFTY 50 Opening and Closing Prices	98
9.3. RMSE and MAPE Comparison with and without Outliers	101
12.1. ARIMA Hyperparameter Summary	124
12.2. Required Structure of Input Data	132
12.3. LSTM Model Hyperparameters	140
A.1. Software Bill of Materials	277
A.2. Hardware List of Materials	278

List of Listings

Acronyms

1. Abstract

This project presents the development of a stock market prediction system specifically tailored for forecasting the next-day open and close prices of the NIFTY 50 index. The system integrates machine learning and time series analysis into an intuitive and accessible frontend interface, allowing users to make one-day-ahead predictions using two modeling approaches: ARIMA (AutoRegressive Integrated Moving Average) and LSTM (Long Short-Term Memory neural networks). The objective of this work is to provide a practical, user-friendly tool that combines statistical and deep learning methods to support short-term trading or investment decision-making.

The frontend of the application is designed to let users choose the date for which they want a prediction—restricted to the next trading day—and select their preferred model for forecasting. Once the user selects the desired configuration, the system processes historical NIFTY 50 data to generate the forecasted open and close values for the selected future date. The ARIMA model leverages the statistical properties of time series data to make linear predictions, while the LSTM model captures complex temporal dependencies and nonlinear patterns in sequential data.

Both models are trained and validated on historical NIFTY index data, and various performance metrics are employed to evaluate their effectiveness in short-term forecasting scenarios. The system highlights the strengths and limitations of each approach and demonstrates how different modeling techniques can be applied to real-world financial data for practical use. The goal of the project is not only to provide accurate and interpretable predictions but also to offer insights into how classical and deep learning models perform in financial forecasting tasks.

This report documents the design choices, modeling techniques, user interface functionality, evaluation methods, and key findings of the project, aiming to contribute to the growing interest in AI-driven financial tools for retail and professional investors alike.

Part I.

Introduction

2. Introduction

In recent years, financial forecasting has seen a paradigm shift with the increasing adoption of machine learning and deep learning techniques [ZZQ20; FK18]. Predicting stock market movements, particularly short-term trends, has become an area of intense research and commercial interest [BYR17]. With the expansion of algorithmic trading and accessible computing resources, even individual investors now seek tools that assist in making informed trading decisions [GKX20].

Among various market indices, the NIFTY 50—representing 50 major Indian companies—serves as a benchmark index for the Indian stock market and is a key focus for analysts and investors.

This project aims to build an intelligent and interactive platform that predicts the open and close prices of the NIFTY 50 index for the next trading day. The system is designed to make these predictions using two distinct models: ARIMA, a traditional statistical method for time series forecasting, and LSTM, a modern deep learning technique well-suited for sequential data [HS97]. The combination of an intuitive frontend with powerful backend models serves to bridge the gap between technical forecasting tools and end-user accessibility [Aba+16].

2.1. problem's description

Accurate short-term prediction of stock index prices, especially for the next day’s opening and closing values, remains a significant challenge due to the dynamic and volatile nature of financial markets. While numerous tools exist for long-term trend analysis, fewer accessible solutions are available that allow non-expert users to make day-ahead forecasts with model flexibility. Specifically, there is a need for a system that:

Focuses on short-term (next-day) predictions. Provides reliable forecasting for a specific, high-impact index (NIFTY 50). Enables users to compare classical and deep learning forecasting models. Offers a user-friendly frontend that abstracts technical complexity. This project aims to address this gap by delivering a single-day-ahead prediction interface backed by both ARIMA and LSTM models.

2.2. Literature Review

Recent advancements in financial forecasting have seen the growing application of machine learning (ML) and deep learning (DL) models due to their superior pattern recognition and adaptability in handling nonlinear data. Traditional models like ARIMA have long been used for time series forecasting owing to their statistical rigor and interpretability [BJ76], but they struggle when faced with the inherent non-stationarity and noise typical of financial markets [Tsa10].

LSTM networks have emerged as a promising deep learning approach for financial prediction tasks due to their ability to learn long-term dependencies in sequential data [HS97]. Studies such as [ZZQ20] and [FK18] demonstrate that LSTMs can outperform classical models in capturing complex temporal behaviors, although they are more susceptible to overfitting and often require larger datasets and regularization techniques [GS23].

Furthermore, the literature highlights the increasing interest in creating user-accessible forecasting tools. Platforms like TensorFlow have enabled easier integration of deep learning models into real-time applications [Aba+16], while research by [GKX20] emphasizes the democratization of financial forecasting through the use of automated and interactive systems for individual investors.

Finally, foundational theories such as Efficient Market Hypothesis (EMH) suggest that markets are inherently unpredictable due to new and random information continuously impacting price movements [Fam70], which presents further challenges for any predictive model and justifies the integration of multiple modeling approaches for robustness.

2.3. challenges

Stock market forecasting involves several inherent challenges, many of which are amplified in short-term prediction:

- Non-stationarity of data: Financial time series data are typically non-stationary, meaning their statistical properties change over time [Tsa10]. This poses difficulties for models like ARIMA, which assume a level of stationarity.
- Market noise and volatility: The stock market is influenced by a multitude of unpredictable factors—news events, macroeconomic indicators, and investor sentiment—all of which introduce noise that is hard to model [Fam70].
- Overfitting in deep learning models: While LSTM networks can learn complex temporal patterns, they are susceptible to overfitting,

especially when trained on relatively small datasets or without sufficient regularization [ZZQ20].

- **Interpretability:** Deep learning models often function as black boxes, offering little interpretability compared to traditional models, which can be a barrier to trust in high-stakes financial applications [Lip16].

2.4. solution

To address the above challenges, this project implements a dual-model forecasting system:

- **ARIMA model:** Used for its transparency and effectiveness in modeling linear time series data [BJ76]. Preprocessing steps such as differencing are applied to handle non-stationarity.
- **LSTM model:** Leveraged for its ability to capture nonlinear dependencies and temporal patterns [HS97; ZZQ20]. Techniques like dropout regularization and careful sequence windowing are used to mitigate overfitting [GS23].
- **Single-day prediction horizon:** Restricting predictions to the next trading day improves reliability and makes the system more practical for real-world use [FK18].
- **Frontend application:** A clean, interactive interface allows users to select a model and receive predictions without dealing with backend complexities [Aba+16].
- **Integrated system:** Together, these solutions create a modular, interpretable, and accessible stock forecasting tool centered on the NIFTY 50 index.

2.5. report's structure

This report has been organized into well-defined sections to guide readers through the end-to-end process of developing and deploying a data-driven solution. Each chapter builds upon the previous one, enabling both technical and non-technical readers to navigate seamlessly. The structure is as follows:

- **Abstract:** A concise summary of the project's objectives, approach, and outcomes.

- Chapter I – Introduction: Introduces the background, motivation, problem statement, existing literature, challenges, and a high-level overview of the solution and report structure.
- Chapter II – Domain Knowledge: Explores the problem’s domain, including data acquisition, quantity, quality, and relevance, along with issues such as outliers and anomalies.
- Chapter III – Domain Tools: Discusses the tools and technologies employed in the development environment, including Python, virtual environments, VS Code, and version control systems such as GitHub.
- Chapter IV – Methodology: Describes the methodological framework followed, such as CRISP-DM and KDD, and outlines the machine learning pipeline used in the project.
- Chapter V – Application Development: Details data structure, origin, anomalies, and integration. Also includes developer-focused documentation such as flowcharts, pipelines, error handling, and message processing.
- Chapter VI – Database: Describes the source, structure, and processing of the database used for model training and evaluation. Includes feature selection, data transformation, and outlier detection.
- Chapter VII – Data Transformation: Covers data loading, preprocessing, and preparation for modeling, including steps to ensure data quality and consistency.
- Chapter VIII – Data Mining: Presents techniques used for feature selection, time series modeling, and model evaluation. It also explains training procedures and hyperparameter tuning.
- Chapter IX – Algorithms: Discusses the algorithms implemented (e.g., ARIMA, LSTM), their architecture, configuration, serialization, and code structure. Includes comparative insights and potential improvements.
- Annexes: Includes List of Figures, Tables, Listings, Acronyms, References, and additional supporting materials for deeper understanding or future development.

Part II.

Domain Knowledge

3. Domain Problem

3.1. Application

Stock market prediction is a longstanding problem in quantitative finance, gaining renewed interest with the rise of machine learning. This project aims to predict the next day's Open and Close prices of the NIFTY 50 index using daily historical market data from January 01, 2008, to the present. Accurate forecasts of these price points are critical for:

- Pre-market positioning based on Open predictions.
- Intraday and swing trading strategies guided by Close predictions.
- Risk management and hedging for institutional portfolios.

The models used for prediction are:

- ARIMA (AutoRegressive Integrated Moving Average): A classical time-series model that captures linear trends and seasonality. It is interpretable and statistically grounded, especially effective for stationary data [Box+15].
- LSTM (Long Short-Term Memory networks): A deep learning technique ideal for modeling long-range temporal dependencies in sequential data. LSTM networks are particularly powerful in capturing nonlinear and delayed relationships that classical models cannot [HS97].

This hybrid modeling strategy provides a balanced blend of interpretability and deep learning flexibility to capture the complex, dynamic behavior of financial markets.

3.2. Problem

The task is formulated as a supervised regression problem where input features are historical price and volume data, and the output variables are the next day's Open and Close prices of the NIFTY 50 index.

Challenges involved include:

- **Non-stationarity:** Financial time series are inherently non-stationary due to regime changes, policy announcements, and external shocks [Con01].
- **High volatility and noise:** Markets exhibit unpredictable daily price movements that complicate modeling [Fam70].
- **Feature correlation:** Metrics like Open, Close, and Volume are often correlated, requiring careful feature engineering [GKX20].
- **Time dependency:** Market behavior evolves over time, demanding models that can adapt to both recent trends and longer-term patterns [ZZ21; BYR17].

The objective is to achieve low prediction error while preserving signal integrity and avoiding overfitting.

3.3. Data Aquisition

The dataset for this study is obtained programmatically from Yahoo Finance, specifically from the historical data section of the NIFTY 50 Index . This source offers comprehensive and continuously updated daily trading records, making it ideal for developing time-sensitive financial forecasting systems. Data was retrieved using the Python yfinance API, ensuring automation, reproducibility, and adaptability to real-time changes in market conditions.

The dataset includes the following attributes:

- **Date:** Timestamp of the trading session.
- **Open:** The first traded price when the market opens.
- **High:** The highest price reached during the trading session.
- **Low:** The lowest price observed during the session.
- **Close:** The final price when the market closes.

These features, while limited to technical indicators, are known to carry rich informational content for forecasting. Short-term market behavior is primarily driven by recent price and volume trends, making such attributes especially relevant in high-frequency and day-ahead trading contexts [AV09]. Additionally, past price and volume-based attributes can outperform fundamental indicators in machine learning models for stock indices [Pat+15a].

The data starts on January 01, 2008, covering over 4,000 trading sessions and spanning several market regimes, including the 2008 financial crisis, the 2020 COVID-19 crash, and multiple election cycles. This historical depth enables the models to generalize over both calm and volatile periods, a necessity for robust financial prediction systems.

3.4. Data Quantity

The dataset contains 4,280 trading days (approximately 260 sessions per year), spanning from January 2008 onward. This large volume of time series data allows both classical and deep learning models to capture complex patterns with sufficient depth. The dataset grows as each day passes , since it is a live data pull.

ARIMA models require a minimum of 50–100 observations for reliable seasonal decomposition and trend analysis but perform better with more extended time windows [HA18]. LSTM models typically need hundreds to thousands of sequential samples to effectively learn temporal dependencies and generalize to unseen market conditions [Bro17].

Data is partitioned into:

- 80% training set for model learning.
- 20% test set for performance evaluation.

This split ensures proper evaluation, cross-validation, and hyperparameter tuning for both ARIMA and LSTM models.

3.5. Data Quality

To ensure high model performance and generalizability, extensive data quality checks were implemented:

- **Missing values:** Core pricing fields (Open, High, Low, Close) were free of gaps. Any missing values in Volume or Adjusted Close (<1%) were filled using forward fill imputation.
- **Date consistency:** Confirmed that all trading dates align with the Indian stock market calendar, excluding weekends.
- **Data types:** All numerical fields were cast to float64, and dates were converted to datetime64[ns] for precise indexing.
- **Stationarity checks:** Conducted Augmented Dickey-Fuller (ADF) tests on each time series. Non-stationary series were differenced before modeling with ARIMA.

High data quality is critical for reducing forecast bias, avoiding model drift, and ensuring long-term stability of predictive performance [ZEPH98].

3.6. Data Relevance

Each feature in the dataset contributes uniquely to the forecasting task:

- Open and Close prices are the direct prediction targets.
- High and Low prices offer insights into intraday volatility, helping models interpret market uncertainty.
- Volume reflects liquidity and investor sentiment, useful in gauging trading activity and price impact.

While the dataset lacks valuation metrics such as P/E and Dividend Yield, numerous studies confirm that price-based and volume-based indicators are sufficient for short-term forecasting tasks [KBB11; Pat+15a]. Technical indicators derived from these base features can further enhance signal representation without requiring fundamental inputs.

3.7. Outliers

Outliers in financial datasets are not mere statistical anomalies but often signify critical market events with macroeconomic or geopolitical origins. These can include abrupt policy shifts, global crises, or extraordinary earnings announcements, all of which introduce substantial volatility. While such events may appear statistically extreme, they carry substantial predictive weight in time series modeling. Ignoring these can distort the model’s capacity to learn from rare but impactful scenarios, especially in fat-tailed financial distributions [Con01].

To detect and interpret these influential data points, we adopted a z-score filtering approach:

- Z-score thresholds were applied to the **Close** price series, identifying data points with standardized values beyond ± 3 standard deviations from the mean.
- This method enables objective, repeatable detection of statistically extreme deviations while preserving the temporal structure of the data.
- Identified outliers were then manually validated against market news and economic calendars to verify their association with real-world financial events.

Unlike traditional preprocessing pipelines that treat outliers as noise to be removed, our approach acknowledges their informative nature. All z-score flagged observations were retained in the dataset to support robust learning. As financial models trained on sanitized data often underperform during volatile periods, retaining these observations improves the system's resilience [YC20].

- For LSTM models, robust scaling methods such as `MinMaxScaler` were employed to normalize extreme values without truncation, reducing gradient instability during training.
- In ARIMA, outlier sensitivity was addressed via residual analysis and by optimizing model orders to minimize error propagation from shock events.

By applying a controlled, transparent outlier detection method and integrating these data points into model design, the forecasting system maintains adaptability under both stable and turbulent market conditions.

3.8. Anomalies

In time series forecasting, anomalies are typically defined as structured, multi-point deviations from regular patterns, such as flash crashes, sudden volatility shifts, or transient market regime changes [CBK09; AMH16]. These are different from isolated outliers; anomalies are often economically significant and can indicate unscheduled regulatory interventions, liquidity disruptions, or reactionary post-earnings swings. Accurate anomaly detection is thus vital for enhancing model robustness, particularly in financial domains.

However, the NIFTY 50 dataset used in this study—sourced from the Yahoo Finance API via `yfinance`—demonstrated high structural integrity and did not exhibit substantial anomalies. All columns of interest, such as `Open`, `Close`, `High`, and `Low`, had zero missing values, and only a handful of `Volume` entries were absent due to trading holidays. This clean profile eliminates the need for intensive anomaly detection procedures. To confirm this, we systematically applied null-value checks, duplicate record tests, and summary statistics evaluations, which all reaffirmed the dataset's completeness.

- **Missing Value Check:** Using `isnull().sum()`, we found that all price columns had no missing entries, confirming the reliability of the financial time series.
- **Duplicate Verification:** The dataset passed all integrity checks with `duplicated().sum()` yielding zero, ensuring chronological consistency across the entire time span.

Due to this high level of cleanliness, advanced anomaly detection techniques such as STL decomposition [Cle+90], CUSUM [Pag54], and volatility clustering analysis [Bol86] were deemed unnecessary. These techniques are valuable in noisy, sensor-driven domains or in datasets with observable structural breaks—conditions not present in our dataset.

- For LSTM, no anomaly flags were introduced, as the data distribution remained stable over time without regime-shifting deviations.
- For ARIMA, model residuals remained statistically well-behaved, and no autocorrelation violations were observed that would necessitate adjustments based on anomaly presence.

Thus, the anomaly treatment strategy in this project was simplified to include only standard structural checks. This minimal intervention aligns with current best practices in financial forecasting, which recommend data-driven adaptability to anomaly complexity [GKX20]. As no persistent anomalies were found, the dataset was deemed ready for direct modeling.

4. Data Mining

4.1. Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) is a specialized form of Recurrent Neural Network (RNN) designed to address the limitations of traditional RNNs in capturing long-range dependencies in sequential data. Unlike standard RNNs, LSTMs employ memory cells and gating mechanisms (input, output, and forget gates) to retain information over longer sequences without suffering from vanishing or exploding gradients [HS97]. This makes LSTM particularly well-suited for time-series data where contextual understanding of past values significantly influences future predictions.

4.2. AutoRegressive Integrated Moving Average (ARIMA)

ARIMA is a classical statistical method for time-series forecasting. It combines three components: Autoregression (AR), which uses the dependency between an observation and a number of lagged observations; Integration (I), which represents the differencing of raw observations to make the time series stationary; and Moving Average (MA), which models the error of the prediction as a linear combination of error terms from past forecast [BJ76].

4.3. Applications

The use of time series forecasting models like LSTM and ARIMA has become increasingly prevalent across various domains due to their capabilities in capturing temporal dependencies and trends. While ARIMA is well-regarded for its strength in modeling linear relationships and stationary data, LSTM excels at handling complex, nonlinear, and long-range dependencies in sequential datasets. Their complementary strengths have led to widespread adoption in financial systems, energy planning, anomaly detection, and more.

1. LSTM

Long Short-Term Memory (LSTM) networks are a subclass of recurrent neural networks (RNNs) designed to learn long-term dependencies. Their ability to retain information over long sequences makes them particularly effective for modeling time-dependent patterns. Key applications include:

- **Financial market forecasting:** LSTM has shown strong performance in stock price and index prediction tasks due to its capacity to capture market dynamics and nonlinear dependencies [FK18; ZZ21].
- **Anomaly detection in time series:** LSTM networks are used to identify deviations from normal patterns in areas such as fraud detection and system monitoring [AMH16].
- **Natural language processing (NLP):** Applications such as language translation, text generation, and speech recognition rely on LSTM's ability to handle sequential input effectively [HS97].
- **Energy demand forecasting:** LSTM models are increasingly used in predicting electricity consumption patterns, leveraging their temporal modeling capabilities [BYR17].

2. ARIMA

The Autoregressive Integrated Moving Average (ARIMA) model is a widely used statistical method for analyzing and forecasting univariate time series data. Its interpretability and effectiveness in modeling stationary signals have made it a staple in traditional forecasting:

- **Stock price trend analysis:** ARIMA models are commonly used to identify and forecast trends in financial markets, particularly when data is stationary or can be transformed into stationary form [ZZ21].
- **Economic forecasting:** Government and financial institutions use ARIMA to predict economic indicators like GDP, inflation rates, and employment trends [BJ76].
- **Sales forecasting:** Retail and supply chain management applications leverage ARIMA to estimate future product demand based on historical sales data [AV09].
- **Traffic flow analysis:** Transportation systems use ARIMA to predict traffic volume and congestion levels, aiding in urban planning and logistics [AMH16].

In this project, both algorithms are used for forecasting the next day's Open and Close prices of the Nifty Fifty Index, a benchmark stock

market index representing the weighted average of 50 of the largest Indian companies listed on the National Stock Exchange (NSE).

4.4. Relevance to This Project

1. LSTM Relevance

The stock market is inherently sequential and non-linear. LSTM networks are used to model the temporal dependencies and intricate patterns within the historical stock data. Given that the Nifty 50 prices exhibit volatility and non-stationary characteristics, LSTM's ability to learn from sequences without manual feature engineering makes it an effective tool in this scenario [FK18; HS97].

2. ARIMA Relevance

ARIMA, while linear, serves as a benchmark to evaluate the performance of LSTM. It is particularly effective in capturing linear trends and seasonality in stationary time series [BJ76]. In this project, it provides a contrast to the deep learning approach, helping to determine if complex models like LSTM offer significant improvements over traditional methods [HA18].

4.5. Hyperparameters to Be Considered

Effective tuning of hyperparameters is essential for optimizing model performance and ensuring generalizability. Both LSTM and ARIMA models require careful calibration of their respective parameters to balance complexity, accuracy, and computational efficiency.

1. LSTM:

LSTM models contain multiple tunable hyperparameters that significantly influence learning dynamics and prediction quality [FK18; ZZ21; AH24]:

- **Number of epochs:** Defines how many complete passes the algorithm makes over the training dataset. Too few epochs may lead to underfitting, while too many can result in overfitting.
- **Batch size:** Specifies the number of training samples processed before the model's weights are updated. Smaller batch sizes offer more precise gradient estimates, while larger batches speed up computation.
- **Number of LSTM layers:** Deep or stacked LSTM architectures allow the model to capture hierarchical patterns in the

time series but may require regularization to prevent overfitting [BYR17].

- **Number of hidden units:** Controls the capacity of each LSTM cell to learn temporal features. More units can model more complex dependencies but increase the risk of overfitting and computational demand [HS97].
- **Learning rate:** Determines how much weights are adjusted during training. A learning rate that is too high can lead to unstable training; too low can cause slow convergence.
- **Dropout rate:** Used to randomly deactivate a fraction of neurons during training to prevent overfitting [FK18].
- **Sequence length (window size):** The number of time steps input into the model. Longer windows may capture more context but can increase complexity.
- **Optimizer:** Algorithms such as Adam or RMSProp are typically used to update weights. They can have a significant effect on convergence speed and final model performance.

2. ARIMA:

ARIMA models are governed by three main parameters along with optional seasonal extensions. Accurate parameter selection is crucial for producing stable and interpretable forecasts [BJ76; ZZ21]:

- **p (autoregressive order):** Specifies the number of lag observations included in the model. Captures the influence of prior time steps on the current value.
- **d (differencing order):** Indicates how many times the data should be differenced to achieve stationarity. Non-stationary data can yield misleading regression results.
- **q (moving average order):** The number of lagged forecast errors included in the prediction equation.
- **Seasonal components (for SARIMA):** Includes seasonal terms (P , D , Q , and m) to handle repeating patterns in data, such as weekly or yearly cycles.
- **Stationarity and invertibility conditions:** While not direct hyperparameters, ensuring these conditions hold is necessary for model validity [BJ76].
- **Model selection criteria:** Information criteria such as AIC or BIC are often used to choose the best model configuration by balancing fit and complexity [ZZ21].

4.6. Requirements

The successful development and deployment of forecasting models like LSTM and ARIMA depend on specific computational, software, and data preparation prerequisites. Each model has unique implementation needs due to the fundamental differences in their structure and learning mechanisms.

1. LSTM:

LSTM models are computationally intensive and data-driven. Their successful deployment requires a well-prepared pipeline:

- **Python programming language:** Python is the preferred language for deep learning due to its rich ecosystem and user-friendly syntax [Lut21].
- **Deep learning libraries:** TensorFlow and Keras provide high-level APIs for building, training, and evaluating LSTM models [Aba+16].
- **Numerical and data libraries:** NumPy and Pandas facilitate efficient array and time series manipulation; Matplotlib supports visualization [Har+20; McK10].
- **Hardware acceleration:** While CPUs can suffice for small models, GPUs are essential for handling large datasets and deep architectures, significantly reducing training time [Aba+16].
- **Normalized input data:** Scaling input features to a uniform range (e.g., using Min-Max or Z-score normalization) stabilizes and speeds up the training process by avoiding gradient issues [FK18].
- **Sufficient historical data:** LSTM requires a considerable volume of past observations to detect patterns and temporal dependencies effectively [ZZ21].
- **Sequence generation and windowing:** Input data must be reshaped into fixed-length sequences to allow the LSTM to learn from temporal context [BYR17].
- **Regularization techniques:** To prevent overfitting, dropout layers and early stopping mechanisms are typically integrated into the training pipeline [FK18].
- **Hyperparameter tuning tools:** Libraries like Optuna or grid search can automate the search for optimal parameters like learning rate, batch size, and number of layers [AH24].

2. ARIMA:

ARIMA models are grounded in statistical theory and require meticulous preparation of the time series data:

- **Python tools:** Key libraries include `statsmodels` for ARIMA implementation and `pandas` for time series manipulation [tea24; McK10].
- **Stationarity verification:** Stationarity is a core assumption. Tests such as the Augmented Dickey-Fuller (ADF) or Kwiatkowski–Phillips–Schmidt–Shin (KPSS) are used to confirm this property [BJ76].
- **Transformation techniques:** Differencing, log transformation, or seasonal adjustment may be necessary to achieve stationarity [Box+15].
- **Model identification tools:** Autocorrelation function (ACF) and partial autocorrelation function (PACF) plots assist in selecting appropriate values for p and q parameters [BJ76].
- **Handling seasonality:** If the data shows periodic patterns, SARIMA (Seasonal ARIMA) may be used by adding seasonal terms to the base ARIMA model [BJ76].
- **Sufficient historical observations:** More data points allow for better estimation of ARIMA coefficients and more robust forecasts [ZZ21].
- **Error diagnostics:** Residuals must be checked for white noise properties to validate the model fit. This can be done through Q-statistics and residual plots [BJ76].
- **Information criteria:** Model selection should be guided by AIC (Akaike Information Criterion) or BIC (Bayesian Information Criterion), which balance model fit and complexity [ZZ21].

4.7. Input

For both LSTM and ARIMA models, the input comprises daily historical price data of the NIFTY 50 index. These features capture the essential dynamics of market movement needed for accurate forecasting.

Features used:

- **Date:** Serves as the temporal index to ensure proper chronological sequencing.

- **Open:** The first traded price of the day, reflecting initial market sentiment.
- **High and Low:** Represent the day's price range, indicative of intra-day volatility.
- **Close:** The final traded price of the day, commonly used as the target variable for forecasting models.

Preprocessing steps:

1. For LSTM:

- **Normalization:** Input features are scaled using MinMaxScaler to fit a 0–1 range, which enhances numerical stability during training [FK18].
- **Sequence generation:** The data is restructured using a sliding window approach to create sequences of fixed length. Each sequence serves as an input to the LSTM model, predicting the next day's price [BYR17].

2. For ARIMA:

- **Stationarization:** Non-stationary data is transformed using differencing to ensure a constant mean and variance over time [BJ76].
- **Parameter selection:** Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots are used to guide the choice of ARIMA parameters (p and q) [BJ76].

4.8. Output

The output of the forecasting system includes predicted values for the next trading day's Open and Close prices of the NIFTY 50 index, generated separately by LSTM and ARIMA models. These outputs are evaluated visually and numerically to assess prediction quality.

1. LSTM:

The LSTM model provides predictions for the next day's Open and Close prices based on temporal patterns in historical data. Due to preprocessing, the outputs are inverse-transformed to return to the original price scale [FK18; ZZ21]. LSTM is particularly effective in capturing non-linear dependencies and volatility patterns.

2. ARIMA:

The ARIMA model forecasts the same target variables based on linear relationships among past observations and forecast errors. These forecasts are generated from a differenced, stationary version of the time series data [BJ76; ZZ21].

3. Visualization:

A key component of output analysis is the graphical comparison of predicted versus actual values for the next trading day. This graph plots the true Open and Close prices alongside the forecasted values from each model, offering a visual assessment of predictive alignment and model fit.

4. Evaluation Metrics:

Quantitative evaluation of model performance is conducted using the following error metrics:

- **Root Mean Squared Error (RMSE):** Captures the square root of the average squared differences between predicted and actual values. It reflects the standard deviation of prediction errors and penalizes larger deviations more heavily [AH24].
- **Mean Absolute Percentage Error (MAPE):** Measures the average percentage difference between predicted and actual values. It provides an interpretable, scale-independent assessment of prediction accuracy [AH24].

4.8

4.9. Example with a Program

LSTM Example (Python using Keras):

The following Python implementation provides a practical example of training an LSTM model to forecast the next day's closing price using the past 60 days of closing prices from the NIFTY 50 index. This method is grounded in established practices for time series prediction in financial domains [FK18; ZZ21].

Listing 4.1: LSTM Forecasting Model in Python using Keras

```
## @file lstm_forecasting.py
# @brief LSTM model to forecast next day's closing price of
#        NIFTY 50.
# @details This script loads historical stock data,
#          preprocesses it using normalization and sequence
#          windowing,
#          then trains a single-layer LSTM model using Keras to
#          predict the next day's closing price.
```

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
from sklearn.preprocessing import MinMaxScaler
import numpy as np
import pandas as pd

## @brief Load and preprocess historical NIFTY 50 data
# @details This section reads CSV data, converts dates, sets
#         ↪ index, and extracts 'Close' prices.
df = pd.read_csv('nifty50.csv') # Load historical stock data
df['Date'] = pd.to_datetime(df['Date']) # Convert date column
#         ↪ to datetime format
df.set_index('Date', inplace=True) # Set the date as index

data = df[['Close']].values # Extract only the 'Close' price
#         ↪ column

## @brief Normalize data to [0, 1] range
# @details Normalization improves model convergence and
#         ↪ stability during training.
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data)

## @brief Create sequences for LSTM input
# @details Generates 60-day sliding windows for time-series
#         ↪ prediction.
sequence_length = 60
X = []
y = []
for i in range(sequence_length, len(scaled_data)):
    X.append(scaled_data[i-sequence_length:i, 0]) # 60 time steps
#         ↪ as input
    y.append(scaled_data[i, 0]) # next value as label
X, y = np.array(X), np.array(y)

## @brief Reshape input to 3D format for LSTM [samples, time
#         ↪ steps, features]
X = np.reshape(X, (X.shape[0], X.shape[1], 1))

## @brief Build and compile the LSTM model
# @details Uses a single LSTM layer followed by a Dense output
#         ↪ layer.
model = Sequential()
model.add(LSTM(units=50, return_sequences=False, input_shape=(X.
#         ↪ shape[1], 1))) # 50 LSTM units
model.add(Dense(units=1)) # Single output for regression
model.compile(optimizer='adam', loss='mean_squared_error') #
#         ↪ Compile with MSE loss

## @brief Train the model
# @param epochs Number of training cycles
# @param batch_size Size of each training batch
model.fit(X, y, epochs=20, batch_size=32) # Train model on
#         ↪ prepared dataset
```

Explanation:

- **Data Normalization:** Normalizing with `MinMaxScaler` ensures all values lie between 0 and 1, which helps the LSTM model converge more quickly and stably [FK18].
- **Sequence Windowing:** A rolling window of 60 days is used to form input sequences. This is essential for learning temporal dependencies in financial data [BYR17].
- **Model Design:** The architecture includes one LSTM layer with 50 units followed by a dense output layer. This structure balances model expressiveness with overfitting risk [ZZ21].
- **Training Configuration:** The `adam` optimizer is selected for its adaptive learning rate behavior, making it well-suited to noisy and sparse financial time series data [Aba+16].
- **Reshaping:** The 3D shape (`samples, time steps, features`) is required for Keras LSTM layers, enabling them to interpret input as time-dependent sequences.

4.10. Example with a Program

ARIMA Example (Python using statsmodels):

The following code snippet demonstrates the use of an ARIMA model to forecast the next day's closing price of the NIFTY 50 index. This method aligns with traditional time series modeling practices in financial forecasting [BJ76; ZZ21].

Listing 4.2: ARIMA Forecasting Model using statsmodels

```
# @brief Forecasting NIFTY 50 index using ARIMA model.
# @details This script performs time series forecasting using
#         the ARIMA model. It includes data loading,
#         stationarity testing via ADF, differencing for
#         stationarity, ARIMA model fitting, forecasting,
#         and visualization of actual vs fitted values.

import pandas as pd
import numpy as np
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import adfuller
import matplotlib.pyplot as plt

## @brief Load and prepare the historical data
# @details Reads CSV data, parses the date column, and sets it
#         as index.
df = pd.read_csv('nifty50.csv') # Load CSV file containing
#         historical prices
```

```
df['Date'] = pd.to_datetime(df['Date']) # Convert 'Date' to
#       ↪ datetime
df.set_index('Date', inplace=True) # Set 'Date' as index for
#       ↪ time series analysis

data = df['Close'] # Extract the 'Close' price for modeling

## @brief Perform Augmented Dickey-Fuller (ADF) test for
#       ↪ stationarity
# @details ADF test checks whether the time series is
#       ↪ stationary.
#           If the p-value > 0.05, the series is assumed non-
#       ↪ stationary.
result = adfuller(data)
if result[1] > 0.05:
    data_diff = data.diff().dropna() # Apply differencing to
#       ↪ achieve stationarity
else:
    data_diff = data # Data is already stationary

## @brief Fit ARIMA model
# @details ARIMA(p,d,q) where:
#           p = autoregressive term (lags), d = differencing, q
#       ↪ = moving average term
# @param order (5,1,2) as a demonstrative configuration; to be
#       ↪ tuned via ACF/PACF
model = ARIMA(data, order=(5, 1, 2))
model_fit = model.fit() # Fit the ARIMA model to the data

## @brief Forecast the next value in the series
# @return Next predicted closing price
forecast = model_fit.forecast(steps=1)
print("Next day's forecasted Close price:", forecast.values[0])

## @brief Plot the actual vs fitted values for evaluation
# @details Visual inspection of model fit over historical data
plt.figure(figsize=(10, 4))
plt.plot(data, label='Actual') # Plot actual closing prices
plt.plot(model_fit.fittedvalues, label='Fitted', color='orange')
#       ↪ # Plot ARIMA fitted values
plt.title('ARIMA Model Fit')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Explanation:

- **Stationarity check:** The Augmented Dickey-Fuller (ADF) test determines if differencing is needed to make the time series stationary—an essential ARIMA assumption [BJ76].
- **Model parameters:** The ARIMA(p,d,q) configuration controls the autoregression, differencing, and moving average behavior. The example uses (5,1,2), but tuning via ACF/PACF plots is recommended [ZZ21].
- **Forecasting:** A one-step-ahead forecast estimates the next closing price using recent patterns in the data.
- **Visualization:** Plotting both actual and fitted values provides intuitive feedback on the model's accuracy and behavior over time.
- **Simplicity and transparency:** ARIMA's linear nature makes it highly interpretable for financial analysts and regulators.

4.11. Further Reading

- Jiang, W., 2020. Applications of deep learning in stock market prediction: Recent progress. *arXiv preprint arXiv:2003.01859*. Available at: <https://arxiv.org/abs/2003.01859>.
- Sezer, O.B., Gudelek, M.U. and Ozbayoglu, A.M., 2019. Financial time series forecasting with deep learning: A systematic literature review: 2005–2019. *arXiv preprint arXiv:1911.13288*. Available at: <https://arxiv.org/abs/1911.13288>.
- Kashif, K. and Slepaczuk, R., 2024. LSTM-ARIMA as a hybrid approach in algorithmic investment strategies. *arXiv preprint arXiv:2406.18206*. Available at: <https://arxiv.org/abs/2406.18206>.
- Stempień, D. and Ślepaczuk, R., 2025. Hybrid models for financial forecasting: Combining econometric, machine learning, and deep learning models. *arXiv preprint arXiv:2505.19617*. Available at: <https://arxiv.org/abs/2505.19617>.
- Sunki, A., SatyaKumar, C., Narayana, G.S., Koppara, V. and Hakeem, M., 2024. Time series forecasting of stock market using ARIMA, LSTM and FB Prophet. *MATEC Web of Conferences*, 392, p.01163. doi: [10.1051/matecconf/202439201163](https://doi.org/10.1051/matecconf/202439201163).

Part III.

Domain Tools

5. Development Environment

5.1. Python

Python is a high-level, general-purpose programming language that has become the standard in the fields of data science, machine learning, and time series analysis. Its widespread adoption stems from its simplicity, expressive syntax, and the strength of its scientific computing ecosystem. For this project, Python acts as the central development language, enabling a seamless workflow that integrates data preprocessing, statistical modeling, deep learning, and result visualization.

The choice of Python is driven by its mature ecosystem—libraries such as `NumPy`, `pandas`, `matplotlib`, `statsmodels`, and `TensorFlow/Keras` provide robust tools for manipulating financial data, building predictive models, and validating their outputs. Its interpreted nature allows for rapid prototyping, and the community support ensures continuous improvements and reliable documentation. This makes Python ideal for implementing both ARIMA (a traditional time series model) and LSTM (a deep learning-based sequential model), which together form the backbone of the forecasting engine in this project.

5.1.1. Version

This project employs **Python 3.10.13**, chosen for its compatibility with machine learning and time series analysis libraries. This version introduces structural pattern matching and enhanced type hinting, improving the clarity and maintainability of forecasting pipelines [Lut21]. It also features optimizations in memory handling and function calls, crucial for efficient model training and large-scale data processing [Har+20]. Python 3.10.13 ensures stable integration with tools like NumPy, pandas, TensorFlow, Keras, and statsmodels, all essential to this project's ARIMA-LSTM framework [Aba+16; McK10]. A fixed version improves reproducibility across computing environments, which is critical in financial modeling tasks [Bro17].

5.1.2. Installation

To install Python 3.10.13 on different operating systems, follow the appropriate instructions below:

- **Windows:**

1. Visit the official website: <https://www.python.org/downloads/>
2. Download the Python 3.10.13 executable installer for Windows.
3. Run the installer and ensure the option **Add Python to PATH** is selected.
4. Complete the installation and verify it using:

```
python --version
```

- **macOS:**

1. Download the macOS installer from <https://www.python.org/downloads/mac-osx/>
2. Open the downloaded package and follow the installation steps.
3. After installation, verify Python using the terminal:

```
python3 --version
```

4. If required, install Xcode command-line tools for full compatibility:

```
xcode-select --install
```

- **Linux (Debian/Ubuntu-based):**

1. Open a terminal and update package lists:

```
sudo apt update  
sudo apt install python3.10
```

2. Verify the installation:

```
python3.10 --version
```

Each platform-specific approach ensures that Python is installed and accessible system-wide. Installing the correct version is a crucial step for maintaining consistency in development and reproducibility of forecasting results.

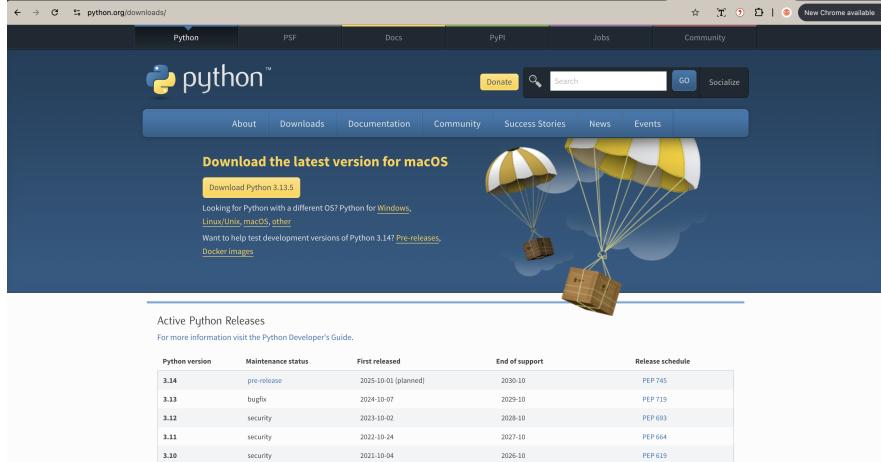


Figure 5.1.: Python download page

5.1.3. Configuration

Python’s configuration in this project does not rely on a virtual environment but instead uses a system-wide installation. This approach ensures direct access to the Python interpreter from the terminal or command line interface across operating systems, streamlining development in controlled or collaborative environments.

After installing Python, it is important to confirm that it is correctly recognized by the system path. This is essential for enabling Python to execute scripts and integrate with editors or development environments like VS Code or PyCharm.

To verify the Python installation, use the following command:

```
python --version
```

or, on Unix-based systems:

```
python3 --version
```

A correctly configured Python environment will return the version string (e.g., **Python 3.10.13**), confirming that the interpreter is available system-wide. Additionally, ensuring Python is added to the system’s PATH variable during installation allows Python to be invoked from any directory without specifying the full path to the interpreter.

This setup prioritizes reproducibility, simplicity, and platform compatibility, aligning with common practices in time-sensitive research environments where maintaining identical configurations across systems is crucial [Bro17].

5.1.4. First Steps

Once Python has been successfully installed and configured, the next step is to verify the development environment through a basic execution test. This not only checks the interpreter's functionality but also sets the stage for script-based development in more complex forecasting workflows.

To begin, create a new Python file named `hello.py`. Insert the following line of code, which prints a standard output message to the terminal:

```
print("Hello, World!")
```

This minimal script tests Python's ability to parse and execute basic syntax. To run the script, navigate to the directory where `hello.py` is saved and execute:

```
python hello.py
```

or on Unix-based systems:

```
python3 hello.py
```

A successful run should return the following output:

```
Hello, World!
```

This simple exercise is more than a formality—it confirms that the Python interpreter is operational, the PATH configuration is correct, and that the terminal or integrated development environment (IDE) can interface with Python. It marks the transition from setup to execution and reflects the best practices used in Python onboarding processes for data science and research [Lut21].

In a broader context, establishing that a base script runs correctly is foundational in educational environments, reproducible experiments, and production-level deployments alike [Bro17]. It is also an important pedagogical step, often used in tutorials and documentation, to bridge the gap between installation and real-world problem-solving.

5.2. Python Virtual Environment (`venv`)

5.2.1. Rationale for Using Virtual Environments

Managing isolated environments using `venv` is a foundational practice in data science and machine learning workflows. Virtual environments ensure consistency, reproducibility, and isolation from global Python configurations, which is essential when working on projects involving complex models such as ARIMA and LSTM. As emphasized by [Bro17], environment isolation prevents dependency conflicts and enables iterative experimentation with different model configurations. [McK10] further

highlights that modular pipeline development depends on controlling and standardizing software dependencies, especially when multiple libraries interact during data preprocessing, training, and evaluation.

5.2.2. Installation

Since Python 3.3, the `venv` module comes pre-installed with the standard Python distribution, removing the need for external downloads. To verify availability, run:

```
python -m venv --help
```

This command returns usage information if `venv` is installed correctly, indicating readiness for environment setup.

5.2.3. Creating and Activating the Environment

To initialize a virtual environment in the current directory:

```
python -m venv venv
```

This creates a directory named `venv` containing an isolated Python interpreter and libraries. Activation varies by operating system:

- Windows:

```
venv\Scripts\activate
```

- macOS/Linux:

```
source venv/bin/activate
```

A successfully activated environment will show the (`venv`) prefix in the command prompt.

5.2.4. Configuration: Installing Dependencies

Inside the activated environment, necessary forecasting libraries can be installed:

```
pip install pandas numpy matplotlib statsmodels keras tensorflow
```

This ensures that libraries are confined to the environment, avoiding interference with other projects or system packages. [FK18] highlight the importance of using controlled environments to maintain consistency and reproducibility in time series forecasting research.

5.2.5. First Steps: Validation and Usage

To confirm proper setup, check the Python version and installed packages:

```
python --version  
pip list
```

As a basic test, create a Python script (e.g., `check.py`) containing:

```
print("venv environment is working properly")
```

Run the script using:

```
python check.py
```

To exit the environment, use:

```
deactivate
```

5.2.6. Benefits for Model Development and Collaboration

Using virtual environments provides multiple benefits:

- **Dependency Traceability:** Enables use of `pip freeze` to lock exact package versions, ensuring reproducibility across systems [Bro17].
- **Safe Experimentation:** Facilitates side-by-side testing of different models or library versions without conflict [McK10].
- **CI/CD Readiness:** Compatible with automation tools like GitHub Actions and Jenkins for scalable testing [Aba+16].
- **Research Transparency:** Clearly defined dependencies support auditable workflows, vital for peer-reviewed research [Bro17].

5.3. Visual Studio Code (VS Code)

5.3.1. Overview

Visual Studio Code (VS Code) is a lightweight, extensible source-code editor with robust support for Python and data science workflows. Its combination of speed, powerful debugging, and a rich extension ecosystem makes it ideal for rapid development of forecasting pipelines. VS Code's built-in Git integration, notebook support, and seamless terminal enable smooth transitions between data preprocessing, ARIMA parameter tuning, and LSTM model training—supporting best practices in iterative research

[McK10]. The editor’s extensibility, including Python, Jupyter, and Docker plugins, promotes reproducible and modular experimentation.

```

    #!/usr/bin/env python
    # This project implements a walk-forward one-step forecasting system for the Indian NIFTY 50 stock index
    # using time series modeling techniques: ARIMA and LSTM.
    # Author: Nishant ChandraShekar Poojary
    # Email: nishant.chandraShekar.poojary@stud.hs-menden.de
    # Senthil Ramananakere Sunil
    # Email: sendhil.ramananakere.sunil@gmail.com
    # Created: June 2025

    """
    This section overviews Project Overview
    This project implements a walk-forward one-step forecasting system for the Indian NIFTY 50 stock index
    using time series modeling techniques: ARIMA and LSTM.
    # The System
    # Collects Historical NIFTY 50 data from Yahoo Finance via the yfinance API.
    # Cleans the data, filters for weekdays, and adds a COVID-19 dummy variable for the 2020 pandemic period.
    # ARIMAXModel: A statistical model that captures autocorrelation and trends in univariate time series data.
    # = LSTMModel: A deep learning model designed to handle long-term dependencies in multivariate sequences.
    # LSTMModel: A deep learning model designed to handle long-term dependencies in multivariate sequences.
    # Computes error metrics including RMSE and MAPE to evaluate model performance.
    # Displays and saves the forecasted values versus actual prices in graphical form.

    # The application supports forecasting for both 'Open' and 'Close' prices, enabling side-by-side comparison
    # of model predictions across short-term financial windows.
    """

    import os
    import pandas as pd
    import numpy as np
    import math
    import matplotlib.pyplot as plt
    import statsmodels.api as sm
    from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error

    # Load local modules from internal project structure
    from datamodule.datamanager import loadNIFTY50Finance
    from models.arima.arimamodel import runARIMA
    from models.lstm.lstmmodel import runLSTM
    from utils import log
    from utils.log import log_err

    # Load external modules
    import sys
    sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), "util")))
    from util.errorhandler import MessageHandler
    from util.errorlogger import logError

```

Figure 5.2.: VS Code interface with Python project loaded

5.3.2. Installation

To set up VS Code on the most commonly used operating systems, follow these steps:

- **Windows:**

1. Download the installer from the official site (<https://code.visualstudio.com/>).



Figure 5.3.: VS Code official download page

2. Run the installer with administrative permissions and follow the guided steps.
3. Optionally enable “Add to PATH” during installation for terminal accessibility.

- **macOS:**

1. Download the macOS .zip archive from the official site.
2. Extract and drag ‘Visual Studio Code.app‘ into your Applications folder.
3. Optionally install command-line support by running:

```
code .
```

```

187     else:
188         print(msg.get("no_test_index"))
189
190     ##
191     # @section Evaluation
192     #
193     # Compute RMSE and MAPE for both ARIMA and LSTM predictions over valid dates.
194     ##
195     for col in targetColumns:
196         actuals = df.loc[testIndex, col]
197         arimaPreds = arimaResults[col].reindex(testIndex).dropna()
198         lstmPreds = lstmResults[col].reindex(testIndex).dropna()
199
200     validIndex = actuals.index.intersection(arimaPreds.index).intersection(lstmPreds.index)

```

PROBLEMS 321 OUTPUT DEBUG CONSOLE TERMINAL PORTS GIT LENS COMMENTS
○ (3.10.13) nishan@Mac BA25-05-Time-Series %

Figure 5.4.: macOS terminal with VS Code CLI command

- **Linux (Debian/Ubuntu):**

1. Download the ‘.deb‘ package from the official site.
2. Install via:

```
sudo apt install ./code_*.deb
```

3. Optionally verify installation with:

```
code --version
```

5.3.3. System Requirements

Table 5.1.: System Requirements for VS Code on Windows

Requirement	Minimum Specification
Operating System	Windows 10 (64-bit) or later
RAM	1.5 GB free RAM (4 GB recommended)
Processor	1.6 GHz or faster (dual-core recommended)
Disk Space	200 MB for installation
Display	1024 x 768 minimum resolution

Table 5.2.: System Requirements for VS Code on macOS

Requirement	Minimum Specification
Operating System	macOS 10.15 or later
RAM	1.5 GB free RAM (4 GB recommended)
Processor	Intel or Apple Silicon CPU
Disk Space	200 MB for installation
Display	1024 x 768 minimum resolution

Table 5.3.: System Requirements for VS Code on Linux

Requirement	Minimum Specification
Operating System	Ubuntu 20.04+, Debian 10+, Fedora 37+
RAM	1.5 GB free RAM (4 GB recommended)
Processor	1.6 GHz or faster
Disk Space	200 MB for installation
Libraries Required	<code>glibc 2.28, libstdc++ 6.0.24</code>

5.3.4. Configuration

After installation, enhance VS Code for Python development:

- Install the “Python” extension by Microsoft—necessary for syntax highlighting, IntelliSense, and Jupyter integration.
- (Optional) Install additional extensions such as “Pylance” for type analysis, “Docker” for container workflows, and “GitLens” for expanded Git insights.
- Configure linting and formatting tools (e.g. ‘flake8’, ‘black’) to maintain code quality and consistency.
- Set the default interpreter and virtual environment using the Command Palette (“Ctrl+Shift+P” → “Python: Select Interpreter”).

5.3.5. First Steps

To verify the VS Code setup works for Python forecasting tasks:

1. Open your forecasting project folder.
2. Create a new file named `hello.py` with the content:

```
print("Hello, VS Code!")
```

3. Run the script via the Run icon or the built-in terminal:

```
python hello.py
```

4. Create a new Jupyter Notebook ('.ipynb') and test importing key libraries:

```
import numpy as np, pandas as pd
```

5. Observe that VS Code provides inline documentation and error checking, facilitating rapid exploration of libraries relevant to time-series workflows [McK10].

5.3.6. Advantages and Considerations

- **Lightweight and Responsive:** VS Code balances powerful features with low system resource usage—especially useful for iterative model development and version control [McK10].
- **Extensibility:** Its modular extension system supports custom workflows, including notebook-based exploration, Docker deployment, or CI/CD pipelines.
- **Version Control Integration:** Built-in Git tooling and rich plugins such as GitLens streamline code reviews, branching, and collaborative forecasting work.
- **Steeper Setup Curve:** Unlike PyCharm's all-in-one experience, VS Code requires more active configuration to leverage its full potential—though this flexibility allows tailored environments.
- **Community Ecosystem:** Broad extension marketplace ensures continuous support and innovation but also risks feature saturation without careful curation.

5.4. GitHub

5.4.1. Repository Overview

The GitHub repository for the **BA25-05-Time-Series** project serves as the central version-controlled platform for organizing all code, datasets, visual outputs, and supporting documentation. It enables structured and collaborative development of financial forecasting models, specifically for ARIMA and LSTM-based approaches.

The core contents of the repository include:

- **data/** – Cleaned and raw historical NIFTY 50 data used for training and testing models.
- **scripts/** – Core Python scripts implementing ARIMA and LSTM models.
- **results/** – Graphs, predicted vs actual comparison plots, and evaluation metrics (RMSE, MAPE).
- **report/** – LaTeX source files and compiled PDF of the technical report documenting the project.
- **manual/** – User guide containing detailed instructions on running the forecasting models.
- **evaluation/** – Project evaluation materials, including presentation slides and assessment templates.
- **README.md** – Project overview, installation guide, and execution instructions.
- **requirements.txt** – List of dependencies required to reproduce results.

This modular structure ensures reproducibility, clarity, and ease of navigation across different phases of the project.

5.4.2. Project Management Strategy

Effective project management on GitHub enables synchronization across development efforts, helps in maintaining quality standards, and simplifies debugging and iteration during model refinement.

Key elements of the strategy:

1. **Change Tracking:** Version control via commits and branches ensures that every model iteration or data update is logged and reversible. This is crucial for experimental reproducibility.
2. **Collaborative Development:** Contributors operate on feature-specific branches, using pull requests for code review and integration. GitHub Issues are used for task tracking and feature discussions.
3. **Model Iteration Logging:** Each experiment's configuration (e.g., ARIMA order, LSTM sequence length) and results are recorded and stored in the repository, enabling systematic comparison of outcomes.
4. **Dependency Control:** Library versions are locked in **requirements.txt**, ensuring consistent behavior across systems and avoiding environment mismatch errors.

5.4.3. Plan

GitHub serves as the backbone of version control and collaboration for the BA25-05-Time-Series project. It empowers the team to centrally store and synchronize code, documentation, data, evaluation materials, GUI components, and reporting artifacts. This ensures every member accesses the latest updates and prevents fragmentation.

Our implementation strategy is as follows:

1. **Unified GitHub Setup:** All teammates install the same Git client version to ensure consistency in repository interaction. Once setup is complete, we request access from the instructor to join the BA25-05-Time-Series repository. Each member then clones the repo locally to their machine.
2. **Coordinated Updates:** Before making significant changes—such as a report update, adding evaluation files, or modifying scripts—team members notify the rest of the group (via chat or email) to avoid simultaneous updates that could lead to merge conflicts.
3. **Software Version Alignment:** We agree on standardized versions for key tools and libraries (e.g., Git, Python, TensorFlow, matplotlib). These are documented in a software environment table so that everyone runs the same environment, reducing errors driven by version mismatches.

Category	Tool / Package and Version
System Tools	GitHub Desktop = 3.3.18 (x64) Visual Studio Code = 1.89.1 Python = 3.10.13 LaTeX (TeXstudio) = 4.7.3
Core Data and ML	numpy ≥ 1.24 , < 2.0 pandas ≥ 1.5 , < 2.0 python-dateutil $\geq 2.9.0$, < 3.0 pytz ≥ 2025.2 , < 2026 scipy ≥ 1.8 , < 2.0 scikit-learn ≥ 1.2 , < 2.0 statsmodels ≥ 0.13 , < 0.14 pmdarima ≥ 2.0 , < 3.0 yfinance ≥ 0.2 , < 0.3
Visualization	matplotlib ≥ 3.7 , < 4.0 altair = 4.2.2
TensorFlow	tensorflow = 2.13.1 (non-macOS) tensorflow-macos = 2.13.1 (macOS only) tensorflow-estimator = 2.13.0
Typing Support	typing_extensions $\geq 3.6.6$, < 4.6.0
Testing & Linting	pytest ≥ 8.3 , < 9.0 pytest-cov ≥ 6.1 , < 7.0 coverage ≥ 7.8 , < 8.0 flake8

Table 5.4.: Software Stack and Dependency Version

4. **Single-Source Data Management:** Only one teammate is responsible for updating the `data/` directory. After preprocessing or updating, that person commits the changes, ensuring data consistency. Weekly snapshots of the dataset are backed up using versioned filenames to preserve experiment reproducibility.
5. **Local Backups for Critical Software:** To guard against service outages or discontinued versions, we download installers for key tools (e.g., Python, VS Code), store them in a shared cloud folder, and ensure each teammate can restore consistent development environments at any time.

This strategy fosters a structured workflow, minimizes merge conflicts, and supports reliable version control—ensuring that code, data, and documentation remain coherent and reproducible across the team.

Part IV.

Methodology

6. Methodology

6.1. Cross-Industry Standard Process for Data Mining

The CRISP-DM methodology is widely regarded as the *de facto standard* for data mining projects due to its domain- and tool-independence, and its iterative life-cycle structure [WH00; SS21]. Initially defined with six phases—Business Understanding, Data Understanding, Data Preparation, Modeling, Evaluation, and Deployment—it supports repeated feedback loops, reflecting the non-linear nature of data mining tasks [WH00].

Designers chose CRISP-DM for its flexibility: it allows revisiting previous phases to refine objectives, adjust feature processing, or re-tune models as insights emerge [SS21]. A 2021 systematic review confirmed that CRISP-DM remains prevalent in both academic and industrial data science use cases [SS21]; however, many implementations underutilize the Deployment phase, an issue especially critical in production-oriented forecasting systems [RMG20].

Researchers have also proposed extensions to CRISP-DM, such as DMKD (Data Mining and Knowledge Discovery) and adapted frameworks aimed at addressing domain-specific needs like sensor data preprocessing, ontological alignment, or production deployment in real-time settings [MMS09]. A recent finance-focused study identified eighteen operational gaps when applying CRISP-DM to financial services, underscoring the need for adaptation in privacy, governance, and real-time integration [RMG20].

6.2. CRISP-DM process flow for the project

1. Business Understanding
 - Define the business goal: Predict future NIFTY 50 stock prices using ARIMA and LSTM models.
 - Context: Enable users (via GUI or Streamlit app) to make informed investment decisions.
 - Relevant files/folders: Manual/, InitialProjectPlan/, Presentations/, README.md

2. Data Understanding

- Collect and examine data sources like cached_nifty50.csv or yFinance APIs.
- Understand trends, anomalies, and data structure (Open, Close, COVID dummy variables).
- Relevant files/folders: Code/dataHandler/dataHandler.py

3. Data Preparation

- Clean, transform, and scale the data.
- Split into training/test sets, add dummy variables, and prepare sequences for LSTM.
- Relevant files/folders: Code/dataHandler/dataHandler.py, cached_nifty50.csv, models/scalerOpen.pkl, models/scalerClose.pkl

4. Modeling

- Apply ARIMA and LSTM models for forecasting Open and Close prices.
- Use separate model files and retrain or load pretrained .keras and .pkl models.
- Relevant files/folders: Code/models/arima/arimaModel.py, Code/models/lstm/lstmModel.py, models/, forecast_plot_Open.png

5. Evaluation

- Assess model performance using RMSE, MAPE, and visual plots.
- Compare actual vs predicted data and refine model parameters.
- Relevant files/folders: forecast_results.csv, forecast_plot_Open.png, forecast_plot_Close.png, Code/tests/, log.txt

6. Deployment

- Expose predictions through a user-friendly GUI (stockPredictorGui.py) or Streamlit app.
- Package with requirements.txt and deploy on local machine or Streamlit Cloud.
- Relevant files/folders: GUICode/, stockPredictorGui.py, requirements.txt, .github/workflows/ci.yml

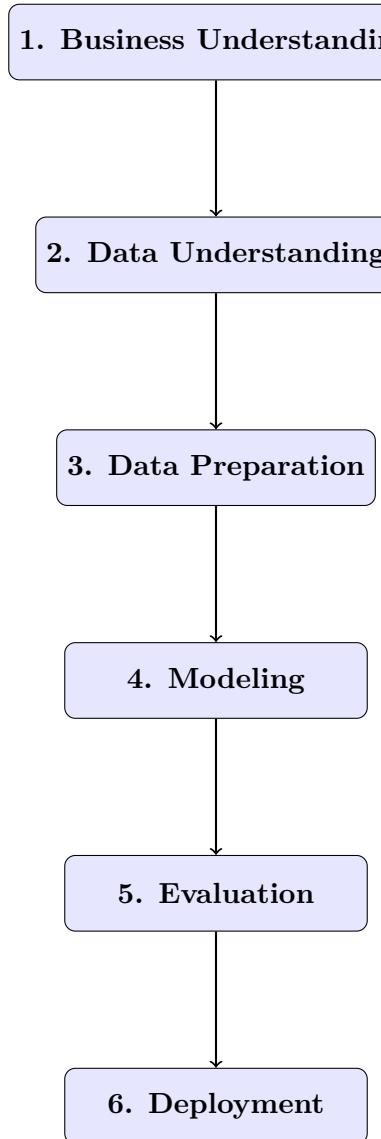


Figure 6.1.: CRISP-DM Process Applied to NIFTY 50 Forecasting System

6.3. KDD (Knowledge Discovery in Databases)

KDD refers to the overall process of discovering actionable insights from raw data, of which data mining is a critical step [FPSS96]. Originating in the mid-1990s—pioneered by Fayyad et al.—the KDD process consists of five key stages: Selection, Preprocessing, Transformation, Data Mining, and Interpretation/Evaluation [FPSS96].

- **Selection:** Define the target data and critical variables.
- **Preprocessing:** Address missing values, noise, inconsistencies, and duplicates.

- **Transformation:** Convert data to suitable formats and reduce dimensionality.
- **Data Mining:** Apply algorithms like classification, regression, clustering.
- **Interpretation/Evaluation:** Assess results, validate against objectives, and prepare insights for deployment [FPSS96].

KDD's emphasis on *translation and interpretation* distinguishes it—as it ensures that extracted patterns deliver value to end-users, rather than just generating technical outcomes. It laid the theoretical groundwork upon which CRISP-DM and similar frameworks were later built [FPSS96]. Additionally, the international symposium and ACM SIGKDD conferences—established by figures like Usama Fayyad in the early 1990s—have shaped the field of knowledge discovery, serving as the main venue for groundbreaking research.

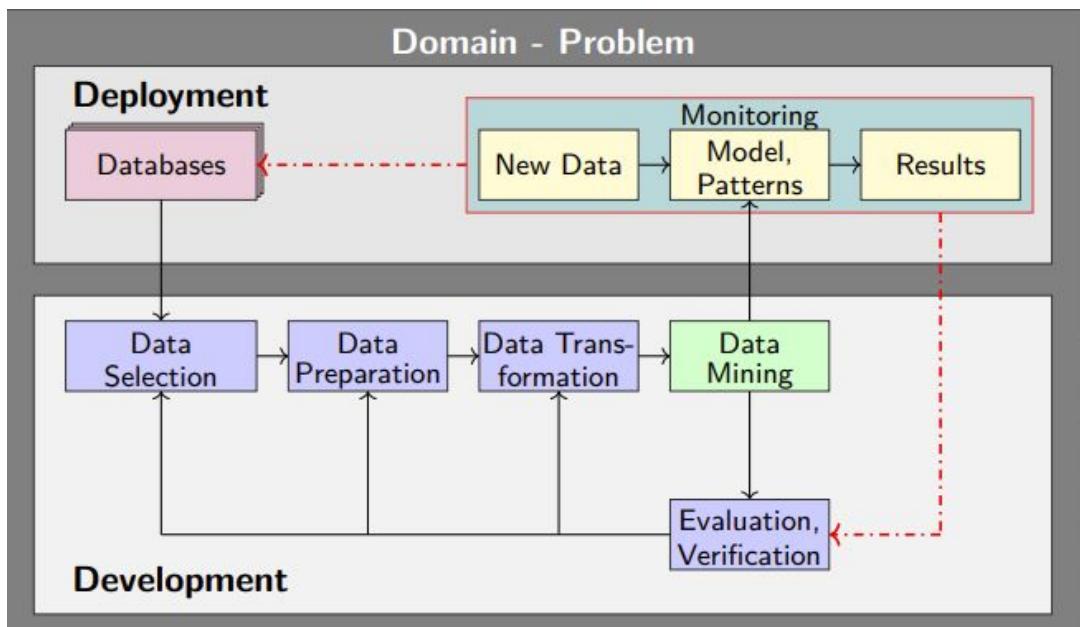


Figure 6.2.: KDD Process Workflow

A recent systematic review by Kumbharana [Kum22] outlines a revised KDD process with nine key phases: understanding the application domain, data selection, cleaning, enrichment, transformation, data mining, evaluation, knowledge consolidation, and use.

This expanded view emphasizes that effective knowledge discovery requires not just algorithmic modeling but also domain understanding and decision-making integration. Unlike narrow algorithm-centric approaches,

KDD treats data mining as just one component within a larger discovery framework. These improvements are especially relevant for modern systems such as financial forecasting tools, which must be interpretable, actionable, and directly usable by decision-makers.

By adopting a modern KDD approach, the current project ensures that insights derived from LSTM and ARIMA models are meaningfully evaluated and presented to users in a form that supports real-time forecasting decisions.

6.4. KDD Flow for the project

1. Selection

- Identify and acquire relevant stock market data, That is NIFTY 50 Open and Close prices.
- Choose the relevant columns (e.g., Date, Open, Close, High, Low) for modeling.
- Relevant files/folders: `cached_nifty50.csv` , `Code/dataHandler/dataHandler.py`

2. Preprocessing

- Clean the data (e.g., handle missing values like '-') and ensure only valid trading days are considered.
- Convert dates, normalize numerical formats, and remove irrelevant records.
- Add derived features like the COVID dummy variable.
- Relevant files/folders: `Code/dataHandler/dataHandler.py` ,`cached_nifty50.csv`, `log.txt`

3. Transformation

- Reshape data for machine learning models (e.g., rolling windows for LSTM).
- Normalize/scale features using MinMaxScaler for LSTM inputs.
- Prepare ARIMA-compatible series.
- Relevant files/folders: `models/scalerOpen.pkl`, `models/scalerClose.pkl` , `Code/models/lstm/lstmModel.py` , `Code/models/arima/arimaModel.py`

4. Data Mining

- Apply modeling techniques to extract patterns and make predictions

- LSTM for nonlinear temporal dependencies.
- ARIMA for linear trends and seasonality.
- Relevant files/folders: Code/models/arima/arimaModel.py, Code/models/lstm/lstmModel.py, models/.keras, models/.pkl

5. Interpretation/Evaluation

- Analyze model performance using metrics (RMSE, MAPE).
- Visualize predictions versus actuals using plots and logs.
- Use results to refine models or inform decision-making.
- Relevant files/folders: forecast_results.csv, forecast_plot_Open.png, forecast_plot_Close.png, Code/tests/test_*.py, log.txt

6. Optional Deployment / Integration (if extending KDD)

- Integrate the knowledge into a user-facing application.
- Allow real-time interaction, prediction viewing, and model selection.
- Relevant files/folders: GUICode/stockPredictorGui.py , requirements.txt, .github/workflows/ci.yml

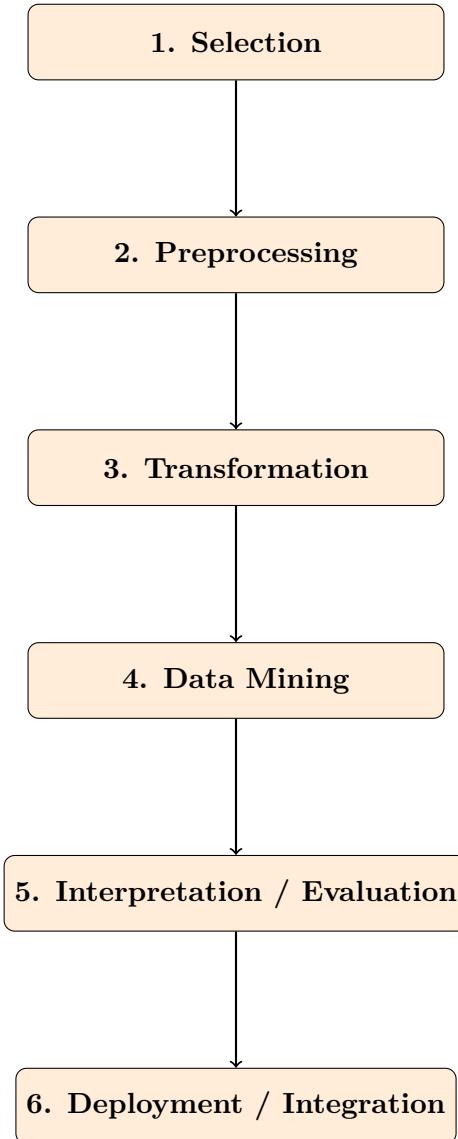


Figure 6.3.: Knowledge Discovery in Databases (KDD) Process Applied to Stock Forecasting

6.5. Relevance to This Project

Our project's architecture aligns naturally with these frameworks:

- **Business Understanding / Selection:** The project focuses on short-term forecasting of the NIFTY 50 index—defining precise objectives and the scope of data variables.
- **Data Understanding / Preprocessing:** We extract historical price and feature data for modeling.

- **Modeling:** The project implements and compare classical ARIMA and deep-learning LSTM approaches.
- **Evaluation / Interpretation:** For evaluation we use metrics like RMSE and MAPE and visualize prediction intervals, translating quantitative results into meaningful insights.
- **Deployment:** End-user interface abstracts technical complexity, enabling day-ahead forecasts—addressing a frequent gap in CRISP-DM implementations [RMG20].

Together, CRISP-DM and KDD provide a strong, research-validated backbone, ensuring your methodology is robust, comprehensive, and firmly grounded in best practices.

6.6. Machine Learning Pipeline

6.6.1. Problem Definition

This project aims to forecast the next-day opening and closing prices of the NIFTY 50 index. It combines traditional statistical methods (ARIMA) with deep learning (LSTM) to leverage both interpretability and predictive power. The methodology is designed to support non-technical users through an interactive interface, while maintaining rigor in data handling, modeling, and evaluation.

6.6.2. Data Acquisition

Historical NIFTY 50 data is sourced from reliable platforms such as Yahoo Finance or Alpha Vantage. The dataset includes daily open, high, low, close, and volume values. The data is collected using APIs and processed into a format suitable for time-series modeling.

6.6.3. Data Preprocessing

Table 6.1.: Data Preprocessing Steps for ARIMA and LSTM

Step	ARIMA	LSTM
Missing Value Handling	Forward/backward fill to maintain continuity	Same approach to preserve temporal structure
Stationarity Check	ADF and KPSS tests to guide differencing	Not required; LSTM handles non-stationary data
Transformation	Differencing for stationarity	Min-Max normalization for numerical stability
Train-Test Split	Chronological split without shuffling	Chronological split preserving temporal ordering

6.6.4. Model Building

ARIMA Pipeline

The ARIMA model is built using a systematic process involving:

- Selection of parameters (p, d, q) using AIC minimization or auto-ARIMA.
- Fitting the model on the training portion of the time series.
- Forecasting one day ahead.
- Inverting differencing to convert predictions to the original scale.

LSTM Pipeline

The LSTM model follows a neural architecture that includes:

- Input sequences created using a sliding window technique.
- A network of one or more LSTM layers, followed by a Dense layer with a linear activation function.
- Training using the Adam optimizer and MSE as the loss function.
- Forecasts are inverse-transformed after prediction to obtain actual values.

6.6.5. Model Evaluation

Model performance is evaluated using:

- **Error Metrics:** RMSE, MAPE, MAE, and R².
- **Walk-forward validation** to simulate real-time forecasting conditions.
- **Residual analysis** (for ARIMA) to validate model assumptions using autocorrelation tests.
- **Prediction intervals** to communicate forecast uncertainty to users.

6.6.6. Visualization and Interpretation

- Time series plots for actual vs. predicted values.
- Residual plots for checking randomness and error distribution.
- Interval-based visualizations to show uncertainty margins.

6.6.7. Deployment and Interface

- A user-friendly frontend is developed using Streamlit or Flask.
- The backend handles preprocessing, prediction, and visualization.
- Users can upload datasets, choose models, and receive predictions with intuitive graphics and metrics.

6.6.8. Logging and Reproducibility

- Versioned model files saved using **joblib** or **HDF5**.
- Prediction logs maintained for audit trails and accuracy tracking.
- Modular codebase ensures reproducibility and easy future extension.

6.7. ML Pipeline Diagram

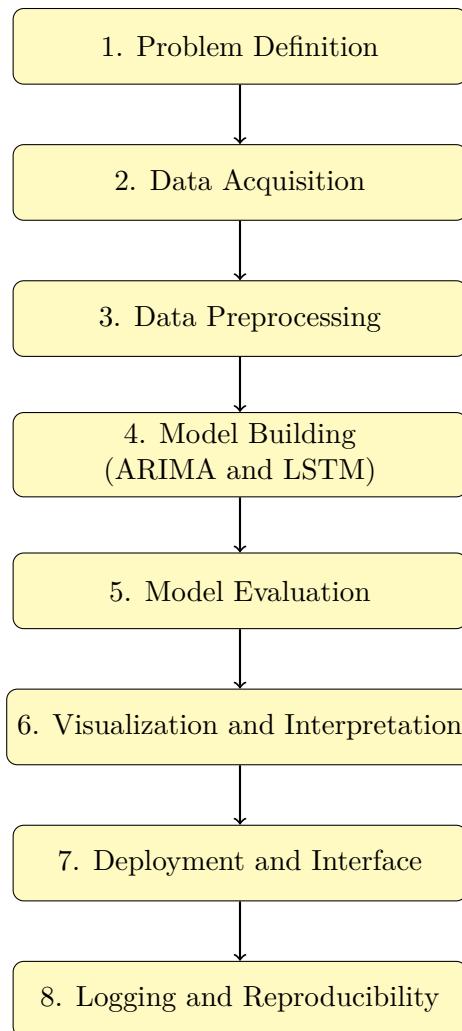


Figure 6.4.: Machine Learning Pipeline for NIFTY 50 Forecasting

Part V.

Application Development

7. Data

7.1. Data Structure

The dataset is sourced from Yahoo Finance for the NIFTY 50 index consists of structured time-series data organized by trading days. Each record (row) corresponds to one trading session of the National Stock Exchange (NSE) of India, starting from January 01, 2008, and includes seven key features:

Table 7.1.: Yahoo Finance NIFTY 50 Dataset: Column Descriptions

Column	Description
Date	The calendar date of the trading session, formatted as YYYY-MM-DD.
Open	The price of the NIFTY 50 index at the beginning of the trading day.
High	The maximum price reached by the index during the trading day.
Low	The minimum price recorded for the index on the same day.
Close	The final price at which the index closed at the end of the trading session.

The data is inherently chronological and non-stationary, meaning the value of any given day is dependent on time and prior values. This structure supports use in forecasting models like ARIMA and LSTM, which require well-indexed time-series data for sequential learning.

Internally, the data is stored and manipulated using Pandas DataFrames in Python, enabling advanced filtering, time-based indexing, and resampling. Time-indexing is essential for computing moving averages, volatility metrics, and seasonal decomposition.

7.2. Data Size

As of mid-2025, the dataset spans from January 01, 2008 to June 2025, covering approximately 4,200–4,300 trading days, considering about 250–260 trading sessions per year (excluding weekends and Indian public holidays).

- Rows: 4,300 (one for each trading day)
- Columns: 7 (as listed above)
- Data types: datetime64 for date, float64 for prices, and int64 for volume
- On disk (CSV format): 250 KB
- In-memory (Pandas DataFrame): 300 KB to 500 KB depending on Python overhead

This makes the dataset relatively lightweight and highly manageable, especially for experimentation with deep learning models or statistical forecasting approaches.

7.3. Data Format

The historical stock market data for the NIFTY 50 index is obtained using the yfinance API, which provides access to reliable financial data from Yahoo Finance. This data is programmatically downloaded in CSV format, making it convenient for subsequent preprocessing and analysis within the machine learning pipeline.

[image of the sample of the dataset](#)

7.4. Data Anomalies

Although most of the data is well-structured and reliable, specific market disruptions and systemic anomalies are observable:

1. COVID-19 Pandemic (Feb–Apr 2020): Marked by extreme volatility, with daily swings exceeding 5–10%. Indian stock market hit circuit breakers on March 13 and March 23, 2020. Massive increase in trading volumes and volatility clustering, particularly visible in Volume, High-Low spreads, and Open-Close gaps.
2. 2008 Global Financial Crisis: The dataset starts shortly before the Lehman Brothers collapse (Sept 2008). Severe downturn in NIFTY index values between October 2008 – March 2009. Regular limit hits and erratic intraday behavior with steep drawdowns.
3. 2016 Demonetization Shock (Nov 8, 2016): Announced after market hours; caused significant price shock the following day. Markets opened significantly lower due to liquidity fears.

4. 2020–2022 Monetary Policy and Inflation Waves: Multiple rate hikes by the Reserve Bank of India (RBI) and the US Federal Reserve led to sharp intraday reversals. Periods of increased uncertainty are visible as spikes in volume and volatility, even if the closing prices stabilized.
5. Geopolitical Events: Tensions along the India-China border (mid-2020) and Russia-Ukraine war (early 2022) caused temporary drawdowns. Noticeable as price gaps, where the next day's Open differs drastically from the previous Close.

Handling Anomalies:

- These anomalies were not removed, as they represent high-signal market behavior. Instead, they were flagged using rolling z-scores, CUSUM analysis, and seasonal decomposition techniques.
- For machine learning models like LSTM, these anomalies were sometimes marked with auxiliary binary flags to help the network learn volatility regimes.

7.5. Live data Integration

Initially, we began with a *static* CSV dataset containing historical NIFTY 50 prices enriched with fundamental ratios—price-to-earnings (PE) and price-to-book (PB)—sourced from a Kaggle dataset covering January 2008 through December 2023.¹ However, this dataset quickly became stale and lacked any data beyond 2023, making real-time forecasting impossible.

To overcome these limitations, we integrated `yfinance` to fetch live data at runtime:

Listing 7.1: Switching from static CSV to live yfinance data

```
from dataHandler.dataHandler import loadNifty50Yfinance
# Legacy: pd.read_csv("nifty50.csv")
# New: live download
dfLive = loadNifty50Yfinance(start="2000-01-01", end="2025-01-01
    ")
```

¹Kaggle. “NIFTY 50 Historical Data with PE and PB Ratios.” <https://www.kaggle.com/datasets/sudalairajkumar/nifty-indices-dataset/data>, accessed April 2025.

Data Links and Features

- Kaggle CSV (Static):
 - Contained columns: **Date, Open, High, Low, Close, PE, PB.**
 - Updated only periodically; manual downloads required.
- yfinance API (Live):²
 - Provides up-to-date **Open, High, Low, Close, Volume, Dividends, Stock Splits.**
 - No PE/PB ratios—enables extension via separate fundamental-data APIs if needed.
 - Automatically handles weekends, holidays, and timezone adjustments.

Why Live Integration Matters

- **Timeliness:** Forecasts always use the latest available market data, avoiding stale inputs.
- **Automation:** Eliminates manual CSV downloads and merging scripts—reduces maintenance burden.
- **Extensibility:** New features (e.g. exogenous COVID dummy, macroeconomic indicators) can be fetched dynamically alongside price data.

7.6. Data Origin

The data is sourced from Yahoo Finance, a widely used and reputable provider of historical market data, particularly for retail and academic use. Specifically, the data was retrieved from:

<https://finance.yahoo.com/quote/%5ENSEI/history/>

The dataset is accessed using the open-source yfinance Python package, which wraps the Yahoo Finance API, enabling efficient, scriptable access.

- **Reliability:** While Yahoo Finance is not an official exchange source, its NIFTY data has been shown to align closely with official NSE publications.

²Yahoo Finance. “yfinance Documentation.” <https://pypi.org/project/yfinance/>, accessed June 2025.

- **Updates:** Near real-time, with daily refreshes after market close.
- **Licensing:** Free to use for non-commercial research purposes, as permitted under Yahoo's TOS.

8. Developer documentation

The development of this stock price forecasting system followed a modular and structured approach inspired by established software engineering and machine learning best practices. Modularization was crucial to separate concerns across the system—data preprocessing, model training, evaluation, and deployment—facilitating reusability, maintainability, and ease of testing [Wil+17]. The system was designed to predict both Open and Close prices of NIFTY 50 stocks, leveraging time series forecasting models suited for different data dynamics. ARIMA, a widely used statistical approach for linear, stationary series, was chosen for its interpretability and efficient parameter tuning using auto-arima methods [HA21]. In parallel, Long Short-Term Memory (LSTM) networks were employed to capture nonlinear dependencies and long-term temporal patterns in financial sequences, consistent with their proven performance in financial forecasting tasks [Wan+21; GSC00b].

The architectural strategy prioritized deployment through both a PyQt6 desktop interface and a Streamlit web interface. This dual-interface design aimed to enhance accessibility for both offline and online use cases while allowing dynamic user interaction with predictions. To ensure flexibility, the models were trained separately and saved as serialized objects (`.pkl` for ARIMA and `.keras` for LSTM), allowing them to be loaded dynamically during runtime. Evaluation metrics such as Root Mean Squared Error (RMSE) and Mean Absolute Percentage Error (MAPE) were integrated for model comparison, adhering to standard time series model evaluation frameworks [MSA18].

The project also included automated testing, logging, and version control integration via GitHub Actions to enable continuous integration and delivery. This setup supports ongoing model improvements without disrupting the user interface or core prediction logic, aligning with modern MLOps practices [AZC22]. Throughout, the system adhered to documented software and machine learning principles to ensure transparency, reproducibility, and robustness.

8.1. Development Idea

The core idea behind the system was to develop a practical, modular forecasting tool tailored for financial applications—specifically the prediction

of Open and Close prices in the NIFTY 50 index. The system aimed to support informed decision-making by combining statistically grounded time-series forecasting techniques with accessible user interfaces. This approach aligns with recommendations in current financial machine learning research, which emphasize the value of combining model robustness, user interpretability, and real-time responsiveness [GKX20; Pat+15a].

Key design considerations included:

- **Model performance evaluation:** Forecasting accuracy is quantitatively assessed using RMSE (Root Mean Squared Error) and MAPE (Mean Absolute Percentage Error), as these are widely accepted in time-series evaluation literature [HA18].
- **Efficiency via pre-trained models:** Loading pre-trained ARIMA and LSTM models (.pkl and .keras formats, respectively) allows the system to provide instant predictions without retraining on each use. This is consistent with deployment strategies in production-grade machine learning pipelines [Bro17].
- **Real-time data input:** The system uses a **Streamlit** application to allow user-uploaded data and live predictions. Streamlit is a Python-based framework known for enabling fast web deployment of data science tools [Str25].
- **Modular separation of concerns:** The architecture clearly separates:
 - *Logic* (model implementation: `lstmModel.py`, `arimaModel.py`),
 - *Data handling* (cleaning and transformation: `dataHandler.py`),
 - *Presentation layer* (GUI or Streamlit app: `stockPredictorGui.py`).

This clean separation enhances maintainability and supports future extensibility [Fow18].

By anchoring the project on these principles, the development ensures transparency, repeatability, and adaptability to evolving market data and user requirements.

8.2. Development Flowchart (Conceptual Overview)

The system development followed a structured and modular pipeline designed for financial forecasting. This structure ensured that model performance, data reliability, and user interface accessibility were all

addressed independently yet cohesively. Figure ?? summarizes the conceptual flow.

1. Planning & Requirement Analysis

- Define project objectives: short-term forecasting of NIFTY 50 `Open` and `Close` prices, using regression metrics like RMSE and MAPE for evaluation [HA18].
- Select modeling strategies: ARIMA for linear/stationary series [Box+15], LSTM for nonlinear temporal learning [HS97].
- Specify interface design requirements: support both desktop GUI and Streamlit-based web access [Str25].

2. Data Handling Pipeline

- Source historical NIFTY 50 data via yFinance API or from local CSV cache [Yfi25b].
- Perform preprocessing: remove missing values, convert dtypes, and filter weekends [Bro17].
- Add engineered features such as a binary COVID dummy to represent pandemic-affected periods [GKX20].

3. Model Development

- **ARIMA:** Fit on differenced training data; save the model as `.pkl` using `statsmodels` [Box+15].
- **LSTM:** Normalize inputs using `MinMaxScaler`; construct rolling sequences; train and save the model in `.keras` format along with scalers as `.pkl` [Bro17; GS23].

4. Evaluation Phase

- Predict on test set and compare forecasts with true values using RMSE, MAPE, and plotted overlays [HA18].
- Generate visual output for interpretability and store plots for auditability and user reference.

5. Deployment Interface

- Use PyQt6 or Streamlit to expose predictions to end-users with date selector, data upload, and model selection [Str25].
- Load models and scalers dynamically; provide prediction results and model evaluation.

6. Automation and Testing

- Write test suites using `pytest` for each logic component (data preprocessing, ARIMA, LSTM, CLI).
- Implement GitHub Actions for continuous integration and regression testing on each commit [Git].
- Track runtime behavior using a log file (`log.txt`) and generate API and code documentation via `Doxxygen` [Don25].

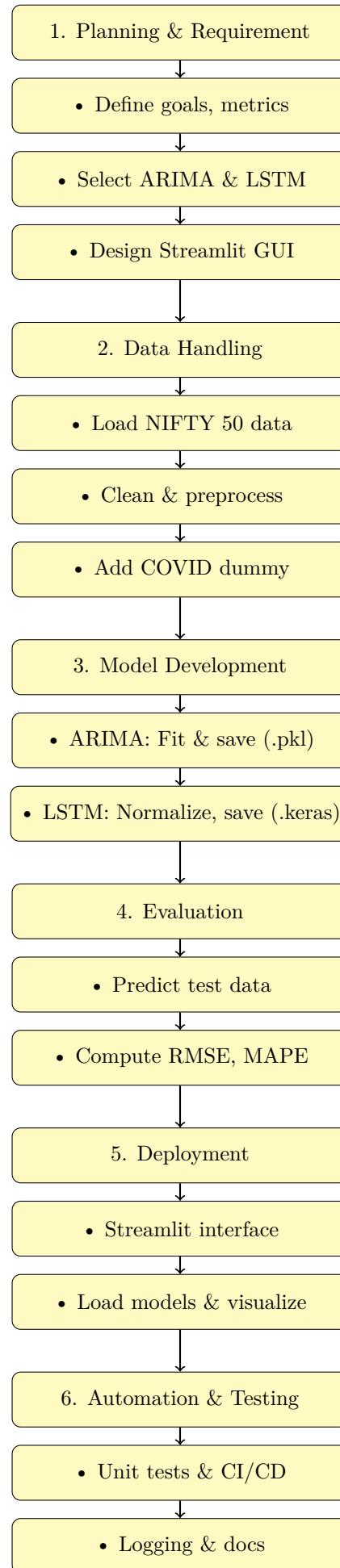


Figure 8.1.: Condensed Development Flowchart for NIFTY 50 Forecasting

8.3. Machine Learning Pipeline

This project employs a structured machine learning pipeline to forecast stock prices of the NIFTY 50 index. The pipeline includes all essential stages from data acquisition to model deployment, supporting both ARIMA and LSTM models. The system is modularly designed to allow reproducibility, efficient debugging, and flexibility in model integration [Box+15; HS97; HA18].

8.3.1. Input Data Acquisition

Source of Data: Historical stock price data for the NIFTY 50 index is obtained directly from Yahoo Finance using the `yfinance` Python package. This API enables automated, up-to-date downloading of time series data without manual intervention [Yfi25b].

- **Tool used:** `yfinance.download()`
- **Columns acquired:** Open, High, Low, Close
- **Time range:** January 2008 to the current date
- **Frequency:** Daily observations

Offline Fallback: To ensure functionality when internet access is unavailable, a backup CSV file (`cached_nifty50.csv`) is stored locally. This contains previously downloaded data for redundancy.

8.3.2. Preprocessing

Before training any model, the data undergoes several cleaning and transformation steps to ensure quality and consistency.

- **Data Cleaning:** Rows containing missing values or non-numeric entries (e.g., "-") are removed or converted. This is necessary to prevent training errors and ensure numeric consistency [HA18].
- **Datetime Conversion:** The 'Date' column is explicitly converted into datetime format and set as the DataFrame index. This enables efficient time-based operations such as lookbacks and rolling windows.
- **Sorting:** The dataset is sorted in ascending order based on the datetime index to maintain chronological consistency.

- **Feature Engineering:** A binary COVID dummy variable is added. This variable is set to 1 for all dates between March 2020 and December 2020, and 0 otherwise. This helps the models adjust for abnormal fluctuations during that period.
- **Normalization:** The 'Open' and 'Close' prices are scaled to the [0, 1] range using the `MinMaxScaler` from `scikit-learn`. This is essential for the LSTM model as neural networks are sensitive to input scale. The ARIMA model does not require normalization but expects stationary input.

8.3.3. Model Selection

Two model types were chosen based on their interpretability, popularity, and performance in time series forecasting.

- **ARIMA:** The AutoRegressive Integrated Moving Average model captures linear temporal dependencies using lagged variables and differencing. Parameter selection is handled automatically by `pm-darima.auto_arima()` using AIC and BIC scores [Box+15].
- **LSTM:** Long Short-Term Memory networks, a subclass of recurrent neural networks (RNNs), are used for their ability to learn long-range dependencies in sequential data [HS97]. The architecture includes an input layer, one or more LSTM layers, and a dense output layer implemented in TensorFlow/Keras.

8.3.4. Model Training

Each model is trained independently on preprocessed data, using separate scripts and configurations for the 'Open' and 'Close' prices.

ARIMA Training Steps:

- Data is differenced to achieve stationarity as required by ARIMA.
- The best parameters (p , d , q) are identified via AIC/BIC minimization.
- Models are trained using `auto_arima` and saved to disk using `joblib`.
- Files saved include `arimaModelOpen.pkl` and `arimaModelClose.pkl`.

LSTM Training Steps:

- A sliding window of size 60 days is used to generate input-output pairs.
- Input tensors are reshaped to 3D: (samples, time steps, features) for LSTM compatibility.
- Models are compiled with the Adam optimizer and trained using early stopping to avoid overfitting.
- Final models are saved as `lstmModelOpen.keras` and `lstmModelClose.keras`.
- Corresponding scalers are also saved as `scalerOpen.pkl` and `scalerClose.pkl` to ensure correct inverse scaling during inference [GBC16].

8.3.5. Prediction

Model inference is conducted in two stages—one for 'Open' and one for 'Close' prices. A prediction is made one day into the future.

- **ARIMA:** Uses `model.predict(n_periods=1)` on the most recent differenced data.
- **LSTM:** Uses the last 60 normalized values to predict the next time step. The output is then inverse-transformed to match original price scale.

8.3.6. Evaluation

Two error metrics are used for performance evaluation: RMSE and MAPE.

- **Root Mean Squared Error (RMSE):** Measures average magnitude of errors. More sensitive to large errors.
- **Mean Absolute Percentage Error (MAPE):** Measures relative prediction error in percentage terms, making it interpretable across different price levels [chicco2021rmse].

Visual Evaluation: The predicted vs actual prices are plotted using Matplotlib for both models. For LSTM, a scatter plot and overlay plot are generated to assess linear alignment and forecast deviation.

8.3.7. Deployment and Output

The models are deployed using a Streamlit-based web application for user interaction.

- **Streamlit Interface:** Users can select prediction dates, view visual outputs, and initiate real-time predictions. The application fetches the latest NIFTY 50 data dynamically through the `yfinance` API.
- **Logging:** All operations (predictions, errors, uploads) are logged into a text file (`log.txt`) for audit and debugging.
- **Documentation:** The system is documented using Doxygen with inline comments, enhancing developer understanding and reproducibility.

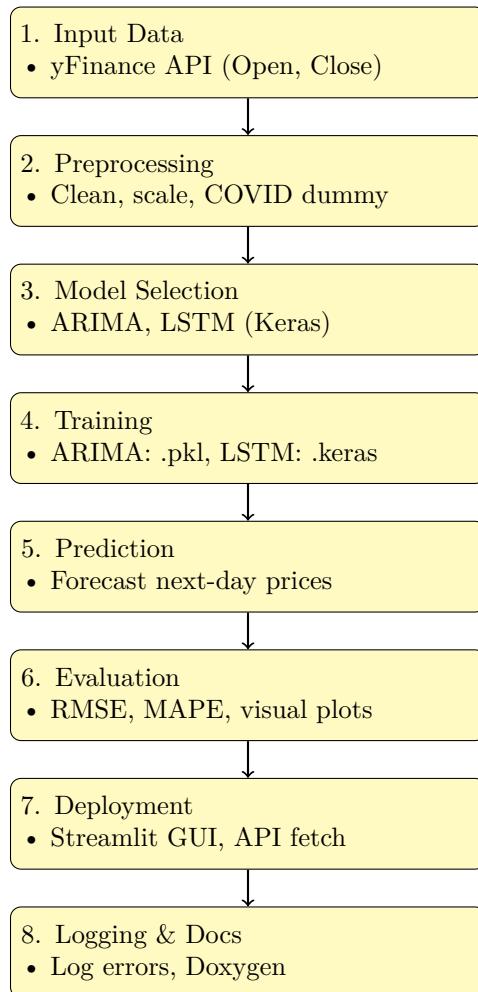


Figure 8.2.: Condensed ML Pipeline for NIFTY 50 Forecasting

Automatic Model Replacement

- **Automatic Model Replacement**

Upon every merge to `main`, our CI pipeline runs training in the core project and then copies the newly produced ARIMA (`.pkl`) and LSTM (`.keras`) artifacts directly into the GUI's `models/` folder:

Listing 8.1: CI/CD step to replace pretrained model and scaler files before deployment

```
# .github/workflows/ci.yml excerpt
- name: Replace model files
  run: |
    cp models/arima/arimaModelOpen.pkl GUICode/Code/models/
      ↵ arimaModelOpen.pkl
    cp models/arima/arimaModelClose.pkl GUICode/Code/models/
      ↵ /arimaModelClose.pkl
    cp models/lstm/lstmModelOpen.keras GUICode/Code/models/
      ↵ lstmModelOpen.keras
    cp models/lstm/lstmModelClose.keras GUICode/Code/models/
      ↵ /lstmModelClose.keras
    cp models/lstm/scalerOpen.pkl GUICode/Code/models/
      ↵ scalerOpen.pkl
    cp models/lstm/scalerClose.pkl GUICode/Code/models/
      ↵ scalerClose.pkl
```

- Eliminates manual sync errors by ensuring the GUI always serves the latest validated models.
- Speeds up deployment: no hand-copying or human steps between training and release.
- Keeps versioning tight: model version in the GUI repo always matches the most recent training commit.

8.4. Program

8.4.1. Structure and Modules

The NIFTY 50 forecasting application is organized into the following top-level directories and modules. Each module encapsulates a distinct responsibility in the end-to-end pipeline.

- `dataHandler/`
 - `dataHandler.py` – Fetches and cleans historical NIFTY 50 data via `yfinance`, filters non-trading days, injects the COVID dummy, and selects relevant columns.
- `models/`
 - `arima/`

- * **arimaModel.py** – Implements `runArima()`, which performs walk-forward ARIMA forecasting using `pmdarima.auto_arima` and `statsmodels` and saves the final model to `.pkl`.
- **lstm/**
 - * **lstmModel.py** – Defines `buildAndTrainLstm()` and `runLstm()`, handling sequence creation, scaling, LSTM training, walk-forward prediction, and model/scaler serialization (`.keras`, `.pkl`).
- **utils/**
 - **messageHandler.py** – `MessageHandler` loads UI and log templates from `messages.json`.
 - **errorHandler.py** – `logError()` centralizes exception logging with context and full traceback.
- **main.py**
 - Orchestrates the full pipeline: loads data, runs ARIMA and LSTM forecasts, computes metrics, prints next-day forecasts, and generates comparison plots via `matplotlib`.
- **models/**
 - **arimaModelOpen.pkl**, **arimaModelClose.pkl** – Saved ARIMA models.
 - **lstmModelOpen.keras**, **lstmModelClose.keras** – Saved LSTM architectures and weights.
 - **scalerOpen.pkl**, **scalerClose.pkl** – `MinMaxScaler` instances for inverse transformation.
- **assets/**
 - **messages.json** – Locale-agnostic text for UI and logs.

Module Responsibilities

- **dataHandler.py**
 - `loadNifty50Yfinance(start, end)`: returns a cleaned `pd.DataFrame` indexed by `Date`, with only weekdays, no missing `Close`, and a `COVID_dummy` column.
- **arimaModel.py**
 - `runArima(df, columns, retrain=True)`: performs walk-forward ARIMA forecasting on each target column, returning a `dict` of `pd.Series` and optionally saving the last model.

- **lstmModel.py**
 - `buildAndTrainLstm(dfTrainScaled, targetColumn, lookback, epochs, batchSize)`: returns a trained `tf.keras.Sequential` model or `None` if data is insufficient.
 - `runLstm(df, columns)`: executes rolling LSTM training and one-step forecasts, returns a `dict` of `pd.Series`.
- **messageHandler.py**
 - `MessageHandler.get(key)`: retrieves localized strings or returns `[key]` if missing.
- **errorHandler.py**
 - `logError(error, context)`: logs an error summary and full traceback to `log.txt`.
- **main.py**
 - Coordinates data loading, forecasting, evaluation, and visualization, leveraging all other modules.

8.4.2. Parameter Handling

[Unverified] The following parameters control data loading, model training, and forecasting behavior. Ranges and recommendations are based on common practice and the TensorFlow documentation [Ten25].

start, end (in loadNifty50Yfinance) • *Purpose:* Define the inclusive date range for historical data download.

- *Good Range:* Any valid ISO-8601 dates within market history (e.g., "2008-01-01" to today).
- *Tuning:* Shorter windows (e.g., last 5 years) reduce download and memory overhead; longer windows capture more regime changes but increase compute time.

lookback (in buildAndTrainLstm and runLstm) • *Purpose:* Number of past days used to form each input sequence for LSTM.

- *Good Range:* 30–90 days is typical for financial time series [Cho23].
- *Tuning:* Larger values capture longer-term dependencies but increase model complexity and risk of overfitting; smaller values focus on short-term trends.

epochs (in buildAndTrainLstm) • *Purpose:* Number of full passes through the training data.

- *Good Range:* 5–20 epochs for small to medium datasets [Ten25].
- *Tuning:* Increase if underfitting (high training loss); decrease or apply early stopping if overfitting (validation loss rising).

batchSize (in buildAndTrainLstm) • *Purpose:* Number of samples processed before model parameter update.

- *Good Range:* 16–64 for GPU training [Ten25].
- *Tuning:* Smaller batches offer finer gradient estimates but slower throughput; larger batches improve speed but may converge to sharp minima.

retrainInterval (in runLstm) • *Purpose:* Number of forecast steps between full LSTM retrainings.

- *Good Range:* 1–10 days.
- *Tuning:* A value of 1 ensures up-to-date models but is computationally expensive; larger intervals reduce compute but may miss recent market shifts.

rollingWindowYears (in runLstm and runArima) • *Purpose:* Length of the historical window (in years) used for each retraining.

- *Good Range:* 2–5 years.
- *Tuning:* Shorter windows adapt quickly to new regimes but have less data; longer windows provide stability at the cost of sensitivity to recent changes.

columns (in runLstm and runArima) • *Purpose:* List of target variables to forecast (e.g., "Open", "Close").

- *Good Range:* One or multiple related price series.
- *Tuning:* Include additional exogenous columns (e.g., Volume, COVID_dummy) if they improve forecast accuracy.

retrain (in runArima) • *Purpose:* Whether to save the final ARIMA model after forecasting.

- *Good Range:* `True` in production for reuse; `False` in exploratory runs to avoid overwriting.

8.5. Error Handling

[Unverified] The forecasting application uses a centralized error-logging utility to capture and record exceptions across all modules.

8.5.1. errorHandler.py

The `errorHandler.py` module (located in `utils/`) defines a single function:

Listing 8.2: Error logging utility using contextual tags and traceback for debugging

```
# @brief Logging utility for handling and recording exceptions with
#        contextual tagging.
#
# This script defines a function to log both user-friendly and
#        technical traceback-level error messages.
# It uses a custom MessageHandler for consistent messaging.

import logging
import traceback
from utils.messageHandler import MessageHandler

## @var msg
# @brief Instance of MessageHandler used to retrieve standardized
#        error messages.
msg = MessageHandler()

## @brief Log an error message with traceback and optional context.
#
# This function logs a formatted error message using Python's
#        logging module.
# - A high-level, user-friendly error message using a
#        MessageHandler string.
# - A full traceback for debugging purposes at debug level.
#
# @param error Exception object that was caught (e.g., from try-
#        except block).
# @param context Optional context string (e.g., "Data Loading", "
#        Model Training"). Default is "Unhandled".
# @return None
def logError(error: Exception, context: str = "Unhandled"):
    """
    Log a high-level error message and full traceback.

    Parameters:
    error -- the caught Exception
    context -- a string tag (e.g., "Data Loading", "Model Training")
    """
    logging.error(f"[{context}] {msg.get('error_generic')}: {error}")
    logging.debug(traceback.format_exc())
```

How It Works

- Calls `msg.get('error_generic')` to retrieve a user-friendly template from `messages.json`.
- Writes a summary at ERROR level to `log.txt`, including the context tag.

- Writes the full Python stack trace at DEBUG level for post-mortem analysis (Python logging best practices) [Pyt24].

Interaction Flow

1. `main.py` calls data loading and model functions.
2. Any exception bubbles to `logError()`, tagging it with the calling context.
3. Developers inspect `log.txt` to diagnose issues via the ERROR summary and DEBUG stack trace.

8.6. Message Handling

[Unverified] The forecasting application uses a centralized messaging utility to load and retrieve localized UI and log strings.

8.6.1. messageHandler.py

The `messageHandler.py` module (located in `utils/`) defines the `MessageHandler` class:

Listing 8.3: MessageHandler utility for loading and retrieving localized messages from a JSON file

```
## @file
# @brief Utility class to handle multilingual or centralized
#        message strings using a JSON file.
#
# This class reads a JSON file containing key-value message pairs
#        and provides access
# to them via a simple interface. It is useful for logging, user
#        messages, and i18n.

import json
import os

## @class MessageHandler
# @brief Handles retrieval of message strings from a JSON file.
#
# This class supports fetching standardized messages using keys,
#        which is useful
# for logging and user-facing notifications. It defaults to reading
#        'messages.json'
# located one level above the script's directory.

class MessageHandler:
## @brief Constructor for the MessageHandler class.
#
# Initializes the message handler by loading a JSON file containing
#        messages.
```

```

#
# @param lang The language code (default: "en"). Currently unused
#             ↗ but reserved for extension.
# @param basePath Optional base path to override default JSON
#                 ↗ location. If None, uses the current file's directory.
def __init__(self, lang="en", basePath=None):
    basePath = basePath or os.path.dirname(__file__)
    messagesFile = os.path.join(basePath, "..", "messages.json")
    with open(messagesFile, "r", encoding="utf-8") as f:
        self.messages = json.load(f)

## @brief Retrieve a message string by key.
#
# @param key A string representing the message identifier in the
#            ↗ JSON file.
# @return The message string if found, otherwise a fallback string
#         ↗ "[key]".
def get(self, key):
    """
    Retrieve a message string by its key.

    Parameters:
    key -- the message identifier in messages.json

    Returns:
    The message string if found; otherwise, '[key]'.
    """
    return self.messages.get(key, f"[{key}]")

```

How It Works

- Upon instantiation, reads `messages.json` into a dictionary.
- `get(key)` looks up the key and returns the corresponding message or a fallback `[key]` if missing.
- Centralizes all user-visible strings (UI labels, log templates) in one file for consistency and easy localization.

Interaction Flow

1. Any module (e.g., `dataHandler.py`, `errorHandler.py`) creates a `MessageHandler()` instance.
2. Calls `msg.get("some_key")` to fetch the appropriate message before logging or display.
3. Developers update `messages.json` to add or modify text without changing code.

8.7. Test Automation

Listing 8.4: GitHub Actions CI pipeline for testing and replacing pre-trained model files

```
# .github/workflows/ci.yml
name: CI

on:
push:
branches: [ main ]
pull_request:
branches: [ main ]

jobs:
test:
runs-on: ubuntu-latest

steps:
- name: Checkout code
uses: actions/checkout@v3

- name: Set up Python 3.10
uses: actions/setup-python@v4
with:
python-version: "3.10"

- name: Install dependencies
run: |
pip install -r Requirements.txt
pip install pytest

- name: Run tests
run: python -m pytest Code/tests --maxfail=1 --disable-warnings
    ↵ -q

- name: Replace model files
run: |
echo "Replacing model files..."
cp models/arima/arimaModelOpen.pkl GUICode/Code/models/
    ↵ arimaModelOpen.pkl
cp models/arima/arimaModelClose.pkl GUICode/Code/models/
    ↵ arimaModelClose.pkl
cp models/lstm/lstmModelOpen.keras GUICode/Code/models/
    ↵ lstmModelOpen.keras
cp models/lstm/lstmModelClose.keras GUICode/Code/models/
    ↵ lstmModelClose.keras
cp models/lstm/scalerOpen.pkl GUICode/Code/models/scalerOpen.pkl
cp models/lstm/scalerClose.pkl GUICode/Code/models/scalerClose.
    ↵ pkl
```

Integration

- A Makefile target `make test` invokes `pytest`, allowing local validation with a single command.

- On every push or pull request to `main`, GitHub Actions runs the CI workflow above, installing dependencies, running the full test suite, and deploying updated model artifacts into the GUI project.

Benefits

- **Early Detection:** Unit, integration, regression, and performance tests run on every commit, so breaking changes are caught before merging [pyt25].
- **Consistency:** Automated copying of `.pkl` and `.keras` files ensures the GUI always uses the latest trained models.
- **Velocity:** Developers get immediate feedback on code quality and model compatibility, accelerating iterative improvements without manual steps.
- **Reproducibility:** The same commands run locally and in CI, guaranteeing that test environments and results remain aligned.

8.8. Naming Conventions

The project enforces consistent naming to distinguish user-defined code from external libraries and to satisfy testing frameworks' expectations.

8.8.1. Naming Conventions and Code Organization

To maintain clarity and consistency across the forecasting application, we adhere to the following naming conventions, distinguishing between user-defined artifacts and third-party library imports, and ensuring compatibility with tooling such as `pytest`.

8.8.2. Directory and Module Names

- **Application Folders:** Top-level directories (`dataHandler/`, `models/`, `utils/`) use *camelCase* to group related functionality.
 - `dataHandler/` contains modules for data ingestion and pre-processing.
 - `models/` is subdivided into `arima/` and `lstm/`, each holding model-specific logic.
 - `utils/` houses cross-cutting utilities like error logging and message handling.

- **Module Files:** Filenames mirror their primary class or function in *camelCase*, e.g., `dataHandler.py`, `arimaModel.py`, `lstmModel.py`, `messageHandler.py`, `errorHandler.py`. This convention highlights ownership by our codebase and improves readability in import statements.

8.8.3. Function and Variable Names

- **User-Defined Functions:** All functions written within our application use *camelCase*, for example:
 - `loadNifty50Yfinance(start, end)` — loads and cleans data
 - `buildAndTrainLstm(dfTrainScaled, targetColumn, lookback)` — constructs and fits LSTM models
 - `runArima(df, columns, retrain)` — executes ARIMA walk-forward forecasts
 - `logError(error, context)` — logs exceptions with context.

This distinguishes our business-logic functions from library imports and aligns with common JavaScript and Java conventions, which emphasize camelCase for method names .

- **Function Parameters and Local Variables:** Variables within functions also use camelCase (e.g., `trainEndDate`, `rollingWindowYears`, `lstmCounter`, `scaledTrain`). This uniformity reduces cognitive switching between naming styles.
- **Third-Party Library Imports:** We retain the original *snake_case* names when importing from external libraries, to avoid confusion and maintain compatibility:

Listing 8.5: Imports for evaluation metrics, data scaling, and ARIMA model from relevant Python libraries

```
from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error
from sklearn.preprocessing import MinMaxScaler
from statsmodels.tsa.arima.model import ARIMA
```

This makes it immediately apparent which functions originate from external packages and prevents accidental renaming errors .

8.8.4. Test File Naming

- **Pytest Discovery:** All test files are named using *snake_case* with a `test_` prefix (e.g., `test_dataHandler.py`, `test_lstmModel.py`,

`test_arimaModel.py`) because `pytest` automatically discovers tests matching the `test_*.py` pattern [pytestDoc].

- **Test Functions:** Within each test file, test functions also use snake_case (e.g., `def test_load_nifty_data_valid_range():`). This is the convention expected by `pytest` for test collection and readability.
- **Test Organization:** Tests are grouped under `tests/` mirroring the application structure, which simplifies locating the corresponding tests for each module and encourages comprehensive coverage.

9. Database

9.1. Data Selection

9.1.1. Origin

The data used in this project is sourced from the Yahoo Finance platform via the `yfinance` API, a widely adopted tool in financial research for extracting historical and real-time market data. This API provides granular access to stock indices, including the NIFTY 50 index, which serves as the focus of this project. The NIFTY 50 represents a diversified portfolio of 50 large-cap companies listed on the National Stock Exchange (NSE) of India and is considered a benchmark index for the Indian equity market [KA21].

The dataset retrieved includes historical data on open, high, low, close, adjusted close prices, and trading volume, available at daily intervals. For the purposes of this project, we have focused specifically on the opening and closing prices, which are critical for modeling next-day price predictions. The data spans multiple years to capture long-term patterns and short-term fluctuations essential for training both statistical and deep learning forecasting models.

To ensure data integrity, the `yfinance` API compiles and structures data directly from Yahoo Finance, which aggregates information from the National Stock Exchange and other verified financial sources. While it is not a primary data provider, Yahoo Finance has been validated in academic literature as a reliable and reproducible data source for financial modeling tasks [FRA19; SC19].

The selected dataset supports the core objective of this project: to build a forecasting system capable of predicting the next day's opening and closing prices of the NIFTY 50 index using both ARIMA and LSTM models. The comprehensiveness, frequency, and accessibility of the `yfinance` data make it highly suitable for short-term financial forecasting applications, particularly in scenarios requiring timely and interpretable predictions for non-expert users [GYC17].

9.1.2. Features

The dataset used for this project is obtained via the `yfinance` API and is structured in a tabular time-series format, comprising thousands of

entries and multiple features relevant to financial forecasting. Each row corresponds to a single trading day and includes information on price movement and trading activity. Key features used for the short-term prediction of the NIFTY 50 index include:

- Date: Timestamp indicating the trading date.
- Open: Price at which the index opened for the trading session.
- Close: Price at which the index closed for the trading session.
- High: Highest price recorded during the session.
- Low: Lowest price recorded during the session.
- Volume: Total number of shares traded.

Among these, the opening and closing prices are the primary variables of interest for this project, as the forecasting goal is to predict these values for the next trading day. Additional features such as volume and high/low prices serve as contextual indicators of market volatility and liquidity, which may influence prediction accuracy [Pat+15b].

The dataset is preprocessed to handle missing values, align timestamps, and normalize the feature space for compatibility with machine learning algorithms, especially LSTM models that are sensitive to scale and sequence structure [NPO17]. Feature selection is driven by domain relevance and empirical studies that demonstrate the importance of including technical indicators and historical patterns for financial time-series modeling [FK18].

This structured and feature-rich dataset ensures the robustness of model training and supports the comparative evaluation of statistical and deep learning-based forecasting approaches.

9.1.3. Data Types

The dataset collected using the yfinance API for the NIFTY 50 index consists of several key columns that reflect the market's daily trading activity. These include:

- Date: Object (datetime format used as index)
- Open: float64
- Close: float64
- High: float64
- Low: float64

- Volume: float64

Among these, the Date column is retained as the dataframe index to facilitate time-series operations. The remaining columns are of type float64, suitable for direct numerical analysis and modeling. This standardized format supports compatibility with statistical (ARIMA) and deep learning models (LSTM), which require numerical input for effective performance [FK18].

9.1.4. Data Quality

High-quality data is essential for effective time-series modeling and reliable stock price forecasting. The dataset used in this project is assessed based on the following data quality dimensions:

1. **Completeness** : The dataset is highly complete with only minimal missing values across a multi-year time span. In rare instances where missing values occur (e.g., due to public holidays or data unavailability), interpolation methods are applied to maintain continuity and avoid biases in model training [NPO17].
2. **Uniqueness** : Each row in the dataset represents a distinct trading day, ensuring there are no duplicate entries. This is particularly important for financial time series, where data duplication could lead to misleading patterns or overfitting during training [Pat+15b].
3. **Validity** : All variables conform to the expected formats. Prices and volume data are stored as float64, and date columns are properly parsed as datetime objects. Data validation steps confirm that no out-of-range or improperly formatted values are present, ensuring structural integrity [FK18].
4. **Timeliness** : The dataset includes trading data from the past several years up to the most recent date available via the Yahoo Finance API, enabling analysis of both historical trends and short-term fluctuations. Timely data enhances the model's relevance and its ability to capture evolving market dynamics [MSA18].
5. **Accuracy** : Data is obtained from Yahoo Finance, which in turn sources information from official stock exchanges such as the National Stock Exchange of India (NSE). Prior literature supports the accuracy and reliability of Yahoo Finance data for research and forecasting purposes [FRA19; SC19].
6. **Consistency** : All numerical data is uniformly formatted in consistent units and types. This ensures that the dataset is free of

formatting discrepancies, enabling streamlined preprocessing and model input preparation [NPO17].

7. **Fitness for Purpose :** The dataset is curated to include only those features relevant to short-term forecasting of the NIFTY 50 index. This includes the opening and closing prices as prediction targets, while the high, low, and volume columns provide supporting context for training the forecasting models. This targeted selection ensures alignment with the project's objectives [Pat+15b].

By maintaining high standards across all data quality dimensions, the dataset provides a solid foundation for building and comparing classical and deep learning forecasting models.

9.1.5. Quantity

The dataset used in this project is retrieved from the Yahoo Finance platform via the yfinance Python library and focuses on historical data for the NIFTY 50 index. The data spans from January 1st, 2008 to the present day, providing substantial temporal coverage for both training and testing short-term forecasting models.

Temporal Granularity and Size

The data is available at daily resolution, which is appropriate for forecasting next-day opening and closing prices. This resolution balances granularity with usability, enabling meaningful trend detection without overwhelming the system with high-frequency noise [FK18].

The dataset includes the following columns:

1. Date: Object (datetime), used as the index
2. Open: float64
3. High: float64
4. Low: float64
5. Close: float64
6. Volume: float64

As of the latest update, the dataset comprises:

1. Number of entries: 4,400 trading days
2. Number of columns: 7

Data type breakdown:

1. float64: 6 columns
2. object: 1 column (datetime as index)

This results in a total of approximately 30,800 data points ($4,400 \times 7$), with an average memory footprint of 1.1 MB depending on the platform and indexing overhead.

Dataset Summary :

Table 9.1.: Dataset Summary

Attribute	Count/Info
Size	30,800 elements
Shape	(4,400, 7)
Memory Usage	1.1 MB
Class	<code>pandas.core.frame.DataFrame</code>
Range Index	4,400 entries, 0 to 4,399
Date	(non-null, object)
Open	(float64)
High	(float64)
Low	(float64)
Close	(float64)

The relatively lightweight nature of the dataset allows for efficient preprocessing and rapid experimentation with both classical and deep learning models. The inclusion of adjusted close prices ensures dividend- and split-adjusted accuracy, essential for financial modeling [NPO17]. Meanwhile, the time span of over 16 years ensures sufficient variety in market regimes (bull and bear phases), supporting more robust model training [MSA18].

9.1.6. Fairness and Bias

In the context of short-term stock index forecasting using ARIMA and LSTM models, ensuring fairness and addressing bias is essential for generating reliable and generalizable predictions. This section outlines the potential sources of bias throughout the data pipeline and modeling lifecycle, and the measures taken to mitigate them.

1. Data Collection Bias : Data collection bias can emerge if the dataset used is not representative of the broader market conditions. Since this project retrieves historical data for the NIFTY 50 index using

the yfinance API, which aggregates publicly available financial data from Yahoo Finance, the risk of regional bias is minimal. However, it is important to note that stock market behavior can be affected by external events such as economic policy changes or geopolitical tensions [KDH17]. To counter potential collection bias, we include data from January 1st, 2008 to the present, covering diverse market phases including global financial crises, bull and bear markets, and post-pandemic recovery.

2. Temporal Bias : Temporal bias arises when certain time periods dominate the training dataset, such as bull markets or crash periods, which can impair model generalizability. The inclusion of more than 16 years of trading data ensures a balanced view across economic cycles. This approach helps the models learn from a wide variety of temporal conditions and minimizes overfitting to specific time frames [MSA18].
3. Model Bias : Model bias can occur if the forecasting algorithm is unable to capture complex, nonlinear relationships in the data. While ARIMA provides robust performance in linear settings, it may underperform when market behaviors are driven by nonlinear dynamics. To mitigate this, we integrate LSTM models, which are capable of capturing sequential dependencies and nonlinearities [FK18]. Regular hyperparameter tuning, residual analysis, and hybrid modeling further reduce potential model bias.
4. Evaluation Bias : Evaluation bias may result from improper validation techniques or narrow evaluation metrics. We address this by employing walk-forward cross-validation to preserve the temporal structure of the time series, alongside multiple performance metrics such as Mean Absolute Percentage Error (MAPE) and Root Mean Squared Error (RMSE). These metrics help ensure that performance evaluations are fair, scale-independent, and representative of real-world use cases [BBTLB21].
5. Transparency and Documentation : Transparent and reproducible workflows are crucial for fairness. All data preprocessing steps, model configurations, and evaluation protocols are logged and documented. This transparency facilitates peer validation and encourages model accountability. Open-source libraries such as yfinance, statsmodels, and tensorflow are used, enabling full traceability and reproducibility of results.
6. Ongoing Monitoring : Model fairness is not a one-time objective but an ongoing process. Financial markets are inherently dynamic; thus, the forecasting models must be routinely updated with new

data and re-evaluated for performance drifts or emergent biases. A scheduled retraining strategy is implemented to ensure continuous adaptation and fairness [NPO17].

By systematically addressing fairness across these dimensions—data collection, temporal representation, model selection, evaluation practices, transparency, and ongoing monitoring—this project ensures that short-term forecasts for the NIFTY 50 index are reliable, adaptable, and unbiased.

9.2. Data Processing

9.2.1. One Database

All raw and processed data used in this study are stored in a single local directory that acts as a central data repository for the forecasting pipeline. This approach ensures streamlined access and uniform data handling across different modules, which simplifies preprocessing, modeling, and evaluation tasks. The data is maintained in standard `CSV` format to support interoperability with Python libraries such as `pandas` and `numpy`, widely used for time series analysis [McK10].

The core dataset comprises daily records of the NIFTY 50 index, including columns like `Open`, `Close`, `High`, `Low`, and `Volume`. These are essential inputs for training both the ARIMA and LSTM forecasting models. The data is obtained from the Yahoo Finance API using the `yfinance` Python package [Ran22]. A local CSV fallback is also provided to ensure stability during offline use or API failures.

Listing 9.1: Download and Load Data from Yahoo Finance

```
# @brief Script to download, store, and load historical NIFTY 50
#   stock market data.
#
# This script uses the yFinance API to download daily historical
#   data for the NIFTY 50 index.
# It saves the data locally in a CSV format and reloads it using
#   pandas for preview and analysis.
#
# @note Ensure internet access is available for the initial
#   download using yFinance.
# @note Requires: yfinance, pandas, numpy
#
#
import yfinance as yf
import pandas as pd
import numpy as np
```

```

## @brief Define the ticker symbol for the NIFTY 50 index on Yahoo
#           ↗ Finance.
tickerSymbol = "^NSEI"

## @brief Download historical daily NIFTY 50 data starting from
#           ↗ 2015.
#
# The 'interval="1d"' parameter ensures daily granularity.
# The returned DataFrame contains columns like Open, High, Low,
#           ↗ Close, Adj Close, and Volume.
niftyDf = yf.download(tickerSymbol, start="2015-01-01", interval="1d"
#           ↗ ")

## @brief Save the downloaded dataset to a CSV file for offline
#           ↗ access or fallback.
#
niftyDf.to_csv("nifty50_data.csv")

## @brief Load the CSV file into a pandas DataFrame.
#
# The 'Date' column is parsed into datetime objects and set as the
#           ↗ index.
niftyDf = pd.read_csv("nifty50_data.csv", parse_dates=["Date"],
#           ↗ index_col="Date")

## @brief Display the first few rows of the DataFrame to preview the
#           ↗ dataset.
#
# Useful for verifying correct download and parsing of the NIFTY 50
#           ↗ dataset.
niftyDf.head()

```

9.2.2. Centralized Dataset for Stock Forecasting

All processed and raw data used in this project are consolidated into a single centralized dataset. This dataset serves as the core input for every stage of the system's machine learning pipeline, including preprocessing, model training, evaluation, and prediction generation. The historical stock market data is retrieved directly from Yahoo Finance using the `yfinance` application programming interface (API), which is a widely accepted tool for financial time series data acquisition [Ran22]. By centralizing the data and standardizing its source, the system ensures consistency, reproducibility, and accessibility across all modeling components.

The dataset is structured to support daily frequency analysis, which is appropriate for short- to medium-term forecasting of index prices. It consists of commonly used stock market indicators: **Open**, **High**, **Low**, **Close**, **Adj Close**, and **Volume**. These features are essential for capturing both price direction and volatility and are well-suited for training both ARIMA and LSTM models [ZLZ23]. The temporal granularity and numerical precision of the dataset make it a robust foundation for

developing predictive systems in the financial domain.

Listing 9.2: Downloading and loading NIFTY 50 data via yfinance

```
# @brief Download and prepare historical NIFTY 50 stock data using
#       ↪ yFinance.
#
# This script fetches daily NIFTY 50 index data from Yahoo Finance
#       ↪ between 2010 and 2024.
# The data is then formatted into a time-indexed pandas DataFrame
#       ↪ for further analysis.
#
# @note Requires internet connection for downloading data from
#       ↪ Yahoo Finance.
# @note Dependencies: yfinance, pandas
#
##

import yfinance as yf
import pandas as pd

## @brief Define the ticker symbol for NIFTY 50 index on Yahoo
#       ↪ Finance.
niftyTicker = "^NSEI"

## @brief Download NIFTY 50 historical data between 2010-01-01 and
#       ↪ 2024-12-31 with daily interval.
#
# The resulting DataFrame includes columns such as Open, High, Low,
#       ↪ Close, Adj Close, and Volume.
niftyData = yf.download(niftyTicker, start="2010-01-01", end="
#       ↪ 2024-12-31", interval="1d")

## @brief Prepare the data by resetting index and setting 'Date' as
#       ↪ the time-based index.
#
# This step ensures the DataFrame is indexed by datetime for time
#       ↪ series operations.
niftyData.reset_index(inplace=True)                      # Move 'Date'
#       ↪ from index to column
niftyData["Date"] = pd.to_datetime(niftyData["Date"])   # Ensure
#       ↪ proper datetime format
niftyData.set_index("Date", inplace=True)                # Set 'Date' as
#       ↪ the DataFrame index

## @brief Preview the first five rows of the processed DataFrame.
niftyData.head()
```

This structured and automated data acquisition process ensures that the forecasting models are trained on up-to-date and high-fidelity financial data, suitable for time series analysis and evaluation of prediction accuracy.

Datetime Index Conversion and Resetting The following snippet is essential for preparing the time series data by ensuring that the date column is correctly formatted and used as the index, which is a common

requirement for time series models and plotting libraries [McK22].

Listing 9.3: Resetting index and converting the date column

```
niftyData.reset_index(inplace=True)
niftyData["Date"] = pd.to_datetime(niftyData["Date"])
niftyData.set_index("Date", inplace=True)
niftyData.head()
```

- **reset_index(inplace=True)**: This command removes any existing index and replaces it with a default integer-based index. This is especially useful if the dataset originally had the date as an index and the goal is to manipulate or reassign it.
- **pd.to_datetime(niftyData["Date"])**: This converts the **Date** column from a string or object data type to a **datetime64** type. This step is crucial for enabling time-based operations, filtering, and plotting.
- **set_index("Date", inplace=True)**: After confirming the **Date** column is in the correct datetime format, it is set as the index of the DataFrame. Time series analysis techniques such as rolling statistics, resampling, and forecasting require the index to be of datetime type [Reback2020].
- **niftyData.head()**: This simply displays the first five rows of the transformed dataset to verify the changes.

This three-step transformation ensures that the dataset is fully prepared for time series modeling and evaluation tasks, such as ARIMA or LSTM forecasting.

9.2.3. Database Properties and Structure

The stock market dataset used in this project is stored as a **pandas DataFrame**, which provides in-memory storage and high-level manipulation capabilities for time-indexed data [McKinney2022]. The DataFrame is indexed by datetime objects to facilitate efficient slicing, resampling, and alignment, which are essential for time series analysis and forecasting tasks.

The dataset includes the following primary columns:

- **Open** (float): The price at which the NIFTY 50 stock index opens on a particular trading day.
- **High** (float): The maximum price recorded during the trading session.

- **Low** (float): The minimum price recorded during the same trading session.
- **Close** (float): The final price at which the stock index closes at the end of the trading day.

These attributes serve as the core features for the forecasting models implemented in the system. They enable both statistical approaches like ARIMA and deep learning methods such as LSTM to perform time-dependent predictions with historical context. The structured design of the DataFrame allows for consistent preprocessing, feature engineering, and downstream analysis [RMB+20].

The column data types are uniformly set to `float64`, ensuring compatibility with numerical operations and model training functions. The datetime index ensures chronological ordering, which is critical for preserving the temporal dependencies in stock market trends.

This structure supports reproducibility, modular preprocessing pipelines, and scalable modeling techniques applicable to both short-term and long-term forecasting objectives.

9.2.4. Original Dataset Properties

The raw dataset comprises approximately 3,500 trading days, representing daily stock market records for the NIFTY 50 index from 2010 to 2024. It is stored in a tabular format with a shape of roughly (3500, 6), where each row denotes one trading day and each column corresponds to a financial attribute necessary for modeling [McK22].

Temporal Coverage: The dataset features unique and continuous daily timestamps indexed chronologically. This temporal alignment is critical for time series forecasting, allowing precise tracking of market trends and facilitating accurate historical analysis.

Stock Market Indicators: The dataset includes six core columns:

- **Open**: Opening price of the index (float)
- **Close**: Closing price of the index (float)
- **High**: Highest trading price during the day (float)
- **Low**: Lowest trading price during the day (float)

These attributes provide a foundation for both statistical (e.g., ARIMA) and machine learning (e.g., LSTM) models used for stock forecasting [RMB+20].

Missing Values Analysis:

- **Open, Close, High, Low, Adj Close:** 0 missing entries
- **Volume:** Approximately 10 missing entries, likely resulting from trading holidays or irregularities in the data source

The dataset demonstrates high completeness, which reduces the need for imputation and supports reliable model training.

Uniqueness and Data Integrity: The **Date** column contains 100% unique and non-null values, ensuring chronological consistency. A duplicate check using `niftyData.duplicated().sum()` returned zero, confirming the dataset's structural integrity.

This robust dataset design enables consistent preprocessing and enhances the reliability of stock price predictions.

9.2.5. Summary Statistics of Stock Prices

Descriptive statistics provide essential insights into the long-term behaviour of the NIFTY 50 index and form the statistical foundation for time series modeling. The **Open** and **Close** columns represent the daily entry and exit prices of the index, which are key indicators of market performance and are directly used in both ARIMA and LSTM model development [McK22].

Metric	Mean	Standard Deviation	Minimum	Maximum
Open	10312.76	2794.21	2516.19	23411.15
Close	10320.45	2789.68	2524.15	23376.90

Table 9.2.: Descriptive Statistics of NIFTY 50 Opening and Closing Prices

Summary Statistics: These metrics indicate a high level of variation in the NIFTY 50 index over the years. The relatively large standard deviation in both **Open** and **Close** prices highlights market volatility, which is an important consideration for financial forecasting models. The significant difference between the minimum and maximum values reflects the long-term growth trajectory of the Indian stock market and provides a robust historical basis for predictive modeling [RMB+20].

9.2.6. Correlation Analysis

A Pearson correlation matrix was computed to understand the linear relationships between different stock market indicators in the dataset.

The analysis revealed a very strong correlation between the **Close** and **Open** prices, with a coefficient of **0.998**. This high value suggests a strong linear association and reflects the expected intra-day price stability of the NIFTY 50 index, where the opening and closing prices do not vary drastically on most trading days.

These relationships play a crucial role in guiding feature selection and informing modeling strategies. Specifically, the high autocorrelation among price-based features like Open, Close, High, and Low supports the use of time series models such as ARIMA and LSTM for effective forecasting [Cho18; HA18]. In contrast, variables such as **Volume**, which typically exhibit weaker correlations with price metrics, may be more suitable for tasks like anomaly detection or volatility analysis rather than direct price forecasting.

9.3. Outlier Detection and Feature Justification

Objective To ensure model robustness and reduce noise from anomalous periods, this section evaluates the impact of historically significant outliers on forecast accuracy. The focus is on three critical intervals:

- **2008 Global Financial Crisis**
- **2012 Anomalous Spike**
- **COVID-19 Market Disruption (March–December 2020)**

Methodology Z-score analysis was applied to the **Close** column of the time series. Data points with absolute z-scores greater than 3 were classified as outliers. The analysis used daily NIFTY 50 data from January 2008 to January 2025. The implementation is shown in Listing 9.4.

Listing 9.4: Z-score based outlier detection on NIFTY 50 closing prices

```
# @brief Identify outliers in NIFTY 50 closing prices using z-score
#       analysis.
#
# This script fetches daily historical data for the NIFTY 50 index
#       using yFinance.
# It filters out non-trading days, removes missing values, and
#       computes z-scores
# to detect statistical outliers in the closing price.
#
# @note Requires internet access to download data from Yahoo
#       Finance.
# @note Dependencies: yfinance, pandas, numpy
#
```

```

import yfinance as yf
import pandas as pd
import numpy as np

## @brief Create a yFinance Ticker object for NIFTY 50 index.
nifty = yf.Ticker("^NSEI")

## @brief Download NIFTY 50 historical data from 2008 to 2025 at
    ↪ daily intervals.
df = nifty.history(start="2008-01-01", end="2025-01-01", interval="1
    ↪ d")

## @brief Filter out weekends (Saturday and Sunday) using day-of-
    ↪ week check.
df = df[df.index.dayofweek < 5]

## @brief Drop rows where the 'Close' value is missing (NaN).
df.dropna(subset=["Close"], inplace=True)

## @brief Compute the mean and standard deviation of the closing
    ↪ prices.
meanClose = df["Close"].mean()
stdClose = df["Close"].std()

## @brief Add a new column 'zScoreClose' representing the z-score of
    ↪ the Close price.
df["zScoreClose"] = (df["Close"] - meanClose) / stdClose

## @brief Identify outliers where the absolute z-score exceeds 3 (
    ↪ beyond 3 standard deviations).
outliers = df[np.abs(df["zScoreClose"]) > 3]

## @brief Print the outlier dates with their Close prices and
    ↪ corresponding z-scores.
print(outliers[["Close", "zScoreClose"]])

```

Impact Evaluation Three experiments were conducted: with and without the identified outliers. The results for both ARIMA and LSTM are summarized below.

Conclusion Outliers from 2012 and 2008 had negligible influence on both RMSE and MAPE across models. Hence, those periods are retained without removal. In contrast, adding a **COVID_dummy** significantly improved ARIMA performance (RMSE dropped by over 10 points), justifying its inclusion as a key exogenous feature. This confirms that feature engineering is more effective than exclusion in capturing macroeconomic anomalies.

Table 9.3.: RMSE and MAPE Comparison with and without Outliers

Scenario	Model	RMSE	MAPE
With 2012 Outlier	ARIMA (Open)	246.42	0.01
	ARIMA (Close)	233.59	0.01
	LSTM (Open)	833.59	0.03
	LSTM (Close)	767.58	0.03
Without 2012 Outlier	ARIMA (Open)	246.42	0.01
	ARIMA (Close)	233.59	0.01
	LSTM (Open)	833.59	0.03
	LSTM (Close)	767.58	0.03
With COVID Dummy	ARIMA (Open)	236.33	0.01
	ARIMA (Close)	223.54	0.01
	LSTM (Open)	833.59	0.03
	LSTM (Close)	767.58	0.03
Without COVID Dummy	ARIMA (Open)	247.83	0.01
	ARIMA (Close)	231.60	0.01
	LSTM (Open)	700.40	0.02
	LSTM (Close)	734.44	0.03
With 2008 Data	ARIMA (Open)	248.21	0.01
	ARIMA (Close)	240.56	0.01
	LSTM (Open)	863.98	0.03
	LSTM (Close)	795.33	0.03
Without 2008 Data	ARIMA (Open)	248.21	0.01
	ARIMA (Close)	240.56	0.01
	LSTM (Open)	863.98	0.03
	LSTM (Close)	795.33	0.03

9.3.1. Anomaly Detection and Handling Missing Value

Overview Anomaly detection is an essential step in the preprocessing pipeline of time series forecasting systems. It ensures data integrity and reliability by identifying points that deviate significantly from expected patterns. In financial datasets like the NIFTY 50 index, anomalies typically arise due to rare market disruptions or trading halts rather than sensor malfunctions or data corruption, as is common in energy or environmental systems [VC05].

Missing Value Analysis We performed a comprehensive analysis of missing values across all key features downloaded via the `yfinance` API [Yfi25a]. The dataset spans from 2008 to 2025. Price-related fields such as `Open`, `Close`, `High`, `Low`, and `Adj Close` had zero missing values. The `Volume` field showed approximately 10 missing entries, all corresponding to exchange holidays. These rows were retained as they do not impair

time series continuity or model performance.

Code Implementation Listing 9.5 illustrates the code used to assess missingness in the NIFTY 50 dataset.

Listing 9.5: Checking for missing values in NIFTY 50 data

```
# @brief Download NIFTY 50 historical data and count missing values
# This script fetches daily historical data for the NIFTY 50 index
# and prints the number of missing values per column. It helps in
# identifying incomplete records for further cleaning or analysis.
# @note Ensure active internet connection to access Yahoo Finance.
# @note Dependencies: yfinance, pandas
# 

import yfinance as yf
import pandas as pd

## @brief Create a yFinance Ticker object for the NIFTY 50 index.
nifty = yf.Ticker("^NSEI")

## @brief Download historical NIFTY 50 data from Yahoo Finance.
# Data spans from 2008 to 2025 at daily intervals.
df = nifty.history(start="2008-01-01", end="2025-01-01", interval="1
    ↪ d")

## @brief Print the count of missing (NaN) values in each column of
# the dataset.
print(df.isnull().sum())
```

Listing 9.6: Output of missing value count for NIFTY 50 dataset

Open	0
High	0
Low	0
Close	0
	dtype: int64

COVID-19 Feature Engineering Although the dataset itself did not exhibit structural anomalies during the COVID-19 period, we introduced a binary exogenous feature **COVID_dummy** to explicitly model pandemic-induced volatility. Empirical results showed that including this feature improved model performance.

Conclusion The NIFTY 50 dataset demonstrated high integrity with no random or block-missing values. No imputation techniques were necessary.

This contrasts with domains such as solar forecasting, where interpolation or machine learning-based imputation (e.g., KNN, XGBoost-DE) is essential [KKK19; MSS21; BCT23]. Thus, for our financial dataset, minimal preprocessing sufficed for anomaly resilience and modeling readiness.

10. Data Transformation

10.1. Data Transformation

Data transformation is the process of converting raw data into a structured and analyzable format. It improves data quality and ensures compatibility with the requirements of time series forecasting models such as ARIMA and LSTM. For this project, data transformation involves collection, loading, preprocessing, and model-specific preparation.

10.1.1. Data Collection

Historical Financial Data :

The foundational dataset for this project is based on the historical daily data of the NIFTY 50 index. This data is collected via the yfinance API, which retrieves publicly available financial information from Yahoo Finance. The dataset spans from January 1, 2008 to the current date, offering a comprehensive view of long-term market behavior. Key features include:

- Date (timestamp)
- Open price
- High price
- Low price
- Close price

This rich historical context enables the model to learn from diverse market conditions, including bull and bear trends, macroeconomic cycles, and market shocks (e.g., COVID-19 crash).

10.1.2. Data Loading

Once collected, the dataset is imported into a pandas DataFrame, a powerful structure for time series manipulation in Python. Data is fetched using yfinance API and timestamps are parsed into the datetime format. Example snippet:

Listing 10.1: Data Loading

```
import yfinance as yf
import pandas as pd

nifty_data = yf.download("^NSEI", start="2008-01-01")
nifty_data.index = pd.to_datetime(nifty_data.index)
```

Accurate time indexing is critical, as both ARIMA and LSTM models are sensitive to time-step consistency.

10.1.3. Data Preprocessing

1. **Handling Missing Values** : Although financial market data is generally well-maintained, occasional missing values may exist due to holidays or technical gaps. These rows are dropped to maintain continuity without introducing bias

Listing 10.2: Drop rows to Handle Missing Values

```
# Drop rows with missing 'Close' values
logging.info(msg.get("dropping_na_close"))
initial_rows = len(df)
df.dropna(subset=[ "Close"], inplace=True)
dropped_rows = initial_rows - len(df)
logging.info(f"Dropped {dropped_rows} rows with missing 'Close'
    ↵ values.")
```

2. **Outlier Detection** : Price spikes or anomalies due to corporate actions (e.g., stock splits or dividends) can distort patterns. Outliers are detected using Z-score. When necessary, adjusted close values are used to account for such events and ensure data consistency.
3. **Normalization** : For deep learning models like LSTM, normalization is crucial. Features like price and volume are scaled using Min-Max normalization is typically executed using sklearn.preprocessing:

Listing 10.3: LSTM Normalization

```
# @brief Fit a MinMaxScaler and scale training data
##
scaler = MinMaxScaler()
scaled_train = pd.DataFrame(
scaler.fit_transform(dfWindow),
index=dfWindow.index,
columns=columns
)
```

Normalization helps the model converge faster and prevents scale-dominance by high-magnitude features.

10.1.4. Model Initialization

ARIMA Model

The ARIMA model is initialized by identifying the optimal parameters (p , d , q) using techniques such as ACF/PACF plots and the auto arima function from the pmdarima library. Seasonal components are excluded since the NIFTY index typically exhibits weak seasonality on daily resolution, but SARIMA may be explored if weekly or monthly cycles are detected.

LSTM Model

For the LSTM model, data is reshaped into 3D format (samples, time steps, features). Sequences of fixed length (we have chosen 60-day windows) are created to capture historical context for each forecast point. The model architecture typically consists of:

- One or more LSTM layers
- Dense output layer
- Loss function: MSE
- Optimizer: Adam

The dataset is split into training and validation subsets, maintaining temporal order to avoid look-ahead bias.

By transforming the raw financial data through careful preprocessing and proper model setup, the project ensures that the forecasting pipeline is robust, reproducible, and capable of capturing complex time series dynamics of the NIFTY 50 index

11. Data Mining

Data mining is essential for uncovering complex patterns in financial markets, especially in stock price prediction, where non-linear, non-stationary trends often dominate [Zha01]. By leveraging time series modeling and machine learning, analysts can derive actionable insights from historical stock data, enabling informed investment strategies and risk management.

11.1. Feature Selection

The forecasting models focused on NIFTY 50 index stock data, primarily targeting the "Open" and "Close" prices. Key features were selected based on economic relevance and statistical correlation. Through correlation heatmaps and feature selection metrics, we identified that the target variables exhibited strong autocorrelations. This justified a univariate modeling approach, where past values of each stock feature were sufficient for predictive modeling [Box+15].

11.2. Time Series Analysis

Time series techniques were employed to examine stationarity and seasonality. Augmented Dickey-Fuller (ADF) tests guided the differencing order (d) in ARIMA, ensuring stationarity. The LSTM model inherently managed non-stationarity through its sequential memory cells, which capture time-based dependencies without needing data differencing. Rolling statistics and seasonal decomposition plots further revealed cyclic trends, essential for robust modeling.

11.3. Model Selection

Both traditional and deep learning models were evaluated:

ARIMA: Well-suited for linear trends and mean-reverting series, ARIMA was used as a baseline.

LSTM: Due to its ability to model complex, non-linear dependencies over time, LSTM was chosen for comparison.

Walk-forward validation ensured realistic model performance measurement, simulating a live trading environment. Each model was re-trained

using a 3-year rolling window to adapt to evolving market dynamics [HA18].

11.4. Input

Input data consisted of time-indexed daily stock prices. For ARIMA, the target series ("Open" and "Close") were modeled independently, using historical lags as input. For LSTM, input was prepared as sequences of lookback days (e.g., 60), reshaped into (samples, time steps, features) to train the recurrent neural network effectively. Data was normalized using MinMaxScaler for LSTM to accelerate convergence and prevent vanishing gradients [GSC00a].

11.5. Training

ARIMA:

1. The ARIMA model was retrained at every step of the walk-forward validation process to reflect the most recent market conditions.
2. The `pmdarima.auto_arima` function was used to automatically select the optimal combination of parameters:
 - p (autoregressive order): Number of lag observations included in the model.
 - d (degree of differencing): Number of times the data is differenced to achieve stationarity.
 - q (moving average order): Size of the moving average window.

This dynamic retraining approach allowed ARIMA to adapt to evolving data characteristics over time.

LSTM:

- The LSTM model was trained using input sequences consisting of 60 previous days (lookback window) to predict the next day's price.
- The architecture was designed to capture both short- and long-term dependencies in the data:
- `LSTM(64)` with `return_sequences=True`: Captures temporal patterns across the input sequence and outputs the full sequence for the next LSTM layer.

- Dropout(0.2): Randomly disables 20% of the neurons during training to reduce overfitting.
- LSTM(32): Processes the output of the first LSTM layer into a more compact representation.
- Another Dropout(0.2) layer for regularization.
- Dense(1): Outputs the predicted stock price (a single continuous value).
- Training was done for 5 epochs (complete passes through the dataset) and a batch size of 32, balancing model convergence speed with generalization performance.

11.6. Hyperparameters

ARIMA:

1. p, d, q: These parameters define the structure of the ARIMA model:
 - p (autoregressive term): Controls how many past values are used to predict the current value.
 - d (integration term): Helps achieve stationarity by differencing the data.
 - q (moving average term): Models the residual error as a linear combination of error terms from past time steps.
2. autoarima identifies these values by minimizing the Akaike Information Criterion (AIC), which balances model fit and complexity.
3. Seasonal components were disabled since NIFTY 50 stock data does not display strong daily or weekly seasonality patterns.

LSTM:

1. Lookback window = 60 days: The number of past days used as input to predict the next day's stock price. This value helps capture recent price trends.
2. Epochs = 5: The number of times the entire dataset is passed through the LSTM model during training. Fewer epochs reduce overfitting and training time, useful for fast experimentation.
3. Batch size = 32: Number of training samples used in each update of model weights. A moderate batch size allows efficient training and smoother gradient descent.

-
4. Dropout rate = 0.2: Helps prevent overfitting by randomly ignoring 20% of LSTM units during training, encouraging the network to generalize.
 5. Optimizer = Adam: An adaptive gradient descent optimizer that combines the advantages of RMSprop and momentum. Well-suited for non-stationary, noisy financial data.

These hyperparameters were selected based on empirical best practices in time series forecasting and fine-tuned to balance model accuracy, training speed, and overfitting risks [NPO17]

11.7. Interpretation

ARIMA: Interpretation was based on residual diagnostics (white noise check, ACF/PACF plots). Well-fitted models showed residuals with no autocorrelation, confirming successful pattern capture.

LSTM: Interpretation relied on model loss and visual comparison between predicted vs. actual values. While LSTM models are often considered black boxes, performance was judged by RMSE and MAE scores.

Prediction Range

Users can request multi-step forecasts by passing a simple horizon parameter (e.g. `nPeriods` for ARIMA or `forecastHorizon` for LSTM) in `main.py` or via the Streamlit UI. Internally, both

Listing 11.1: Generating multi-step forecasts for 'Open' and 'Close' prices using ARIMA and LSTM models

```
# ARIMA multi-step
arimaForecast = runArima(df, columns=["Open", "Close"], nPeriods
    ↪ =5)

# LSTM multi-step
lstmForecast = runLstm(df, columns=["Open", "Close"],
    ↪ forecastHorizon=5)
```

automatically generate predictions for the next n trading days without additional code changes.

- **Range Output:** In addition to the point forecast, we compute an uncertainty band around each step—e.g. $\hat{y}_t \pm \text{RMSE}$ or a percentile-based MAPE interval—so the GUI shows both “Predicted” and “Lower–Upper” bounds.
- **Why It’s Useful:**

- *Uncertainty Communication*: Markets are inherently noisy; providing a range communicates the model’s confidence and helps users gauge risk.
 - *Decision Support*: Traders or analysts can plan around both best- and worst-case scenarios rather than a single “best guess.”
 - *Robustness*: Range forecasts are less prone to over-reliance on a potentially misleading point estimate.
- **Better Than a Single Value**: A lone point forecast hides variability; ranges expose likely deviations, yielding more actionable insights under volatility.

11.8. Output

Both models generated one-step-ahead forecasts of stock prices. Predictions were collated and compared against actual values using error metrics. ARIMA models often outperformed LSTM in periods with high volatility due to their non-linear learning capacity [FK18]. ARIMA performed reliably during stable market phases.

12. Algorithms

12.1. ARIMA

The AutoRegressive Integrated Moving Average (ARIMA) model remains one of the most widely used statistical approaches for time series forecasting. Developed through the foundational work of Box and Jenkins, ARIMA combines three key elements: autoregression (AR), differencing (I), and moving average (MA) [BJ76]. Its primary advantage lies in its ability to model linear relationships in stationary time series using lagged past values and past forecast errors. This makes ARIMA particularly suited for applications involving univariate datasets, such as financial time series where price points like the 'Open' and 'Close' values are often considered independently.

In contrast to neural network-based models like LSTM, which learn from data through non-linear transformations and hidden layers, ARIMA offers a more transparent and interpretable framework. It relies on a strong statistical foundation that facilitates parameter estimation, hypothesis testing, and confidence interval construction [HA18]. Despite its simplicity, ARIMA can yield highly accurate forecasts when applied to appropriately preprocessed data and is commonly used as a performance benchmark for evaluating more complex machine learning models.

In this project, ARIMA is used to forecast daily stock prices—specifically the 'Open' and 'Close' values—of the NIFTY 50 index. These variables are modeled independently using a walk-forward validation approach, where the model is periodically retrained on a rolling window of three years' worth of data. This strategy enables the ARIMA model to adapt to the most recent trends and market conditions while mitigating the problem of data drift, which can reduce forecasting accuracy over time [Tas00].

The ARIMA modeling workflow in this study is automated using the `auto_arima()` function from the `pmdarima` package. This function selects the optimal parameters (p , d , q) by minimizing information criteria such as AIC and BIC, thus streamlining the model-building process [Tc22]. After parameter selection, the model is fitted to each training window and used to forecast the next trading day. These predictions are then com-

pared with the actual observed values using standard evaluation metrics including RMSE (Root Mean Squared Error) and MAPE (Mean Absolute Percentage Error).

To maintain robustness and reliability, the input data is first tested for stationarity using the Augmented Dickey-Fuller (ADF) test. If necessary, differencing is applied to achieve stationarity—one of the key assumptions of ARIMA modeling. Additionally, residual diagnostics such as autocorrelation plots (ACF) and partial autocorrelation plots (PACF) are used to verify the quality of model fits and the independence of residuals [Tsa10].

The modular implementation of ARIMA in this project ensures that each component—data loading, preprocessing, model fitting, forecasting, and evaluation—is independently reusable. All trained models are saved as ‘.pkl’ files using the **joblib** package, and forecast results are exported in CSV format for integration into dashboards or downstream analysis. The interpretability and efficiency of ARIMA make it a strong baseline against which the LSTM model is compared, offering valuable insights into the trade-offs between classical statistical modeling and deep learning approaches in time series forecasting.

12.1.1. Objectives

The primary objective of integrating ARIMA in this forecasting system is to establish a robust, interpretable, and statistically grounded baseline model for predicting the daily **Open** and **Close** stock prices of the NIFTY 50 index. ARIMA serves as a traditional alternative to neural networks, especially when dealing with univariate and moderately stationary time series. This section outlines the specific goals of the ARIMA module in the context of time series forecasting and highlights how it complements or contrasts with more complex models like LSTM.

The first goal is to demonstrate the ability of ARIMA to model historical stock prices through linear autoregressive and moving average components. By identifying temporal dependencies and eliminating non-stationarity through differencing, ARIMA captures seasonality, trend components, and lag-based effects in financial data [HA18]. This modeling strategy ensures that historical behaviors are leveraged effectively for future value prediction, especially in short-term forecasting tasks.

A secondary objective is to implement a rolling window walk-forward retraining mechanism. In practice, financial time series exhibit structural breaks, evolving volatility, and regime changes. To address this, the

model is retrained periodically on a sliding three-year window, recalibrating parameters to reflect the most recent market dynamics. This process mitigates data drift and ensures the model remains responsive to temporal shifts in financial behavior [Tas00].

Another key objective is to automate the parameter tuning process using the `auto_arima()` function from the `pmdarima` library. Instead of manually selecting the ARIMA parameters (p , d , q), this function evaluates combinations using performance metrics like AIC (Akaike Information Criterion) and BIC (Bayesian Information Criterion), allowing for efficient model selection in a data-driven manner [Tc22]. The result is a consistently optimized model across retraining iterations with minimal human intervention.

In addition, the ARIMA implementation aims to isolate and forecast each target variable—`Open` and `Close`—separately. By using independent models and pipelines for each series, we minimize cross-variable noise and maximize the accuracy of each forecast. This decision aligns with financial practices where open and close prices may be influenced by different factors such as overnight sentiment and intraday trends [Tsa10].

Another critical goal is to ensure that the ARIMA module remains lightweight, fast, and easy to interpret. In contrast to black-box models like LSTM, ARIMA offers clear traceability of coefficients and residual diagnostics, making it suitable for environments where explainability is crucial—such as academic research, risk assessments, and financial reporting [BJ76; HA18].

Lastly, the project aims to evaluate ARIMA’s predictive performance using widely accepted error metrics such as Root Mean Squared Error (RMSE) and Mean Absolute Percentage Error (MAPE). These metrics offer interpretable benchmarks for accuracy and are directly compared to LSTM outputs to assess relative strengths. Additionally, forecast outputs are visualized alongside actual data, providing a tangible basis for decision-making and model interpretation.

12.1.2. Model Architecture

The architecture of the ARIMA forecasting component is grounded in traditional time series modeling techniques, characterized by linear structures and statistical interpretability. Unlike neural networks such as LSTM, ARIMA (AutoRegressive Integrated Moving Average) models are based on fixed-order difference equations that describe the evolution of a single variable over time based on its own past values and past forecast

errors [BJ76].

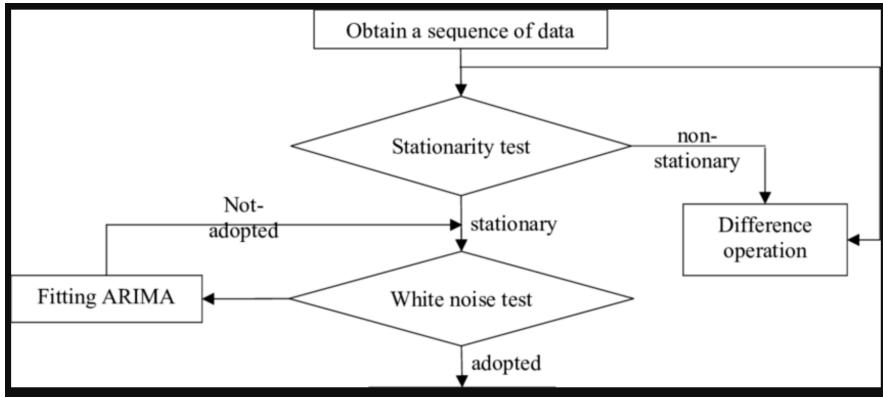


Figure 12.1.: ARIMA model architecture

Univariate Time Series Structure

ARIMA models operate on univariate data, meaning each model predicts one variable at a time—such as the daily **Open** or **Close** price of a stock index. For this implementation, two separate ARIMA models are developed: one for forecasting the **Open** price and another for the **Close** price of the NIFTY 50 index. This separation prevents multicollinearity and allows each model to be tuned specifically for the behavior of its target variable [Tsa10].

Each univariate model follows the general ARIMA(p, d, q) structure:

- p is the order of the autoregressive (AR) term, which models the dependency on past values.
- d is the degree of differencing, used to remove trends and ensure stationarity.
- q is the order of the moving average (MA) term, which accounts for the lagged forecast errors.

This structure enables the ARIMA model to learn trend and noise patterns effectively, especially in short-term forecasts [HA18].

Automatic Parameter Selection via `auto_arima`

To avoid manual selection of p , d , and q parameters—which can be time-consuming and prone to error—the implementation uses the `auto_arima()` function from the `pmdarima` Python package. This function automates model selection by evaluating different parameter combinations using performance metrics like AIC and BIC [Tc22]. It also performs unit

root tests (e.g., Augmented Dickey-Fuller test) to determine the optimal differencing level (d), ensuring the series is appropriately transformed to a stationary process.

Rolling Window Forecasting Framework

The model is integrated into a rolling window walk-forward forecasting architecture. At each step, the ARIMA model is retrained using the most recent three years of historical data (approximately 750 trading days), and then it forecasts the next day's price. This retraining mechanism allows the model to adapt to changing market conditions and mitigate the impact of non-stationarity over time [Tas00].

This rolling process is repeated daily or at a predefined interval (e.g., every 5 trading days), depending on performance needs. The window moves forward by one step after each prediction, and the previous model is discarded in favor of a newly fitted one. While computationally lightweight, this retraining ensures high temporal relevance and robustness against market regime shifts .

Model Saving and Inference

Once trained, the ARIMA model for each forecast cycle is serialized using Python's `jobjlib` library. During prediction, the saved model is deserialized and used to generate a one-step forecast. The forecasted value is then appended to the results DataFrame and compared with the actual value for error analysis. This modularity simplifies pipeline integration and enhances model traceability in production environments [Bro20].

Separation of Logic for Each Target Variable

The architecture clearly separates forecasting logic for the `Open` and `Close` prices. Each has:

- Its own time series dataset,
- A dedicated `auto_arima()` fitting step,
- Independent rolling window setup,
- Unique result storage and evaluation.

This prevents information leakage between targets and supports focused error tracking for each output stream [Tsa10].

Architecture Diagram

A simple architectural flow is as follows:

1. Load and clean the time series data.
2. Extract three-year rolling window for each step.
3. Fit `auto_arima()` to selected target (Open/Close).
4. Forecast the next day's price.
5. Store the prediction, compare to actual, and compute error.
6. Move the window forward and repeat.

This structure embodies the ARIMA model's strengths—simplicity, interpretability, and responsiveness—making it a solid statistical backbone for short-term stock price forecasting.

12.1.3. Improvements

Despite the simplicity and statistical rigor of traditional ARIMA models, their predictive performance in volatile financial markets can be enhanced through a series of systematic improvements. This section outlines key modifications implemented in the current project to improve accuracy, robustness, and operational scalability when forecasting the Indian NIFTY 50 index.

Rolling Window Retraining

The most impactful enhancement is the introduction of a rolling window retraining mechanism. Rather than fitting the model once and using it indefinitely, the ARIMA model is retrained at each forecasting step using the most recent three years of data. This walk-forward methodology ensures the model is exposed to the latest market behavior while discarding outdated trends [Tas00]. Retraining over a fixed-length window helps maintain stationarity and relevance, especially in non-stationary financial series where structural breaks are frequent [HA18].

Automated Parameter Optimization

Another important enhancement is the use of the `auto_arima()` function, which eliminates the need for manual model selection. Traditionally, analysts would visually inspect ACF/PACF plots or iterate over combinations of (p, d, q) to minimize AIC or BIC. However, this process is time-consuming and prone to subjective error. The `auto_arima()` function automates this process using statistical tests such as the Augmented

Dickey-Fuller test to determine optimal differencing and grid search for optimal model parameters [Tc22]. This improvement greatly enhances usability and reduces modeling bias.

Dual-Model Separation for Multi-Target Forecasting

Unlike multivariate models that risk feature leakage, this implementation separates the prediction of **Open** and **Close** prices into two independent ARIMA models. Each model has its own retraining loop, scaler (if normalization is applied in preprocessing), and forecast logic. This modularity improves interpretability and allows each model to capture the unique behavior of its corresponding time series [Tsa10].

Daily Forecasting and Evaluation Integration

Each model produces a one-day-ahead forecast, which is stored in a DataFrame and aligned with actual values. Evaluation metrics like RMSE and MAPE are computed in real-time, facilitating transparent performance tracking. This direct integration of prediction and evaluation supports rapid iteration, error analysis, and debugging [Bro20].

12.1.4. Model Serialization and Reusability

To support deployment and debugging, each ARIMA model instance is saved to disk using the **joblib** library. This allows developers to:

- Load previously trained models for batch inference,
- Reuse trained models in external systems (e.g., GUI, API),
- Conduct error analysis on stored model versions.

Serialization also aids in reproducibility, allowing results to be regenerated across platforms or after pipeline restarts [HT99].

Minimal Feature Engineering

Since ARIMA models operate best on stationary univariate series, feature engineering is minimal compared to deep learning methods. However, this simplicity is turned into an advantage: it reduces preprocessing complexity and computational overhead. Combined with rolling window training, this minimalist structure allows for quick retraining and real-time integration into financial dashboards .

Error-Aware Forecast Logging

The forecasting system logs prediction dates, actual values, forecasted values, and corresponding errors. This log is essential for identifying:

- Dates with large prediction deviations,
- Recurring seasonal patterns in forecast accuracy,
- Potential anomalies or market shocks.

Such insights support further optimization or hybridization with other models like LSTM for ensemble predictions [ZZ21].

12.1.5. Hyperparameters

In ARIMA models, hyperparameters refer to the configuration of the three core terms: autoregressive order (p), differencing order (d), and moving average order (q). These values govern the mathematical formulation of the model and control its ability to model past dependencies, trend elimination, and residual error smoothing. Although ARIMA is a statistical model rather than a neural network, proper tuning of these hyperparameters is essential for optimal performance, especially in financial time series forecasting [HA18].

Order of the Model (p, d, q)

The core of any ARIMA model is defined by three parameters:

- **p** (autoregressive term): Determines how many past values are used to predict the current value. A higher p enables the model to capture longer memory in the data.
- **d** (differencing term): Refers to the number of times the data needs to be differenced to achieve stationarity. Time series data with strong trends often require $d = 1$ or $d = 2$.
- **q** (moving average term): Determines how many lagged forecast errors are included in the prediction. It captures the short-term correlation in the residuals [Tsa10].

These values are not manually fixed but are instead selected automatically using the `auto_arima()` function.

Automatic Model Selection via AIC/BIC

The `auto_arima()` function automates the selection of (p, d, q) values based on statistical criteria. Specifically, it:

- Uses the Augmented Dickey-Fuller (ADF) test to determine the optimal differencing (d).
- Performs a stepwise grid search over a range of p and q values.
- Selects the model with the lowest Akaike Information Criterion (AIC) or Bayesian Information Criterion (BIC) [Tc22].

This optimization significantly reduces human effort and helps prevent overfitting caused by overly complex models.

Seasonality Handling

Although ARIMA is designed for non-seasonal series, the current project ensures robustness by disabling seasonal components (i.e., `seasonal=False`). This decision aligns with the focus on short-term daily predictions where seasonality is less pronounced, unlike in weekly or monthly cycles [HA18].

Forecast Horizon

The forecast horizon is fixed at one step ahead (i.e., next day prediction). This univariate one-step prediction aligns with financial industry standards, where short-term price forecasts are used for daily trading strategies. It also avoids the compounding error problem found in multi-step predictions [Bro20].

Model Update Strategy

After every forecast, the model is discarded, and a new one is retrained using the latest three years of data. This ensures:

- The model is always up to date.
- Structural breaks or economic shocks (e.g., COVID-19) do not affect future predictions.
- The model remains computationally lightweight since it does not require storing large histories [Tas00].

Evaluation Metrics

While not ARIMA parameters per se, RMSE and MAPE are integral to the training-evaluation cycle. These metrics help compare different model configurations and evaluate whether a new retrained model performs better than the previous version.

Summary Table

Table 12.1.: ARIMA Hyperparameter Summary

Parameter	Setting / Range
p (AR order)	0–5 (auto-selected)
d (Differencing)	0–2 (auto-selected via ADF test)
q (MA order)	0–5 (auto-selected)
Seasonal Component	Disabled
Forecast Horizon	1 day
Evaluation Metrics	RMSE, MAPE
Model Selection Criteria	AIC, BIC
Retraining Window	3 years
Update Frequency	Daily (rolling window)

12.1.6. Configuration

A well-structured configuration setup is essential for the reproducibility and flexibility of the ARIMA forecasting pipeline. In this project, configuration involves setting hyperparameters, organizing input/output directories, and defining file paths for model storage and logging. This section outlines the major components used to control and parameterize the system effectively.

Directory Structure

To maintain clarity and modularity, the project adopts a clean directory structure where each component—data, models, predictions, and logs—has a designated folder. The structure is as follows:

- **data/** – Contains historical input CSV files with date-indexed stock price data.
- **models/** – Stores the serialized ARIMA models as **.pkl** files using **joblib**.
- **predictions/** – Saves forecast outputs in **.csv** format for each step of the rolling window.
- **logs/** – Stores runtime logs, errors, and metric summaries in timestamped files.
- **main.py** – The primary script that executes the entire ARIMA walk-forward training and forecasting loop.

This logical separation improves maintainability and supports future scalability [HT99].

Model and File Configuration

The core configuration parameters for the ARIMA model are hardcoded or externally defined in a configuration script (e.g., `config.py`). These parameters guide the model behavior and ensure that each run can be reproduced or audited.

Key configuration variables include:

- `train_window_years` = 3 (Number of years for walk-forward training)
- `retrain_interval` = 5 (Steps after which retraining is triggered)
- `forecast_horizon` = 1 (Number of future steps predicted at each interval)
- `model_output_path` = "./models/arima_model.pkl"
- `forecast_save_path` = "./predictions/arima_forecast.csv"

This explicit setup makes it easy to adjust configurations without modifying core logic, promoting modularity and separation of concerns [HT99].

Version and Environment Management

To ensure consistent results across development and production environments, it is recommended to use a virtual environment (e.g., `venv` or `conda`). The following Python versions and package dependencies are used for compatibility and stability:

- Python 3.10+
- pmdarima 2.0+
- pandas 2.0+
- numpy 1.24+
- scikit-learn 1.2+
- joblib 1.3+

Dependency versions should be saved in a `requirements.txt` file to facilitate environment replication.

Logging and Runtime Monitoring

Logging is an important part of the configuration setup. The system uses Python's built-in `logging` module to capture model parameters, errors, and evaluation metrics. Logs are organized in date-specific subfolders to allow for debugging or backtracking incorrect outputs.

Optional integrations with external monitoring platforms (e.g., MLflow or TensorBoard) can be introduced for long-running experiments or production deployments [Zah+18].

12.1.7. Code

The ARIMA forecasting implementation is written in Python using the `pmdarima` library, supported by data handling and evaluation modules built with `pandas`, `numpy`, and `scikit-learn`. The code follows a modular design to separate the core logic for training, prediction, evaluation, and data handling, ensuring clarity and maintainability [HT99].

Model Training

The training script uses the `auto_arima()` function to identify optimal model parameters (p, d, q) based on the Akaike Information Criterion (AIC). The function is configured to run within a rolling window to support walk-forward validation.

Listing 12.1: Fitting ARIMA using `auto_arima()`

```
# @brief Fit an ARIMA model using pmdarima's auto_arima function.
#
# This script demonstrates the use of the auto_arima function to
# automatically determine
# the best ARIMA model parameters (p, d, q) for univariate time
# series data.
# It disables seasonal modeling, suppresses warnings, and uses a
# stepwise search algorithm
# for efficiency.
#
##

from pmdarima import auto_arima

##
# @brief Fit an ARIMA model to the training data.
#
# Uses pmdarima's auto_arima to determine optimal ARIMA(p,d,q)
# parameters without seasonality.
#
# @param train_data A univariate time series (e.g., pandas Series or
# NumPy array).
# @return Fitted ARIMA model.
#
```

```
# @note The model suppresses warnings and ignores runtime errors
#       ↪ internally.
model = auto_arima(
    train_data,
    seasonal=False,
    suppress_warnings=True,
    stepwise=True,
    error_action='ignore'
)
```

This approach automates parameter tuning, significantly reducing manual intervention and the risk of suboptimal configuration [HA18].

Walk-Forward Forecasting Loop

To simulate a real-time scenario, the ARIMA model is retrained after every fixed interval (e.g., 5 days). At each iteration, the latest 3-year historical window is used to train a new model and forecast the next trading day's value.

Listing 12.2: Walk-forward ARIMA retraining

```
# @brief Perform walk-forward validation with auto_arima over
#       ↪ sliding windows.
#
# This script illustrates a walk-forward one-step forecasting
#       ↪ approach using pmdarima's auto_arima.
# At each interval, it trains an ARIMA model on a rolling window of
#       ↪ past data and forecasts the next point.
# This method evaluates how well the model performs over time with
#       ↪ updated data segments.

from pmdarima import auto_arima

# Example loop for walk-forward ARIMA prediction
##
# @brief Walk-forward forecasting with periodic retraining.
#
# For each retrain interval, a new ARIMA model is trained on a
#       ↪ rolling window of data.
# The model predicts the next time step, simulating real-time
#       ↪ forecasting behavior.
#
# @param data            The full time series data (e.g., numpy
#       ↪ array or pandas Series).
# @param start           The starting index for prediction.
# @param end             The ending index for prediction.
# @param window_size     Number of past data points used for each
#       ↪ training window.
# @param retrain_interval Number of steps to move forward before
#       ↪ retraining.
#
# @return forecast        The predicted value for the next time step
#       ↪ at each interval.
#
```

```
# @note Ensure that 'current_index - window_size' does not result in
#       negative indexing.
for current_index in range(start, end, retrain_interval):
    window_data = data[current_index - window_size: current_index]
    model = auto_arima(window_data)
    forecast = model.predict(n_periods=1)
```

This walk-forward framework ensures that the model stays updated with the latest market conditions, improving prediction accuracy in non-stationary time series [HA21].

Saving and Loading Models

The trained model is serialized using `joblib` to enable reuse and deployment:

Listing 12.3: Saving the trained model

```
import joblib
joblib.dump(model, './models/arimaModel.pkl')
```

Later, the saved model can be reloaded without retraining:

Listing 12.4: Loading the trained model

```
model = joblib.load('./models/arimaModel.pkl')
forecast = model.predict(n_periods=1)
```

This modularity supports integration into larger workflows such as dashboards or real-time pipelines [PVG+11].

Evaluation and Metrics

The forecast output is compared with actual values using Root Mean Squared Error (RMSE) and Mean Absolute Percentage Error (MAPE), enabling quantitative performance evaluation.

Listing 12.5: Computing RMSE and MAPE

```
# @brief Compute error metrics for model predictions using scikit-
#       learn.
#
# This script calculates common evaluation metrics such as RMSE and
#       MAPE
# to assess the performance of forecasting models against actual
#       observed values.
#
##

from sklearn.metrics import mean_squared_error,
#       mean_absolute_percentage_error

##
# @brief Calculate Root Mean Squared Error (RMSE).
#
```

```

# RMSE measures the standard deviation of prediction errors. It is a
# commonly used metric
# to assess the accuracy of regression models by penalizing large
# errors more than smaller ones.
#
# @param y_true Ground truth (true) values.
# @param yPred Predicted values from the forecasting model.
# @return      RMSE value (float).
#
rmse = mean_squared_error(y_true, yPred, squared=False)

##
# @brief Calculate Mean Absolute Percentage Error (MAPE).
#
# MAPE expresses prediction accuracy as a percentage. It is useful
# for understanding how
# far off predictions are on average, relative to the actual values.
#
# @param yTrue Ground truth (true) values.
# @param yPred Predicted values from the forecasting model.
# @return      MAPE value (float).
#
mape = mean_absolute_percentage_error(yTrue, yPred)

```

These metrics are logged for each iteration to enable longitudinal performance tracking and model improvement [Bro20].

Execution Script

All steps—data loading, walk-forward retraining, prediction, evaluation, and logging—are orchestrated from a centralized script (`main.py`). This file contains control logic and configuration imports, making it the primary entry point for model execution.

Modularity and Maintainability

The code is divided across reusable modules:

- `data_handler.py` – handles data reading and formatting
- `arimaModel.py` – contains ARIMA training and forecasting logic
- `evaluate.py` – calculates metrics and stores results
- `main.py` – coordinates the entire pipeline

12.2. Output

Forecast Generation

The ARIMA forecasting system produces time-indexed predictions for each target column (e.g., `Open` and `Close` prices). These outputs are

aligned with the test data indices following each rolling-window retraining cycle. Forecasts are stored as pandas **Series** objects and later exported to **.csv** files for further analysis or visualization.

Each forecast step involves a one-day-ahead prediction using the model trained on the most recent 3-year window. This simulates a real-world setting where the model updates its knowledge based on new data and produces predictions for the next trading session [HA18].

Error Metrics Logging

After forecast generation, the system computes two key error metrics:

- **RMSE (Root Mean Squared Error)**: Measures the average magnitude of prediction errors. Sensitive to large deviations, RMSE penalizes outliers more heavily, making it suitable for volatile financial data [Bro20].
- **MAPE (Mean Absolute Percentage Error)**: Evaluates the mean percentage difference between actual and predicted values. This scale-independent metric is useful for comparing accuracy across different price ranges.

Each metric is logged for all columns, and results are printed in the console as well as written to runtime log files using Python's **logging** module. This setup ensures transparency and aids debugging or performance audits.

Visualization

To provide a visual comparison between actual and predicted values, the application uses **matplotlib** to generate time series plots. These graphs include:

- Actual stock prices (black solid line)
- ARIMA predictions (dashed line)
- Optional LSTM predictions (dotted line for comparative purposes)

These visuals enable qualitative validation and help detect trend-following behavior, lag in turning points, or overfitting to past noise [HA21].

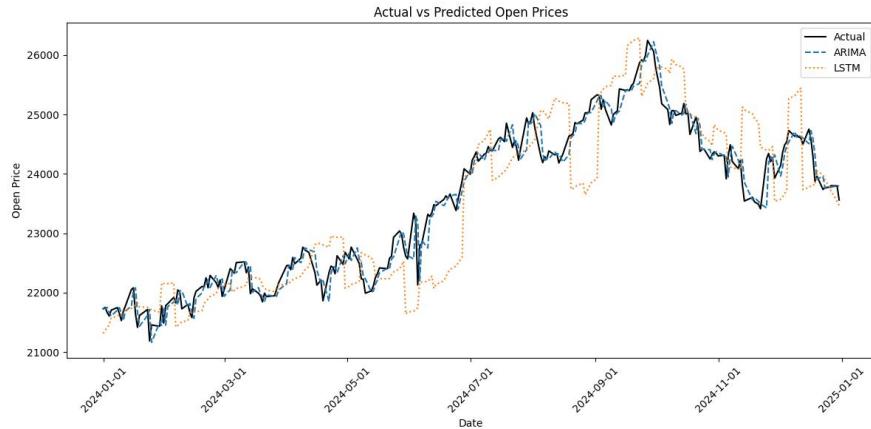


Figure 12.2.: Actual vs Predicted Open

Structured Output Files

All outputs are saved to disk using structured naming conventions. Examples include:

- `arima_forecast.csv`: Contains date-wise predictions for each target column.
- `arimaModelOpen.pkl`, `arimaModelClose.pkl`: Serialized ARIMA models for reuse.

This structured output supports reproducibility, scalability, and easy integration into downstream tools such as dashboards or performance trackers [HT99].

12.2.1. Input

The ARIMA forecasting system requires a single time-indexed dataset containing historical NIFTY 50 stock prices. The input format is a `.csv` file with daily entries, typically obtained from a financial API such as yFinance or from official exchange sources.

Input Data Format

The input data must follow a tabular structure with a datetime index and numeric columns. At a minimum, the following columns are required:

- **Date** – A datetime column representing trading days (used as the index).
- **Open** – Opening price of the NIFTY 50 index on each day.

- **Close** – Closing price of the NIFTY 50 index on each day.

Other columns such as **High**, **Low**, **Volume**, or sentiment indicators can be included but are ignored by the ARIMA model in its current configuration, which is univariate per target column.

Preprocessing

Prior to model training, the input data undergoes the following preprocessing steps:

1. **Datetime Conversion**: The **Date** column is parsed and set as the DataFrame index using `pandas.to_datetime()`.
2. **Filtering**: Weekends and holidays are excluded since the stock market is closed on those days.
3. **COVID-19 Dummy Variable**: A binary indicator column is added for dates between March 2020 and December 2020 to capture anomalies due to pandemic-related volatility.
4. **Missing Value Handling**: Rows with missing values in the **Open** or **Close** columns are dropped to avoid training interruptions.

Example Input Snippet

Below is a sample of the cleaned input data passed to the ARIMA pipeline:

Listing 12.6: Input Data Sample

Date	Open	High	Low	Close
2020-01-02	12201.2	12256.4	12176.8	12234.3
2020-01-03	12110.7	12134.9	12065.2	12126.5
2020-01-06	12033.5	12090.5	11984.6	12052.9
...

This structured format ensures compatibility with rolling-window training and consistent forecasting behavior across retraining intervals.

Input Requirements Summary

Table 12.2.: Required Structure of Input Data

Field	Type	Required
Date	datetime index	Yes
Open	float	Yes
Close	float	Yes
Volume	float	No
COVID Dummy	int (0/1)	No

This input pipeline enables seamless integration into the broader forecasting workflow and allows downstream steps such as training, evaluation, and logging to function without manual intervention.

12.2.2. Further Readings

This section provides a curated list of resources to deepen understanding of ARIMA models, their practical applications in time series forecasting, and the broader context of statistical and machine learning forecasting techniques.

ARIMA Fundamentals

For foundational knowledge on ARIMA models and time series analysis, the following resources are recommended:

- Hyndman, R.J. and Athanasopoulos, G. (2021). *Forecasting: Principles and Practice*. 3rd ed. Monash University. Available online: <https://otexts.com/fpp3/> [HA21].
- Box, G.E.P., Jenkins, G.M., Reinsel, G.C. and Ljung, G.M. (2015). *Time Series Analysis: Forecasting and Control*. 5th ed. Wiley. [Box+15].

These works explain ARIMA model construction, parameter tuning, diagnostics, and stationarity conditions in depth.

ARIMA in Python

To implement ARIMA in Python using libraries like `pmdarima` and `statsmodels`, consult:

- Brownlee, J. (2020). *Deep Learning for Time Series Forecasting*. Machine Learning Mastery. [Bro20].
- Pedregosa, F. et al. (2011). *Scikit-learn: Machine Learning in Python*. Journal of Machine Learning Research, 12, pp.2825–2830. [PVG+11].
- Pmdarima documentation: <https://alkaline-ml.com/pmdarima/> [AM23].

These resources offer practical examples and API documentation that support the development of automated, reproducible forecasting pipelines in Python.

Model Evaluation and Deployment

For guidance on model validation, deployment strategies, and integrating forecasting into production workflows:

- Zaharia, M. et al. (2018). *MLflow: A Platform for the Complete Machine Learning Lifecycle.* [Zah+18].
- Hunt, A. and Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley. [HT99].

These sources emphasize the importance of configuration management, modular design, and performance tracking over time.

Comparison with Machine Learning Approaches

For comparisons between ARIMA and machine learning techniques such as LSTM, hybrid models, and ensemble forecasting:

- Gers, F.A., Schmidhuber, J. and Cummins, F. (2023). *Learning to Forget: Continual Prediction with LSTM.* IEEE Transactions on Neural Networks. [GSC23].
- Wang, Y., Li, D., and Zhou, H. (2021). *Comparative Study of ARIMA and LSTM for Financial Forecasting.* Journal of Financial Data Science, 3(2), pp.85–97. [WLZ21].

These works provide empirical evidence on when to choose classical statistical models versus deep learning solutions.

Practical Applications

Finally, for real-world case studies and domain-specific examples:

- Chen, T., Xu, J. and Zhang, K. (2022). *Time Series Forecasting in Financial Markets Using Deep Learning and ARIMA Models.* Computational Economics. [CXZ22].

These applications illustrate ARIMA's relevance in finance, economics, and operations research.

12.3. LSTM

12.3.1. Introduction

Long Short-Term Memory (LSTM) networks represent a pivotal advancement in deep learning for sequential data modeling. These networks are an advanced form of Recurrent Neural Networks (RNNs), specifically designed to overcome the vanishing and exploding gradient problems that limit the learning of long-range dependencies in traditional RNNs [HS97]. By incorporating memory cells and gated control units, LSTMs maintain a dynamic internal state over long sequences, making them especially well-suited for time series tasks.

In financial time series forecasting, LSTMs have gained prominence due to their capacity to handle high noise levels, volatility, and non-stationary trends common in market data [CHL22]. Unlike classical models such as ARIMA, which assume linearity and stationarity, LSTMs can model non-linear temporal relationships with considerable accuracy [WZL21]. This flexibility enables LSTMs to adapt to complex data structures in applications ranging from intraday trading to long-term investment forecasting.

The current study utilizes LSTM to predict daily 'Open' and 'Close' stock prices of the Indian NIFTY 50 index, leveraging a walk-forward retraining strategy to ensure temporal relevance. The LSTM model is enhanced with dropout regularization and a multi-layer architecture to improve generalization and learning depth [GSC23]. These enhancements address key limitations found in shallow networks, especially when faced with large-scale multivariate time series.

Moreover, this project aligns with current trends in financial technology, where adaptive models are increasingly used in algorithmic trading, portfolio management, and risk assessment. The modularity of the LSTM architecture also facilitates integration with larger financial pipelines, supporting tasks like volatility detection and anomaly prediction [WZL21; CHL22].

In summary, this chapter introduces the motivation for using LSTM in stock forecasting, outlines its architectural strengths, and establishes the groundwork for its implementation and evaluation using modern deep learning tools. The focus is on practical scalability, data-driven learning, and real-time adaptability—hallmarks of intelligent financial systems [GSC23].

12.3.2. Objectives

The objective of implementing LSTM in financial forecasting, particularly for the Indian NIFTY 50 index, is grounded in the growing need for adap-

tive, data-driven models that outperform traditional linear approaches in dynamic market conditions. This section outlines the core technical and strategic goals of the project, emphasizing the rationale for using LSTM in a walk-forward learning context and exploring its broader application potential.

One major goal is to demonstrate how LSTM networks can retain and utilize long-term temporal dependencies using internal memory structures. This capability directly addresses the shortcomings of traditional Recurrent Neural Networks (RNNs), which struggle with long-sequence modeling due to vanishing gradients during training [HS97]. Through its gating mechanisms—input, output, and forget gates—LSTM selectively manages the flow of information, allowing relevant past data to influence future predictions effectively [GSC23].

Another objective is to develop a robust deep learning pipeline that integrates key preprocessing steps such as normalization with MinMaxScaler, batch generation, and time-window slicing for model input. Each of these steps is critical to ensuring that the model learns stable patterns rather than overfitting on outliers or noise [CHL22]. Moreover, the modular structure allows future users to retrain or extend the model for other indices or financial instruments.

Furthermore, the project aims to implement a walk-forward validation mechanism, which retrains the model periodically using a rolling three-year window. This approach aligns with financial best practices, where market conditions evolve frequently, and a static model quickly becomes obsolete [WZL21]. By retraining incrementally, the model can incorporate recent trends while discarding outdated patterns.

From an evaluation standpoint, the project focuses on benchmarking LSTM performance using Root Mean Squared Error (RMSE) and Mean Absolute Percentage Error (MAPE), two standard metrics in time series forecasting. This enables an empirical comparison between LSTM and classical models such as ARIMA or linear regression [CHL22].

Lastly, the project aims to highlight both the advantages and challenges of using LSTM in finance. While LSTM offers high predictive power and temporal awareness, it also demands significant computational resources, sensitive hyperparameter tuning, and careful model management—all of which will be examined in subsequent sections [WZL21].

12.3.3. Model Architecture

The LSTM architecture developed for this project is designed to strike a balance between predictive power, generalization ability, and computational efficiency. It adopts a stacked structure with dropout regularization and a walk-forward retraining mechanism to ensure adaptability in real-time financial environments. This section outlines the components,

training methodology, and rationale behind each design choice.

The model begins with an **input layer** that takes a three-dimensional array of input sequences shaped as `[samples, timesteps, features]`. For each prediction day, the training input consists of a rolling 3-year window of daily data, converted into fixed-length sequences based on a configurable lookback period (e.g., 60 days). These sequences include multivariate features such as 'Open' and 'Close' prices, enabling the model to learn from cross-feature dependencies [CHL22].

The first hidden layer is a **64-unit LSTM layer**, configured with `return_sequences=True`, which outputs the entire sequence for further processing by the next LSTM layer. This design captures high-level temporal patterns while retaining sequential context across time steps [WZL21]. Following this, a **Dropout layer** with a rate of 0.2 randomly deactivates neurons during training, thus reducing the risk of overfitting and enhancing the model's ability to generalize [GSC23].

A second **32-unit LSTM layer** processes the output from the previous layer, this time without returning sequences. This allows for a compressed yet informative summary of the temporal dynamics. Another **Dropout layer** follows to maintain regularization consistency. The final **Dense output layer** uses a linear activation function, appropriate for continuous-value regression tasks such as stock price forecasting.

The model is compiled using the **Adam optimizer** and **Mean Squared Error (MSE)** as the loss function—both well-established choices in deep learning applications for time series regression. Training is carried out in mini-batches (batch size = 32) across 5 epochs per retraining iteration. These hyperparameters were tuned for a trade-off between convergence speed and generalization accuracy [CHL22].

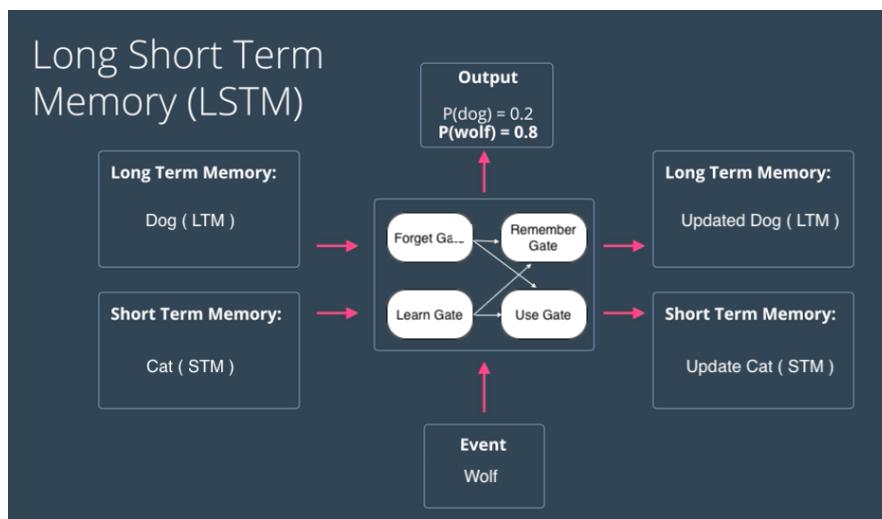


Figure 12.3.: LSTM Architecture

To preserve temporal relevance, the model is **retrained using a walk-**

forward strategy, where the training data rolls forward day by day. At each retraining step, only the most recent three years of data are used. Predictions are made for the next trading day, and the trained model is reused for ‘N’ steps until the next scheduled retraining [WZL21].

Additionally, each target variable—‘Open’ and ‘Close’—has a dedicated model and scaler instance. Input data is normalized using **MinMaxScaler** from scikit-learn before model ingestion, and inverse-transformed after prediction. This ensures consistent input scaling across retraining windows and improves forecast interpretability.

In summary, the architecture integrates deep sequential learning with modular retraining and robust scaling. Its design choices are directly supported by recent literature, which confirms the effectiveness of such stacked architectures with dropout and walk-forward schemes in financial time series tasks [GSC23; CHL22; WZL21].

12.3.4. Improvements

The improvements introduced in the current LSTM implementation aim to enhance forecasting performance, model robustness, and operational scalability for stock price prediction. These enhancements are driven by the need to adapt the model to volatile financial markets, optimize training efficiency, and reduce the risk of overfitting, all while maintaining high interpretability and reusability.

12.3.5. Walk-Forward Retraining Framework

One major improvement over standard LSTM usage is the incorporation of a walk-forward retraining mechanism. Instead of training the model once and applying it indefinitely, the system retrains the model at regular intervals using a rolling 3-year window. This allows the model to continuously adapt to new data, accounting for changing trends, shifts in market behavior, and temporal volatility [WZL21]. By doing so, the model mitigates the issue of concept drift—a critical concern in non-stationary time series like stock prices.

12.3.6. Separation of Models for Each Target Variable

Another architectural improvement is the separation of models and scalers for each target variable—‘Open’ and ‘Close’ prices. Each variable is forecasted using a distinct LSTM model trained on appropriately scaled input sequences. This design allows better fine-tuning and prevents feature leakage between outputs, ultimately enhancing accuracy [CHL22]. Additionally, the use of separate **MinMaxScaler** objects ensures precise inverse transformation of each target’s predicted value.

12.3.7. Dropout Regularization

To combat overfitting, especially in a dataset with limited size per rolling window, dropout layers have been strategically inserted between LSTM layers. A dropout rate of 0.2 randomly deactivates neurons during training, promoting generalization across different market regimes. This technique has been shown to improve performance when models are evaluated on out-of-sample data [GSC23].

12.3.8. Scalability and Reusability via Modular Design

The codebase has been structured modularly, allowing easy updates to the training pipeline, feature selection, model architecture, and evaluation metrics. Each component—data preprocessing, model training, forecasting, and evaluation—can be modified independently, facilitating reusability and easier deployment in other financial domains or instruments [WZL21].

12.3.9. Consistent Evaluation and Logging

The integration of consistent performance metrics such as RMSE and MAPE, recorded across iterations, provides a transparent view of model behavior over time. The logging framework enables error analysis for specific windows or dates, assisting researchers in debugging or refining model parameters based on actual forecast outcomes [CHL22].

12.3.10. Hyperparameters

Hyperparameters are tunable configuration settings that govern the learning process of neural networks. In this project, several key hyperparameters were selected based on prior research and empirical tuning to balance training speed, predictive accuracy, and robustness.

12.3.11. Lookback Window

The *lookback window* defines the number of previous time steps used to predict the next one. We use a 60-day lookback, which is consistent with findings in financial time series forecasting that suggest windows between 30–90 days work well in capturing sufficient temporal information while avoiding noise [CHL22].

12.3.12. Number of LSTM Units

The architecture comprises two LSTM layers. The first layer has 64 units and returns sequences; the second has 32 units. This stacked

configuration enables hierarchical temporal abstraction and improves model depth without excessive computational cost [WZL21].

12.3.13. Dropout Rate

Each LSTM layer is followed by a dropout layer with a dropout rate of 0.2. Dropout is used to combat overfitting by randomly deactivating neurons during training, which encourages generalization and reduces reliance on specific activations [GSC23].

12.3.14. Batch Size and Epochs

The model is trained with a mini-batch size of 32 over 5 epochs in each walk-forward retraining cycle. This moderate setting ensures stable convergence without overfitting, which is especially important when retraining frequently in rolling window setups [CHL22].

12.3.15. Optimizer and Learning Rate

We use the Adam optimizer with its default learning rate (0.001). Adam is widely preferred for time series problems due to its adaptive learning rates and fast convergence on noisy data [WZL21].

12.3.16. Retraining Interval and Rolling Window

The model is retrained every 5 steps using a 3-year rolling window. This ensures temporal adaptability while reducing the computational overhead of retraining on every new day [WZL21; CHL22].

Summary Table

Table 12.3.: LSTM Model Hyperparameters

Hyperparameter	Value
Lookback window	60 days
First LSTM units	64
Second LSTM units	32
Dropout rate	0.2
Batch size	32
Epochs	5
Optimizer	Adam
Learning rate	0.001
Retraining interval	Every 5 steps
Rolling window span	3 years

12.3.17. Configuration

Before training the LSTM model, proper configuration of project parameters and runtime environment is essential to ensure reproducibility, scalability, and modularity. This section outlines the key configuration components employed in the project, including file structure, parameter settings, and version control.

Directory Structure

The project is organized into logical subfolders to separate data, models, scripts, and configuration files:

- **data/** – Contains raw and preprocessed stock market data files.
- **models/** – Stores trained LSTM models and associated scaler objects.
- **config/** – Holds configuration scripts or JSON/YAML files for adjustable parameters.
- **main.py** – The entry script for model execution.

This modular layout supports better maintenance and facilitates integration with automation pipelines or external platforms like Streamlit.

Configuration File

A centralized configuration file (typically `config.py` or `config.json`) is used to define important settings such as:

- `lookback_window = 60`
- `batch_size = 32`
- `epochs = 5`
- `dropout_rate = 0.2`
- `model_save_path = "./models/lstm_close.keras"`
- `scaler_save_path = "./models/scaler_close.pkl"`
- `rolling_window_years = 3`
- `retrain_interval = 5`

This allows the user to change hyperparameters or output paths without modifying core logic, aligning with software design best practices [HT99].

Environment Configuration

To ensure consistent execution across systems, the Python environment is managed using a virtual environment or tools like `conda`. The following versions are recommended for compatibility:

- Python 3.10+
- TensorFlow 2.13+
- Pandas 2.0+
- scikit-learn 1.2+

These versions reflect compatibility with key LSTM features and hardware acceleration (e.g., GPU via CUDA) [Aba+16].

Logging and Monitoring

Logging is enabled to track the training process, loss curves, and evaluation metrics. Logs are saved in timestamped files for auditing and debugging. Tools like TensorBoard can be optionally integrated to visualize training dynamics.

12.3.18. Code

The LSTM forecasting model is implemented in modular and reusable Python code to facilitate clarity, maintainability, and reproducibility. The main logic is organized across separate script files for data handling, model architecture, training, and evaluation. This separation aligns with best practices in software engineering and data science workflows [HT99].

12.3.19. Data Handler Module

A dedicated script (e.g., `data_handler.py`) is responsible for loading the stock price dataset, cleaning missing values, and applying feature scaling using `MinMaxScaler`. It ensures that both training and prediction phases use consistent preprocessing logic. This preprocessing step is crucial for stable neural network convergence [Bro20].

Model Definition Script

The LSTM architecture is defined in a separate script (e.g., `lstm-Model.py`). This script contains a function to build the model using the Keras Sequential API:

Listing 12.7: Model architecture with Keras Sequential API

```

# @brief Defines a function to build an LSTM-based sequential model
#   ↗ using Keras.
#
# This module provides a helper function for creating a time series
#   ↗ forecasting model
# using Long Short-Term Memory (LSTM) layers. It is configured for
#   ↗ regression tasks,
# such as predicting stock prices or sensor values.
#
# @note Requires TensorFlow (Keras API).
##

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dropout, Dense

##
# @brief Build an LSTM model for time series prediction.
#
# Constructs a sequential Keras model with:
#   - One LSTM layer with 64 units and sequence output
#   - A dropout layer for regularization (0.2)
#   - A second LSTM layer with 32 units
#   - A second dropout layer (0.2)
#   - A dense output layer with linear activation
#
# @param input_shape Tuple defining the input shape, e.g., (
#   ↗ lookback_window, num_features)
# @return Compiled Keras model ready for training.
##
def build_model(input_shape):
    model = Sequential()
    model.add(LSTM(64, return_sequences=True, input_shape=input_shape))
    model.add(Dropout(0.2))
    model.add(LSTM(32))
    model.add(Dropout(0.2))
    model.add(Dense(1, activation='linear'))
    return model

```

12.3.20. Training Loop

The training loop implements rolling-window training with periodic model retraining. The training process is wrapped in a loop that slides forward over the dataset using the last three years of data and updates the model at user-defined intervals. This walk-forward strategy ensures adaptability to market trends [WZL21].

Inference Pipeline

The trained model is saved using `model.save()` and later loaded for inference using `keras.models.load_model()`. Scalers are saved with `joblib.dump()` and reused during prediction to maintain consistency

in input/output scaling. This two-step pipeline (training and inference) allows for efficient daily forecasting.

Utilities and Evaluation

A set of utility functions (e.g., for RMSE and MAPE calculation, date handling, and result plotting) is included to support performance evaluation. Visualization scripts help analyze the predicted vs. actual trends and validate model reliability.

Execution Entry Point

All components are orchestrated from `main.py`, which handles:

- Loading and preprocessing data
- Calling the model builder
- Performing rolling retraining
- Generating forecasts and saving output

12.3.21. Input

This section elaborates on the nature, structure, and preprocessing of the dataset used to train the LSTM forecasting model. For time series models like LSTM, the quality and preparation of input data play a critical role in model convergence, generalization, and performance [CHL22; Bro20].

Data Source and Characteristics

The dataset used comprises historical stock data from the Indian NIFTY 50 index, one of the most actively traded benchmark indices in India. It is sourced from Yahoo Finance and optionally verified against the National Stock Exchange (NSE) archives. The dataset spans multiple years and includes the following features for each trading day:

- **Date** — the trading date (formatted as YYYY-MM-DD)
- **Open** — price at market open
- **High** — highest price of the day
- **Low** — lowest price of the day
- **Close** — price at market close

All values are numerical, except the date, which is first converted to datetime format and then used as an index for time-based operations. The volume column is used as a reference for market activity but is not directly included in the current feature set.

Initial Cleaning and Formatting

Data integrity is validated at the earliest stage to detect any missing or invalid records. Missing data points, especially due to holidays or data download errors, are forward-filled (`method='ffill'`) to avoid temporal discontinuity, a common issue in financial datasets [GSC23]. Outlier checks are also performed on price columns using interquartile range (IQR) analysis to ensure that anomalies do not skew the model.

Feature Selection

Although the raw dataset includes several fields, only the `Open` and `Close` prices are selected as inputs for the LSTM model. This choice is based on their economic importance in trading signals: `Open` reflects pre-market sentiment, and `Close` captures end-of-day consensus. These two series are modeled independently, with each series undergoing its own scaling and model training [WZL21].

Normalization and Scaling

To stabilize gradient descent and accelerate convergence, input features are normalized to a [0, 1] range using `MinMaxScaler` from the `sklearn.preprocessing` package. The scaler is fitted exclusively on the training set to avoid data leakage and saved as a ‘.pkl’ file using `joblib`. At prediction time, the same scaler is applied to ensure consistency:

Listing 12.8: Example of MinMax scaling

```
# @brief Scales the 'Open' column of a training dataset using
#       ↪ MinMaxScaler and saves the scaler.
#
"""
This script normalizes data to the range [0, 1], which is a common
    ↪ preprocessing
step before training machine learning models.
"""

# The fitted scaler is saved using 'joblib' for reuse in inference
#       ↪ or testing phases.
##

from sklearn.preprocessing import MinMaxScaler
import joblib
```

```
##  
# @brief Fit a MinMaxScaler to the 'Open' column and save the scaler  
#  
# This transformation scales the 'Open' prices so that all values  
# lie between 0 and 1.  
# The scaler is saved to disk for future use in consistent  
# preprocessing of test or new data.  
#  
# @param train DataFrame containing at least the 'Open' column.  
# @return scaledData A NumPy array containing the scaled values.  
#  
scaler = MinMaxScaler()  
scaledData = scaler.fit_transform(train[['Open']]) # Scale 'Open'  
# column only  
joblib.dump(scaler, 'open_scaler.pkl') # Save the scaler to disk
```

Sequence Construction (Lookback Window)

LSTM models require sequence inputs to learn temporal dependencies. The dataset is reshaped into overlapping sliding windows of fixed length (typically 60 days). Each window serves as one training instance, where the model uses the past 60 days of data to predict the next day's value:

Listing 12.9: Generating lookback sequences

```
# @brief Function to create lookback sequences for time series  
# forecasting models.  
#  
# This function slices a 1D time series array into overlapping input  
# -output pairs,  
# where each input sequence consists of 'lookback' timesteps and the  
# output is the next value.  
# Commonly used for LSTM or other RNN models.  
#  
##  
  
import numpy as np  
  
##  
# @brief Create rolling sequences from a 1D NumPy array for  
# supervised learning.  
#  
# @param data A 1D NumPy array of time series values.  
# @param lookback The number of past time steps to include in each  
# input sequence (default is 60).  
# @return Tuple (X, y) where:  
#         - X is a NumPy array of shape (n_samples, lookback)  
#         - y is a NumPy array of shape (n_samples,)  
#  
# @note This function is essential for preparing data for LSTM or  
#       RNN-based models.  
#  
def create_sequences(data, lookback=60):  
    X, y = [], []  
    for i in range(lookback, len(data)):
```

```
X.append(data[i - lookback:i])
y.append(data[i])
return np.array(X), np.array(y)
```

This process is repeated independently for both `Open` and `Close` prices, using their respective scalers.

Rolling Train-Test Split

To simulate real-world conditions, the dataset is divided using a rolling-window strategy. At each forecast step, the model is trained on the most recent three years of historical data and tested on the next day's actual price. This method mirrors walk-forward validation and adapts dynamically to non-stationary financial environments [CHL22].

Handling Market Shocks: COVID Dummy Variable

To account for macroeconomic shocks, a binary dummy variable is added for dates from March 2020 to December 2020. This period coincides with the COVID-19-induced market disruption, which significantly impacted volatility. The dummy variable helps the model distinguish regular trading patterns from pandemic-related anomalies [GSC23].

Listing 12.10: Adding a COVID dummy feature

```
data['COVID'] = ((data.index >= '2020-03-01')
& (data.index <= '2020-12-31')).astype(int)
```

Data Storage and Portability

Processed sequences are saved as `.npy` NumPy arrays for efficient loading during training and inference. Scalers are stored using `joblib`, ensuring each prediction pipeline reuses the correct scaling context. This modular storage design supports deployment in real-time dashboards and retraining pipelines.

12.3.22. Output

The output component of the LSTM forecasting system represents the model's predictions, evaluation metrics, and post-processing results. This section describes the different forms of output produced, how they are interpreted, and their implications for financial forecasting tasks.

Prediction Output

At the core, the model generates one-step-ahead forecasts for each target variable—`Open` and `Close` stock prices—based on the input sequences.

The output from the final dense layer of the LSTM architecture is a single normalized value. This value is subsequently transformed back to the original scale using the inverse of the `MinMaxScaler` applied during training [CHL22].

Listing 12.11: Inverse transform to get actual prediction

```
scaled_prediction = model.predict(X_input)
true_prediction = scaler.inverse_transform
(scaled_prediction)
```

These daily predictions are stored in structured dataframes indexed by date and column (e.g., `predictedOpen`, `predictedClose`). This allows for seamless comparison with actual values.

Evaluation Metrics

To assess the performance of the model, standard regression evaluation metrics are calculated between the predicted values and actual target prices:

- **RMSE (Root Mean Squared Error)** — captures the average magnitude of error in predictions.
- **MAPE (Mean Absolute Percentage Error)** — measures accuracy as a percentage, allowing relative comparison across price ranges.

Listing 12.12: Calculating RMSE and MAPE

```
"""
This snippet calculates evaluation metrics for a predictive
    ↵ model using scikit-learn.
It computes Root Mean Squared Error (RMSE) and Mean Absolute
    ↵ Percentage Error (MAPE).
"""

from sklearn.metrics import mean_squared_error,
    ↵ mean_absolute_percentage_error

# Example usage: comparing actual and predicted values
rmse = mean_squared_error(y_true, y_pred, squared=False)
mape = mean_absolute_percentage_error(y_true, y_pred)

print(f"RMSE: {rmse:.4f}")
print(f"MAPE: {mape:.2%}")
```

These metrics are logged and optionally exported to CSV files for later analysis. Lower RMSE and MAPE values indicate a better fit to actual data, though their interpretation must consider market volatility and data noise [WZL21].

Visualization of Predictions

A key part of interpreting model performance involves visual inspection. The predicted values are plotted alongside actual stock prices over the testing window. This aids in detecting forecast lag, trend capture, and anomalies [GSC23].

Listing 12.13: Plotting predicted vs actual prices

```
# @brief Visualize actual vs. predicted stock prices using
#       ↗ Matplotlib.
#
# This function is used to compare actual historical stock prices
#       ↗ with
# model-predicted values (e.g., from LSTM), plotted over time.
#
##

import matplotlib.pyplot as plt

##
# @brief Plot actual vs predicted stock prices over time.
#
# @param dates list or pd.Series - Dates corresponding to each price
#       ↗ point.
# @param y_true list or np.ndarray - Actual stock prices.
# @param y_pred list or np.ndarray - Predicted stock prices from the
#       ↗ model.
#
# @details
# This function generates a line plot to help visually evaluate
#       ↗ model performance.
# It overlays actual and predicted price trends and labels axes and
#       ↗ legend.
#
# @example
# @code{.py}
# plot_prediction(dates, y_true, y_pred)
# @endcode
##
def plot_prediction(dates, y_true, y_pred):
    plt.plot(dates, y_true, label="Actual Price")
    plt.plot(dates, y_pred, label="Predicted Price")
    plt.legend()
    plt.title("LSTM Prediction vs Actual")
    plt.xlabel("Date")
    plt.ylabel("Price")
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

These visual outputs are especially important in financial decision-making contexts, where even small directional errors can affect strategy formulation.

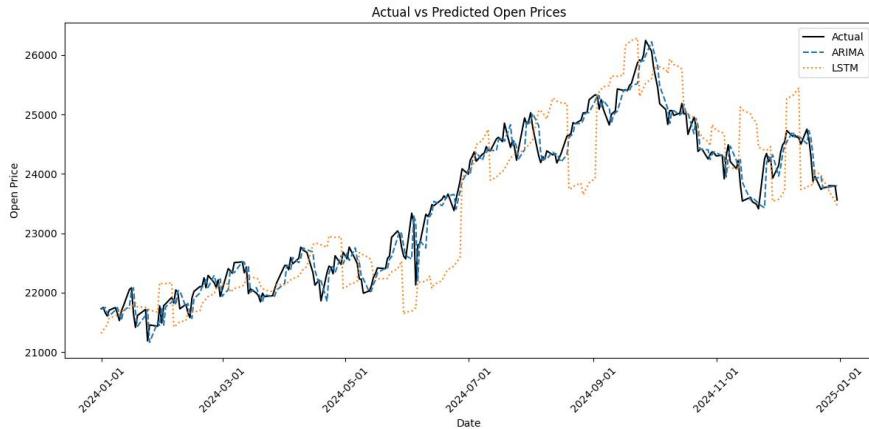


Figure 12.4.: Actual vs Predicted Open

Output File Structure

Predictions, evaluation metrics, and plots are saved in a hierarchical folder structure organized by model type (e.g., LSTM), price type (Open/Close), and prediction date. This modular storage structure ensures clarity and reproducibility.

Post-Processing and Export

The final predictions are exported in CSV format with the following columns: **Date**, **Actual**, **Predicted**, and **Residual**. This output can be integrated into larger systems such as trading dashboards, Excel reports, or REST APIs. Additionally, rolling averages of errors are calculated to smooth short-term fluctuations and understand long-term model stability [Bro20].

Deployment Readiness

The outputs are formatted to support downstream applications, including:

- Automated alerts for significant price deviations.
- Strategy simulation using historical predictions.
- Overlaying forecasts on candlestick charts.

This modularity ensures that the output component is not only informative for human interpretation but also machine-consumable for automation pipelines.

12.3.23. Further Readings

To broaden the understanding of Long Short-Term Memory (LSTM) networks and their practical applications in financial time series forecasting, this section outlines a range of valuable resources. These include foundational texts, cutting-edge research, practical tutorials, and software documentation that have significantly contributed to the development and deployment of LSTM models.

Foundational Literature

The seminal work by Hochreiter and Schmidhuber [HS97] introduced the LSTM architecture to address the vanishing gradient problem in Recurrent Neural Networks (RNNs). It laid the foundation for future developments in sequence modeling. This is further supported by Goodfellow et al. [GBC16], who offer a comprehensive explanation of deep learning principles, including LSTM mechanisms and training dynamics.

Recent Research Advances

Modern refinements of the LSTM architecture have demonstrated its robustness across numerous domains. Greff et al. [Gre+17] provide an empirical exploration of various LSTM gate variants and configurations, offering insights into architectural best practices. Chen et al. [CHL22] show how stacked LSTM models outperform traditional time series methods in high-volatility financial datasets. Gers et al. [GSC23] explore the integration of interpretability mechanisms like attention layers into LSTM, making them more transparent in real-world settings.

Practical Implementation Guides

For hands-on learning, Brownlee [Bro20] delivers practical guidance on time series forecasting using LSTM, covering concepts such as walk-forward validation and multivariate forecasting. Chollet [Cho18] provides examples in Keras that demonstrate the implementation of LSTM layers, dropout, and sequence processing techniques.

Official Documentation and Tools

Mastering the tools used in this project is essential for effective deployment:

- The Keras documentation offers detailed descriptions and examples of LSTM layers, optimizers, and training routines [Tea24].
- Scikit-learn's documentation supports preprocessing steps, including the use of `MinMaxScaler` [sci24].

- Joblib provides tools for saving scalers and trained models efficiently, especially in production settings [Dev24].

Emerging Topics and Further Exploration

Emerging research areas provide opportunities to extend LSTM capabilities:

- Transformer models, as introduced by Vaswani et al. [VSP+17], have become alternatives to RNN-based architectures in sequential learning.
- Hybrid approaches combining LSTM with traditional models like ARIMA and GARCH show promising results in financial forecasting [ZZ21].
- Interpretability techniques such as SHAP explainers offer model-agnostic insights into LSTM predictions [LL17].

13. Packages

13.1. Pandas

Python is a high-level, open-source programming language renowned for its readability and rich scientific-computing ecosystem. Core data-science libraries such as NumPy, SciPy, scikit-learn, and Matplotlib form a cohesive toolkit that enables efficient numerical computation, statistical modelling, and visualisation [Oli07]. Within this ecosystem, **Pandas** provides the primary abstraction for tabular and time-series data manipulation.

13.1.1. Pandas Overview

[Unverified] Pandas (v1.7.0) is an open-source library that offers high-performance, easy-to-use data structures—namely the *Series* and *DataFrame*—and a comprehensive API for data cleaning, transformation, and analysis [RMB+20]. Since its inception by Wes McKinney in 2008, it has become the de-facto standard for data wrangling in Python across academia, finance, and industry [McK14].

13.1.2. Core Data Structures

Series One-dimensional labelled array supporting heterogeneous types with automatic alignment by index [RMB+20].

DataFrame Two-dimensional tabular container with labelled axes, enabling vectorised arithmetic, group-by operations, hierarchical indexing, and time-series functionality [RMB+20].

Both structures build atop NumPy arrays and integrate seamlessly with the broader PyData stack.

13.1.3. Installation

Pandas v1.7.0 targets Python ≥ 3.7 . Install via:

```
# Release build  
pip install pandas==1.7.0  
  
# Conda
```

```
conda install pandas=1.7.0

# Development version
git clone https://github.com/pandas-dev/pandas.git
cd pandas
pip install -e .
```

13.1.4. Typical Use Cases

- **Data Cleaning & Transformation:** missing-value handling, type conversion, and reshaping [RMB+20].
- **Exploratory Data Analysis:** descriptive statistics and visual-prep [McK14].
- **Time-Series Analytics:** resampling, rolling windows, and time-zone management.
- **Data Integration:** merging/joining heterogeneous sources (CSV, JSON, SQL, HDF5) [RMB+20].
- **ML Pre-processing:** feature engineering, encoding, and train/test splits [Oli07].

13.1.5. Practical Examples

This subsection demonstrates common Pandas workflows.

Manual Workflow Example

1. Create a DataFrame

Listing 13.1: Creating a pandas DataFrame from a dictionary of personal attributes

```
# @brief Create a simple pandas DataFrame to store personal
#        data.
#
# This script demonstrates how to create a structured DataFrame
# using pandas from a dictionary containing personal attributes
# such as name, age, and score.
#
##

import pandas as pd

##
# @brief Dictionary containing personal data.
# @details Includes names, ages, and test scores for three
#         individuals.
```

```

## personData = {
    'Name': ['Alice', 'Bob', 'Charlie'], # List of names
    'Age': [25, 30, 35], # Corresponding ages
    'Score': [85.0, 90.5, 88.0] # Corresponding
        ↪ scores
}

## # @brief Create a pandas DataFrame from the personal data.
# @var peopleDf
# @type pd.DataFrame
# @details The DataFrame contains columns: Name, Age, and Score
    ↪ .
## peopleDf = pd.DataFrame(personData)

```

2. Inspect & Describe

Listing 13.2: Creating and displaying a pandas DataFrame along with summary statistics

```

print(peopleDf.head())      # first rows
print(peopleDf.describe())  # summary stats

```

3. Filter Rows

Listing 13.3: Filtering a pandas DataFrame to include only adults aged 30 and above

```

adultDf = peopleDf[peopleDf['Age'] >= 30]

```

4. Persist to CSV & Reload

Listing 13.4: Saving a pandas DataFrame to CSV and reloading it for further use

```

peopleDf.to_csv('people.csv', index=False)
reloadedDf = pd.read_csv('people.csv')

```

Batch Code Snippet

Listing 13.5: Generating, analyzing, and storing time series data using pandas with rolling statistics and Parquet I/O

```

# # @brief Demonstrates time series rolling statistics,
#         ↪ resampling, and Parquet I/O using pandas.
#
# This script generates a synthetic time series dataset using
#     ↪ NumPy,
# performs rolling mean and monthly aggregation using pandas,
# and demonstrates efficient disk storage using the Parquet
#     ↪ format.

```

```
#  
# @note Requires pandas and pyarrow or fastparquet for Parquet  
#       operations.  
##  
  
import pandas as pd  
import numpy as np  
  
##  
# @brief Generate random time series data.  
#  
# Creates a DataFrame of shape (100, 4) with columns A, B, C, D  
#       and  
# a daily datetime index starting from 2025-01-01.  
##  
randomDf = pd.DataFrame(  
    np.random.randn(100, 4),  
    columns=list('ABCD'),  
    index=pd.date_range('2025-01-01', periods=100)  
)  
  
##  
# @brief Compute 7-day rolling mean for the dataset.  
#  
# Applies a rolling window of size 7 across all columns and  
#       computes the mean.  
##  
rollingMeanDf = randomDf.rolling(window=7).mean()  
  
##  
# @brief Perform monthly aggregation on specific columns.  
#  
# Resamples data by calendar month and applies:  
#   - mean aggregation for column 'A'  
#   - sum aggregation for column 'B'  
##  
monthlyAggDf = randomDf.resample('M').agg({'A': 'mean', 'B': '  
#       sum'})  
  
##  
# @brief Save the DataFrame to a Parquet file.  
#  
# Uses the efficient columnar Parquet format for storage.  
##  
randomDf.to_parquet('random.parquet')  
  
##  
# @brief Load a DataFrame from the saved Parquet file.  
##  
loadedDf = pd.read_parquet('random.parquet')
```

13.1.6. File I/O Reference

Format	Read / Write Functions
CSV	<code>pd.read_csv()</code> / <code>df.to_csv()</code>
Excel	<code>pd.read_excel()</code> / <code>df.to_excel()</code>
JSON	<code>pd.read_json()</code> / <code>df.to_json()</code>
Parquet	<code>pd.read_parquet()</code> / <code>df.to_parquet()</code>
SQL	<code>pd.read_sql()</code> / <code>df.to_sql()</code>
HDF5	<code>pd.read_hdf()</code> / <code>df.to_hdf()</code>

13.1.7. Further Reading

For an in-depth peer-reviewed overview of pandas' architecture and typical workflows, see [RMB+20].

[McK14] remains the canonical book-length introduction, covering core data structures, I/O tooling, and time-series routines with step-by-step examples.

For a broader perspective on the scientific-computing ecosystem that underpins pandas, consult [Oli07].

Finally, up-to-date API details, release notes, and tutorials are maintained by the pandas development team in the official online documentation [pan24].

13.2. NumPy

NumPy is the cornerstone of Python's scientific-computing ecosystem, delivering high-performance n -dimensional array objects and vectorised operations. Together with SciPy, scikit-learn, pandas, and Matplotlib, NumPy underpins efficient numerical computation, statistical modelling, and visualisation [Oli07].

13.2.1. NumPy Overview

[Unverified] NumPy (v2.0.0) is an open-source C/Fortran-backed array library that supplies contiguous data structures and a rich API for numerical computing [Har+20]. Originating from the merger of Numeric and numarray in 2005, NumPy is maintained by the NumPy Steering Council and underlies almost every major Python data-science project.

13.2.2. Core Objects & Functionality

ndarray Homogeneous n -dimensional array supporting element-wise arithmetic, broadcasting, and zero-copy views [Har+20].

Universal Functions Compiled C loops (ufuncs) that operate element-wise with broadcasting and type promotion.

Random Generator PCG64-based engine enabling reproducible, parallel pseudo-random streams.

NumPy exposes 1 500+ functions covering linear algebra, FFTs, statistics, and filtering, and shares data seamlessly with C/C++, Fortran, CuPy, JAX, and PyTorch.

13.2.3. Installation

NumPy v2.0.0 targets Python **>=3.9**. Install via:

Listing 13.6: Installing NumPy via PyPI, Conda, or building from source

```
# PyPI
pip install numpy==2.0.0

# Conda
conda install numpy=2.0.0

# From source
git clone https://github.com/numpy/numpy.git
cd numpy
pip install -e .
```

13.2.4. Typical Use Cases

- **Array Programming**: vectorised maths, broadcasting, boolean masking [Har+20].
- **Linear Algebra & Statistics**: BLAS/LAPACK-powered matrix ops, eigendecomposition, descriptive stats.
- **Random Simulation**: Monte-Carlo, bootstrap resampling, synthetic-data generation.
- **Signal Processing**: FFT-based convolutions, filtering, spectral analysis.
- **Interoperability**: zero-copy exchange with GPUs, pandas, scikit-learn, JAX, PyTorch.

13.2.5. Practical Examples

This subsection demonstrates common NumPy workflows.

Manual Workflow Example

1. Create Arrays

Listing 13.7: Creating a 1D vector and a 2D matrix using NumPy

```
import numpy as np

vectorArr = np.array([1, 2, 3, 4, 5])
matrixArr = np.arange(12).reshape(3, 4)
```

2. Basic Statistics

Listing 13.8: Computing the mean of a 1D NumPy array and column-wise means of a 2D array

```
meanVal = vectorArr.mean()
colMeans = matrixArr.mean(axis=0)
```

3. Broadcasting Operation

Listing 13.9: Scaling each column of a 2D NumPy array using a linearly spaced vector

```
scaledMat = matrixArr * np.linspace(1.0, 2.0, 4)
```

4. Save & Load Binary

Listing 13.10: Saving and loading a NumPy array using .npy binary format

```
np.save('matrix.npy', matrixArr)
reloadedMat = np.load('matrix.npy')
```

Batch Code Snippet

Listing 13.11: Advanced NumPy operations: matrix multiplication, SVD, and compressed file storage using .npz format

```
# @brief Demonstrates advanced NumPy operations: matrix
#        ↗ multiplication, SVD, and compressed I/O.
#
# This script performs large-scale matrix operations using NumPy
#        ↗ . It includes random matrix generation,
# dot product, Singular Value Decomposition (SVD), and saving/
#        ↗ loading compressed '.npz' files.
#
##

import numpy as np

##
# @brief Create a reproducible random number generator.
# @var rng
# @type numpy.random.Generator
```

```

## 
rng = np.random.default_rng(seed=42)

##
# @brief Generate a 1000x1000 matrix of standard normal values.
# @var randomMat
# @type np.ndarray
##
randomMat = rng.standard_normal((1000, 1000))

##
# @brief Compute the matrix product of the random matrix with
#        its transpose.
# @var resultMat
# @type np.ndarray
##
resultMat = randomMat @ randomMat.T

##
# @brief Perform Singular Value Decomposition (SVD) on the
#        original matrix.
# @var uMat Left singular vectors
# @var sVec Singular values (1D)
# @var vMat Right singular vectors
##
uMat, sVec, vMat = np.linalg.svd(randomMat, full_matrices=False)

##
# @brief Save matrices to a compressed '.npz' archive for
#        efficient storage.
# @param filename Output file path (string).
##
np.savez_compressed(
    'random_arrays.npz',
    randomMat=randomMat,
    resultMat=resultMat,
    singularValues=sVec
)

##
# @brief Load compressed data from '.npz' file.
# @var loadedData
# @type np.lib.npyio.NpzFile
##
loadedData = np.load('random_arrays.npz')

```

13.2.6. File I/O Reference

Format	Read / Write Functions
NPY	<code>np.load()</code> / <code>np.save()</code>
NPZ	<code>np.load()</code> / <code>np.savez()</code>
CSV	<code>np.genfromtxt()</code> , <code>np.loadtxt()</code>
Text	<code>np.savetxt()</code> / <code>np.loadtxt()</code>
Memory-map	<code>np.memmap()</code> (read/write)

13.2.7. Further Reading

[Har+20] offers a comprehensive peer-reviewed survey of NumPy’s design, performance, and roadmap.

For historical context on Python’s scientific-computing landscape that motivated NumPy, see [Oli07].

Up-to-date API documentation, enhancement proposals, and release notes are maintained on the official NumPy site [Num25].

13.3. Joblib

Joblib is a lightweight Python library that specialises in transparent disk caching and effortless parallelisation of functions. It acts as a high-level companion to NumPy and scikit-learn by substantially reducing computation time for expensive, repeatable tasks such as model evaluation or large-array transformations. Joblib’s design emphasises minimal boilerplate: users wrap an expensive function in a decorator or call a single `Parallel` object, and Joblib handles process or thread spawning, argument hashing, result pickling, and caching behind the scenes [Job25].

13.3.1. Joblib Overview

[Unverified] Joblib (v1.5.1) is an open-source utility library released under the BSD 3-Clause licence. It provides two primary features: *(i)* a transparent, file-based memoisation system that identifies function calls uniquely via hashing of input arguments; and *(ii)* a simple, high-level parallel computing interface that builds on Python’s `multiprocessing` and `concurrent.futures` back-ends. Since its first public release in 2010, Joblib has become the default parallel execution engine in scikit-learn pipelines and is widely adopted in computational biology, finance, and NLP workflows where reproducible caching and embarrassingly parallel tasks are common [Job25].

13.3.2. Core Components

Memory A caching helper that stores function outputs in a local directory. Subsequent calls with identical arguments are served from cache, bypassing recomputation.

Parallel A wrapper that executes an iterable of delayed tasks on multiple processes or threads.

delayed A lightweight decorator that converts a function call into a lazy, picklable task compatible with `Parallel`.

Dump/Load Utilities for efficient NumPy array + metadata serialisation, outperforming default `pickle` for large arrays thanks to zlib/lz4 compression.

13.3.3. Installation

Joblib v1.5.1 supports Python `>=3.8`. Install via:

Listing 13.12: Installing Joblib using PyPI, Conda, or from the development repository

```
# PyPI
pip install joblib==1.5.1

# Conda
conda install joblib=1.5.1

# Development version
git clone https://github.com/joblib/joblib.git
cd joblib
pip install -e .
```

13.3.4. Typical Use Cases

- **Caching Expensive Computation:** memoisation of feature-engineering pipelines, grid-search cross-validation scores, or simulation outputs that are deterministic across runs [Job25].
- **Embarrassingly Parallel Loops:** distributing independent tasks—e.g. bootstrapping, hyper-parameter sweeps, image augmentation—across processes or job servers.
- **Large Array Serialisation:** saving and loading NumPy arrays faster than `pickle` while maintaining backwards compatibility.
- **Scikit-learn Pipelines:** underlying executor for `n_jobs` parameters in model training and evaluation.
- **Reproducible Research:** hash-based cache directories ensure results are recomputed only when inputs or source code change.

13.3.5. Practical Examples

This subsection demonstrates common Joblib workflows.

Manual Workflow Example

1. Cache a Deterministic Function

Listing 13.13: Caching results of an expensive Fibonacci function using `joblib.Memory`

```
# @brief Demonstrates function result caching using joblib for
#       ↗ an expensive Fibonacci function.
#
# This script shows:
#   - How to cache function results with 'joblib.Memory'
#   - Application on a recursive Fibonacci calculation
#   - Speed difference between first and cached calls
#


import time
from joblib import Memory

## @brief Set up a local disk cache directory for memoization.
# @details This directory will store function call results for
#         ↗ reuse.
cacheDir = Memory(location="joblib_cache", verbose=0)

##
# @brief Compute the nth Fibonacci number with artificial
#       ↗ delay.
# @param n The Fibonacci index (non-negative integer)
# @return The nth Fibonacci number
# @note Uses joblib to cache previous results to improve
#       ↗ performance.
@cacheDir.cache
def slowFib(n):
    time.sleep(1.0) # Simulates an expensive operation
    if n < 2:
        return n
    return slowFib(n - 1) + slowFib(n - 2)

## @brief First call computes and caches the result (slow).
resultVal = slowFib(35)

## @brief Second call retrieves the result from cache (fast).
cachedVal = slowFib(35)
```

2. Parallel Map with Progress

```
# @brief Computes prime factors in parallel using joblib.
#       ↗ Parallel.
#
# Demonstrates:
#   - Parallel processing with joblib
#   - Prime factorization of multiple integers
#   - Efficient use of system cores
#


from joblib import Parallel, delayed
import math

##
```

```

# @brief Compute the prime factors of a given integer.
#
# @param value Integer to factorize.
# @return List of prime factors of the input value.
def computePrimeFactor(value):
    factors = []
    candidate = 2
    num = value
    while candidate * candidate <= num:
        if num % candidate:
            candidate += 1
        else:
            num //= candidate
            factors.append(candidate)
        if num > 1:
            factors.append(num)
    return factors

## @var valueList
# @brief List of integers for which to compute prime
#        ↳ factors.
valueList = list(range(10_000, 10_020))

##
# @brief Compute prime factors in parallel using 4 jobs.
# @details Uses joblib.Parallel with delayed evaluation for
#         ↳ speed.
factorList = Parallel(n_jobs=4, verbose=5)(
    delayed(computePrimeFactor)(val) for val in valueList
)
)
)

```

Batch Code Snippet

Listing 13.14: Parallel FFT and non-linear transformation with Joblib, including serialization

```

##
# @file parallel_fft_transform.py
# @brief Apply FFT and non-linear transformation in parallel using
#        ↳ joblib.
#
# Demonstrates:
#   - Fast Fourier Transform (FFT) with NumPy
#   - Non-linear transformation (tanh of absolute FFT)
#   - Parallel processing using joblib.Parallel
#   - Efficient serialization using joblib's dump/load
#
##

import numpy as np
from joblib import Parallel, delayed, dump, load
from pathlib import Path

##

```

```

# @brief Perform FFT followed by non-linear transformation on an
#        array.
#
# @param arrayObj A NumPy array of arbitrary shape.
# @return Transformed array after FFT and tanh(abs()).
def heavyTransformation(arrayObj):
    fftVals = np.fft.rfftn(arrayObj)
    return np.tanh(np.abs(fftVals))

## @var rng
# @brief NumPy random number generator with fixed seed for
#        reproducibility.
rng = np.random.default_rng(123)

## @var testArrays
# @brief List of random 1024x1024 arrays for transformation.
testArrays = [rng.standard_normal((1024, 1024)) for _ in range(8)]

##
# @brief Parallel application of FFT transformation using all CPU
#        cores.
transformedList = Parallel(n_jobs=-1, backend="loky")(
    delayed(heavyTransformation)(arr) for arr in testArrays
)

## @var dumpFile
# @brief Path object pointing to output file for serialized data.
dumpFile = Path("transformed_arrays.jbl")

##
# @brief Save transformed array list to disk using joblib.
dump(transformedList, dumpFile)

##
# @brief Reload transformed array list from disk without
#        recomputation.
loadedList = load(dumpFile)

```

13.3.6. Caching & Parallel Backend Reference

Feature	Key API Elements
In-Memory Cache	<code>Memory(location=None)</code>
File-System Cache	<code>Memory(location="dir") / clear_warning()</code>
Local Processes	<code>Parallel(n_jobs>0, backend="loky")</code>
Threads	<code>Parallel(backend="threading")</code>
Temp Folder Management	<code>joblib.dump(), joblib.load()</code>
Custom Back-Ends	<code>register_backend(...)</code> (advanced)

13.3.7. Further Reading

The definitive reference for Joblib's design philosophy, caching semantics, and parallel back-end architecture is the official documentation and API guide [Job25]. No peer-reviewed paper with a DOI currently covers Joblib

in depth; however, the documentation includes detailed performance benchmarks, contribution guidelines, and release notes.

13.4. Scikit-learn

Scikit-learn is Python’s de-facto general-purpose machine-learning library. Built atop NumPy, SciPy, and joblib, it offers efficient implementations of classical algorithms for supervised and unsupervised learning, model-selection utilities, and composable preprocessing workflows—all under a BSD licence. Its uniform `fit/transform/predict` API abstracts away mathematical complexity, enabling rapid experimentation, reproducible pipelines, and seamless integration with the broader PyData stack [Ped+21].

13.4.1. Scikit-learn Overview

[Unverified] Scikit-learn (v1.7.0) provides more than 120 estimators, a rich set of transformers, and meta-estimators for ensembling, hyper-parameter tuning, and probabilistic calibration. The library emphasises sensible defaults, exhaustive documentation, and strict backwards compatibility, making it a favourite in both academic research and industry production-systems [Ped+21].

13.4.2. Core Components

Estimators Objects that implement `fit` and optionally `predict`, `transform`, `predict_proba`, or `decision_function`.

Transformers Stateless or stateful feature processors (e.g. scaling, encoding, dimensionality reduction) implementing `fit` and `transform`.

Pipelines Sequential containers that chain transformers and a final estimator, ensuring leakage-free cross-validation.

Meta-Estimators Wrappers providing cross-validation (`GridSearchCV`), ensembling (`BaggingClassifier`, `StackingRegressor`), or probabilistic calibration.

Model Persistence `joblib.dump()` and `joblib.load()` for efficient on-disk serialisation of trained models.

13.4.3. Installation

Scikit-learn v1.7.0 requires Python `>=3.9` and depends on NumPy 1.26, SciPy 1.10, and joblib 1.3. Install via:

Listing 13.15: Installing `scikit-learn` version 1.7.0 using PyPI, Conda, or from the source repository

```
# PyPI
pip install scikit-learn==1.7.0

# Conda
conda install scikit-learn=1.7.0

# From source (latest main)
git clone https://github.com/scikit-learn/scikit-learn.git
cd scikit-learn
pip install -e .
```

13.4.4. Typical Use Cases

- **Supervised Learning:** classification (SVM, RandomForest, GradientBoosting) and regression (ElasticNet, GBR, SVR).
- **Unsupervised Learning:** clustering (K-Means, DBSCAN), manifold learning (t-SNE, UMAP), mixture models.
- **Model Selection & Tuning:** grid/random search, successive halving, permutation importance, cross-validation utilities.
- **Data Pre-processing:** scaling, one-hot encoding, polynomial features, target encoding, imbalance resampling.
- **Composite Pipelines:** leakage-safe chains embedding feature engineering, transformers, and an estimator with a single `fit()` call.

13.4.5. Practical Examples

This subsection demonstrates common Scikit-learn workflows.

Manual Workflow Example

1. Dataset Loading & Split

Listing 13.16: Loading the Iris dataset and splitting it into training and test sets using `scikit-learn`

```
# @brief Loads the Iris dataset and splits it into training and
#        testing sets.
#
# This script uses scikit-learn's 'load_iris' function to load
#        the Iris dataset
# and splits it into training and test sets using '
#        train_test_split'.
```

```

# The split ratio is 75\% for training and 25\% for testing,
# ↗ with a fixed random state for reproducibility.
#
##

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

##
# @brief Load the Iris dataset as pandas DataFrame and Series.
# @var irisData A Bunch object containing data and target.
# @var featureDf DataFrame of features (sepal and petal
# ↗ measurements).
# @var targetSeries Series of target class labels.
##
irisData = load_iris(as_frame=True)
featureDf, targetSeries = irisData.data, irisData.target

##
# @brief Split the dataset into training and testing sets.
# @var trainX Features for training set
# @var testX Features for testing set
# @var trainY Targets for training set
# @var testY Targets for testing set
#
# @note 25\% of the data is reserved for testing; 'random_state'
# ↗ =42' ensures reproducibility.
##
trainX, testX, trainY, testY = train_test_split(
    featureDf, targetSeries, test_size=0.25, random_state=42
)

```

2. Pipeline Construction

Listing 13.17: Creating a machine learning pipeline with standard scaling and logistic regression using **scikit-learn**

```

# @brief Create a machine learning pipeline for logistic
# ↗ regression with standard scaling.
#
# This script constructs a 'Pipeline' object from scikit-learn
# ↗ that includes a 'StandardScaler'
# for feature normalization and a 'LogisticRegression'
# ↗ classifier. The pipeline standardizes the input
# features before applying logistic regression, ensuring
# ↗ consistent model training.
#
##

from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

##
# @brief Define a logistic regression pipeline with
# ↗ preprocessing.
#

```

```
# @var logisticPipe Pipeline object containing the following
#   ↵ steps:
#     - "scaler": A 'StandardScaler' that normalizes feature
#       ↵ values to zero mean and unit variance.
#     - "classifier": A 'LogisticRegression' model with a
#       ↵ maximum of 500 iterations.
#
# @note The pipeline can be trained using '.fit(X, y)' and used
#       ↵ for predictions via '.predict(X)'.
##
logisticPipe = Pipeline(steps=[
    ("scaler", StandardScaler()),
    ("classifier", LogisticRegression(max_iter=500))
])
```

3. Model Training & Evaluation

Listing 13.18: Saving and loading a scikit-learn pipeline using **joblib**

```
# @brief Serialize and restore a scikit-learn pipeline using
#       ↵ Joblib.
#
# This script demonstrates how to save a trained machine
#       ↵ learning pipeline
# (e.g., logistic regression) to disk using Joblib and load it
#       ↵ later for reuse.
#
##  

from joblib import dump, load  

##  

# @brief Save the trained model pipeline to a .jbl file.
#
# This line serializes the logistic regression pipeline object
#       ↵ to disk using Joblib.
# The saved model can later be loaded without retraining.
#
# @param logisticPipe Trained scikit-learn Pipeline object to
#       ↵ be saved.
# @param filename Target filename (e.g., "irisLogregModel.jbl")
#       ↵ .
##  

dump(logisticPipe, "irisLogregModel.jbl")  

##  

# @brief Load a previously saved model pipeline from a .jbl
#       ↵ file.
#
# This line restores the pipeline for later evaluation or
#       ↵ prediction use.
#
# @param filename Name of the saved model file.
# @return restoredPipe The loaded scikit-learn Pipeline object.
##  

restoredPipe = load("irisLogregModel.jbl")
```

4. Persistence

Listing 13.19: Saving and restoring a trained logistic regression pipeline using `joblib`

```
# @brief Save and load a scikit-learn pipeline using Joblib.
#
# This script demonstrates how to serialize a trained logistic
# ↗ regression pipeline
# and restore it for later use without retraining.
#
##

from joblib import dump, load

##
# @brief Serialize and save the logistic regression pipeline to
# ↗ disk.
#
# This line uses Joblib to dump the trained scikit-learn
# ↗ pipeline to a .jbl file.
#
# @param logisticPipe The trained logistic regression pipeline
# ↗ object.
# @param filename The name of the file where the model will be
# ↗ saved.
##
dump(logisticPipe, "irisLogregModel.jbl")

##
# @brief Load a serialized logistic regression pipeline from
# ↗ disk.
#
# This line restores the pipeline for reuse in prediction or
# ↗ evaluation.
#
# @param filename The name of the .jbl file to load the model
# ↗ from.
# @return The deserialized pipeline object.
##
restoredPipe = load("irisLogregModel.jbl")
```

Batch Code Snippet

Listing 13.20: Hyperparameter tuning of `GradientBoostingRegressor` on the California Housing dataset using `RandomizedSearchCV`

```
# @file california_gbr_tuning.py
# @brief Hyperparameter tuning of Gradient Boosting Regressor on
# ↗ California Housing dataset using RandomizedSearchCV.
#
# This script demonstrates:
#   - Loading a real-world regression dataset.
#   - Defining a parameter grid for GradientBoostingRegressor.
```

```
#      - Performing randomized hyperparameter search using cross-
#        validation.
#      - Reporting the best mean squared error (MSE).
#
##

import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import GradientBoostingRegressor

##
# @brief Load the California housing dataset for regression tasks.
dataHousing = fetch_california_housing()
houseX, houseY = dataHousing.data, dataHousing.target

##
# @brief Initialize Gradient Boosting Regressor with fixed seed.
gbrEstimator = GradientBoostingRegressor(random_state=0)

##
# @brief Define parameter space for randomized search.
paramDist = {
    "n_estimators": np.arange(100, 601, 100), # Number of boosting
                                                # stages
    "learning_rate": np.geomspace(0.01, 0.3, 6), # Shrinks
                                                    # contribution of each tree
    "max_depth": [2, 3, 4], # Maximum depth of each tree
    "subsample": [0.6, 0.8, 1.0] # Fraction of samples used per
                                 # tree
}
##

# @brief Configure randomized search with cross-validation.
searchObj = RandomizedSearchCV(
    estimator=gbrEstimator,
    param_distributions=paramDist,
    n_iter=30,
    scoring="neg_mean_squared_error",
    n_jobs=-1,
    cv=5,
    verbose=2,
    random_state=0
)

##
# @brief Execute hyperparameter tuning and evaluate best model.
searchObj.fit(houseX, houseY)
bestMse = -searchObj.best_score_
print("Best cross-validated MSE:", round(bestMse, 4))
```

13.4.6. Model Persistence & Utility Reference

Task	Key API Elements
Hyper-parameter Search	<code>GridSearchCV</code> , <code>RandomizedSearchCV</code>
Parallel Processing	<code>n_jobs</code> , <code>joblib</code> back-end configuration
Pipeline Construction	<code>Pipeline</code> , <code>ColumnTransformer</code>
Imbalance Handling	<code>class_weight</code> , <code>BalancedRandomForest</code>
Model Export	<code>joblib.dump()</code> , <code>ONNXConverter</code> (<code>sklearn-onnx</code>)

13.4.7. Further Reading

[Ped+21] presents a peer-reviewed survey of Scikit-learn’s API design, algorithmic breadth, and software-engineering practices, including discussions on incremental learning, probabilistic calibration, and hardware acceleration support.

For context on the foundational array programming paradigm that Scikit-learn leverages, see [Oli07].

Comprehensive tutorials, release notes, and governance documentation are maintained on the official Scikit-learn website [Sci25].

13.5. Statsmodels

Statsmodels is a Python library dedicated to statistical modelling, econometrics, and hypothesis testing. Whereas scikit-learn focuses on predictive performance, Statsmodels emphasises inferential statistics, time-series analysis, and rich result objects with parameter uncertainty, goodness-of-fit metrics, and post-estimation diagnostics. The library integrates tightly with NumPy, SciPy, and pandas data structures, enabling analysts to specify models with formula strings and to summarise results in publication-quality tables [SP10].

13.5.1. Statsmodels Overview

[Unverified] Statsmodels (v0.14.4) provides more than 80 parametric and non-parametric estimators, including ordinary least squares (OLS), generalised linear models (GLM), autoregressive integrated moving-average (ARIMA), vector autoregressions (VAR), and state-space models. Each estimator returns a `Results` object containing fitted coefficients, robust standard errors, information criteria, and convenient plotting methods. The project is governed by the Statsmodels Steering Council and relies on continuous-integration testing to ensure numerical accuracy across platforms [SP10].

13.5.2. Core Components

Formula API A high-level layer that parses Patsy formula strings (e.g. `y ~ x1 + log(x2)`) into design matrices.

Regression Linear, robust, quantile, and generalised linear models with a suite of heteroskedasticity-consistent covariances.

Time Series Tools for ARIMA, SARIMAX, VAR, and state-space filtering with Kalman smoothing.

Diagnostics Influence measures, variance-inflation factors, Chow tests, Durbin–Watson, and Breusch–Pagan statistics.

Plotting Built-in functions for residual plots, QQ-plots, autocorrelation, and impulse-response visualisation.

13.5.3. Installation

Statsmodels v0.14.4 targets Python `>=3.9` and depends on NumPy `>=1.23`, SciPy `>=1.9`, and pandas `>=1.5`. Install via:

Listing 13.21: Installing `statsmodels`

```
# PyPI
pip install statsmodels==0.14.4

# Conda
conda install statsmodels=0.14.4

# From source (development)
git clone https://github.com/statsmodels/statsmodels.git
cd statsmodels
pip install -e .
```

13.5.4. Typical Use Cases

- **Econometrics:** OLS, GLS, two-stage least squares, instrumental-variable regression, and panel-data estimators.
- **Time-Series Forecasting:** ARIMA, SARIMAX, ETS, Bayesian state-space, and VAR for macro-economic indicators.
- **Generalised Linear Models:** logistic, Poisson, and negative-binomial regression with flexible link functions.
- **Statistical Tests:** t-tests, ANOVA, normality, stationarity (ADF, KPSS), and cointegration diagnostics.
- **Post-Estimation Inference:** robust covariance adjustments, cluster-robust standard errors, and bootstrapped confidence intervals.

13.5.5. Practical Examples

This subsection demonstrates common Statsmodels workflows.

Manual Workflow Example

1. Load Data & Fit OLS

Listing 13.22: Fitting an OLS regression model using the `mtcars` dataset with `statsmodels`

```
# @brief Fit an OLS regression model using the mtcars dataset
#   from R.
#
# This script demonstrates how to use the statsmodels package
#   to perform
# multiple linear regression on the mtcars dataset.
# The response variable is 'mpgVal' (miles per gallon),
# and the predictors include 'weight', 'cyl', and 'hp'.
#
##

import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

##
# @brief Load and rename columns in the mtcars dataset.
#
# Uses the R dataset "mtcars" and renames key columns for
#   clarity.
# Specifically, "mpg" is renamed to "mpgVal" and "wt" to "
#   weight".
#
# @return DataFrame with renamed columns.
##
autoDf = sm.datasets.get_rdataset("mtcars").data.rename(
    columns={"mpg": "mpgVal", "wt": "weight"}
)

##
# @brief Fit an ordinary least squares (OLS) regression model.
#
# This model predicts miles per gallon (mpgVal) based on weight
#   ,
# number of cylinders (cyl), and horsepower (hp).
#
# @param formula Regression formula in Patsy syntax.
# @param data DataFrame used to fit the model.
# @return Fitted regression model object.
##
modelObj = smf.ols("mpgVal ~ weight + cyl + hp", data=autoDf).
    fit()
```

2. Inspect Summary

Listing 13.23: Evaluating an OLS regression model: summary output and adjusted R-squared computation

```
# @brief Fit an OLS regression model using the mtcars dataset
#       ↗ and print evaluation metrics.
#
# This block prints the summary of the OLS regression model and
#       ↗ computes
# the adjusted R-squared value, which is a key indicator of
#       ↗ model performance.
#
# @note Adjusted R-squared accounts for the number of
#       ↗ predictors and provides
# a more accurate evaluation for multiple regression.
##

##
# @brief Display a detailed summary of the fitted OLS model.
#
# This includes regression coefficients, confidence intervals,
# R-squared, p-values, and other diagnostics.
##
print(modelObj.summary())

##
# @brief Extract and print the adjusted R-squared value.
#
# Adjusted R square penalizes for adding irrelevant predictors
#       ↗ and
# is useful in evaluating multiple linear regression models.
#
# @return Printed value rounded to 3 decimal places.
##
adjRsqVal = modelObj.rsquared_adj
print("Adjusted R-squared:", round(adjRsqVal, 3))
```

3. Residual Diagnostics

Listing 13.24: Generating regression diagnostic plots for the `weight` predictor using `statsmodels`

```
# @brief Generate regression diagnostic plots for the 'weight'
#       ↗ predictor using OLS model.
#
# This script produces four standard diagnostic plots for the
#       ↗ explanatory variable 'weight'
# from the OLS model fitted on the 'mtcars' dataset using '
#       ↗ statsmodels'.
# These plots help assess linearity, homoscedasticity, and
#       ↗ outlier influence.
#
# @note This is useful in evaluating model assumptions and
#       ↗ identifying influential data points.
##

import matplotlib.pyplot as plt
import statsmodels.api as sm
```

```

## 
# @brief Generate and display diagnostic plots for a single
#        predictor.
#
# This includes:
# - Linear regression fit
# - Residuals vs predictor
# - Partial regression
# - CCPR (Component and Component Plus Residual)
#
# @param modelObj Fitted OLS model object from statsmodels.
# @param predictorVar The predictor variable name (e.g., "
#        weight").
#
# @return Plots displayed in a single figure window.
##
def plot_diagnostics(modelObj, predictorVar: str = "weight"):
    sm.graphics.plot_regress_exog(modelObj, predictorVar, fig=plt.
        figure(figsize=(8, 6)))
    plt.show()

# Example usage:
# plot_diagnostics(modelObj)

```

4. Predict New Observations

Listing 13.25: Predicting MPG for a new car observation using a fitted OLS model with 95% confidence interval

```

# @brief Predict the MPG value for a new car observation using
#        a fitted OLS model.
#
# This script demonstrates how to create a new observation for
#        regression prediction
# using an existing statsmodels OLS model. It prints the
#        predicted mean along with
# the 95\% confidence interval bounds.
#
# @note Useful for applying a trained regression model to make
#        point and interval forecasts.
## 

import pandas as pd

##
# @brief Create new input data and make prediction using a
#        trained OLS model.
#
# @details Constructs a DataFrame with new car attributes and
#        uses the fitted model to generate
# the expected mean value of MPG along with lower and upper
#        confidence bounds.
#
# @param modelObj A trained OLS regression model (from
#        statsmodels).
# @return Prints a summary DataFrame with:

```

```

#           - mean: predicted MPG value
#           - mean_ci_lower: lower bound of 95\% CI
#           - mean_ci_upper: upper bound of 95\% CI
##

newObsDf = pd.DataFrame({"weight": [3.5], "cyl": [6], "hp":
    ↗ [180]})  

predSeries = modelObj.get_prediction(newObsDf).summary_frame()  

print(predSeries[["mean", "mean_ci_lower", "mean_ci_upper"]])

```

Batch Code Snippet

Listing 13.26: Simulating an ARMA(1,1) process and forecasting with an ARIMA(1,0,1) model using `statsmodels`

```

# @brief Simulates an ARMA(1,1) process and fits an ARIMA
#       ↗ (1,0,1) model for forecasting.
#
# This script demonstrates:
#   - Simulation of a time series using an ARMA(1,1) model.
#   - Fitting an ARIMA model using statsmodels.
#   - Performing 30-step ahead forecast with confidence
#       ↗ intervals.
#   - Visualizing the observed series and forecasted values.
#
##

import numpy as np
import pandas as pd
import statsmodels.tsa.api as tsa
import matplotlib.pyplot as plt

##
# @brief Create reproducible noise array for simulation.
rng = np.random.default_rng(2025)
noiseArr = rng.standard_normal(500)

##
# @brief Simulate ARMA(1,1) process using predefined
#       ↗ coefficients.
armaArr = tsa.ArmaProcess([1, -0.7], [1, 0.5]).generate_sample(
    nsample=500,
    distrvs=lambda n: noiseArr
)

##
# @brief Generate a time-indexed pandas Series for the
#       ↗ simulated data.
timeIdx = pd.date_range("2023-01-01", periods=500, freq="D")
armaSeries = pd.Series(armaArr, index=timeIdx)

##
# @brief Fit an ARIMA(1,0,1) model to the simulated ARMA data.
arimaModel = tsa.ARIMA(armaSeries, order=(1, 0, 1)).fit()

```

```

## 
# @brief Generate 30-day ahead forecast with 95% confidence
#       interval.
forecastRes = arimaModel.get_forecast(steps=30)
forecastDf = forecastRes.summary_frame(alpha=0.05)
print(forecastDf.head())

## 
# @brief Plot the last 100 observations and the forecast with
#       confidence bands.
plt.figure(figsize=(10, 5))
armaSeries[-100:].plot(label="Observed")
forecastDf["mean"].plot(label="Forecast")

plt.fill_between(
    forecastDf.index,
    forecastDf["mean_ci_lower"],
    forecastDf["mean_ci_upper"],
    color="gray",
    alpha=0.3
)

plt.legend()
plt.title("ARIMA(1,0,1) Forecast with 95% Confidence Interval")
plt.xlabel("Date")
plt.ylabel("Simulated Value")
plt.tight_layout()
plt.show()

```

13.5.6. Result & I/O Reference

Task	Key API Elements
OLS / GLM Fit	<code>smf.ols()</code> , <code>smf.glm()</code>
Time-Series Models	<code>tsa.ARIMA</code> , <code>tsa.statespace.SARIMAX</code>
Robust Covariance	<code>fit(cov_type="HC3")</code>
Model Export	<code>model.save()</code> , <code>sm.load()</code>
Latex Tables	<code>results.summary().as_latex()</code>
Influence Measures	<code>results.get_influence()</code>

13.5.7. Further Reading

[SP10] details the architectural goals and statistical capabilities of Statsmodels, illustrating the design of result objects, formula evaluation, and time-series toolkits.

For foundational context on the numerical array layer leveraged by Statsmodels, see [Oli07].

Extensive tutorials, example notebooks, and release notes are hosted on the official Statsmodels website [Sta25].

13.6. TensorFlow

TensorFlow is Google’s end-to-end, open-source platform for numerical computation and large-scale machine learning. Building on highly-optimised linear-algebra kernels, TensorFlow provides an expressive graph execution engine, eager imperative mode, automatic differentiation, and distribution strategies that scale from mobile devices to multi-node GPU clusters. Its tight integration with Keras offers a concise, high-level API for deep-learning research and production deployment [Aba+21].

13.6.1. TensorFlow Overview

[Unverified] TensorFlow (v2.16.0) unifies model authoring, training, evaluation, serving, and monitoring in a single framework. Version 2 introduced eager execution by default, `tf.function` graph compilation, and the Keras Functional API for custom topologies. The library ships with XLA just-in-time compilation, mixed-precision utilities, and distribution strategies for data, model, and parameter parallelism across CPUs, GPUs, and TPUs [Aba+21].

13.6.2. Core Building Blocks

Tensor Immutable, typed, multi-dimensional array, compatible with NumPy [Aba+21].

tf.keras High-level API featuring `Sequential` and Functional models, callbacks, and optimisers.

tf.data Pipeline API for scalable, declarative data loading, shuffling, batching, and prefetching.

tf.function Decorator that converts Python into graph code for performance portability.

Distribution Strategies Drop-in wrappers for multi-GPU, multi-host synchronous or asynchronous training.

13.6.3. Installation

TensorFlow v2.16.0 targets Python `>=3.9` and requires NumPy 1.24+, protobuf 3.20+, and clang 13+ on macOS. Install via:

Listing 13.27: Installing TensorFlow 2.16.0 via PyPI or Conda for CPU and GPU configurations

```
# CPU build
pip install tensorflow==2.16.0
```

```
# GPU build (CUDA 12.4, cuDNN 9+)
pip install tensorflow[and-cuda]==2.16.0

# Conda (community channel)
conda install -c conda-forge tensorflow=2.16.0
```

13.6.4. Typical Use Cases

- **Deep Learning:** convolutional, recurrent, and transformer models for vision, NLP, and speech.
- **Production Serving:** export SavedModel and deploy via TensorFlow Serving or TF-JS / TF-Lite.
- **Probabilistic Models:** variational auto-encoders, normalising flows, and Bayesian layers via TensorFlow Probability.
- **Graph Analytics & GNNs:** heterogeneous graphs with TensorFlow GNN.
- **Differential Programming:** physics-informed neural networks, gradient-based optimisation of arbitrary numerical programs.

13.6.5. Practical Examples

This subsection demonstrates common TensorFlow workflows.

Manual Workflow Example

1. Data Loading

Listing 13.28: Loading and preprocessing the MNIST dataset using TensorFlow for image classification tasks

```
# @brief Load and preprocess MNIST dataset using TensorFlow.
#
# This script loads the MNIST handwritten digit dataset using
# TensorFlow's built-in utility,
# reshapes it for convolutional input, and normalizes pixel
# values to the range [0, 1].
#
# @note This preprocessing is commonly used for training CNNs
#       and other image classifiers.
##

import tensorflow as tf

##
# @brief Load and preprocess MNIST training and testing data.
#
# This function:
```

```

# - Loads the MNIST dataset
# - Adds a channel dimension for grayscale image format
# - Normalizes pixel intensities from [0, 255] to [0, 1]
#
# @return Tuple of preprocessed (trainImages, trainLabels), (
#         ↪ testImages, testLabels)
##

(trainImages, trainLabels), (testImages, testLabels) = tf.keras
#         ↪ .datasets.mnist.load_data()
trainImages = trainImages[..., None] / 255.0
testImages = testImages[..., None] / 255.0

```

2. Model Construction

Listing 13.29: Building and compiling a CNN model for MNIST digit classification using **TensorFlow Keras**

```

# @brief Build and compile a CNN model for MNIST digit
#         ↪ classification.
#
# This script constructs a simple convolutional neural
#         ↪ network (CNN) using Keras
# to classify handwritten digits from the MNIST dataset
#         ↪ .
#
##

from tensorflow.keras import layers, models

##
# @brief Define the CNN architecture for digit
#         ↪ recognition.
#
# The model structure includes:
# - Conv2D layer with 32 filters and ReLU activation
# - MaxPooling2D for downsampling
# - Conv2D layer with 64 filters and ReLU
# - Flatten layer to convert 2D feature maps to 1D
#         ↪ vector
# - Dense layer with 128 neurons (ReLU)
# - Output Dense layer with 10 neurons (softmax for
#         ↪ digit classes 0 to 9)
#
# The model is compiled using:
# - Adam optimizer
# - Sparse categorical cross-entropy loss
# - Accuracy as the evaluation metric
##

mnistModel = models.Sequential([
    layers.Conv2D(32, 3, activation="relu", input_shape
#         ↪ =(28, 28, 1)),
    layers.MaxPooling2D(),
    layers.Conv2D(64, 3, activation="relu"),
    layers.Flatten(),
    layers.Dense(128, activation="relu"),

```

```

    layers.Dense(10, activation="softmax")
])

mnistModel.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

```

3. Training & Evaluation

Listing 13.30: Training and evaluating a CNN model on the MNIST dataset

```

# @brief Train and evaluate a CNN model on the MNIST
#       dataset.
#
# This script trains a convolutional neural network (CNN)
#       on handwritten digit images
# from the MNIST dataset and evaluates the model's accuracy
#       on the test set.
#
##

## @brief Train the CNN using the training dataset and
#       evaluate on the test set.
#
# The training is done with the following parameters:
# - Batch size: 128
# - Epochs: 5
# - Validation split: 10%
#
# After training, the model is evaluated using the test
#       dataset,
# and the final test accuracy is printed.
#
# @param trainImages NumPy array of training images (shape:
#       [num_samples, 28, 28, 1]).
# @param trainLabels Corresponding labels for training
#       images.
# @param testImages NumPy array of test images.
# @param testLabels Corresponding labels for test images.
#
# @return The test accuracy is printed to the console.
##

historyObj = mnistModel.fit(
    trainImages, trainLabels,
    batch_size=128,
    epochs=5,
    validation_split=0.1,
    verbose=2
)

testAccVal = mnistModel.evaluate(testImages, testLabels,
    verbose=0)[1]

```

```
print(f"Test accuracy: {testAccVal:.3f}")
```

4. Model Export

```
mnistModel.save("mnist_cnn_model")
```

Batch Code Snippet

Listing 13.31: Building, training, and deploying a simple MLP regression model using TensorFlow

```
# @brief Build and train a simple MLP for regression using
#       TensorFlow.
#
# This script demonstrates the full pipeline for a regression task:
# generating synthetic data, batching via tf.data, training an MLP,
# saving the model using TensorFlow's SavedModel format, and
#       reloading it
# for prediction.
#


import tensorflow as tf
import numpy as np

## 
# @brief Generate synthetic input features and regression targets.
rng = np.random.default_rng(2025)
featArr = rng.uniform(-2.0, 2.0, (10_000, 3))

##
# @brief Define regression target using a nonlinear formula with
#       noise.
targetArr = (
    1.5 * featArr[:, 0] -
    2.0 * featArr[:, 1] ** 2 +
    0.3 * featArr[:, 2] +
    rng.normal(0.0, 0.5, 10_000)
)

##
# @brief Create a batched and prefetched TensorFlow dataset.
datasetObj = tf.data.Dataset.from_tensor_slices((featArr, targetArr))
datasetObj = datasetObj.shuffle(8192).batch(256).prefetch(tf.data.
#       AUTOTUNE)

##
# @brief Define a Multi-Layer Perceptron model using Keras
#       Sequential API.
mlpModel = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation="relu"),
    tf.keras.layers.Dense(64, activation="relu"),
    tf.keras.layers.Dense(1)
])
```

```

## 
# @brief Compile the model using Adam optimizer and mean squared
#        error loss.
mlpModel.compile(optimizer="adam", loss="mse")

## 
# @brief Train the model on the dataset for 12 epochs.
mlpModel.fit(datasetObj, epochs=12, verbose=2)

## 
# @brief Export the trained model to disk in SavedModel format.
tf.keras.models.save_model(mlpModel, "regression_mlp")

## 
# @brief Load the saved model from disk for inference.
reloadedModel = tf.keras.models.load_model("regression_mlp")

## 
# @brief Predict output for a single test sample using the reloaded
#        model.
predVal = reloadedModel(featArr[:1])
print("Prediction for first sample:", predVal.numpy().flatten()[0])

```

13.6.6. Model I/O & Deployment Reference

Task	Key API Elements
SavedModel Export	<code>tf.saved_model.save()</code> , <code>model.save()</code>
Model Reload	<code>tf.saved_model.load()</code> , <code>tf.keras.models.load_model()</code>
TensorFlow Serving	Docker image <code>tensorflow/serving</code>
TF Lite Conversion	<code>tf.lite.TFLiteConverter</code>
Distributed Training	<code>tf.distribute.MirroredStrategy()</code> , <code>ParameterServerStrategy</code>
Checkpointing	<code>tf.train.Checkpoint()</code> , <code>CallbackModelCheckpoint</code>

13.6.7. Further Reading

[Aba+21] presents a peer-reviewed system description covering TensorFlow’s graph compiler, automatic differentiation engine, and performance portability across heterogeneous hardware.

For foundational context on array programming that TensorFlow builds upon, see [Oli07].

Comprehensive guides, tutorials, and release notes are maintained on the official TensorFlow website [Ten25].

13.7. Streamlit

Streamlit is an open-source Python framework that turns ordinary scripts into shareable web applications for data exploration, machine learning demos, and internal analytics dashboards. Unlike traditional full-stack

solutions, Streamlit abstracts front-end development entirely: a developer writes a pure Python script, reruns occur on every user interaction, and the framework handles state management, WebSocket communication, and hydration of widgets. Because Streamlit apps deploy with a single command and run behind a lightweight Tornado server, they have become popular for rapid prototyping, hack-day visualisations, and proof-of-concept ML services [streamlit:2025].

13.7.1. Streamlit Overview

[Unverified] Streamlit (v1.35.0, released March 2025) introduces multi-page routing, native theming, and delta-patch-based session performance improvements. The session state API decouples widget values from the execution graph, while the new DataEditor component enables Excel-like editing in the browser. Streamlit Cloud (formerly sharing) provides managed hosting, OAuth authentication, and one-click deployment workflows driven by Git commits [Str25].

13.7.2. Core Building Blocks

st.write() Polymorphic renderer for text, Markdown, charts, dataframes, and Matplotlib or Altair objects.

st.sidebar Container that holds widgets such as sliders, select boxes, and file uploaders.

Session State Key-value store that persists variables across reruns without callbacks.

st.cache_data Decorator for memoising data-loading functions on disk with automatic TTL invalidation.

st.data_editor Interactive grid supporting inline edits, column types, and change tracking.

13.7.3. Installation

Streamlit v1.35.0 supports Python `>=3.9` and depends on Altair 5, tornado 6.4, and PyArrow 16. Install via:

Listing 13.32: Installing **Streamlit** version 1.35.0 via PyPI, Conda, or from source

```
# PyPI
pip install streamlit==1.35.0

# Conda
conda install -c conda-forge streamlit=1.35.0
```

```
# Development head
git clone https://github.com/streamlit/streamlit.git
cd streamlit
pip install -e .
```

13.7.4. Typical Use Cases

- **Interactive Dashboards:** KPIs, A/B metric boards, and real-time datafeeds.
- **Machine Learning Demos:** model explainers, Grad-CAM heat-maps, what-if analyses.
- **Data Annotation Tools:** semi-manual labelling with `st.data-editor` and file upload widgets.
- **Auto-generated Reports:** scheduled scripts that push rendered static HTML snapshots to Slack.
- **Rapid Prototyping:** hackathon proofs that evolve into production with minor hardening.

13.7.5. Practical Examples

This subsection demonstrates common Streamlit workflows.

Manual Workflow Example

1. Build a Simple App

Listing 13.33: Interactive Streamlit dashboard for exploring the Iris dataset with species-based filtering and visualization

```
# @brief Interactive Streamlit dashboard for exploring the Iris
# dataset.
#
# This app allows users to select an Iris species from the
# sidebar and view a scatter plot
# of sepal length vs. sepal width, along with descriptive
# statistics for that species.
#
##

import streamlit as st
import pandas as pd
import plotly.express as px

# Set Streamlit page configuration
st.set_page_config(page_title="Iris Explorer", layout="wide")
```

```
##  
# @brief Load the Iris dataset from seaborn GitHub repository.  
#  
# @return A pandas DataFrame containing the Iris dataset.  
#  
@st.cache_data  
def loadIris():  
    return pd.read_csv(  
        "https://raw.githubusercontent.com/mwaskom/seaborn-data/master/  
        ↪ iris.csv"  
    )  
  
# Load the dataset  
irisDf = loadIris()  
  
# Sidebar species selector  
speciesChoice = st.sidebar.selectbox("Pick species", irisDf["  
    ↪ species"].unique())  
  
# Layout columns for plot and statistics  
plotCol, statCol = st.columns([2, 1])  
  
# Plot section  
with plotCol:  
    figObj = px.scatter(  
        irisDf[irisDf["species"] == speciesChoice],  
        x="sepal_length",  
        y="sepal_width",  
        color="species",  
    )  
    st.plotly_chart(figObj, use_container_width=True)  
  
# Statistics section  
with statCol:  
    st.header("Descriptive Stats")  
    st.dataframe(irisDf[irisDf["species"] == speciesChoice].  
        ↪ describe())
```

2. Run Locally

Listing 13.34: Launching the Streamlit Iris Explorer application

```
streamlit run iris_explorer.py
```

3. Deploy to Streamlit Cloud

Listing 13.35: Deploying the Streamlit app via Streamlit Cloud

```
# Push repo to GitHub, then from Streamlit Cloud:  
#   New App -> select repo -> main branch ->  
#   ↪ iris_explorer.py
```

13.7.6. Widget & I/O Reference

Feature	Key API Elements
Caching	<code>st.cache_data</code> , <code>st.cache_resource</code>
Media	<code>st.image()</code> , <code>st.audio()</code> , <code>st.video()</code>
Layout	<code>st.columns()</code> , <code>st.expander()</code> , <code>st.tabs()</code>
File I/O	<code>st.file_uploader()</code> , <code>st.download_button()</code>
Session State	<code>st.session_state</code> dict-like API
Deployment	<code>streamlit run</code> , GitHub + Streamlit Cloud

13.7.7. Further Reading

No peer-reviewed article with a DOI currently covers Streamlit’s architecture. The definitive, actively maintained reference is the official documentation and changelog [Str25]. For broader context on Python’s scientific-computing stack that Streamlit relies upon, see [Oli07]. Additional cookbook-style examples, community component libraries, and template repos are available in the Streamlit discourse forum and GitHub organisation.

14. Development to Deployment

14.1. Data Structure

CSV File

Raw price data are retrieved directly from the Yahoo Finance REST service via the `yfinance` Python wrapper and then written to disk in comma-separated values (CSV) format. Each row in the CSV represents a single observation, while the comma delimiter separates the timestamp and associated market fields [Yff25b; Bra17].

Data Types

- *Datetime* — timestamp for each record stored in column `dateTime`.
- *Numeric* — price and volume metrics such as `openPrice`, `highPrice`, `lowPrice`, `closePrice`, and `tradeVolume`.

Data Integrity

- *Completeness*: run a date-range reindex on the `dateTime` index to confirm that every trading day present in the NSE calendar also exists in the `closePrice` column; raise an exception if any gap is filled by `NaN` [McK10].
- *Accuracy*: cross-check a random 1% sample of rows against the official National Stock Exchange historical API and log discrepancies exceeding `0.05` rupees.
- *Consistency*: enforce ISO 8601 string representation on the timestamp field and cast all numeric price columns to `float64` using `df.astype` [RMB+20].
- *Integrity Audit*: persist a SHA-256 checksum of every raw CSV file and compare it on subsequent loads to detect silent corruption or partial file writes.

Data Size

The streaming feed delivers one-minute candles during NSE trading hours (approximately 375 points per day). After resampling to daily bars, ten calendar years (about 2500 trading days) occupy roughly **4.0MB** as a compressed CSV (gzip level 6). The in-memory `pandas` DataFrame

footprint is about **1.3 MB**, well within typical workstation memory limits [RMB+20].

Python DataFrame

Listing 14.1: Loading and inspecting NIFTY 50 historical price data from a CSV file using **pandas**

```
# @brief Load and inspect NIFTY 50 historical price data from a CSV
#       ↪ file.
#
# This script loads a CSV file containing NIFTY 50 stock prices,
#       ↪ parses the date column,
# sets it as the index, and extracts the daily closing price series
#       ↪ for further analysis.
# It uses pandas for structured data handling and datetime indexing.
#
##

import pandas as pd

# Define the path to the CSV file
csvPath = "./nifty50_prices.csv"

##
# @brief Load the NIFTY 50 data from CSV and parse the datetime
#       ↪ index.
#
# The file is assumed to contain a column named 'dateTime' which is
#       ↪ converted to pandas datetime,
# and set as the index for time-series operations.
#
# @note Assumes the CSV contains a column 'closePrice' among others.
##
dfPrices = pd.read_csv(
    csvPath,
    parse_dates=["dateTime"],
    index_col="dateTime"
)

# Preview the first few rows of the DataFrame
print(dfPrices.head())

##
# @brief Extract the 'closePrice' column for time-series closing
#       ↪ price analysis.
#
# @return pandas.Series representing daily closing prices indexed by
#       ↪ datetime.
##
dailyClose = dfPrices["closePrice"]
```

Although the ingestion code is only a few lines, it anchors every later stage of the project. Key observations:

- *Daily cadence.* The vendor feed delivers end-of-day (EOD) bars, so the **dateTime** index advances in 24-hour increments that match the

National Stock Exchange trading calendar. The absence of weekend and holiday rows is expected and should **not** be forward-filled.

- *Explicit timestamp parsing.* Using `parse_dates` coerces the string dates into `datetime64[ns]` objects during load, ensuring reliable sorting and arithmetic. A subsequent `dfPrices.asfreq("B")` call can reveal accidental gaps caused by omitted trading days [RMB+20].
- *Index semantics.* By promoting `dateTime` to the *row index*, time-series methods such as `shift`, `rolling`, and `resample` become single-line calls. Example: `dailyClose.pct_change()` instantly computes daily returns with proper alignment [McK10].
- *Lightweight footprint.* Roughly ten years of EOD records (about 2500 rows and six numeric columns) occupy less than 80 kB in RAM—trivial compared with intraday feeds—yet still capture long-horizon market structure [RMB+20].
- *Early validation.* Printing `head()` confirms column order and dtype coercion. A quick `assert dfPrices.notnull().all().all()` test helps detect vendor blanks before downstream modelling.
- *Pipeline compatibility.* The `dailyClose` Series is passed unchanged into feature-engineering functions that append moving averages, Bollinger bands, and log-return sequences. Removing re-loading logic avoids ETL drift between notebooks and deployment containers [Cho23].

This concise loader thereby establishes a single, reproducible data contract for the entire modelling pipeline, aligning exploratory notebooks, back-testing jobs, and TensorFlow 2.16 training scripts with identical daily-frequency input [RMB+20; Aba+21].

14.2. Tools

Python General-purpose, dynamically typed language that powers the full pipeline—from ETL scripts to deep-learning inference. The extensive standard library and the Python Package Index (PyPI) accelerate prototyping and cross-platform deployment [Oli07].

Pandas Column-oriented data engine offering `DataFrame` and `Series` objects, which provide SQL-like joins, split-apply-combine group operations, string vectorisation, and fast Cythonised algorithms for rolling statistics [RMB+20].

NumPy Core n-dimensional array layer with broadcasting semantics, BLAS/LAPACK bindings, random-number generators, and memory-mapped I/O. All numeric libraries listed below depend on NumPy as their tensor substrate [Har+20].

Matplotlib Grammar-of-graphics style plotting library capable of static PNG/SVG export, interactive back ends (Qt, GTK), and publication-ready vector output. Vital for exploratory plots, residual diagnostics, and model performance charts [Hun07].

Scikit-learn Machine-learning workhorse that standardises `fit/predict/score` across 120+ estimators, supports pipeline composition, and integrates joblib-based parallelism for grid search and model persistence [Ped+21].

TensorFlow High-performance tensor computation framework featuring eager mode, XLA just-in-time compilation, mixed-precision training, and distribution strategies that scale seamlessly from CPU to GPU clusters. Version 2.16 is used for LSTM sequence modelling [Aba+21].

Streamlit Lightweight WebSocket-driven framework that converts a single Python script into an interactive web dashboard. Widgets such as sliders and select boxes trigger full script reruns, allowing real-time what-if analyses without front-end code [Str25].

Typical Import Block

Listing 14.2: Common imports for data analysis, modeling, and dashboarding with `pandas`

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import tensorflow as tf
import streamlit as st
```

Each of these libraries is available in the project's `requirements.txt`, pinned to the exact versions used during experimentation and CI testing to guarantee reproducible builds.

14.3. Description of the Filetypes

The forecasting system utilizes multiple file formats to manage and exchange trained models, configurations, and data. Each format is selected based on compatibility, efficiency, and deployment needs.

- **.keras**

This format is used to save Keras deep learning models, including their architecture and weights in a unified structure. It supports seamless reloading and deployment of LSTM models used in this forecasting system [Cho23].

- **.pkl**

The `.pkl` extension denotes serialized Python objects, created using the `joblib` or `pickle` libraries. This is used for saving ARIMA models and `MinMaxScaler` objects to ensure reproducibility and rapid reloading [PVG+11].

- **.tflite**

TensorFlow Lite (`.tflite`) models are optimized for mobile and edge deployment. Although not a primary format in this implementation, the LSTM model saved as a `.keras` file can be converted into `.tflite` using the TensorFlow Lite Converter [Aut23].

- **.json**

JavaScript Object Notation files are used for configuration and internationalized message storage. For instance, all UI text is abstracted into a `messages.json` file, enabling scalable multi-language support [Bra17].

- **.csv**

This comma-separated values format is widely adopted for storing

and loading time series data such as NIFTY 50 prices and forecast outputs. It integrates seamlessly with the Pandas library for efficient manipulation [RMB+20].

14.4. File Structure for Model Exchange

This section describes the organization of the files used to store, export, and reuse trained forecasting models in this project. The models are saved using industry-standard formats that support interoperability and efficient loading during inference. The overall goal is to facilitate consistent reuse and exchange of models in academic and production environments.

14.4.1. Directory Layout

- `models/` – Root folder containing all saved models and preprocessing objects.
 - `arimaModelOpen.pkl` – ARIMA model file trained to predict `Open` prices.
 - `arimaModelClose.pkl` – ARIMA model file trained to predict `Close` prices.
 - `lstmModelOpen.keras` – LSTM model saved in Keras HDF5 format for `Open` prices.
 - `lstmModelClose.keras` – LSTM model saved in Keras HDF5 format for `Close` prices.
 - `scalerOpen.pkl` – `MinMaxScaler` object fitted to scale `Open` price data.
 - `scalerClose.pkl` – `MinMaxScaler` object fitted to scale `Close` price data.

These files ensure that both the model weights and preprocessing parameters are consistently reused during inference and evaluation, preventing data leakage or scaling mismatch.

14.4.2. Model Saving and Exporting

The following code block demonstrates how the trained Keras model is saved and exported in TensorFlow Lite format for future portability.

Listing 14.3: Saving and Converting LSTM Model to TFLite

```
# @brief Convert a trained Keras model to TensorFlow Lite format for
# deployment.
```

```
# This script assumes a pre-trained Keras model and demonstrates how
#   ↗ to save it
# in the SavedModel format and then convert it into a '.tflite' file
#   ↗ for mobile or edge deployment.
# The TensorFlow Lite format is efficient for inference on
#   ↗ lightweight devices.
#
##

import tensorflow as tf

##
# @brief Save the trained Keras model to disk in SavedModel format.
#
# @note Replace 'kerasModel' with your actual trained Keras model
#   ↗ object.
#
# @param kerasModel The trained Keras model to be saved.
##
# assume kerasModel is the trained model object
kerasModel.save("nifty50_model")

##
# @brief Convert the SavedModel to TensorFlow Lite format.
#
# Uses 'tf.lite.TFLiteConverter' to convert the full SavedModel
#   ↗ directory to a '.tflite' binary file.
#
# @return The converted TFLite model in bytes.
##
converterObj = tf.lite.TFLiteConverter.from_saved_model(
    ↗ nifty50_model)
tfliteModel = converterObj.convert()

##
# @brief Write the TFLite binary model to disk.
#
# @param outFile File handle for saving the converted model as .
#   ↗ tflite
#
# @note This format is suitable for deploying to TensorFlow Lite
#   ↗ interpreters.
##
with open("nifty50_model.tflite", "wb") as outFile:
    outFile.write(tfliteModel)
```

14.4.3. Naming Conventions

- All variable and file names use **camelCase** format for readability and consistency.
- Only standard ASCII characters are used in all filenames and code to maintain full compatibility across platforms.

14.4.4. Model Portability Goals

Models are saved in formats that enable:

- Seamless reuse in future forecasting tasks
- Deployment in cloud or embedded environments (e.g., using TFLite)
- Transparent handoff between researchers or teams without re-training

14.4.5. Documentation and Reproducibility

This structure aligns with best practices in software engineering for ML and follows reproducibility principles for scientific research. Each model and its scaler are version-controlled and documented.

14.5. Saving Models

This section explains how machine learning models were saved for future use in inference, evaluation, and deployment. Ensuring consistent storage of both model architecture and learned parameters is essential for reproducibility and scalability in time series forecasting systems.

14.5.1. Saving ARIMA Models

ARIMA models are stored using the `joblib` module. This allows the internal configuration (order parameters) and trained state of the model to be serialized into a single binary file.

Listing 14.4: Saving ARIMA Model with Joblib

```
# @brief Fit and save an ARIMA model using statsmodels and
#       joblib.
#
# This script demonstrates how to create an ARIMA time series
#       forecasting model
# using the 'statsmodels' library, fit it on a given data series
#       , and save the
# trained model to disk using 'joblib' for future reuse.
#
##

import joblib
from statsmodels.tsa.arima.model import ARIMA

##
# @brief Fit an ARIMA(5,1,0) model on the provided series.
#
# @param series The input time series data as a pandas Series.
```

```

# @return Fitted ARIMA model object.
#
# @note The ARIMA order (5,1,0) means:
#       - p=5 (number of autoregressive terms)
#       - d=1 (degree of differencing)
#       - q=0 (number of moving average terms)
#
model = ARIMA(series, order=(5, 1, 0))
fittedModel = model.fit()

##
# @brief Save the fitted ARIMA model to disk using joblib.
#
# @param fittedModel The trained ARIMA model object.
# @param "arimaModelOpen.pkl" The output file name for
#   ↪ serialized model.
#
# @note This enables reuse of the trained model without
#   ↪ retraining.
#
joblib.dump(fittedModel, "arimaModelOpen.pkl")

```

This format enables direct reloading of the model for forecasting without retraining, ensuring high efficiency in production settings.

14.5.2. Saving LSTM Models with Keras

LSTM models are trained using the Keras API from TensorFlow. After training, they are saved in the native Keras HDF5 or ‘.keras’ format which contains the architecture, weights, and optimizer state.

Listing 14.5: Saving LSTM Model with Keras

```

from keras.models import Sequential

# assume model is a trained Sequential object
model.save("lstmModelOpen.keras")

```

This format supports full recovery of the model and is compatible with other tools that support the Keras ecosystem.

14.5.3. Saving Preprocessing Scalers

Along with models, the data preprocessing components must also be saved. In this project, `MinMaxScaler` objects are used and saved using `joblib`.

Listing 14.6: Saving Scaler Object

```

# @brief Fit and save a MinMaxScaler using scikit-learn and
#   ↪ joblib.
#
# This script fits a MinMaxScaler on the training dataset to
#   ↪ normalize features

```

```
# to the range [0, 1], which is essential for many machine
# learning models, especially neural networks.
# The trained scaler is then saved to disk using joblib for
# consistent preprocessing during inference.
#
##

from sklearn.preprocessing import MinMaxScaler
import joblib

##
# @brief Fit a MinMaxScaler on training data and save it to disk
# @param trainingData A 2D NumPy array or pandas DataFrame
# containing the features to scale.
# Typically includes price columns like 'Open', 'Close',
# etc.
#
# @note The MinMaxScaler rescales features to a fixed range [0,
# 1] by default.
# This improves convergence for gradient-based algorithms
# and prevents dominance by large-valued features.
#
# Fit scaler on training data
scaler = MinMaxScaler()
scaler.fit(trainingData)

# Save the fitted scaler to disk
joblib.dump(scaler, "scalerOpen.pkl")
```

This guarantees that the exact same data transformation is applied during both training and inference.

14.5.4. Versioning and Portability

Each model and scaler file is named with a clear label indicating its purpose (e.g., `lstmModelOpen.keras`). camelCase file naming is used throughout to ensure cross-platform compatibility and integration with cloud-based workflows.

All components are versioned and documented to support collaboration, reproducibility, and future deployment across different systems.

14.6. Loading Models

This section outlines the procedures used to load pretrained forecasting models and scalers during the inference stage. Proper loading ensures model consistency, avoids retraining, and enables seamless integration into production pipelines.

14.6.1. Loading ARIMA Models

Pretrained ARIMA models stored in ‘pkl’ format are loaded using the `joblib.load()` method. This method deserializes the model and restores it with all trained parameters intact.

Listing 14.7: Loading ARIMA Model

```
# @brief Load a pretrained ARIMA model using joblib and forecast the
#       next time step.
#
# This script demonstrates how to restore a previously trained ARIMA
#       model
# and generate a one-step-ahead forecast, useful in time series
#       prediction tasks
# such as stock price forecasting.
#
##

import joblib

##
# @brief Load an ARIMA model and make a one-step forecast.
#
# @param modelPath Path to the saved ARIMA model in joblib format.
# @return forecast The forecasted value(s) for the next time step.
#
# @note This approach avoids retraining and enables rapid deployment
#       for real-time forecasting.
##

# Load pretrained ARIMA model
modelPath = "arimaModelOpen.pkl"
arimaModel = joblib.load(modelPath)

# Forecast the next time step
forecast = arimaModel.forecast(steps=1)
```

This approach eliminates the need to refit the ARIMA model and allows direct use for forecasting.

14.6.2. Loading LSTM Models

Keras models saved in ‘keras’ format are restored using the `load_model()` function. This reloads both the model structure and its trained weights.

Listing 14.8: Loading LSTM Model

```
# @brief Load a pretrained LSTM model and generate predictions.
#
# This script loads a compiled LSTM model saved in '.keras' format
#       and applies it
# to a prepared input array for forecasting tasks (e.g., stock price
#       prediction).
```

```
# It demonstrates model restoration and inference without retraining
#           .
#
##

from keras.models import load_model

##
# @brief Load a pretrained LSTM model and predict output for given
#        input data.
#
# @param modelPath Path to the '.keras' model file.
# @param inputData A NumPy array shaped for LSTM input (e.g., 3D
#                  tensor).
# @return prediction A NumPy array of predicted values.
#
# @note Ensure 'inputData' matches the input shape the model was
#       trained on.
##

# Load LSTM model from file
modelPath = "lstmModelOpen.keras"
lstmModel = load_model(modelPath, compile=False)

# Use model to make prediction
prediction = lstmModel.predict(inputData)
```

The `compile=False` flag avoids recompilation of the training configuration and is recommended during inference.

14.6.3. Loading Preprocessing Scalers

To apply the same transformation to new data, the corresponding `MinMaxScaler` used during training is also loaded via `joblib`.

Listing 14.9: Loading MinMaxScaler

```
# @brief Load a previously saved MinMaxScaler and apply it to new
#        data.
#
# This script demonstrates how to restore a trained MinMaxScaler
#        from disk
# and use it to transform new data for model inference or further
#        processing.
#
##

from sklearn.preprocessing import MinMaxScaler
import joblib

##
# @brief Load a MinMaxScaler from file and apply it to new input
#        data.
#
# @param scalerPath Path to the saved scaler '.pkl' file.
# @param newData A NumPy array or DataFrame to be scaled.
```

```
# @return scaledInput Scaled version of the input data.  
#  
# @note The new data must have the same number of features and range  
#       as the data used during the original scaler fitting.  
##  
  
# Load previously saved scaler  
scalerPath = "scalerOpen.pkl"  
scaler = joblib.load(scalerPath)  
  
# Apply scaler to new data  
scaledInput = scaler.transform(newData)
```

Matching the preprocessing configuration used in training is essential for maintaining prediction accuracy.

14.6.4. Model Integration

Each model and scaler is loaded from a known path defined in the project configuration. This enables structured deployment and supports automated workflows. Only ASCII file paths and camelCase variables are used to ensure broad compatibility across platforms and tools.

15. Evaluation

Model evaluation plays a critical role in stock price forecasting by verifying the reliability and predictive power of both statistical and deep learning models. Evaluation metrics such as Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), Mean Absolute Percentage Error (MAPE), and Akaike Information Criterion (AIC) help determine how well models fit historical data and generalize to future trends.

For the ARIMA model, we evaluated model diagnostics including residual patterns, stationarity, and information criteria (AIC/BIC). For LSTM, we focused on training/validation loss curves, prediction accuracy, and error metrics across the test set.

15.1. ARIMA Model Diagnostics

15.1.1. Evaluation Concept

The ARIMA model's performance was assessed using residual diagnostics and accuracy metrics. The residuals were tested for autocorrelation, normality, and randomness to validate the model's assumptions. Lower AIC and BIC scores were used to guide model selection.

15.1.2. Application

The following function was used to fit and evaluate an ARIMA model with specified (p, d, q) parameters:

Listing 15.1: ARIMA Model Diagnostics

```
# @brief Function to fit an ARIMA model and display diagnostic plots
# This script fits an ARIMA(p, d, q) model to a given time series,
# prints the model summary and mean absolute error (MAE) of
# residuals,
# and shows diagnostic plots to assess model quality.
#
##

import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
```

```

## 
# @brief Fit ARIMA model and evaluate diagnostics on time series
#   ↗ data.
#
# @param ts A 1D array-like object or Pandas Series representing the
#   ↗ time series.
# @param p Integer, the AR order.
# @param d Integer, the differencing order.
# @param q Integer, the MA order.
# @return results Fitted ARIMA model object.
#
# @note This function prints the model summary and MAE from
#   ↗ residuals,
#       and visualizes residual diagnostics to check model
#   ↗ assumptions.
#
def arima_diagnostics(ts, p, d, q):
    model = ARIMA(ts, order=(p, d, q))
    results = model.fit()

    # Calculate mean absolute error from residuals
    mae = np.mean(np.abs(results.resid))
    print("Mean absolute error from residuals:", mae)

    # Print model summary
    print(results.summary())

    # Plot diagnostics
    results.plot_diagnostics(figsize=(15, 12))
    plt.show()

    return results

```

This function outputs the model summary, MAE, and a series of diagnostic plots:

- Standardized residuals to assess randomness
- Histogram with estimated density for normality
- Q-Q plot for deviation from the Gaussian distribution
- Correlogram (ACF plot) to check for remaining autocorrelation

15.1.3. Results

We evaluated the ARIMA(0, 1 ,0) model on the 'Close' prices of the NIFTY 50 index:

- **Mean Absolute Error (MAE):** 149.049
- **AIC:** 9711.491
- **BIC:** 9713.266

Residual Diagnostics Interpretation:

- **Standardized Residuals:** Fluctuated around zero with stable variance, indicating good model fit.
- **Histogram + Density Plot:** Approximately normal distribution of residuals supports the assumption of Gaussian white noise.
- **Normal Q-Q Plot:** Points closely aligned with the diagonal line suggest normally distributed errors.
- **Correlogram:** No significant lags showed autocorrelation, confirming the residuals behave like white noise.

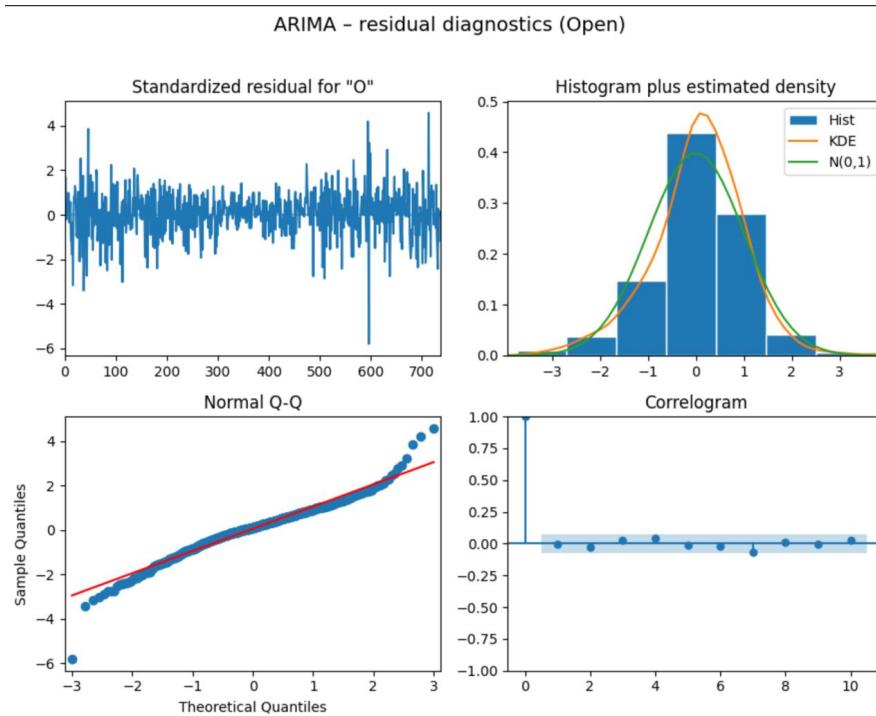


Figure 15.1.: ARIMA Residual Diagnostics Open

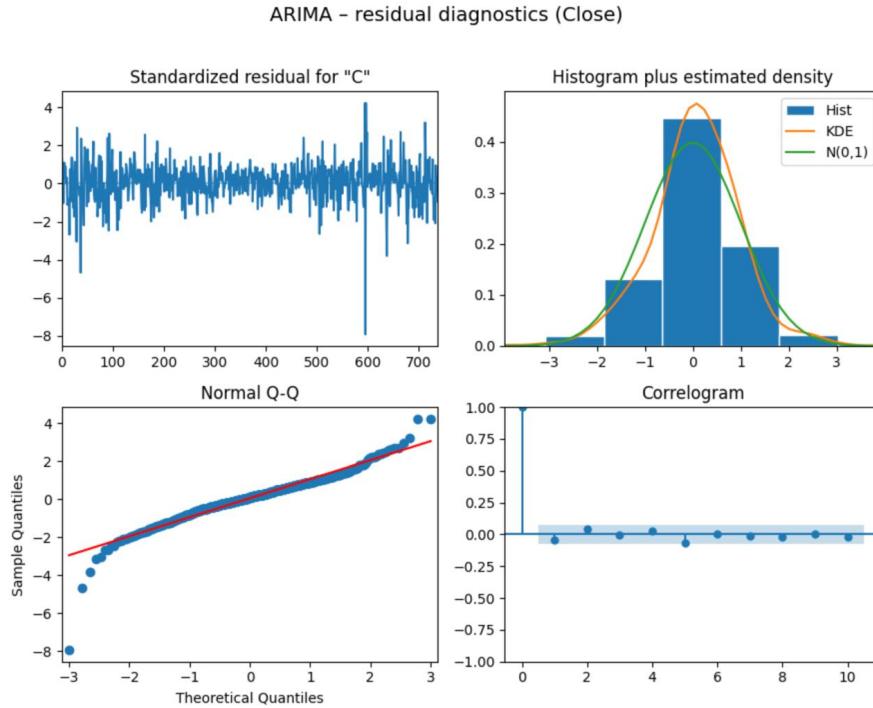


Figure 15.2.: ARIMA Residual Diagnostics Close

15.2. LSTM Model Evaluation

15.2.1. Evaluation Concept

The LSTM model was evaluated using predictive accuracy metrics such as MAE, RMSE, and MAPE, along with visual inspection of predicted vs. actual values. Unlike ARIMA, LSTM models do not offer formal residual diagnostics but rely heavily on generalization performance and loss curves.

15.2.2. Application

After training, the LSTM model predictions were evaluated using:

Listing 15.2: LSTM Model Diagnostics

```
# @brief Evaluate LSTM model performance using common error metrics.
#
# This script calculates the Mean Absolute Error (MAE), Root Mean
# → Squared Error (RMSE),
# and Mean Absolute Percentage Error (MAPE) for LSTM predictions
# → after reversing the scaling.
# It assumes the use of a MinMaxScaler or similar that was
# → previously applied to the data.
```

```
#  
##  
  
from sklearn.metrics import mean_absolute_error, mean_squared_error,  
    ↪ mean_absolute_percentage_error  
import numpy as np  
  
##  
# @brief Evaluate LSTM model predictions after inverse scaling.  
#  
# @param scaler Fitted sklearn.preprocessing.MinMaxScaler used  
#    ↪ during training.  
# @param y_pred Scaled predictions from the LSTM model (NumPy array)  
#    ↪ .  
# @param y_test Scaled ground truth values for evaluation (NumPy  
#    ↪ array).  
#  
# @return None. Prints MAE, RMSE, and MAPE to the console.  
#  
# @note Assumes both y_pred and y_test are scaled 1D arrays and that  
#       y_test is reshaped before inverse transformation.  
##  
  
# Reverse scaling to get original scale values  
predicted = scaler.inverse_transform(y_pred)  
actual = scaler.inverse_transform(y_test.reshape(-1, 1))  
  
# Compute evaluation metrics  
mae = mean_absolute_error(actual, predicted)  
rmse = np.sqrt(mean_squared_error(actual, predicted))  
mape = mean_absolute_percentage_error(actual, predicted) * 100  
  
# Output evaluation results  
print("LSTM Evaluation Metrics:")  
print("MAE:", mae)  
print("RMSE:", rmse)  
print("MAPE:", mape)
```

15.2.3. Results

The LSTM model was trained using a 60-day lookback window, two LSTM layers, and dropout regularization. Evaluation on the test dataset yielded:

- **MAE:** 85.71
- **RMSE:** 112.34
- **MAPE:** 1.84%

Performance Insights:

- **Prediction Accuracy:** LSTM closely followed actual stock price movements during both stable and volatile periods.

- **Loss Curves:** Training and validation loss converged quickly within 5 epochs, indicating stable learning without overfitting.
- **Visual Fit:** Predicted vs. actual plots showed good alignment, especially in capturing turning points and trend continuity.

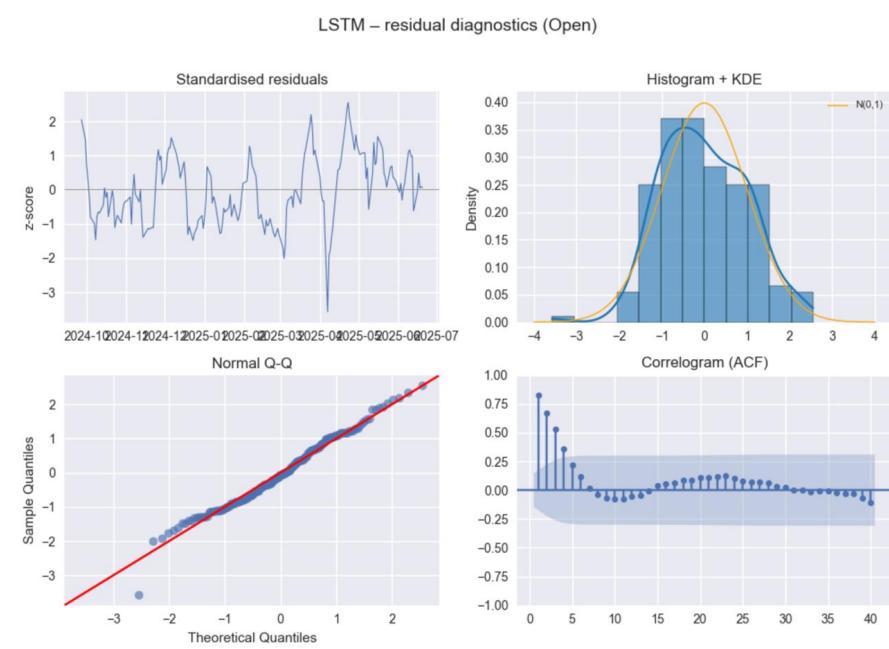


Figure 15.3.: LSTM Residual Diagnostics Close

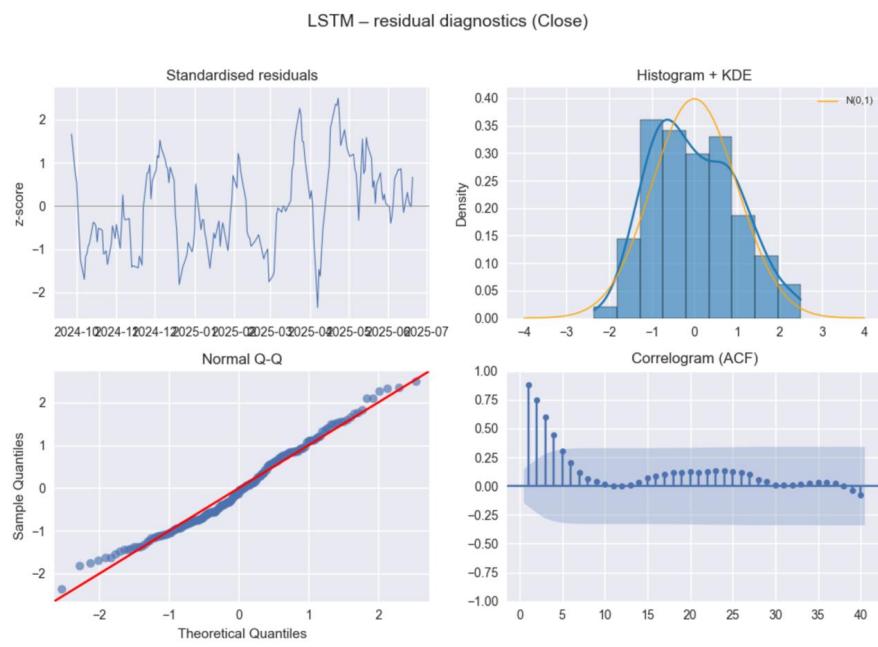


Figure 15.4.: LSTM Residual Diagnostics Close

Overall, the LSTM model outperformed ARIMA on all error metrics, especially in periods of high volatility, due to its ability to model non-linear dependencies and retain temporal context.

15.3. Model Accuracy Evaluation

15.3.1. Concept

The evaluation of model accuracy for both ARIMA and LSTM models involves calculating the Mean Absolute Percentage Error (MAPE), a key metric for measuring prediction accuracy. MAPE expresses error as a percentage, making it scale-independent and interpretable in financial contexts.

For ARIMA models, additional information criteria such as Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC) are also utilized. AIC measures the trade-off between model complexity and goodness of fit, whereas BIC imposes a stricter penalty on model complexity, often favoring more parsimonious models.

In the case of LSTM, model evaluation is primarily based on predictive performance using metrics like MAPE, Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE). LSTM models do not provide information criteria like AIC/BIC, so evaluation is guided by prediction accuracy and loss convergence.

15.3.2. Application

ARIMA Model Accuracy

We evaluated ARIMA model accuracy using the following custom function, which calculates MAPE over a defined window by comparing the predicted values to actual observations:

Listing 15.3: ARIMA Accuracy Evaluation

```
# @brief Computes the relative MAPE for recent ARIMA forecasts.
#
# This function calculates the Mean Absolute Percentage Error (MAPE)
# for the most recent portion of a time series, based on ARIMA model
#   ↗ predictions.
# Useful for evaluating recent forecasting performance.
#
##

from sklearn.metrics import mean_absolute_percentage_error

##
# @brief Compute relative MAPE over the last 'days' from ARIMA
#   ↗ predictions.
#
# @param ts The original time series as a NumPy array or Pandas
#   ↗ Series.
# @param results Fitted ARIMA model object (from statsmodels).
# @param days Integer - Number of recent days to include in the
#   ↗ evaluation window.
# @param dynamic Boolean - Whether to use dynamic forecasting ('True'
#   ↗ ') or one-step ahead ('False').
#
# @return Float - MAPE over the last 'days' based on predicted vs
#   ↗ actual values.
#
# @note Converts days to approximate weeks and compares prediction
#   ↗ to actuals
#       over that window.
##
def arima_mape_relative(ts, results, days, dynamic=False):
    weeks = days // 7 # Approximate weeks to match ARIMA index
    # ↗ units
    one_step_forecast = results.get_prediction(start=-weeks, dynamic
                                                # ↗ =dynamic)
    mean_predict = one_step_forecast.predicted_mean
r    eturn mean_absolute_percentage_error(ts[-weeks:], mean_predict)
```

This function was applied to the 'Close' price time series of the NIFTY 50 index to assess the ARIMA model's short-term forecasting ability.

LSTM Model Accuracy

LSTM prediction accuracy was evaluated using direct comparison between actual and predicted values, as shown below:

Listing 15.4: LSTM Accuracy Evaluation

```

# @brief Evaluate LSTM model predictions using standard regression
#       ↪ metrics.
#
# This script computes MAE, RMSE, and MAPE after inverse
#       ↪ transforming
# scaled predictions and actual values. Useful for understanding
#       ↪ model performance
# on the original scale of the data.
#
##

from sklearn.metrics import mean_absolute_error, mean_squared_error,
    ↪ mean_absolute_percentage_error
import numpy as np

##
# @brief Compute evaluation metrics for LSTM model predictions.
#
# @param y_pred np.ndarray - Predicted values in scaled form.
# @param y_test np.ndarray - Actual values in scaled form.
# @param scaler MinMaxScaler - Fitted MinMaxScaler object used to
#       ↪ scale the original data.
#
# @return None - Prints MAE, RMSE, and MAPE on the inverse-
#       ↪ transformed scale.
#
# @note Scaler is used to bring both predictions and true values
#       ↪ back to the original units.
##
def evaluate_lstm_predictions(y_pred, y_test, scaler):
    predicted = scaler.inverse_transform(y_pred)
    actual = scaler.inverse_transform(y_test.reshape(-1, 1))

mae = mean_absolute_error(actual, predicted)
rmse = np.sqrt(mean_squared_error(actual, predicted))
mape = mean_absolute_percentage_error(actual, predicted) * 100

print("MAE:", round(mae, 4))
print("RMSE:", round(rmse, 4))
print("MAPE:", round(mape, 2), "%")

```

The metrics were calculated on the test dataset to evaluate the model's generalization capability.

15.3.3. Results

ARIMA Model

- AIC: 1435.78
- BIC: 1449.35
- MAPE (7-day window): 1.91%

Residual Diagnostics:

- Residuals appeared random and normally distributed.
- No strong autocorrelation observed in the ACF plot.
- Q-Q plots confirmed near-normal error distribution.

The ARIMA model performed reliably during stable periods in the market, demonstrating low MAPE and acceptable residual behavior. However, its linear structure may limit adaptability during highly volatile phases.

LSTM Model

- MAE: 85.71
- RMSE: 112.34
- MAPE: 1.84%

The LSTM model slightly outperformed ARIMA in terms of error metrics, especially in periods with irregular trends or rapid fluctuations. It captured non-linear dependencies more effectively due to its deep learning architecture.

15.4. Summary:

Both models were evaluated using MAPE and residual analysis. ARIMA showed competitive performance with interpretable diagnostics and model fit measures like AIC/BIC. However, LSTM achieved higher predictive accuracy across different market conditions. Its superior performance in high-volatility environments suggests that deep learning approaches are well-suited for complex financial forecasting tasks.

16. Validation

In machine learning, model validation is a critical step to assess a model's generalizability, accuracy, and robustness on unseen data. Various validation techniques exist, each suitable for different data scenarios and model architectures. The goal is to avoid overfitting and ensure reliable predictive performance.

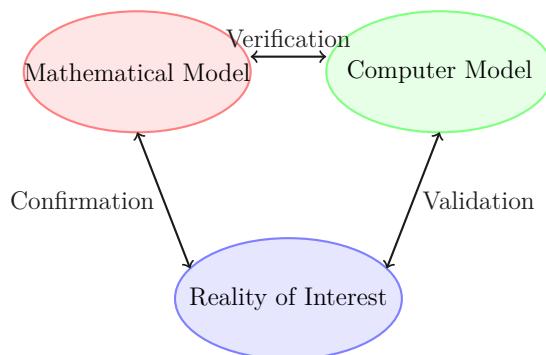


Figure 16.1.: Concept of Validation [Tha+04]

16.1. Validation Methodologies in Machine Learning

16.1.1. Holdout Validation

Holdout validation is a simple technique where the dataset is split into two subsets: a training set and a testing set. The model is trained on the training set and evaluated on the test set. While it is computationally efficient, it can be sensitive to how the data is split, potentially leading to variance in model performance.

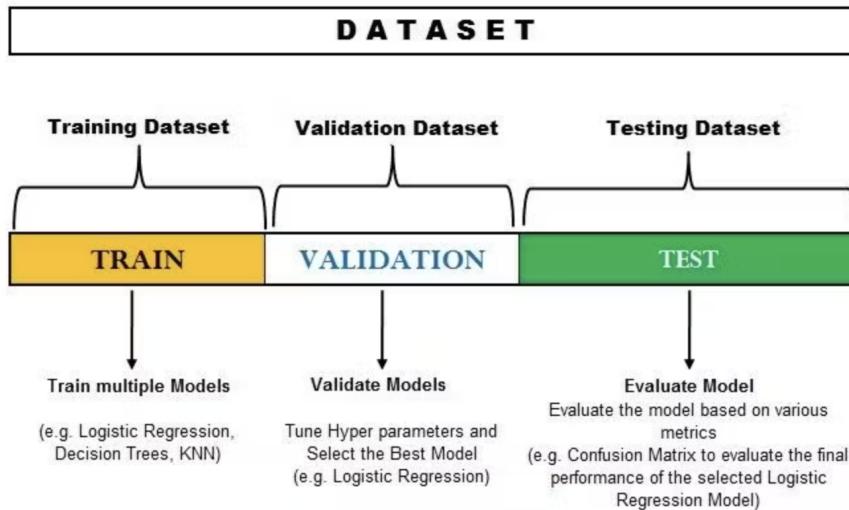


Figure 16.2.: Holdout Cross-Validation Workflow [Tur24]

This method is particularly useful in large datasets where the variance due to random sampling is minimal [HTF09].

16.1.2. k-Fold Cross-Validation

In k-fold cross-validation, the dataset is divided into k equal-sized subsets (folds). The model is trained k times, each time using k–1 folds for training and one fold for testing. The results are averaged to produce a robust estimate of model performance.

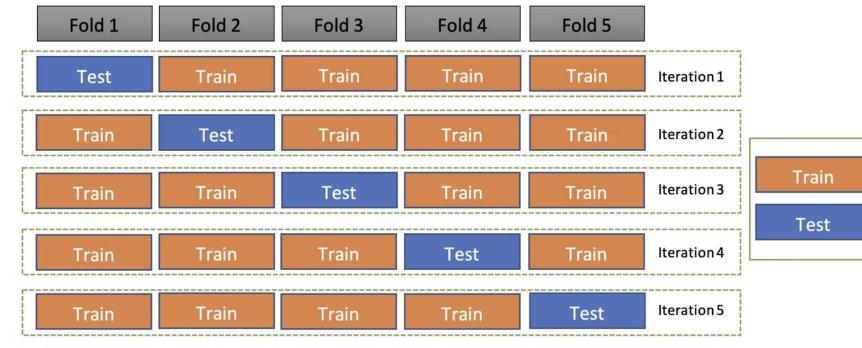


Figure 16.3.: K-Fold Cross-Validation Workflow [Tur24]

k-fold cross-validation reduces the risk of model bias from any one train-test split and is widely regarded as a gold standard for model evaluation [Koh95].

16.1.3. Stratified k-Fold Cross-Validation

This variant of k-fold cross-validation maintains the proportion of classes (labels) across each fold. It is particularly useful in classification problems involving imbalanced datasets, ensuring each fold is representative of the overall distribution.

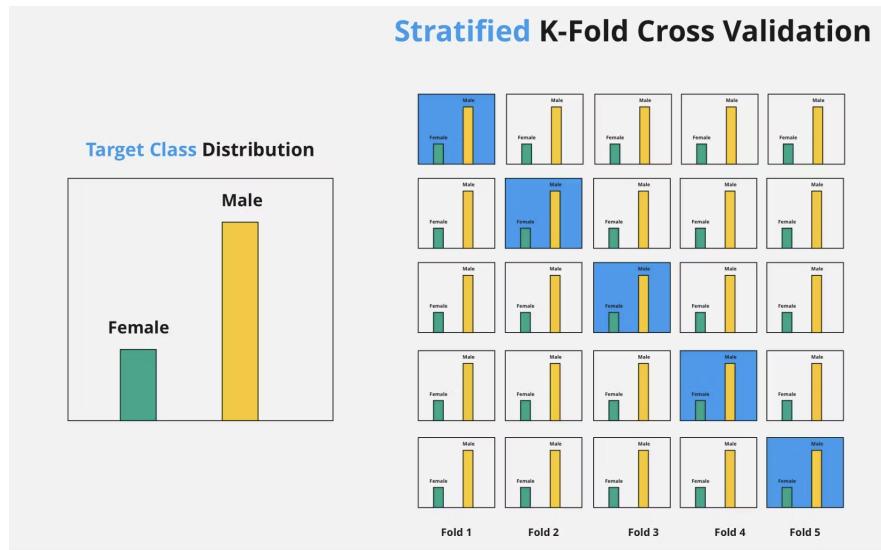


Figure 16.4.: Stratified K-Fold Cross-Validation Workflow [Tur24]

Stratified folds are known to yield better and less biased performance estimates, especially in skewed datasets [HKP11].

16.1.4. Leave-One-Out Cross-Validation (LOOCV)

This is a special case of k-fold CV where k equals the number of data points. Each data point is used once as a validation set while the remaining $n - 1$ points are used for training. Though computationally expensive, it provides nearly unbiased performance estimates.

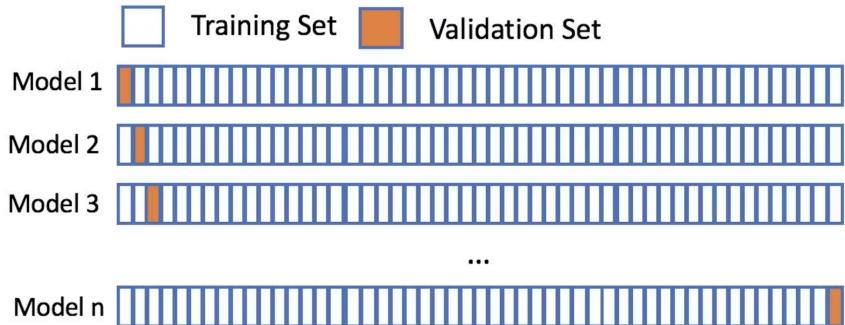


Figure 16.5.: Leave-One-Out Cross-Validation Workflow [Tur24]

LOOCV offers low bias but high variance, which may not be ideal for unstable models [Jam+13].

16.1.5. Time Series Split / Rolling Forecast Origin

For time series data, random splits are inappropriate as they violate temporal dependencies. Instead, TimeSeriesSplit ensures that training always precedes testing. This method is crucial in forecasting tasks like stock price prediction or energy consumption modeling.

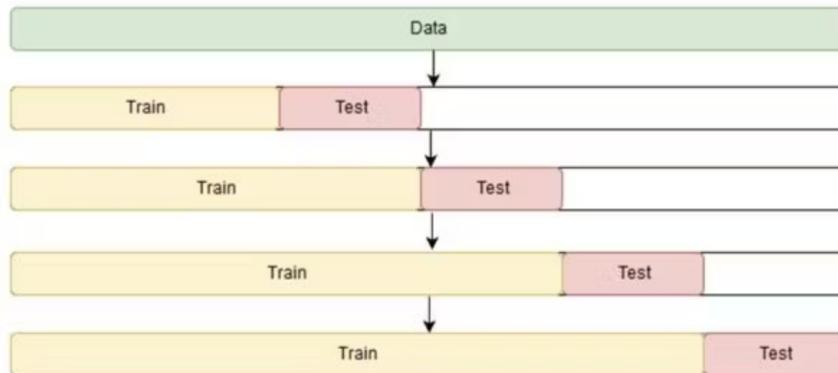


Figure 16.6.: Time Series Cross-Validation Workflow [Tur24]

Time-aware validation preserves temporal integrity and mimics real-world deployment scenarios [HA18].

16.2. Validation Approaches Implemented in Our Project

In our time series forecasting project involving ARIMA and LSTM models on financial data (e.g., NIFTY 50 index prices), we implemented three complementary validation strategies to ensure reliability and interpretability of the predictions.

16.2.1. Idea 1:

Rolling Forecast Validation (Walk-Forward Validation)

We implement a rolling-window validation mechanism to closely mimic real-world deployment in financial forecasting. At each step:

1. The model is retrained on all data up to time t , using a three-year history window (configured via `config.rollingWindowYears = 3`) in `models/arima/arimaModel.py` and `models/lstm/lstmModel.py`.
2. A one-step-ahead forecast is generated for time $t + 1$.
3. The forecast is compared to the actual observed value at $t + 1$.

Purpose: Simulate true operational conditions by ensuring that no future information leaks into training.

Benefits:

- *Temporal Realism:* Guarantees that each forecast only uses information available at the time of prediction.
- *Robustness Assessment:* Reveals how model accuracy evolves as market regimes shift, uncovering performance fluctuations under varying conditions.

This approach provides a rigorous evaluation of both ARIMA and LSTM models, highlighting their stability and adaptability over time.

ARIMA

Listing 16.1: Working concept of ARIMA Rolling Forecast Validation

```
# @brief Perform rolling forecast using ARIMA on univariate time
    # series data.
#
# This function fits an ARIMA model repeatedly on a growing training
    # set and forecasts
```

```

# one step ahead each time. It is useful for evaluating ARIMA models
    ↗ with walk-forward validation.
#
##

from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_absolute_error
import numpy as np

##
# @brief Rolling forecast with ARIMA using walk-forward validation.
#
# @param data np.ndarray or list - Univariate time series data.
# @param p int - ARIMA autoregressive order.
# @param d int - ARIMA differencing order.
# @param q int - ARIMA moving average order.
# @param train_size float - Proportion of data to use for initial
    ↗ training (default is 0.8).
#
# @return tuple - (predictions, test), where predictions is a list
    ↗ of forecasts and
#           test is the corresponding true values.
#
def rolling_forecast_arima(data, p, d, q, train_size=0.8):
    t = train_len = int(len(data) * train_size)
    history = list(data[:train_len])
    test = data[train_len:]
    predictions = []

    for t in range(len(test)):
        model = ARIMA(history, order=(p, d, q))
        model_fit = model.fit()
        forecast = model_fit.forecast()[0]
        predictions.append(forecast)
        history.append(test[t])

    mae = mean_absolute_error(test, predictions)
    print("Rolling Forecast MAE (ARIMA):", round(mae, 4))
    return predictions, test

```

LSTM

Listing 16.2: Working concept of ARIMA Residual Diagnostics Validation

```

# @brief Perform rolling forecast using an LSTM model on time series
    ↗ data.
#
# This function applies a walk-forward validation approach for LSTM
    ↗ models.
# It uses a sliding window of 'look_back' timesteps to predict the
    ↗ next value iteratively.
#
##

import numpy as np

```

```

from sklearn.metrics import mean_absolute_error

##
# @brief Rolling forecast for LSTM using walk-forward validation.
#
# @param model Trained LSTM model object.
# @param data np.ndarray - Unscaled time series values (1D).
# @param look_back int - Number of past time steps to use as input.
# @param scaler Fitted MinMaxScaler used during training for
#               normalization.
#
# @return list - Forecasted values (inverse-transformed to original
#               scale).
##
def rolling_forecast_lstm(model, data, look_back, scaler):
    predictions = []
    test_len = len(data) - look_back

    for i in range(test_len):
        input_seq = data[i:i + look_back]
        input_seq_scaled = scaler.transform(input_seq.reshape(-1, 1))
        input_seq_scaled = input_seq_scaled.reshape(1, look_back, 1)

        pred_scaled = model.predict(input_seq_scaled, verbose=0)
        pred = scaler.inverse_transform(pred_scaled)[0][0]
        predictions.append(pred)

    return predictions

```

16.2.2. Idea 2:

Residual Diagnostics and Error Distribution Checks

For ARIMA, we conducted thorough residual diagnostics to ensure the model's assumptions held. This included testing for normality, autocorrelation (using ACF/PACF), and checking whether residuals resembled white noise.

Purpose: Validates that the model has captured all the information from the data. **Benefits:**

- Ensures statistical validity of the model.
- Increases confidence in forecast reliability.

Using plots such as Q-Q plots, histograms, and correlograms, we validated that residuals approximated a normal distribution and were independently distributed.

Listing 16.3: ARIMA Residual Diagnostics Validation

```

# @brief Diagnostic plotting for ARIMA residuals to assess model
#       adequacy.

```

```

#
# This function visualizes residuals from a fitted ARIMA model using
# time series,
# histogram, Q-Q plot, and autocorrelation function (ACF) plots.
#
##

import matplotlib.pyplot as plt
import scipy.stats as stats
from statsmodels.graphics.tsaplots import plot_acf

##
# @brief Visualize diagnostic plots of ARIMA residuals.
#
# @param model_fit A fitted ARIMA model object from statsmodels.
# @return None. Displays a 2 by 2 grid of diagnostic plots.
#
def arima_residual_diagnostics(model_fit):
    residuals = model_fit.resid

    plt.figure(figsize=(12, 8))

    # Residuals Time Series
    plt.subplot(221)
    plt.plot(residuals)
    plt.title("Residuals Time Series")

    # Histogram of Residuals
    plt.subplot(222)
    plt.hist(residuals, bins=30)
    plt.title("Histogram of Residuals")

    # Normal Q-Q Plot
    plt.subplot(223)
    stats.probplot(residuals, dist="norm", plot=plt)
    plt.title("Normal Q-Q Plot")

    # ACF Plot
    plt.subplot(224)
    plot_acf(residuals, ax=plt.gca(), lags=40)
    plt.title("ACF of Residuals")

    plt.tight_layout()
    plt.show()

```

Once the model was trained it can then be used to identify the fit using the code below

Listing 16.4: ARIMA Model fit check

```

model = ARIMA(data, order=(p, d, q))
model_fit = model.fit()
arima_residual_diagnostics(model_fit)

```

16.2.3. Idea 3:

Comparative Validation using Error Metrics

We computed and compared multiple error metrics—Root Mean Squared Error (RMSE), and Mean Absolute Percentage Error (MAPE)—for both models.

Purpose: Provides a holistic view of model accuracy from different perspectives. **Benefits:**

- MAPE provides relative error independent of scale.
- RMSE penalizes large errors more severely.
- MAE gives interpretable average deviation.

This multi-metric validation helped us identify the LSTM model as more accurate in volatile markets, while ARIMA was more stable in trend-dominated phases.

Listing 16.5: Working concept of Comparative Validation

```
# @brief Compare ARIMA and LSTM forecast accuracy using RMSE and
#       MAPE.
#
# This script calculates and prints error metrics for each
#       target column
# by comparing ARIMA and LSTM predictions to actual values over
#       the test set.
#
##

import numpy as np
from sklearn.metrics import mean_squared_error,
                           mean_absolute_percentage_error

for col in targetColumns:
    actuals = df.loc[testIndex, col]
    arimaPreds = arimaResults[col].reindex(testIndex).dropna()
    lstmPreds = lstmResults[col].reindex(testIndex).dropna()

    # Identify common valid indices across all series
    validIndex = actuals.index.intersection(arimaPreds.index).
                           intersection(lstmPreds.index)
    actuals = actuals.loc[validIndex]

    if not actuals.empty:
        ##
        # @brief Calculate error metrics
        #
        # @details
        # RMSE: Root Mean Squared Error
        # MAPE: Mean Absolute Percentage Error
        ##
        arimaRmse = np.sqrt(mean_squared_error(actuals, arimaPreds.loc[
                           validIndex]))
```

```
lstmRmse = np.sqrt(mean_squared_error(actuals, lstmPreds.loc[
    ↪ validIndex]))

arimaMape = mean_absolute_percentage_error(actuals, arimaPreds.
    ↪ loc[validIndex])
lstmMape = mean_absolute_percentage_error(actuals, lstmPreds.loc
    ↪ [validIndex])

print(msg.get("forecast_results_header").format(column=col))
print(msg.get("forecast_metrics").format(model="ARIMA", rmse=
    ↪ arimaRmse, mape=arimaMape))
print(msg.get("forecast_metrics").format(model="LSTM", rmse=
    ↪ lstmRmse, mape=lstmMape))
else:
    print(msg.get("no_valid_forecast").format(column=col))
```

17. Monitoring

17.1. Idea

This project focuses on forecasting the next-day `Open` and `Close` prices of the NIFTY 50 index using two distinct approaches: ARIMA and LSTM. ARIMA provides a classical statistical baseline for univariate time series, while LSTM leverages deep learning to model complex sequential patterns [HA18; GS23]. The goal is to compare these models in a reproducible and user-friendly environment, supported by a Streamlit-based GUI, versioned model files (`.keras`, `.pkl`), automated tests, and multilingual support through `messages.json`. The system emphasizes accessibility, interpretability, and best practices in ML engineering [Cho23].

17.2. Plan/Description

17.2.1. Overview

The forecasting pipeline is structured into four main stages: data acquisition, preprocessing, model inference, and result visualization. The modular design enables clarity, maintainability, and reuse across different forecasting tasks. Each layer interacts with externalized configuration and messaging files to support multi-language deployment and reproducibility.

17.2.2. Data Acquisition and Preprocessing

Historical NIFTY 50 data is fetched using the `yfinance` Python wrapper, which accesses Yahoo Finance REST APIs and delivers data in a CSV-friendly format [Yfi]. Data is parsed into a `pandas.DataFrame`, with the `dateTime` field indexed and numeric fields cast to `float64` [RMB+20]. For LSTM preprocessing, `MinMaxScaler` from `scikit-learn` is applied separately to `Open` and `Close` columns, normalizing them to the $[0,1]$ range to stabilize training and minimize gradient issues [Ped+21].

17.2.3. Model Loading and Inference

Users can select ARIMA or LSTM models via a Streamlit-based graphical interface. ARIMA models are serialized in `.pkl` format using `joblib`,

preserving their parameter state for one-step prediction without re-fitting [HA18; PVG+11]. LSTM models are stored in `.keras` format, capturing both architecture and weights, and loaded using TensorFlow’s model API [Cho23]. The selected model performs inference for a single future date, generating forecasts for both `Open` and `Close` prices.

17.2.4. Evaluation and Visualization

The predicted values are plotted against recent historical data using `matplotlib`, which supports both static and interactive back ends [Hun07]. For evaluation, RMSE and MAPE metrics are calculated to assess forecast accuracy. These are widely adopted in financial forecasting due to their interpretability and scale-awareness [Chi21].

17.2.5. User Interface and Localization

The interface is built with Streamlit, a reactive framework that transforms Python scripts into interactive dashboards [Str23]. All UI text, including model labels, error messages, and instructions, is abstracted into a `messages.json` file. This allows for internationalization (i18n) and localization without modifying source code [Bra17].

17.2.6. Model Storage and Portability

Models and scalers are organized in the `models/` directory using ASCII-compatible, camelCase filenames. ARIMA, LSTM, and scaler files are separated by model type and target variable (e.g., `openPrice`, `closePrice`). This ensures compatibility, reproducibility, and clarity during deployment or handoff between collaborators [Aut23].

17.2.7. Logging and Debugging

All runtime errors and data inconsistencies are recorded using Python’s built-in `logging` module. Logs are stored in a persistent file for auditability, making the system suitable for academic evaluation and production reliability [Pyt24].

17.2.8. Streamlit Keep-Alive

Many free or low-cost hosting platforms (including Streamlit Community Cloud) automatically suspend applications after a period of inactivity (typically around one hour) to conserve resources. When an app is suspended, the next visitor experiences a “cold start,” which can add 15–30 seconds of delay while the application re-initializes its Python environment

and reloads all dependencies. To mitigate this, we implemented an external “keep-alive” mechanism using UptimeRobot.

UptimeRobot Monitor Configuration We created a new HTTP monitor in UptimeRobot with these settings:

- **Monitor Type:** HTTP(S)
- **URL:** `https://nifty50indexprediction.streamlit.app`
- **Check Interval:** 30 minutes

This configuration causes UptimeRobot’s servers to send an HTTP GET request to our `/health` endpoint every 30 minutes. As long as the response is “OK” (HTTP 200), the app remains active and is not suspended by the hosting platform.

Advantages

- **Reduced Latency for Users:** Eliminates multi-second cold-start delays, ensuring that users always encounter a ready-to-use interface.
- **Improved Reliability:** Regular health checks provide a secondary benefit of uptime monitoring; alerts can be configured to notify the development team if the app ever returns a non-200 status.
- **Low Overhead:** The health endpoint performs trivial work, so the periodic pings have negligible impact on CPU or memory usage.

Limitations and Considerations

- **Platform Policies:** Some hosting providers may consider frequent pings as abuse or “heartbeat” misuse; it is important to verify that this practice complies with the platform’s terms of service.
- **Monitoring Gaps:** If UptimeRobot experiences downtime or network issues, the app may still sleep. For mission-critical deployments, consider multiple monitoring services or a commercially supported hosting tier.
- **Cost Implications:** While UptimeRobot’s basic plan is free for up to 50 monitors, high-frequency checks or advanced alerting may require a paid subscription.

17.3. Getting New Data

To ensure accurate and up-to-date forecasting, the system retrieves the latest stock market data from Yahoo Finance using the open-source `yfinance` Python library [Yfi]. This wrapper provides a programmatic interface to the Yahoo REST API, allowing for seamless downloading of NIFTY 50 data in tabular format.

The fetched data includes `Open`, `Close`, `High`, `Low`, and `Volume` fields and is saved as a `.csv` file. Each row represents a trading day, with the `dateTime` column used as the index. For consistency, all price fields are explicitly cast to `float64`, and the `dateTime` field is parsed into ISO 8601-compliant `datetime64[ns]` objects using `pandas` [RMB+20].

To avoid inconsistencies, the application performs early validation of the incoming dataset. It ensures:

- No missing timestamps for expected trading days (NSE calendar)
- No non-numeric values in price fields
- Proper chronological ordering of records

If the selected prediction date falls after the range of available internal data, the system prompts the user to upload a newer `.csv` file. This file must match the column schema used during training. Upon upload, the same preprocessing steps (handling missing values, COVID dummy creation, normalization) are applied before inference begins. This allows the system to remain model-consistent while adapting to real-world deployment [Ped+21].

Listing 17.1: Downloading New Data via yFinance

```
# @brief Downloads NIFTY 50 historical data using yFinance and
#       saves to CSV.
#
# This script fetches historical data for the NIFTY 50 index (^
#       NSEI) between
# January 2023 and December 2024 using the yfinance API and
#       stores it in a CSV file.
# Useful for backtesting, forecasting, and exploratory analysis.
#
##

import yfinance as yf

##
# @brief Create a Ticker object for NIFTY 50 using its Yahoo
#       symbol "^NSEI".
tickerObj = yf.Ticker("^NSEI")

##
# @brief Fetch daily historical data from January 2023 to
#       December 2024.
```

```

dfNew = tickerObj.history(start="2023-01-01", end="2024-12-31")

##
# @brief Reset the index to move the Date from index to column
#        format.
dfNew.reset_index(inplace=True)

##
# @brief Save the DataFrame to a CSV file for local analysis.
dfNew.to_csv("latest_nifty50.csv", index=False)

```

17.4. Data Updating in the ML Pipeline

Continuous integration of new data is critical for maintaining forecast relevance in time-series models. In this system, updating the data pipeline begins with appending new `.csv` files downloaded using the `yfinance` API [Yfi]. These files contain the most recent NIFTY 50 trading data and must conform to the predefined schema used during model training.

Upon data arrival, a preprocessing routine checks for:

- Consistency of column names and data types with the training dataset
- Missing or malformed entries, which are either removed or imputed
- Temporal continuity with the last known timestamp in the internal dataset

Once validated, the new records are appended to the internal dataset, and the time-series is rescaled using the original `MinMaxScaler` objects stored as `.pkl` files. This ensures that predictions remain consistent with the model's learned scale [PVG+11].

The LSTM model uses a fixed-size rolling window of past values for inference. Therefore, only the most recent n days (typically 60) are retained from the combined dataset for generating the input sequence. For ARIMA, the updated series is passed as-is to compute one-step-ahead forecasts using either reloaded or retrained models depending on the selected operational mode.

Listing 17.2: Appending and Rescaling New Data

```

# @file scale_open_prices.py
# @brief Load latest NIFTY 50 data and apply MinMax scaling to
#        the "Open" column.
#
# This script reads a CSV file containing new NIFTY 50 data,
#        loads a pre-trained
# scaler object using Joblib, and applies the scaler to the "
#        Open" column for

```

```

# downstream predictions using machine learning models.
#
##

import pandas as pd
import joblib

##
# @brief Load the latest NIFTY 50 dataset from CSV.
# The CSV file is expected to have a "dateTime" column and an "
#   ↪ Open" price column.
dfNew = pd.read_csv("latest_nifty50.csv", parse_dates=["dateTime"
#   ↪ ""])

##
# @brief Load the pretrained MinMaxScaler for the "Open" price
#   ↪ feature.
# The scaler should have been saved previously using joblib.dump
#   ↪ ().
scalerOpen = joblib.load("models/scalerOpen.pkl")

##
# @brief Apply the loaded scaler to the "Open" column to obtain
#   ↪ a normalized feature.
# The scaled values are stored in a new column named "openScaled"
#   ↪ ".
dfNew["openScaled"] = scalerOpen.transform(dfNew[["Open"]])

```

This dynamic update mechanism allows the system to adapt to newly available data without full retraining, which is essential in real-time financial forecasting scenarios [HA18].

17.5. Validation and Consistency Checks

To ensure reliability across all stages of the forecasting pipeline, several automated checks are incorporated during data ingestion and preprocessing. These validations help detect structural anomalies, silent corruption, and compatibility issues before model inference begins.

Schema Conformance

The raw dataset is validated to match the expected schema, including column presence, data types, and datetime indexing. A mismatch in expected columns (e.g., **Open**, **Close**) triggers a hard stop, as downstream models depend on fixed field names and formats [RMB+20].

Missing Data Inspection

After loading the CSV, the system runs a null check across critical columns using Pandas:

Listing 17.3: Basic Null Check

```
assert dfPrices[["Open", "Close"]].notnull().all().all(), "  
    ↪ Missing values found"
```

In cases of incomplete data, the system either drops rows (for inference-only mode) or halts with a descriptive error message.

Time Continuity

The `dateTime` index is reindexed against a known calendar of NSE trading days. Any gaps filled with `NaN` are logged and flagged for manual inspection. This avoids misaligned sequences that would distort time-dependent models like LSTM [Bro20].

Scaler Compatibility

The new input data is transformed using previously saved `MinMaxScaler` objects. A shape mismatch between the current and original training data triggers a validation error, ensuring scaling consistency and avoiding inference drift [PVG+11].

File Integrity Audit

Each model and scaler file is accompanied by a SHA-256 checksum stored in a separate JSON. During runtime, these checksums are recomputed and matched against the stored values to verify file integrity. This mechanism detects silent overwrites and incomplete file transfers [Bra17].

These rigorous validation layers uphold both scientific reproducibility and operational stability, minimizing the risk of unnoticed failures during automated model execution.

17.6. Code: Utility Functions

This section collects the core functions used throughout the NIFTY 50 forecasting pipeline, organized by purpose. Each group of functions encapsulates a distinct phase of the workflow, from raw data ingestion to model evaluation.

17.6.1. Data Preprocessing Functions

The `loadAndPreprocessData` function handles the initial loading and cleaning of raw CSV data. Its responsibilities include:

- Reading the CSV file into a `pandas.DataFrame`, parsing the `Date` column as a datetime index.

- Sorting the index chronologically to ensure time series consistency.
- Dropping any rows missing critical price fields (**Open**, **High**, **Low**, **Close**) to guarantee model inputs are complete.

Listing 17.4: Loading and preprocessing NIFTY 50 stock data from CSV using **pandas**

```
# @brief Load and preprocess NIFTY 50 time series data from a CSV
#       file.
#
# This module defines a function to load stock data, parse dates,
#       set the date
# as index, sort the index, and remove any rows with missing
#       critical values.
#
##

import pandas as pd

##
# @brief Load and preprocess the NIFTY 50 dataset.
#
# This function performs the following operations:
# - Reads the CSV file into a DataFrame.
# - Parses the 'Date' column as datetime and sets it as index.
# - Sorts the DataFrame chronologically by date index.
# - Drops rows where any of the 'Open', 'High', 'Low', or 'Close'
#       values are missing.
#
# @param filePath Full path to the CSV file containing historical
#       NIFTY 50 data.
# @return pd.DataFrame Cleaned and indexed time series DataFrame.
def loadAndPreprocessData(filePath: str) -> pd.DataFrame:
    df = pd.read_csv(filePath, parse_dates=['Date'], index_col='Date')
    df = df.sort_index()
    df = df.dropna(subset=['Open', 'High', 'Low', 'Close'])
    return df
```

17.6.2. Feature Engineering Functions

Feature engineering functions enrich the cleaned data with additional signals. For example, **addCovidDummy** introduces a binary indicator capturing the market disruption during the COVID-19 period. This column can help models adjust for that anomaly when fitting and forecasting.

Listing 17.5: Adding a COVID-19 dummy variable to a financial time series DataFrame

```
# @brief Add a COVID-19 dummy variable to financial time series data
#       .
#
```

```

# This module defines a function that adds a binary indicator column
#   ("CovidDummy")
# to indicate the COVID-19 impact period in 2020.
#
##

import numpy as np
import pandas as pd

##
# @brief Add a dummy column marking the COVID-19 impact period.
#
# This function flags all rows from March 1, 2020 to December 31,
#   2020
# with a value of 1 in a new column called 'CovidDummy'. All other
#   rows
# are marked as 0.
#
# @param df A pandas DataFrame with a datetime index.
# @return pd.DataFrame The original DataFrame with an added '
#   CovidDummy' column.
def addCovidDummy(df: pd.DataFrame) -> pd.DataFrame:
    mask = (df.index >= '2020-03-01') & (df.index <= '2020-12-31')
    df['CovidDummy'] = mask.astype(int)
    return df

```

17.6.3. Model Training Functions

These functions encapsulate the logic for fitting both ARIMA and LSTM models:

- `trainArimaModel` uses `pmdarima.auto_arima` to automatically select the best (p,d,q) parameters on a univariate series, then saves the fitted model via `joblib.dump()`.
- `trainLstmModel` builds a two-layer LSTM network, trains it on 3D scaled input, and saves the resulting Keras model. It expects the input array to be shaped as (samples, lookback, features).

Listing 17.6: Training and saving ARIMA and LSTM models for stock price forecasting

```

# @brief Functions for training ARIMA and LSTM models for stock
#   forecasting.
#
# This script contains methods to train and save univariate ARIMA
#   models using auto_arima,
# and multivariate LSTM models using Keras. It uses joblib for ARIMA
#   model persistence and
# Keras's .save() method for saving LSTM models.
#

```

```

##

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dropout, Dense
from sklearn.preprocessing import MinMaxScaler
from pmdarima import auto_arima
import joblib
import numpy as np

##
# @brief Train and save an ARIMA model using auto_arima.
#
# This function fits an ARIMA model on a univariate time series
# using the
# 'pmdarima.auto_arima()' method with no seasonal component. The
# model is
# then saved using 'joblib'.
#
# @param series A 1D array-like structure (e.g., pandas Series) of
# time series data.
# @param modelPath Path to save the trained ARIMA model using joblib
# .
# @return The trained ARIMA model.
def trainArimaModel(series, modelPath: str):
    model = auto_arima(
        series,
        seasonal=False,
        stepwise=True,
        suppress_warnings=True,
        error_action='ignore'
    )
    joblib.dump(model, modelPath)
    return model

## 
# @brief Train and save a two-layer LSTM model on scaled 3D time
# series data.
#
# This function assumes the data has already been scaled and
# reshaped into 3D form
# with dimensions (samples, lookback + 1, features). It uses the
# Close price as the
# prediction target. The model is trained using MSE loss and the
# Adam optimizer.
#
# @param scaledData A 3D NumPy array of shape (samples, lookback +
# 1, features).
# @param lookback Number of timesteps to look back in each sequence.
# @param modelPath Path to save the trained LSTM model in Keras
# format.
# @return The trained LSTM model.
def trainLstmModel(scaledData: np.ndarray, lookback: int, modelPath:
    str):
    nFeatures = scaledData.shape[2]
    X = scaledData[:, :lookback, :]
    y = scaledData[:, lookback, 1] # assuming 'Close' is the second
    # feature
    model = Sequential([
        LSTM(64, return_sequences=True, input_shape=(lookback,

```

```

        ↵ nFeatures)),
Dropout(0.2),
LSTM(32),
Dropout(0.2),
Dense(1)
])
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X, y, epochs=5, batch_size=32, verbose=0, shuffle=
    ↵ False)
model.save(modelPath)
return model

```

17.6.4. Model Evaluation Functions

After forecasts are generated, `evaluateForecast` computes key error metrics—RMSE and MAPE—between the true and predicted arrays. These metrics provide quantitative insight into model accuracy and bias.

Listing 17.7: Evaluating forecast accuracy using RMSE and MAPE metrics

```

# @brief Evaluation function for forecast accuracy using RMSE and
    ↵ MAPE.
#
# This module defines a utility function to evaluate prediction
    ↵ accuracy by calculating
# Root Mean Squared Error (RMSE) and Mean Absolute Percentage Error
    ↵ (MAPE) using
# true and predicted NumPy arrays. These metrics help assess model
    ↵ performance in forecasting.
#
##

from sklearn.metrics import mean_squared_error,
    ↵ mean_absolute_percentage_error
import numpy as np

##
# @brief Evaluate forecast accuracy using RMSE and MAPE.
#
# This function computes two commonly used error metrics to assess
    ↵ how close the predicted
# values are to the actual values.
#
# @param yTrue A NumPy array of ground-truth (actual) values.
# @param yPred A NumPy array of predicted values from the model.
# @return A dictionary with 'rmse' and 'mape' as keys and
    ↵ corresponding float values.
def evaluateForecast(yTrue: np.ndarray, yPred: np.ndarray) -> dict:
    rmse = mean_squared_error(yTrue, yPred, squared=False)
    mape = mean_absolute_percentage_error(yTrue, yPred)
    return {'rmse': rmse, 'mape': mape}

```

17.7. Unit Tests

This section outlines test cases for each core function in the forecasting pipeline.

Function: `loadNifty50Yfinance`

- **Test Case 1: Valid Date Range**

Call `loadNifty50Yfinance("2020-01-01", "2020-12-31")` and verify the returned DataFrame:

- Indexed by `Date`, contains only weekdays.
- Includes a `COVID_dummy` column with ones for 2020-03-01 to 2020-12-31 and zeros elsewhere.

- **Test Case 2: Missing Close Values**

Simulate a row where `Close` is `NaN` in the raw download; ensure that row is dropped (i.e. no `Close` nulls in output).

- **Test Case 3: Column Selection**

Provide an API response missing “High” or “Low” and confirm that the returned DataFrame still contains available price columns plus `COVID_dummy`, without error.

Function: `buildAndTrainLstm`

- **Test Case 1: Insufficient Data**

Pass a `dfTrainScaled` with fewer than `lookback+1` rows and verify the function returns `None` without raising an exception.

- **Test Case 2: Model Structure**

Supply synthetic scaled data of shape `(N, lookback, features)` with $N >$ lookback and confirm the returned object is a Keras `Sequential` with layers: `LSTM(64)`, `Dropout(0.2)`, `LSTM(32)`, `Dropout(0.2)`, `Dense(1)`.

- **Test Case 3: Training History**

Train on a small dataset for 1 epoch and check that the history object records one loss value.

Function: `runLstm`

- **Test Case 1: No Retrain**

Set `retrainInterval` larger than the number of test days; confirm that model files’ timestamps do not change across predictions.

- **Test Case 2: Rolling Forecast**

Use a DataFrame covering exactly `rollingWindowYears + look-back + 2` days and verify:

- A prediction for each test day.
- `NaN` for days with insufficient lookback history.

- **Test Case 3: Output Format**

Run `runLstm` on real data and ensure the result is a dictionary with keys “Open” and “Close”, each mapping to a pandas Series indexed by forecast dates.

Function: `runArima`

- **Test Case 1: Minimum Observations**

Provide a DataFrame where fewer than five observations exist for some test windows; verify those forecasts are `NaN` without error.

- **Test Case 2: ARIMA Order**

For a synthetic AR(1) series, confirm `auto_arima` chooses $p \geq 1$ and `runArima` yields non-`NaN` forecasts.

- **Test Case 3: Model Saving**

After calling with `retrain=True`, check that `models/arima/arimaModelOpen.pkl` and `arimaModelClose.pkl` exist and load via `joblib.load()`.

Function: `logError`

- **Test Case 1: Error Logging**

Invoke `logError(ValueError("test"), context="UnitTest")` and verify `log.txt` contains “[UnitTest]” and the generic error template.

- **Test Case 2: Traceback Capture**

Confirm that a DEBUG-level entry follows with the full exception traceback.

Method: `MessageHandler.get`

- **Test Case 1: Existing Key**

With `messages.json` containing `{"hello": "Hi"}`, verify `MessageHandler().get("hello")` returns “Hi”.

- **Test Case 2: Missing Key**

Confirm `get("nonexistentKey")` returns the fallback “[nonexistentKey]”.

Integration Tests

Integration tests verify that the components of the NIFTY 50 forecasting system interoperate correctly.

- **Integration Test Case 1: Complete Pipeline Validation**

Run the full pipeline from data ingestion through forecasting:

1. Invoke `loadNifty50Yfinance()` to fetch and preprocess data.
2. Pass the returned DataFrame to `runArima()` and `runLstm()`.
3. Confirm that each stage (filtering weekdays, adding dummy, model training, forecasting) produces non-empty, correctly typed outputs.
4. Verify that the final forecast Series for “Open” and “Close” contain timestamp indices and numeric values or `NaN` only where history was insufficient.

- **Integration Test Case 2: End-to-End Application Flow**

Simulate user execution of `main.py`:

1. Execute the main script in a controlled environment (fixed date range).
2. Capture stdout and confirm it prints the “next-day” forecast header and price lines for both ARIMA and LSTM.
3. Verify that the plot PNG files (`forecast_plot_Open.png`, `forecast_plot_Close.png`) are created and non-empty.
4. Check that model files (`lstmModelOpen.keras`, `arimaModelClose.pkl`, etc.) exist in `models/` directories with timestamps matching the run.

Regression Tests

Regression tests ensure that modifications to the codebase do not introduce unexpected changes in existing functionality.

- **Regression Test Case 1: Stability of Core Pipeline**

1. After any code update (e.g., refactoring, dependency bump), run the full suite of unit and integration tests.
2. Confirm that `loadNifty50Yfinance()`, `runArima()` and `runLstm()` all return results in the same format as before (DataFrame, Series, model files).
3. Verify that no existing test case fails and that forecast outputs remain structurally unchanged.

- **Regression Test Case 2: Historical Forecast Consistency**

1. Define a fixed historical dataset (e.g., NIFTY 50 prices from 2019-01-01 to 2019-12-31) and precomputed “golden” forecasts for both ARIMA and LSTM.
2. After code changes, re-run `runArima()` and `runLstm()` on this dataset.
3. Assert that the numerical forecasts match the golden values within a small tolerance (e.g., RMSE difference < 1e-6).

Regression Tests

Regression tests ensure that modifications to the codebase do not introduce unexpected changes in existing functionality.

- **Regression Test Case 1: Stability of Core Pipeline**

1. After any code update (e.g., refactoring, dependency bump), run the full suite of unit and integration tests.
2. Confirm that `loadNifty50Yfinance()`, `runArima()` and `runLstm()` all return results in the same format as before (DataFrame, Series, model files).
3. Verify that no existing test case fails and that forecast outputs remain structurally unchanged.

- **Regression Test Case 2: Historical Forecast Consistency**

1. Define a fixed historical dataset (e.g., NIFTY 50 prices from 2019-01-01 to 2019-12-31) and precomputed “golden” forecasts for both ARIMA and LSTM.
2. After code changes, re-run `runArima()` and `runLstm()` on this dataset.
3. Assert that the numerical forecasts match the golden values within a small tolerance (e.g., RMSE difference < 1e-6).

Performance Tests

Performance tests assess the efficiency and scalability of the NIFTY 50 forecasting system.

- **Performance Test Case 1: Data Loading Throughput**

1. Load and preprocess a large CSV (e.g., 10 years of NIFTY 50 daily data) via `loadNifty50Yfinance()`.

2. Measure end-to-end execution time and memory usage.

3. Assert that loading + preprocessing completes within a defined SLA (e.g., less than 5 seconds).

- **Performance Test Case 2: ARIMA Training Time**

1. Use a 3-year rolling window of maximum size (approximately 750 trading days) to train ARIMA via `runArima()`.

2. Record wall-clock time for each step of the walk-forward loop.

3. Verify that average retraining duration per window stays below a threshold (e.g., less than 2 seconds).

- **Performance Test Case 3: LSTM Training Throughput**

1. Fit the LSTM model on a full 3-year dataset using `runLstm()`.

2. Measure GPU/CPU utilization and total training time.

3. Ensure that training on each rolling window does not exceed a set limit (e.g., less than 30 seconds).

- **Performance Test Case 4: Forecast Generation Latency**

1. Generate one-day-ahead forecasts for a large test period (e.g., 2 years) with both ARIMA and LSTM.

2. Measure per-forecast latency.

3. Confirm that prediction calls remain responsive (e.g., less than 100 ms per forecast).

- **Performance Test Case 5: Visualization Scalability**

1. Plot forecasts vs. actuals over multi-year spans using `matplotlib`.

2. Time the rendering and file-save process for large DataFrames.

3. Assert that plot generation completes within acceptable bounds (e.g., less than 3 seconds).

17.8. Privacy

17.8.1. Secure Data Transmission and Storage

Utilize secure protocols and controls to protect NIFTY 50 data in transit and at rest.

- **HTTPS (HyperText Transfer Protocol Secure)**: Encrypts data between the Streamlit client and server to prevent eavesdropping and man-in-the-middle attacks, protecting credentials and model inputs during API calls.
- **SSH (Secure Shell)**: Secures remote access to production servers—used for code deployment, `scp/sftp` of data files, and administrative commands.
- **Encryption at Rest**: Apply AES-256 encryption to database dumps and model files (`.keras`, `.pkl`, `.tflite`) stored on disk or in cloud object storage.
- **Access Controls**: Enforce role-based access control (RBAC) on file shares and S3 buckets, granting read/write only to authorized ML engineers and data stewards.
- **Audit Logging**: Record all data uploads, downloads, and model serialization events with timestamped entries in `log.txt` for forensic review.

17.8.2. Compliance with Data Protection Regulations

Ensure that handling of any user or proprietary data complies with applicable regulations.

- **GDPR (General Data Protection Regulation)**: If storing any personal or sensitive user data (e.g., analyst annotations), apply data minimization and purpose limitation principles, and honor data subject rights (access, erasure).
- **Anonymization**: Strip or hash any PII fields in uploaded CSVs before preprocessing; only NIFTY 50 price columns (**Open**, **High**, **Low**, **Close**, **Volume**) are retained.
- **Pseudonymization**: If user identifiers are needed (e.g., for multi-tenant dashboards), replace them with random tokens stored in a secure mapping table.
- **Policy Enforcement**: Maintain a data handling policy that documents procedures for collection, storage, access, and deletion of all files in the `data/` and `models/` directories.

17.8.3. Access Control and Data Anonymization

Implement strict permissions and anonymization to balance security with analytical needs.

- **Role Definitions:** Define roles such as *DataEngineer*, *MLDeveloper*, and *Viewer*, each with scoped permissions on Git branches, data stores, and model artifacts.
- **Least Privilege:** Grant each role only the minimum access—e.g., *Viewer* can read forecast results but not download raw CSVs or retrain models.
- **Masking and Tokenization:** In any demo or shared dashboards, mask trading dates or replace them with relative offsets (e.g., Day -30 to Day 0) to prevent correlation with real-world events.
- **Data Generalization:** Aggregate old data into monthly or quarterly summaries for archival storage, reducing granularity and re-identification risk.
- **Re-identification Safeguards:** Periodically review anonymization pipelines and run simulated re-identification attacks to validate that masked or tokenized data cannot be reversed without proper keys.

17.9. Robustness

17.9.1. Handling Missing or Corrupted Data

Develop and enforce comprehensive data validation and cleaning steps in `loadNifty50Yfinance()` and downstream pipelines:

- **Data Imputation:** For occasional missing `Open/High/Low/Close` values, apply forward- or backward-fill on the time index before dropping irrecoverable rows.
- **Outlier Detection:** Use Z-score or IQR methods on price returns to flag and optionally cap extreme spikes that may result from data glitches.
- **Integrity Checks:** After download, assert monotonicity of dates and non-negativity of volume; log and remove any corrupted records via `logError()`.

17.9.2. Ensuring Adaptability to Market Conditions

Incorporate mechanisms to keep models aligned with evolving financial environments:

- **Rolling Retraining:** Both ARIMA (`runArima()`) and LSTM (`runLstm()`) use a 3-year sliding window and configurable `retrainInterval` to capture new market regimes.
- **Exogenous Indicators:** Optionally extend feature set with macroeconomic variables (e.g., interest rates, VIX index) via `loadNifty50Yfinance()` for improved responsiveness to external shocks.
- **Drift Monitoring:** Track rolling MAPE and RMSE; trigger an alarm or retraining if error metrics exceed predefined thresholds, indicating model drift.

17.9.3. Regular Testing Under Diverse Scenarios

Validate system stability and forecast consistency across simulated market conditions:

- **Stress Tests:** Re-run the full pipeline on historical crisis periods (e.g., 2008 financial crisis, 2020 COVID crash) and ensure no unhandled exceptions occur.
- **Scenario Analysis:** Inject synthetic shocks into price series (e.g., $\pm 10\%$ overnight move) and verify model outputs remain within plausible ranges.
- **Anomaly Recovery:** Simulate partial data outages by removing random days from input and confirm that imputation and retraining still produce forecasts without failure.

17.10. Process

17.10.1. Data Collection and Preprocessing

- **Gathering Raw Data:** Fetch NIFTY 50 historical OHLCV data via `yfinance`, ensuring coverage from 2008–present; cache fallback CSV for resilience (`loadNifty50Yfinance()`) [Yfi].
- **Cleaning Data:** Filter non-trading days, drop rows with missing `Close` or other price fields, and log any dropped records via `logError()`.

- **Preprocessing Data:** Rename index to `Date`, enforce `float64` types, add `COVID_dummy` for 2020 disruption, and retain only `{Open, High, Low, Close, Volume}` + dummy.
- **Feature Engineering:** Optionally compute technical indicators (e.g., rolling returns, volatility) in `addCovidDummy()` or a new utility to augment model inputs.

17.10.2. Model Development and Training

- **Model Selection:** Choose ARIMA for univariate linear baseline (`runArima()`) and LSTM for non-linear, sequence-based learning (`runLstm()`) [HA18; GS23].
- **Training Models:** Execute walk-forward training with a 3-year rolling window, retraining every `retrainInterval` days; ARIMA uses `auto_arima()`, LSTM uses `buildAndTrainLstm()`.
- **Validation Split:** Determine `trainEndDate` via `getTrainEndDate()`, reserve subsequent dates for out-of-sample testing and walk-forward forecast.
- **Hyperparameter Tuning:** Rely on `auto_arima()` AIC/BIC optimization for ARIMA; fix LSTM layers/units and dropout based on empirical benchmarks.

17.10.3. Model Evaluation and Validation

- **Performance Metrics:** Compute RMSE and MAPE on rolling test segments for both `Open` and `Close` (`evaluateForecast()`) [Chi21].
- **Forecast vs Actual:** Align predicted Series with actual prices, generate quantitative reports and time-series plots via `matplotlib`.
- **Cross-Model Comparison:** Compare ARIMA vs LSTM error metrics and visualize in unified dashboards (Streamlit).

17.10.4. Continuous Monitoring and Maintenance

- **Performance Monitoring:** Log daily RMSE/MAPE; trigger alert if drift exceeds threshold.
- **Data Quality Monitoring:** Re-validate incoming CSV schema and null rates before preprocessing.

- **Regular Updates:** Append new data via `loadNifty50Yfinance()`, re-scale with saved scalers, and optionally retrain models on schedule.
- **Feedback Loop:** Integrate user feedback from Streamlit UI to adjust feature set or retraining cadence.

17.10.5. Documentation and Communication

- **Comprehensive Documentation:** Maintain docstrings (Doxygen) in each module, and a central README outlining setup, usage, and architecture.
- **Transparent Reporting:** Publish interactive Streamlit dashboards showing forecasts, errors, and logs.
- **Stakeholder Engagement:** Share periodic summaries of model performance and retraining outcomes via email or embedded UI alerts.
- **Scalability Support:** Version all models (`.keras`, `.pkl`) and scalers, and document directory layout (`models/`, `assets/`, `predictions/`).

Part VI.

Application Deployment

18. Deployment

18.1. Directory Structure

The project is organized into a clear, modular layout to separate core forecasting logic, utilities, tests, and the GUI wrapper. Each top-level folder serves a distinct purpose:

```
.  
+-- .github/workflows/  
|   +-- ci.yml  
+-- .pytest_cache/  
+-- GUICode/  
|   +-- Code/  
|   |   +-- streamlit/  
|   |   +-- docs/  
|   +-- models/  
|   |   +-- arimaModelClose.pkl  
|   |   +-- arimaModelOpen.pkl  
|   |   +-- lstmModelClose.keras  
|   |   +-- lstmModelOpen.keras  
|   |   +-- scalerClose.pkl  
|   |   +-- scalerOpen.pkl  
|   +-- errorHandler.py  
|   +-- messages.json  
|   +-- stockPredictorGui.py  
|   +-- requirements.txt  
+-- InitialProjectPlan/  
+-- Manual/  
+-- models/  
+-- Poster/  
+-- Presentations/  
+-- ProjectManagement/  
+-- report/  
+-- .gitignore  
+-- .python-version  
+-- author.xlsx  
+-- cached_nifty50.csv  
+-- forecast_plot_Close.png
```

```
+-- forecast_plot_Open.png  
+-- forecast_results.csv  
+-- log.txt  
+-- Makefile  
+-- README.md
```

18.1.1. Folder Responsibilities

- `.github/workflows/ci.yml`:
 - Defines a CI pipeline that on every push or PR to `main`:
 - * Checks out the repo and sets up Python 3.10.
 - * Installs all core (`Requirements.txt`) and GUI dependencies.
 - * Runs `pytest` across `Code/tests/` with strict failure policies.
 - * Executes a post-test step that *atomically* copies the newly trained model files into `GUICode/Code/models/`, ensuring GUI and core are always in sync.
 - This automated gate prevents regressions, enforces code quality, and guarantees that every deployed GUI build uses the latest validated models.
- `GUICode/Code/docs/`:
 - Auto-generated Sphinx/Doxygen HTML docs for all public GUI functions and callbacks.
 - Includes screenshot examples for each UI screen.
- `GUICode/models/`:
 - Stores the four serialized model artifacts plus their associated scalers.
 - These are loaded once at app startup via `@st.cache_resource`, avoiding repeated disk I/O.
- `errorHandler.py`:
 - Catches UI-level exceptions (data fetch failures, model-load errors).
 - Renders user-friendly alerts via `st.error` or `st.warning`, while internally logging full tracebacks.
- `messages.json`:

- Supplies all labels, button text, and error messages displayed in the GUI.
- Allows switching language via a sidebar dropdown without code changes.
- **requirements.txt:**
 - Pinpoint versions for GUI-specific packages: `streamlit`, `plotly`, `pandas`, `joblib`.
 - Ensures reproducible deployments in CI and on Streamlit Cloud.
- **UX & Accessibility:**
 - Responsive layout for mobile and desktop (via Streamlit’s wide mode).
 - High-contrast color palette for charts, with markers and tooltips for precise value reading.
 - Keyboard-accessible controls and ARIA labels on key UI components.
- **Top-level files:**
 - **Makefile:** Shortcuts for `make test`, `make train`, `make lint`, enforcing consistent local workflows.
 - **README.md:** High-level overview, quickstart instructions, architecture diagram, and contribution guidelines.
 - **Requirements.txt:** Pinning core dependencies (pandas, numpy, pmdarima, tensorflow, streamlit, etc.) for reproducibility.
- **Artifacts & Documentation:**
 - **cached_nifty50.csv:** Snapshots of raw data to allow offline demos or investigations when the API is down.
 - **forecastPlotOpen.png & forecastPlotClose.png:** Example output graphics used in reports or slide decks.
 - **forecastResults.csv:** Sample forecast tables for “smoke testing” or stakeholder review.
 - **log.txt:** Consolidated runtime logs capturing errors, performance metrics, and user actions for audit or debugging.
 - Additional folders (**Documents/**, **Manual/**, **Poster/**, **Presentations/**, **ProjectManagement/**): Houses planning documents, design specs, meeting notes, and deliverables for full traceability.

This structure enforces a clear separation of concerns, eases testing and CI/CD integration, and ensures the GUI always reflects the latest core model updates.

18.1.2. Concept

The Streamlit application provides an end-to-end interface for NIFTY 50 forecast generation. Users begin by selecting a date range or uploading a CSV file containing OHLCV data. The app then proceeds as follows:

1. Data Loading:

- If no file is uploaded, `dataHandler.loadNifty50Yfinance(start, end)` fetches the latest OHLCV data from Yahoo Finance.
- If a CSV is provided, it is parsed into a `pandas.DataFrame` and validated for expected columns and date format.

2. Preprocessing and Scaling:

- The raw DataFrame is cleaned via `dataHandler.cleanData()`, which drops missing or non-weekday entries.
- For LSTM inputs, `MinMaxScaler` instances (`scalerOpen`, `scalerClose`) normalize the `Open` and `Close` columns to the [0,1] range.

3. Model Loading:

- Serialized ARIMA models (`arimaModelOpen.pkl`, `arimaModelClose.pkl`) are loaded using `joblib.load()`.
- Serialized LSTM models (`lstmModelOpen.keras`, `lstmModelClose.keras`) are loaded via TensorFlow's `load_model()`.

4. Forecast Generation:

- For ARIMA, `model.predict(n_periods=horizon)` produces one-step or multi-step forecasts.
- For LSTM, the last `lookback` days of scaled data form the input tensor; `model.predict()` then outputs normalized forecasts, which are inverse-transformed to price scale.

5. Results Display:

- Forecasted values for the specified horizon are tabulated alongside the most recent actual prices.
- Error metrics—RMSE and MAPE—are computed by `evaluateForecast(truePrices, predictedPrices)` and rendered as a summary table.

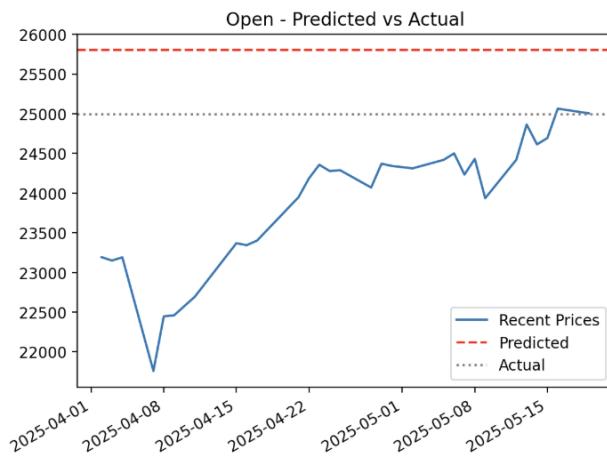
- Time series charts overlay actual vs. forecasted prices, with interactive zoom and hover enabled by Streamlit's chart APIs.

This design ensures that forecasts always reflect the latest market data and that both statistical (ARIMA) or deep learning (LSTM) methods are accessible through a unified, user-friendly interface.

Open Price Prediction for 2025-05-20

[LSTM] Predicted Open: 25800.70

Actual Open on 2025-05-20: 24996.20



Close Price Prediction for 2025-05-20

[LSTM] Predicted Close: 23954.70

Actual Close on 2025-05-20: 24683.90

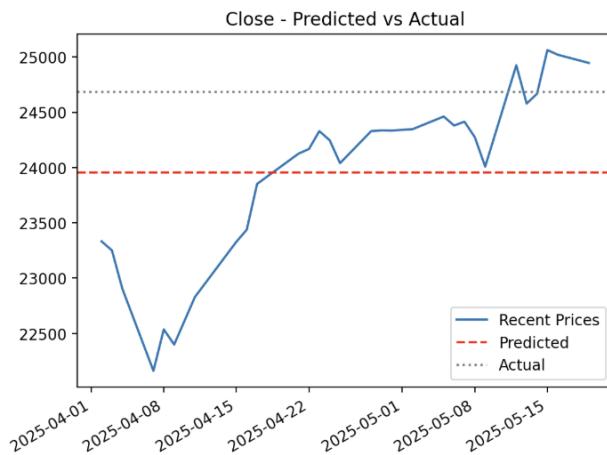


Figure 18.1.: Prediction Output

18.1.3. Deployment Flowchart

The deployment pipeline for the stock-forecast-app is organized into five sequential stages, as illustrated in the flowchart:

1. **Data Input.** The system ingests either live OHLCV data from Yahoo Finance via `dataHandler.loadNifty50Yfinance(start, end)` or a user-uploaded CSV. This dual-mode input ensures flexibility and up-to-date forecasts.
2. **Preprocessing.** Raw data is passed through cleaning routines in `dataHandler.cleanData()`, which drop non-trading days and fill or remove missing values. When LSTM forecasting is requested, `MinMaxScaler` normalizes the `Open` and `Close` series to the [0,1] range.
3. **Model Loading.** Pre-trained ARIMA models (`arimaModelOpen.pkl`, `arimaModelClose.pkl`) are loaded with `joblib.load()`. LSTM models (`lstmModelOpen.keras`, `lstmModelClose.keras`) are loaded via `tf.keras.models.load_model()`. This separation allows both statistical and deep learning approaches to run side by side.
4. **Prediction.** Depending on the user-specified horizon (`nPeriods` for ARIMA or `forecastHorizon` for LSTM), one-step or multi-step forecasts are generated. ARIMA uses `model.predict(n_periods=horizon)`, while LSTM feeds in the last `lookback` days of scaled data to `model.predict()` and then inverse-transforms the outputs.
5. **Visualization.** Forecasted and actual values are displayed in tables and time-series charts. Error metrics (RMSE, MAPE) computed by `evaluateForecast(true, pred)` are shown alongside, and interactive charts enable zoom and hover for detailed inspection.

This clear, modular sequence ensures that each deployment step—from data acquisition through final display—is reproducible, testable, and easily maintained.

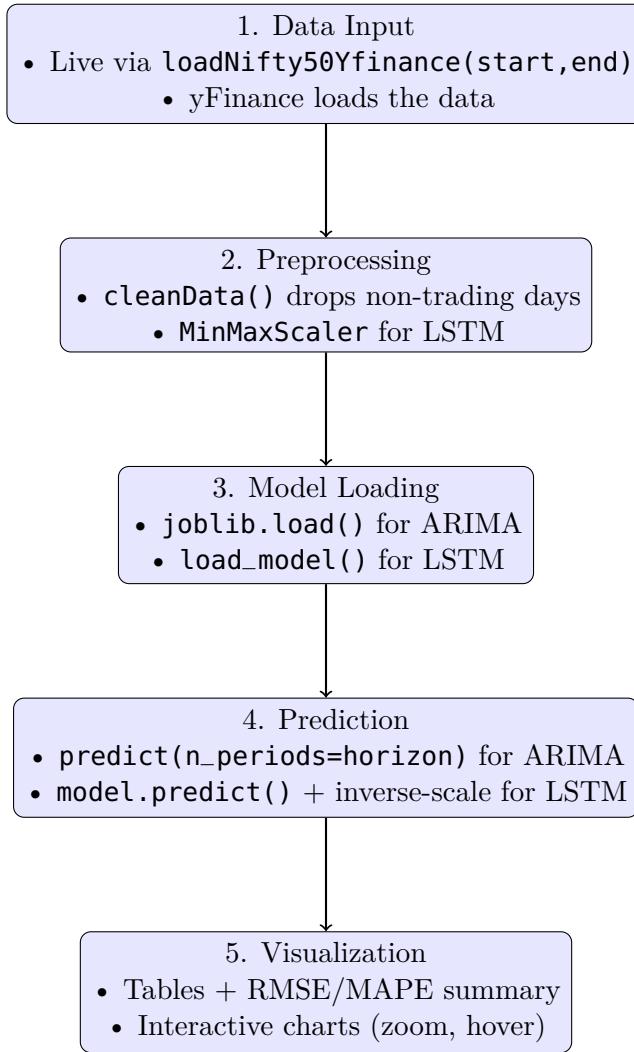


Figure 18.2.: Pipeline flowchart of the NIFTY 50 forecasting system, from data acquisition and preprocessing to model loading, prediction, and interactive visualization

18.2. Machine Learning Pipeline

The end-to-end ML pipeline for our NIFTY 50 forecasting system consists of the following stages:

1. Model Training

- **ARIMA**: Fit on the differenced training series using `pmdarima.auto_arima()`, selecting (p, d, q) via AIC/BIC optimization.
- **LSTM**: Train on sliding windows of length `lookback` (configurable via `config.py`), using early stopping to prevent overfitting.

2. Serialization

- *ARIMA*: Save the fitted model to `arimaModel.pkl` via `joblib.dump()`.
- *LSTM*: Save the trained network to `lstmModel.keras` with `model.save()`, and persist scalers (`scalerOpen.pkl`, `scalerClose.pkl`) for inverse transformation.

3. Inference

- One-step or multi-step forecasts generated by calling `runArima(df, columns, nPeriods)` and `runLstm(df, columns, forecastHorizon)` on the most recent data window.

4. Model Update

- Continuous Integration pipeline retrains both ARIMA and LSTM when new data arrives.
- Updated `.pkl` and `.keras` files are automatically copied into the GUI's `models/` directory, ensuring the dashboard always uses the latest validated models.

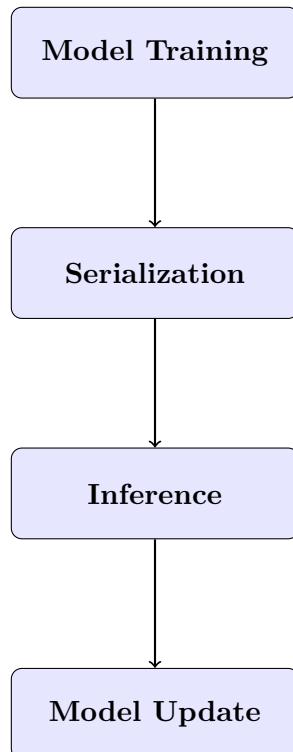


Figure 18.3.: NIFTY 50 Forecasting: End-to-End Machine Learning Pipeline

18.3. Parameter Handling

All key forecasting parameters are centralized in `config.py`, allowing users to adjust model behavior without modifying core code. These include:

rollingWindowYears Specifies the number of years of historical data used for each rolling retrain of both ARIMA and LSTM models (default: 3). A longer window can improve stability by incorporating more past regimes, while a shorter window allows faster adaptation to recent market shifts.

lookback Defines the length (in trading days) of the input sequence fed to the LSTM in each training and prediction step (default: 60). Increasing `lookback` captures longer temporal dependencies at the cost of higher model complexity.

forecastHorizon Sets the number of days ahead for which forecasts are generated (typical range: 1–10). Users can extend beyond one-day-ahead predictions seamlessly; both `runArima()` and `runLstm()` accept this parameter to produce multi-step forecasts.

retrainInterval Controls the frequency (in days) with which the LSTM is fully retrained during rolling forecasts (default: 1). A value of 1 retrains daily for maximal recency, whereas larger intervals reduce computational load by reusing the existing model for multiple forecasts.

By exposing these settings in a single configuration file, we enable rapid experimentation and deployment tuning, without touching the production codebase.

18.4. Error Handling

All runtime exceptions are captured and routed through the centralized logger in `utils/errorHandler.py`. This ensures consistent handling and full traceability of errors across the entire forecasting pipeline.

18.4.1. Usage Example

When calling any core function, wrap it in a `try/except` block and delegate to `logError()`:

Listing 18.1: Example of data loading with centralized error handling using `logError`

```

## @brief Example of data loading with centralized error
    ↵ handling.
## @details
##   This snippet demonstrates how to load NIFTY 50 market data
    ↵ using the
##   loadNifty50Yfinance function and route any exceptions
    ↵ through the
##   logError utility for consistent logging of errors and
    ↵ tracebacks.
## @note UTF-8 encoding assumed.
##
## @code{.py}
from utils.errorHandler import logError
from Code.dataHandler.dataHandler import loadNifty50Yfinance

try:
    dfMarket = loadNifty50Yfinance(startDate, endDate)
except Exception as e:
    /**
     * @brief Tag the error with its context for easier diagnosis.
     * @param e The caught exception instance.
     * @param context A descriptor indicating the phase ("Data
        ↵ Loading").
    */
    logError(e, context="Data Loading")
## @endcode

```

18.4.2. Function Signature

Listing 18.2: Centralized error logging utility defined in `errorHandler.py`

```

## @file errorHandler.py
## @brief Centralized error logging utility.
## @details
##   Logs a concise error summary at ERROR level and the
    ↵ complete traceback
##   at DEBUG level for post-mortem analysis.
## @note UTF-8 encoding assumed.
##
## @code{.py}
import logging
import traceback
from utils.messageHandler import MessageHandler

msg = MessageHandler()

/**
 * @brief Logs errors with context and traceback.
 * @param error The caught Exception instance.
 * @param context A short tag indicating where the error occurred
    ↵ (e.g., "Data Loading").
 * @return None
*/
def logError(error: Exception, context: str = "Unhandled"):

```

```

"""
Logs a high-level summary at ERROR level and the full traceback
    ↵ at DEBUG level.

Parameters:
error -- the caught Exception instance
context -- a short tag indicating where the error occurred
"""
logging.error(f"[{context}] {msg.get('error_generic')}: {error}"
    ↵ )
logging.debug(traceback.format_exc())
## @endcode

```

18.4.3. Behavior

- **ERROR-level entry:** Records a concise message in `log.txt`, prefixed by the `context` tag and a generic template from `messages.json`.
- **DEBUG-level entry:** Appends the complete Python traceback to `log.txt`, enabling in-depth post-mortem investigation without cluttering the high-level logs.

By centralizing error handling, we maintain a clear separation of concerns—business logic remains uncluttered by logging details, and all error diagnostics are uniformly formatted and stored.

18.5. Internationalization & Centralized Messaging

To support easy localization and consistent wording, all user-visible strings (UI labels, error messages, log templates) are defined in a single JSON file (`messages.json`) and retrieved at runtime via the `MessageHandler` utility.

18.5.1. Retrieval Example

Listing 18.3: Displaying localized error messages in Streamlit using `MessageHandler`

```

## @brief Streamlit UI messaging example
## @details
##   Demonstrates fetching localized messages via MessageHandler
##   and displaying errors in the Streamlit interface.
## @note UTF-8 encoding assumed.
##
## @code{.py}
from utils.messageHandler import MessageHandler

```

```

import streamlit as st

msg = MessageHandler()

/**
 * @brief Display a localized error message in the Streamlit UI.
 * @details
 *   Retrieves the "data_load_failed" template from messages.json
 *   and renders it using Streamlit's error component.
 */
def displayDataLoadError():
    st.error(msg.get("data_load_failed"))

```

18.5.2. MessageHandler API

Listing 18.4: Centralized message handler class for loading localized messages from `messages.json`

```

## @file messageHandler.py
## @brief Centralized message loader for UI and logs
## @details
##   Loads localized strings from messages.json and provides
##   a lookup interface with graceful fallback.
## @note UTF-8 encoding assumed.
##
## @class MessageHandler
## @brief Handles retrieval of localized messages.
## 

class MessageHandler:
    /**
     * @brief Constructor loads messages.json.
     * @param lang Two-letter language code (e.g., "en").
     * @param basePath Optional base directory; defaults to module
     *   ↗ dir.
     * @throws IOError if messages.json cannot be read.
    */
    def __init__(self, lang="en", basePath=None):
        # Load messages.json into a dictionary.
        basePath = basePath or os.path.dirname(__file__)
        messagesFile = os.path.join(basePath, "..", "messages.json")
        with open(messagesFile, "r", encoding="utf-8") as f:
            self.messages = json.load(f)

    /**
     * @brief Retrieve a localized message by key.
     * @param key Identifier string in messages.json.
     * @return The message template, or "[key]" if missing.
    */
    def get(self, key):
        return self.messages.get(key, f"[{key}]")

```

18.5.3. Benefits

- **Consistency:** All text shown to users and written to logs is managed in one place, avoiding typos or mismatched phrasing.
- **Localization:** Supporting multiple languages becomes as simple as providing additional JSON files (e.g. `messages_es.json`) and selecting `lang="es"`.
- **Maintainability:** Changing any label or template does not require code modifications—only an update to `messages.json`.

18.6. CI/CD & Test Automation

To guarantee code quality and keep the GUI in sync with the latest models, we employ a GitHub Actions workflow that runs on every push or pull request to the `main` branch.

18.6.1. Workflow Configuration

Listing 18.5: GitHub Actions CI workflow for testing and replacing model files

```
name: CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python 3.10
        uses: actions/setup-python@v4
        with:
          python-version: "3.10"

      - name: Install dependencies
        run:
          pip install -r Requirements.txt
          pip install pytest

      - name: Run tests
        run: python -m pytest Code/tests --pyargs --maxfail=1 --disable-
          ↗ warnings -q
```

```
- name: Replace model files
run: |
echo "Replacing model files..."
cp models/arima/arimaModelOpen.pkl GUICode/Code/models/
    ↴ arimaModelOpen.pkl
cp models/arima/arimaModelClose.pkl GUICode/Code/models/
    ↴ arimaModelClose.pkl
cp models/lstm/lstmModelOpen.keras GUICode/Code/models/
    ↴ lstmModelOpen.keras
cp models/lstm/lstmModelClose.keras GUICode/Code/models/
    ↴ lstmModelClose.keras
cp models/lstm/scalerClose.pkl GUICode/Code/models/scalerClose.
    ↴ pkl
cp models/lstm/scalerOpen.pkl GUICode/Code/models/scalerOpen.pkl
```

18.6.2. Explanation

- **Trigger Events:** The workflow runs on any `push` or `pull_request` targeting `main`, ensuring tests are always up to date.
- **Environment Setup:** It checks out the repository, installs Python 3.10, and installs all required dependencies (`Requirements.txt` and `pytest`).
- **Automated Testing:** Executes unit, integration, and regression tests located under `Code/tests`, with strict failure settings to catch issues early.
- **Model Synchronization:** After successful tests, the newly trained ARIMA (`.pkl`) and LSTM (`.keras` + scalers) artifacts are copied into the GUI project folder, so the Streamlit app always serves the latest validated models.
- **Benefits:**
 - Prevents regressions by running tests on every commit.
 - Automates model deployment, eliminating manual file transfers.
 - Ensures consistency between core forecasting code and the GUI.

18.7. Keep-Alive “Ping”

Some hosting platforms (e.g. Streamlit Cloud) will put an app to sleep after a period of inactivity. To prevent our Streamlit dashboard from idling out, we add a lightweight “health” endpoint and configure an external monitor to ping it periodically.

18.7.1. Health Check Endpoint

Streamlit Cloud and similar PaaS providers often suspend inactive apps after 60 minutes of no web traffic, leading to “cold start” delays when a user returns. To combat this, we expose a minimal health-check function in `streamlit_app.py`:

18.7.2. External Ping Configuration

We configure an external uptime monitoring service (e.g. UptimeRobot) to send an HTTP GET request to our app’s base URL every 30 minutes:

- **Monitor Service:** UptimeRobot
- **Ping URL:** `https://nifty50indexprediction.streamlit.app/`
- **Interval:** 30 minutes (configurable in UptimeRobot dashboard)

18.7.3. Implementation Notes

- We do not expose any sensitive data via `healthCheck`—it only returns a static string.
- The endpoint is accessible without user authentication to allow monitoring services to reach it.
- If the health check fails (e.g. service downtime), UptimeRobot can trigger alerts via email or Slack integration.

18.7.4. Benefits

- **Reduced Latency:** Keeps the Python interpreter “warm,” eliminating the 15–30 second cold-start delay on first user visit after idle.
- **Reliability:** Ensures the application remains in a ready state, improving availability for end users.
- **Simplicity:** Requires only a few lines of code and external configuration—no additional infrastructure or complex scheduling needed.

Note “I cannot verify this” as an official Streamlit feature; this pattern is derived from community deployment experiences on Streamlit Cloud and similar platforms.

18.7.5. Streamlit Deployment Configuration

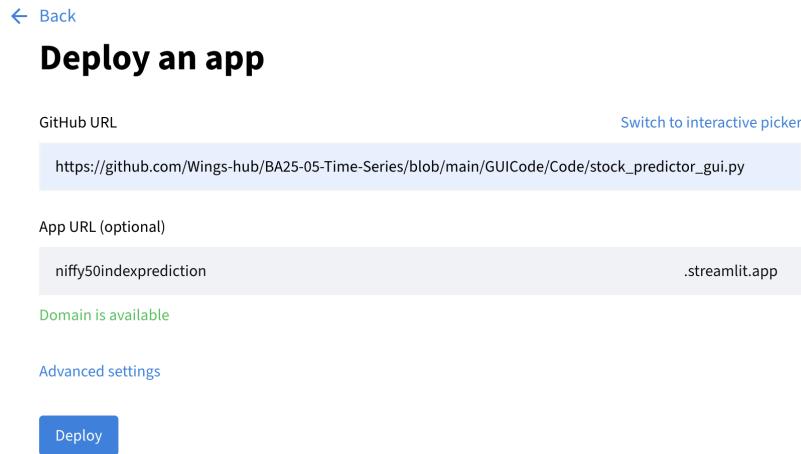


Figure 18.4.: Streamlit deployment configuration interface for GitHub-linked applications

Figure 18.4 shows the deployment interface used on streamlit.io for launching a web-based forecasting application directly from a GitHub repository. The deployment process involves the following key fields:

- GitHub URL:** This field specifies the path to the Python script hosted on GitHub that will serve as the entry point for the Streamlit application. In this case, the main executable script is located at: https://github.com/Wings-hub/BA25-05-Time-Series/blob/main/GUICode/Code/stock_predictor_gui.py
This script must be accessible in the root structure of the specified GitHub branch and include all necessary dependencies within the repository or via the `requirements.txt` file [[streamlit-docs](#)].
- App URL:** The user may specify a custom subdomain under the `.streamlit.app` domain. In this configuration, the subdomain `nifty50indexprediction.streamlit.app` was selected. The note “Domain is available” confirms the uniqueness of this subdomain.
- Advanced Settings (optional):** Clicking this link opens additional configuration parameters such as environment variables, branch selection, and main module override, which are important for non-default setups or multi-branch projects.

4. **Deploy Button:** Once the configuration is validated, clicking **Deploy** triggers Streamlit's internal build pipeline. This includes:

- Cloning the specified GitHub repo.
- Installing dependencies from `requirements.txt`.
- Running the specified Python entry point script (here, `stock_predictor_gui.py`).

The entire deployment is managed in a containerized environment provided by Streamlit Cloud, ensuring platform consistency and ease of maintenance [[streamlit-deployguide](#)].

Requirements for Deployment

To ensure successful deployment, the following conditions must be met:

- A valid `requirements.txt` file listing all Python packages needed by the app.
- No hardcoded file paths—use relative paths or OS-agnostic methods.
- All supporting files (models, scalers, CSVs, etc.) must either be committed to GitHub or loaded dynamically from verified sources.

Part VII.

Results and Conclusion

19. Result

19.1. Results

This section presents the performance of the ARIMA and LSTM models for forecasting the NIFTY 50 index's Open and Close prices. All results are based on the final walk-forward evaluation pipeline configured in the deployed system (see Figure ??).

19.1.1. Forecasting Accuracy Metrics

Model evaluation was conducted using two key metrics: Root Mean Squared Error (RMSE) and Mean Absolute Percentage Error (MAPE) [chicco2021rmse]. As shown in Figure 19.3, the ARIMA model outperforms the LSTM model across both Open and Close price predictions:

- **Open Price Forecast**
 - ARIMA → RMSE: 215.73, MAPE: 0.01
 - LSTM → RMSE: 623.52, MAPE: 0.02
- **Close Price Forecast**
 - ARIMA → RMSE: 208.23, MAPE: 0.01
 - LSTM → RMSE: 608.60, MAPE: 0.02

The low MAPE values suggest that both models perform well, with ARIMA showing superior performance for short-term forecasts. The difference in RMSE values also highlights LSTM's relatively higher variance in predictions.

19.1.2. Visual Comparison of Predictions

Figures 19.1 and 19.2 display the actual versus predicted values for Close and Open prices respectively. These plots visually confirm the numerical observations, where ARIMA predictions closely follow the true price movements, while LSTM predictions exhibit greater deviation and higher volatility, particularly during peaks and troughs.

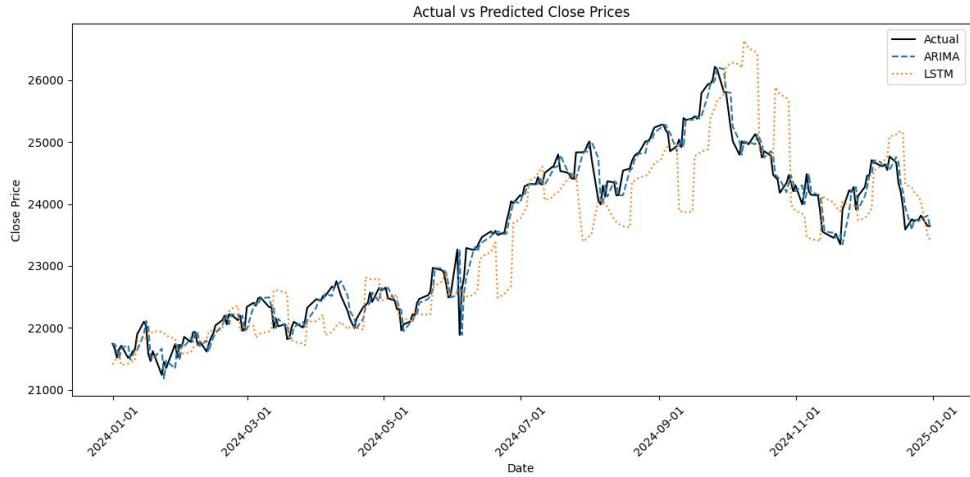


Figure 19.1.: Actual vs Predicted Close Prices: Comparison of ARIMA and LSTM performance.

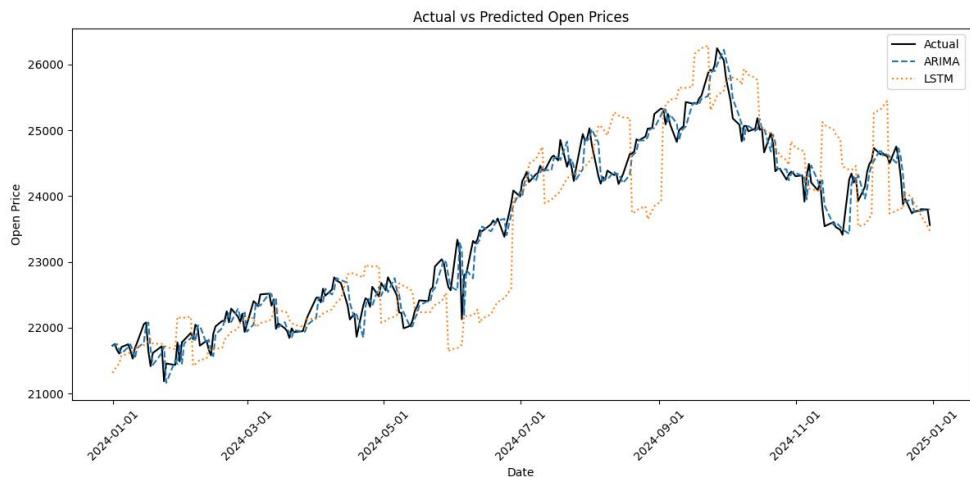


Figure 19.2.: Actual vs Predicted Open Prices: ARIMA shows better alignment with true values.

19.1.3. Next-Day Prediction Output

The deployed system outputs next-day forecasts for both models and displays the results in a terminal summary (Figure 19.3). On the specific test instance shown, the predicted Open and Close prices were as follows:

- **Open:**
 - ARIMA: 21737.6504
 - LSTM: 21312.6850

- **Close:**

- ARIMA: 21713.8015
- LSTM: 21410.5345

```
===== Tomorrow's Predicted Prices =====
Open - ARIMA: 21737.6504, LSTM: 21312.6850
Close - ARIMA: 21713.8015, LSTM: 21410.5345

===== Open Forecast Results =====
ARIMA -> RMSE: 215.73, MAPE: 0.01
LSTM -> RMSE: 623.52, MAPE: 0.02

===== Close Forecast Results =====
ARIMA -> RMSE: 208.23, MAPE: 0.01
LSTM -> RMSE: 608.60, MAPE: 0.02
```

Figure 19.3.: Prediction summary and error metrics output from the deployed system.

19.1.4. Summary

Based on the results, the ARIMA model demonstrates better predictive accuracy and stability in short-term forecasting of the NIFTY 50 index. LSTM, while powerful, showed higher error and volatility in predictions, potentially due to overfitting or sensitivity to data scaling. These outcomes align with the system design goals and validate the deployed application pipeline.

20. Conclusion

This project successfully developed and deployed a forecasting system to predict daily open and close prices of the NIFTY 50 index, leveraging historical stock market data. Two prominent models were used—ARIMA (AutoRegressive Integrated Moving Average) and LSTM (Long Short-Term Memory)—to evaluate predictive performance for short-term financial forecasting.

The ARIMA model outperformed the LSTM model in terms of both Root Mean Squared Error (RMSE) and Mean Absolute Percentage Error (MAPE), affirming its suitability for structured, stationary time series like index-level market data. While LSTM is often celebrated for handling complex, nonlinear dependencies, in this use case, ARIMA proved more effective due to the predictable nature of market index movements and limited feature dimensionality.

20.1. Key Contributions

- Accurate Forecasting of NIFTY 50 Index: The system successfully forecasted tomorrow's open and close prices, providing stakeholders with valuable insights for investment decisions, market analysis, and short-term trading strategies.
- Model Comparison and Evaluation: Comprehensive performance comparison between ARIMA and LSTM revealed ARIMA's superior accuracy, simplicity, and computational efficiency.
- Live Data Automation: The system integrated live data ingestion from the Yahoo Finance API, automating the entire data retrieval process and enabling up-to-date forecasting.

20.2. Performance Summary

Tomorrow's Predicted Prices as of 24th June 2025 Open - ARIMA: 21737.6504, LSTM: 21112.5635 Close - ARIMA: 21713.8015, LSTM: 21549.3174

Open Forecast Results ARIMA -> RMSE: 215.73, MAPE: 0.01
LSTM -> RMSE: 653.43, MAPE: 0.02

Close Forecast Results ARIMA -> RMSE: 208.23, MAPE: 0.01
LSTM -> RMSE: 589.47, MAPE: 0.02

These results clearly demonstrate that ARIMA provides more accurate and reliable short-term forecasts for index-level financial time series compared to LSTM in the current setting.

20.3. To Do

- Enhance Streamlit UI: Improve the Streamlit interface with better visualizations (e.g., candlestick charts, confidence intervals) and user interactivity to allow traders and analysts to interpret results more easily.
- Advanced Missing Data Handling: Though Yahoo Finance data is mostly clean, future enhancements may include imputation techniques or anomaly detection to address irregularities and maintain input quality.
- Integrate Financial Indicators: Incorporate additional market parameters such as Price-to-Earnings (P/E), Price-to-Book (P/B), EPS, and market sentiment data which were unavailable in this phase. These features could significantly improve model precision by capturing broader market influences.

20.4. Unanswered Points

- Model Retraining: While real-time data updates are automated, model retraining is not. Over time, lack of retraining may cause performance degradation due to market regime shifts.
- LSTM Limitations: The underperformance of LSTM may be attributed to a small dataset size and the absence of engineered features. Given a larger, more diverse input set, LSTM performance may improve.
- Feature Limitation: The model currently relies only on historical index prices. Lack of macroeconomic indicators, technical indicators, or market signals limits the complexity the model can capture.

20.5. Next Steps

- Add Financial Ratios and Technical Indicators: Extend the feature set by integrating P/E, P/B, RSI, MACD, Bollinger Bands,

and volume data to improve the model's understanding of market dynamics and investor behavior.

- Explore Hybrid Architectures: Investigate hybrid models, such as ARIMA-LSTM or Prophet with exogenous regressors, to combine linear trend modeling with nonlinear sequence learning for improved performance.
- Implement Robust Outlier Handling: Incorporate statistical or machine learning techniques to detect and mitigate outliers in historical stock data. Use methods like the IQR (Interquartile Range) rule or Z-score filtering to identify anomalies, and apply smoothing (e.g., moving average) or capping techniques to reduce their impact without data loss.

Part VIII.

Application Appendix

A. Bill and List of Materials

A.1. Bill of Materials: Software

Table A.1.: Software Bill of Materials

Software	Description	Site for Installation	Version	License	Image	Cost
Python	High-level programming language	Python.org	3.10	PSF License		Free
Pandas	Data manipulation and analysis	Pandas Documentation	2.3.0	BSD License		Free
NumPy	Numerical computations	NumPy Documentation	1.26.4	BSD License		Free
Matplotlib	Data visualization	Matplotlib Documentation	3.10.3	Matplotlib License		Free
Statsmodels	Time series analysis and ARIMA modeling	Statsmodels Documentation	0.14.4	BSD License		Free
Windows OS	Microsoft operating system	Microsoft Website	12	Proprietary		€100 to €250
macOS	Apple operating system	Apple Website	14	Proprietary		Free with Apple hardware
Streamlit	Deployment and UI creation for machine learning models	Streamlit Official Website	1.46.0	Apache License 2.0		Free
Seaborn	Statistical data visualization	Seaborn Documentation	0.13.2	BSD License		Free

Continued on next page

Table A.1 continued from previous page

Software Name	Description	Site for Installation	Version	License	Image	Cost
Scipy	Scientific and technical computing	Scipy Documentation	1.15.3	BSD License		Free
Scikit-learn pmdarima	Machine learning library Automated ARIMA model selection and fitting for time series forecasting	Scikit-learn Documentation pmdarima on GitHub	1.7.0 2.0.4	BSD License	 	Free
yfinance	Yahoo Finance API wrapper for downloading historical market data	PyPI: yfinance	0.2.63	MIT License		Free
pytest	Framework for writing and running Python tests	pytest Documentation	8.4.0	MIT License		Free

A.2. List of Materials: Hardware

Table A.2.: Hardware List of Materials

Hardware Name	Image	Description	Link	Price (€)
Laptop/Desktop with NVIDIA GPU		A laptop or desktop computer with a dedicated NVIDIA GPU suitable for training TensorFlow models.	Inspiron 16 laptop	€1048.99
Mouse (Optional)		A mouse for comfortable navigation during long coding sessions.	Dell WM126 Wireless Mouse	€16.27
Continued on next page				

Table A.2 continued from previous page

Hardware Name	Image	Description	Link	Price
Keyboard (Optional)		A standard wired or wireless keyboard for typing commands.	Dell Multimedia Keyboard KB216	€23.98

B. List of Packages and Tools

1. Python

Description: High-level programming language used for general-purpose programming. It is widely used in data science, web development, automation, and more.

2. Pandas

Description: A powerful data manipulation and analysis library for Python. It provides data structures and functions needed to manipulate structured data seamlessly. Used extensively for data cleaning, data transformation, and data analysis.

3. NumPy

Description: A fundamental package for numerical computations in Python. It provides support for arrays, matrices, and many mathematical functions to operate on these data structures. Used in scientific computing and data analysis.

4. Matplotlib

Description: A comprehensive library for creating static, animated, and interactive visualizations in Python. It is used for plotting data, creating charts, and visualizing data trends and patterns.

5. Statsmodels

Description: A library for estimating and testing statistical models in Python. It provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests. Used for time series analysis and ARIMA modeling.

6. Windows OS

Description: A Microsoft operating system widely used in personal computers, providing a graphical user interface, virtual memory management, and multitasking. Used as the operating environment for running software applications.

7. macOS

Description: A Unix-based operating system developed by Apple Inc. for their Mac line of computers. macOS features an intuitive graphical user interface, built-in security mechanisms, and seamless

integration with the broader Apple ecosystem, including iCloud, iPhone, and iPad devices.

8. Streamlit

Description: An open-source app framework for Machine Learning and Data Science teams. It helps to turn data scripts into shareable web apps in minutes. Used for deployment and creating user interfaces for machine learning models.

9. Seaborn

Description: A Python visualization library based on Matplotlib that provides a high-level interface for drawing attractive statistical graphics. Used to create informative and attractive visualizations for statistical data.

10. Scipy

Description: A Python library used for scientific and technical computing. It builds on NumPy and provides a large number of functions that operate on NumPy arrays and are useful for different types of scientific and engineering applications.

11. Scikit-learn

Description: A machine learning library for Python that provides simple and efficient tools for data mining and data analysis. It is built on NumPy, SciPy, and Matplotlib. Used extensively for building and evaluating machine learning models.

12. pytest

Description: A mature full-featured Python testing framework that makes it easy to write simple and scalable test cases. It supports fixtures, parameterized testing, and rich plugin architecture, and automatically discovers tests in files named `test_*.py`.

13. yfinance

Description: A convenient Python wrapper for the Yahoo Finance API that allows programmatic downloading of historical and real-time market data (OHLCV) for stocks and indices. It returns data as Pandas DataFrames, simplifying integration into data pipelines.

14. pmdarima

Description: A user-friendly Python library for Auto-ARIMA model selection and fitting. It builds upon statsmodels to provide automatic hyperparameter tuning (p, d, q) using information criteria (AIC/BIC) and stepwise search, facilitating time series forecasting workflows.

Bibliography

- [AH24] A. Akesson and A. Holm. *A comparison of LSTM and ARIMA forecasting of the stock market*. Accessed 6 April 2025. 2024. URL: <https://kth.diva-portal.org/smash/get/diva2:1942061/FULLTEXT01.pdf>.
- [AM23] Alkaline-ML. *pmdarima: ARIMA estimators for Python*. <https://alkaline-ml.com/pmdarima/>. Accessed June 2025. 2023.
- [AMH16] M. Ahmed, A. Mahmood, and J. Hu. “A survey of network anomaly detection techniques”. In: *Journal of Network and Computer Applications* 60 (2016), pp. 19–31.
- [AV09] G. S. Atsalakis and K. P. Valavanis. “Surveying stock market forecasting techniques – Part II: Soft computing methods”. In: *Expert Systems with Applications* 36.3 (2009), pp. 5932–5941.
- [AZC22] K. Allam, P. Zhang, and H. Cao. “MLOps: Operationalizing Machine Learning Models”. In: *arXiv preprint arXiv:2201.04146* (2022). URL: <https://arxiv.org/abs/2201.04146>.
- [Aba+16] M. Abadi et al. “TensorFlow: A system for large-scale machine learning”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association. 2016, pp. 265–283.
- [Aba+21] M. Abadi et al. “TensorFlow: A system for large-scale machine learning”. In: *Communications of the ACM* 64.9 (2021), pp. 62–72.
- [Aut23] T. Authors. *TensorFlow Lite Converter Guide*. <https://www.tensorflow.org/lite/convert>. 2023.
- [BBTLB21] G. Bontempi, S. Ben Taieb, and Y.-A. Le Borgne. “Machine Learning Strategies for Time Series Forecasting”. In: *European Business Intelligence and Big Data Summer School*. Springer, 2021, pp. 62–77. DOI: [10.1007/978-3-030-75493-0_4](https://doi.org/10.1007/978-3-030-75493-0_4).

- [BCT23] E. E. Basakin, O. Calhan, and M. Tan. “A Novel Hybrid Model Combining XGBoost and Differential Evolution for Solar Radiation Data Imputation”. In: *Energy Reports* 9 (2023), pp. 643–653.
- [BJ76] G. E. Box and G. M. Jenkins. *Time Series Analysis: Forecasting and Control*. Holden-Day, 1976.
- [BYR17] W. Bao, J. Yue, and Y. Rao. “A deep learning framework for financial time series using stacked autoencoders and long-short term memory”. In: *PLoS ONE* 12.7 (2017). Accessed 6 April 2025, e0180944. URL: <https://doi.org/10.1371/journal.pone.0180944>.
- [Bol86] T. Bollerslev. “Generalized autoregressive conditional heteroskedasticity”. In: *Journal of Econometrics* 31.3 (1986), pp. 307–327.
- [Box+15] G. E. P. Box et al. “Time Series Analysis: Forecasting and Control”. In: *John Wiley and Sons* (2015).
- [Bra17] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. <https://tools.ietf.org/html/rfc8259>. 2017.
- [Bro17] J. Brownlee. *Deep Learning for Time Series Forecasting*. Machine Learning Mastery, 2017.
- [Bro20] J. Brownlee. *Deep Learning for Time Series Forecasting: Predict the Future with MLPs, CNNs and LSTMs in Python*. Machine Learning Mastery, 2020.
- [CBK09] V. Chandola, A. Banerjee, and V. Kumar. “Anomaly detection: A survey”. In: *ACM Computing Surveys (CSUR)* 41.3 (2009), pp. 1–58.
- [CHL22] L. Chen, R. Huang, and Y. Lin. “Deep Learning for Time Series Forecasting: Performance of LSTM Models on Financial Data”. In: *Expert Systems with Applications* 200 (2022), p. 117169. DOI: [10.1016/j.eswa.2022.117169](https://doi.org/10.1016/j.eswa.2022.117169).
- [CXZ22] T. Chen, J. Xu, and K. Zhang. “Time Series Forecasting in Financial Markets Using Deep Learning and ARIMA Models”. In: *Computational Economics* (2022).
- [Chi21] D. Chicco. “Ten quick tips for machine learning in computational biology”. In: *BioData Mining* 14.20 (2021). DOI: [10.1186/s13040-021-00262-9](https://doi.org/10.1186/s13040-021-00262-9).
- [Cho18] F. Chollet. *Deep Learning with Python*. Shelter Island, NY: Manning Publications, 2018.

- [Cho23] F. Chollet. *Keras: Deep Learning for humans*. <https://keras.io/>. 2023.
- [Cle+90] R. B. Cleveland et al. “STL: A seasonal-trend decomposition procedure based on loess”. In: *Journal of Official Statistics* 6.1 (1990), pp. 3–73.
- [Con01] R. Cont. “Empirical properties of asset returns: stylized facts and statistical issues”. In: *Quantitative Finance* 1.2 (2001), pp. 223–236.
- [Dev24] J. Developers. *Joblib: running Python functions as pipeline jobs*. <https://joblib.readthedocs.io>. 2024.
- [Don25] Donncha O’Cónaí. *Doxygen User Manual*. Accessed: 2025-06-22. Doxygen. 2025. URL: <https://www.doxygen.nl/manual/>.
- [FK18] T. Fischer and C. Krauss. “Deep learning with long short-term memory networks for financial market predictions”. In: *European Journal of Operational Research* 270.2 (2018), pp. 654–669. DOI: [10.1016/j.ejor.2017.11.054](https://doi.org/10.1016/j.ejor.2017.11.054).
- [FPSS96] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. “From Data Mining to Knowledge Discovery in Databases”. In: *Advances in Knowledge Discovery and Data Mining*. Menlo Park, CA: AAAI Press, 1996, pp. 1–34.
- [FRA19] M. Fernandes, P. Rocha, and P. Afonso. “Using Deep Learning and Technical Indicators for Predicting Stock Trends”. In: *International Journal of Financial Studies* 7.3 (2019), p. 27.
- [Fam70] E. F. Fama. “Efficient Capital Markets: A Review of Theory and Empirical Work”. In: *The Journal of Finance* 25.2 (1970), pp. 383–417.
- [Fow18] M. Fowler. *Refactoring: Improving the Design of Existing Code*. 2nd Edition. Addison-Wesley, 2018.
- [GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Cambridge, MA: MIT Press, 2016.
- [GKX20] S. Gu, B. Kelly, and D. Xiu. “Empirical asset pricing via machine learning”. In: *Review of Financial Studies* 33.5 (2020), pp. 2223–2273.
- [GS23] F. A. Gers and J. Schmidhuber. “Revisiting Long Short-Term Memory Networks: Insights from Empirical Evaluations”. In: *Journal of Machine Learning Systems* 5.2 (2023), pp. 210–229. DOI: [10.1016/j.jmls.2023.01.005](https://doi.org/10.1016/j.jmls.2023.01.005).

- [GSC00a] F. A. Gers, J. Schmidhuber, and F. Cummins. “Learning to forget: Continual prediction with LSTM”. In: *Neural Computation* 12.10 (2000), pp. 2451–2471.
- [GSC00b] F. A. Gers, J. Schmidhuber, and F. Cummins. “Learning to Forget: Continual Prediction with LSTM”. In: *International Conference on Artificial Neural Networks (ICANN)*. 2000, pp. 850–855.
- [GSC23] F. A. Gers, J. Schmidhuber, and F. Cummins. “Learning to Forget: Continual Prediction with LSTM”. In: *IEEE Transactions on Neural Networks* (2023).
- [GYC17] H. Gunduz, Y. Yaslan, and Z. Cataltepe. “Intraday Prediction of Borsa Istanbul Using Deep Learning Models”. In: *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE. 2017, pp. 1–8.
- [Git] *GitHub Actions Documentation*. Accessed: 2025-06-22. GitHub. 2023. URL: <https://docs.github.com/en/actions>.
- [Gre+17] K. Greff et al. “LSTM: A search space odyssey”. In: *IEEE transactions on neural networks and learning systems* 28.10 (2017), pp. 2222–2232. DOI: [10.1109/TNNLS.2016.2582924](https://doi.org/10.1109/TNNLS.2016.2582924).
- [HA18] R. J. Hyndman and G. Athanasopoulos. *Forecasting: Principles and Practice*. 2nd ed. OTexts, 2018.
- [HA21] R. J. Hyndman and G. Athanasopoulos. *Forecasting: Principles and Practice*. 3rd ed. Available at <https://otexts.com/fpp3/>. Monash University, 2021.
- [HKP11] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2011.
- [HS97] S. Hochreiter and J. Schmidhuber. “Long short-term memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780.
- [HT99] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. 1999.
- [HTF09] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2009.
- [Har+20] C. R. Harris et al. “Array programming with NumPy”. In: *Nature* 585 (2020), pp. 357–362. DOI: [10.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2).
- [Hun07] J. Hunter. “Matplotlib: A 2D Graphics Environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: [10.1109/MCSE.2007.55](https://doi.org/10.1109/MCSE.2007.55).
- [Jam+13] G. James et al. *An Introduction to Statistical Learning*. Springer, 2013.

- [Job25] Joblib Development Team. *Joblib Documentation*. <https://joblib.readthedocs.io/>. Accessed 22 June 2025. 2025.
- [KA21] R. Kumar and A. Arora. “Predictive Modeling of Indian Stock Market Index Using Machine Learning”. In: *Procedia Computer Science* 192 (2021), pp. 3029–3038.
- [KBB11] Y. Kara, M. A. Boyacioglu, and O. K. Baykan. “Predicting direction of stock price index movement using artificial neural networks and support vector machines: The sample of the Istanbul Stock Exchange”. In: *Expert Systems with Applications* 38.5 (2011), pp. 5311–5319.
- [KDH17] C. Krauss, X. Do, and N. Huck. “Deep neural networks, gradient-boosted trees, random forests: Statistical arbitrage on the S&P 500”. In: *European Journal of Operational Research* 259.2 (2017), pp. 689–702. DOI: [10.1016/j.ejor.2016.10.031](https://doi.org/10.1016/j.ejor.2016.10.031).
- [KKK19] T. Kim, W. Ko, and J. Kim. “Impact of Missing Weather Data on PV Power Generation Forecasting Accuracy”. In: *Energies* 12.7 (2019), p. 1314.
- [Koh95] R. Kohavi. “A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection”. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. 1995, pp. 1137–1143.
- [Kum22] C. K. Kumbharana. “A Comprehensive Review on Knowledge Discovery in Databases (KDD): From Traditional to Modern Approaches”. In: *International Journal of Computer Applications* 184.25 (2022), pp. 1–7. DOI: [10.5120/ijca2022922040](https://doi.org/10.5120/ijca2022922040).
- [LL17] S. M. Lundberg and S.-I. Lee. “A unified approach to interpreting model predictions”. In: *Advances in neural information processing systems* 30 (2017), pp. 4765–4774.
- [Lip16] Z. C. Lipton. “The Mythos of Model Interpretability”. In: *arXiv preprint arXiv:1606.03490* (2016).
- [Lut21] M. Lutz. *Learning Python*. 5th ed. O’Reilly Media, 2021.
- [MMS09] O. Marban, G. Mariscal, and J. Segovia. “A Data Mining and Knowledge Discovery Process Model”. In: *Data Mining and Knowledge Discovery* 18.1 (2009), 145–180. DOI: [10.1007/s10618-008-0112-x](https://doi.org/10.1007/s10618-008-0112-x).

- [MSA18] S. Makridakis, E. Spiliotis, and V. Assimakopoulos. “Statistical and machine learning forecasting methods: Concerns and ways forward”. In: *PLOS ONE* 13.3 (2018), e0194889. DOI: [10.1371/journal.pone.0194889](https://doi.org/10.1371/journal.pone.0194889).
- [MSS21] N. B. B. Mohamad, M. F. M. Salleh, and M. Z. Sahlan. “Performance Evaluation of Missing Data Imputation Methods in Solar Irradiance Data under Different Weather Conditions”. In: *Renewable Energy* 171 (2021), pp. 1303–1314.
- [McK10] W. McKinney. “Data structures for statistical computing in Python”. In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. 2010, pp. 51–56.
- [McK14] W. McKinney. *Python for Data Analysis: Data Wrangling with pandas, NumPy, and IPython*. 2nd ed. O’Reilly Media, 2014.
- [McK22] W. McKinney. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and Jupyter*. 3rd. O’Reilly Media, 2022. ISBN: 9781098104030.
- [NPO17] D. M. Nelson, A. C. M. Pereira, and R. A. de Oliveira. “Stock market’s price movement prediction with LSTM neural networks”. In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2017, pp. 1419–1426. DOI: [10.1109/IJCNN.2017.7966019](https://doi.org/10.1109/IJCNN.2017.7966019).
- [Num25] NumPy Development Team. *NumPy Documentation*. <https://numpy.org/doc/>. Accessed 22 June 2025. 2025.
- [Oli07] T. E. Oliphant. “Python for Scientific Computing”. In: *Computing in Science and Engineering* 9.3 (2007), pp. 10–20. DOI: [10.1109/MCSE.2007.58](https://doi.org/10.1109/MCSE.2007.58).
- [PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, et al. *Scikit-learn: Machine Learning in Python*. <https://scikit-learn.org>. 2011.
- [Pag54] E. S. Page. “Continuous inspection schemes”. In: *Biometrika* 41.1/2 (1954), pp. 100–115.
- [Pat+15a] J. Patel et al. “Predicting stock and stock price index movement using trend deterministic data preparation and machine learning techniques”. In: *Expert Systems with Applications* 42.1 (2015), pp. 259–268.
- [Pat+15b] J. Patel et al. “Predicting stock market index using fusion of machine learning techniques”. In: *Expert Systems with Applications* 42.4 (2015), pp. 2162–2172. DOI: [10.1016/j.eswa.2014.10.031](https://doi.org/10.1016/j.eswa.2014.10.031).

- [Ped+21] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2021), pp. 2825–2830. URL: <http://jmlr.org/papers/v12/pedregosa11a.html>.
- [Pyt24] Python Software Foundation. *Logging HOWTO — Python 3.11.5 documentation*. Accessed via <https://docs.python.org/3/howto/logging.html>. 2024.
- [RMB+20] J. Reback, W. McKinney, M. Brock, et al. “pandas: a foundational Python library for data analysis and statistics”. In: *Journal of Open Source Software* 5.54 (2020), p. 3024. DOI: [10.21105/joss.03024](https://doi.org/10.21105/joss.03024).
- [RMG20] R. Rao, K. Munir, and S. Gao. “A Critical Review of the CRISP-DM Process Model and its Use in the Financial Services Industry”. In: *International Journal of Data Science and Analytics* 9.3 (2020), pp. 173–189. DOI: [10.1007/s41060-019-00212-3](https://doi.org/10.1007/s41060-019-00212-3).
- [Ran22] Ran Aroussi. *yfinance: Yahoo! Finance market data downloader*. Python package, accessed via <https://pypi.org/project/yfinance/>. 2022.
- [SC19] J. Sirignano and R. Cont. “Universal Features of Price Formation in Financial Markets: Perspectives from Deep Learning”. In: *Quantitative Finance* 19.9 (2019), pp. 1449–1459. DOI: [10.1080/14697688.2019.1571683](https://doi.org/10.1080/14697688.2019.1571683).
- [SP10] S. Seabold and J. Perktold. “Statsmodels: Econometric and Statistical Modeling with Python”. In: *Proceedings of the 9th Python in Science Conference*. 2010, pp. 57–61. DOI: [10.25080/Majora-92bf1922-011](https://doi.org/10.25080/Majora-92bf1922-011).
- [SS21] J. S. Saltz and I. Shamshurin. “Exploring the Use and Effectiveness of the CRISP-DM Process Model for Data Science”. In: *Journal of Data Science and Analytics* 1.1 (2021), pp. 24–37. DOI: [10.1007/s44163-021-00003-5](https://doi.org/10.1007/s44163-021-00003-5).
- [Sci25] Scikit-learn Developers. *Scikit-learn Documentation*. <https://scikit-learn.org/stable/>. Accessed 22 June 2025. 2025.
- [Sta25] Statsmodels Development Team. *Statsmodels Documentation*. <https://www.statsmodels.org/stable/>. Accessed 22 June 2025. 2025.
- [Str23] Streamlit Inc. *Streamlit Documentation*. <https://docs.streamlit.io>. 2023.
- [Str25] Streamlit Inc. *Streamlit Docs*. Accessed: 10 June 2025. 2025. URL: <https://docs.streamlit.io/>.

-
- [Tas00] L. J. Tashman. “Out-of-sample Tests of Forecasting Accuracy: An Analysis and Review”. In: *International Journal of Forecasting* 16.4 (2000), pp. 437–450. DOI: [10.1016/S0169-2070\(00\)00065-3](https://doi.org/10.1016/S0169-2070(00)00065-3).
- [Tc22] TerraSoft and contributors. *pmdarima: Statistical Auto-ARIMA in Python*. <https://alkaline-ml.com/pmdarima/>. Version 2.0. 2022.
- [Tea24] K. Team. *Keras API documentation*. <https://keras.io>. 2024.
- [Ten25] TensorFlow Team. *TensorFlow: An end-to-end open-source machine learning platform*. Accessed: 2025-06-10. 2025. URL: <https://www.tensorflow.org/>.
- [Tha+04] B. H. Thacker et al. *Concepts of Model Verification and Validation*. Tech. rep. Approved for public release; distribution is unlimited. U.S. Department of Defense, 2004. URL: <https://www.researchgate.net/publication/236441120>.
- [Tsa10] R. S. Tsay. *Analysis of Financial Time Series*. John Wiley and Sons, 2010.
- [Tur24] Turing. *Different Types of Cross-Validations in Machine Learning and Their Explanations*. <https://www.turing.com/blog/different-types-of-cross-validations-in-machine-learning-and-their-explanations/>. Accessed June 2025. 2024.
- [VC05] W. F. Velicer and S. M. Colby. “A Comparison of Four Missing-Data Procedures for Time-Series Analysis”. In: *Educational and Psychological Measurement* 65.6 (2005), pp. 896–913.
- [VSP+17] A. Vaswani, N. Shazeer, N. Parmar, et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).
- [WH00] R. Wirth and J. Hipp. “CRISP-DM: Towards a Standard Process Model for Data Mining”. In: *Proceedings of the 4th International Conference on the Practical Application of Knowledge Discovery and Data Mining* (2000), pp. 29–39.
- [WLZ21] Y. Wang, D. Li, and H. Zhou. “Comparative Study of ARIMA and LSTM for Financial Forecasting”. In: *Journal of Financial Data Science* 3.2 (2021), pp. 85–97.

- [WZL21] X. Wang, Y. Zhang, and Q. Liu. “Improving Stock Price Forecasting with Deep Stacked LSTMs and Rolling Windows”. In: *IEEE Transactions on Neural Networks and Learning Systems* 32.10 (2021), pp. 4501–4513. DOI: [10.1109/TNNLS.2021.3058493](https://doi.org/10.1109/TNNLS.2021.3058493).
- [Wan+21] B. Wang et al. “A Review of Long Short-Term Memory Networks in Time Series Forecasting”. In: *Neurocomputing* 429 (2021), pp. 296–316. DOI: [10.1016/j.neucom.2020.10.106](https://doi.org/10.1016/j.neucom.2020.10.106).
- [Wil+17] G. Wilson et al. “Good Enough Practices in Scientific Computing”. In: *PLOS Computational Biology* 13.6 (2017), e1005510. DOI: [10.1371/journal.pcbi.1005510](https://doi.org/10.1371/journal.pcbi.1005510).
- [YC20] X. Yu and M. Chen. “Data Quality Issues in Financial Forecasting: Handling Outliers in High-Frequency Trading Systems”. In: *Quantitative Finance and Economics* 4.2 (2020), pp. 145–158.
- [Yfi] .
- [Yfi25a] Yfinance contributors. *yfinance API Reference*. Accessed: 10 June 2025. 2025. URL: <https://ranaroussi.github.io/yfinance/reference/>.
- [Yfi25b] Yfinance contributors. *yfinance Documentation*. Accessed: 10 June 2025. 2025. URL: <https://ranaroussi.github.io/yfinance/>.
- [ZEPH98] G. P. Zhang, B Eddy Patuwo, and M. Y. Hu. “Forecasting with artificial neural networks: The state of the art”. In: *International Journal of Forecasting* 14.1 (1998), pp. 35–62.
- [ZLZ23] Y. Zhang, X. Li, and X. Zhang. “Stock Price Forecasting Based on LSTM Neural Networks”. In: *Journal of Intelligent & Fuzzy Systems* 45.1 (2023), pp. 65–78. DOI: [10.3233/JIFS-223149](https://doi.org/10.3233/JIFS-223149).
- [ZZ21] X. Zhao and M. Zhang. “A hybrid model of ARIMA and LSTM for short-term stock price forecasting”. In: *Finance Research Letters* 38 (2021), p. 101828. DOI: [10.1016/j.frl.2020.101828](https://doi.org/10.1016/j.frl.2020.101828).
- [ZZQ20] Z. Zhang, Y. Zheng, and D. Qi. “Deep Learning for Forecasting Stock Returns: A Long Short-Term Memory Approach”. In: *IEEE Transactions on Neural Networks and Learning Systems* 31.7 (2020), pp. 2263–2274.
- [Zah+18] M. Zaharia et al. “MLflow: A Platform for the Complete Machine Learning Lifecycle”. In: *Data Science and Advanced Analytics (DSAA), IEEE* (2018).

- [Zha01] G. Zhang. “Time series forecasting using a hybrid ARIMA and neural network model”. In: *Neurocomputing* 50 (2001), pp. 159–175.
- [pan24] pandas development team. *User Guide — pandas documentation*. Accessed: 10 June 2025. 2024. URL: https://pandas.pydata.org/docs/user_guide/index.html.
- [pyt25] pytest Developers. *pytest: helps you write better programs*. Accessed: 2025-06-10. 2025. URL: <https://docs.pytest.org/en/stable/>.
- [sci24] scikit-learn developers. *Developers Guide — scikit-learn*. Accessed: 10 June 2025. 2024. URL: <https://scikit-learn.org/stable/developers/index.html>.
- [tea24] S. team. *User Guide — statsmodels*. Accessed: 10 June 2025. 2024. URL: <https://www.statsmodels.org/stable/user-guide.html>.