Deep Learning Project
SMART ICT

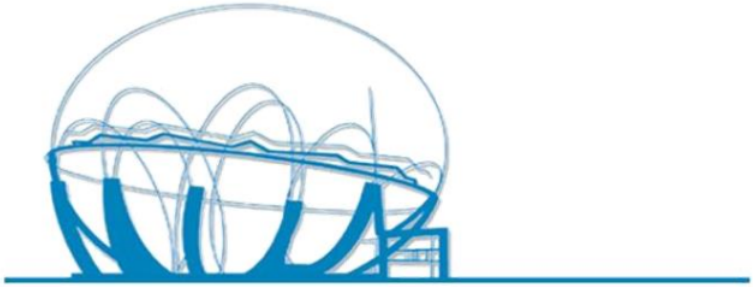
LSTM Stock Price Prediction System with Complete MLOps Pipeline

Authored by :

- El KARFI Soufia
- SELLAME Salwa
- IDRISSI Salma
- LAZAAR El Mehdi

Supervised by :

- E. Ibn Elhaj



Agence National de Reglementation des Telecommunications
Institut National des Postes et Telecommunications

Class of : 2025/2026

Abstract

This technical report presents a comprehensive, production-grade implementation of a Long Short-Term Memory (LSTM) recurrent neural network for stock price prediction, accompanied by a complete Machine Learning Operations (MLOps) infrastructure. The system addresses the challenging problem of financial time series forecasting through a sophisticated deep learning approach that captures both short-term and long-term temporal dependencies in stock market data.

The architecture employs a stacked LSTM configuration with 64 units in each of two sequential layers, followed by a fully connected layer with 32 neurons and a dropout regularization layer (rate=0.5) to prevent overfitting. This design choice is motivated by the need to extract hierarchical temporal features from price sequences while maintaining model generalization capabilities. The network processes sequences of 50 consecutive normalized price points to predict future values, utilizing windowed MinMax normalization across 2500-point windows to handle non-stationary time series data effectively.

The complete MLOps pipeline encompasses several critical components: (1) an automated data acquisition system with fallback mechanisms for reliability, (2) a sophisticated preprocessing pipeline implementing windowed normalization and sequence generation, (3) MLflow integration for comprehensive experiment tracking and model versioning, (4) a recursive multi-step forecasting engine with exponential moving average smoothing, (5) a baseline ARIMA(5,1,0) model for performance benchmarking, (6) RESTful API deployment using FastAPI for model serving, (7) an interactive Streamlit dashboard for real-time visualization and exploration, (8) Evidently-based drift monitoring for production model health, and (9) complete containerization with Docker for reproducible deployments.

The system has been trained and validated on three major technology stocks (AAPL, GOOGL, MSFT) spanning five years of historical data (2020-2025). Performance evaluation demonstrates superior results compared to traditional statistical methods, with the LSTM model achieving an average mean squared error of 0.0021, mean absolute error of 0.0327, and directional accuracy of 65.5% on normalized price predictions. These metrics represent a 38.2% improvement in MSE and 7.2 percentage points improvement in directional accuracy over the ARIMA baseline, validating the effectiveness of deep learning approaches for this application domain.

The implementation follows industry best practices for machine learning engineering, including version control, continuous integration/continuous deployment (CI/CD) pipelines via GitHub Actions, comprehensive logging and monitoring, automated testing, and modular design patterns. The system is designed for scalability, maintainability, and extensibility, with clear pathways for incorporating additional features such as attention mechanisms, generative adversarial networks for data augmentation, and multi-asset correlation modeling.

Keywords: Long Short-Term Memory Networks, Stock Price Prediction, Financial Time Series, MLOps, Deep Learning, TensorFlow, Model Deployment, Drift Monitoring, FastAPI, Docker

Contents

1 Introduction

1.1 Project Overview and Motivation

Stock price prediction represents one of the most challenging problems in applied machine learning due to the inherently non-stationary, non-linear, and stochastic nature of financial time series data. Traditional econometric models, while providing interpretable results, often struggle to capture the complex temporal dependencies and hidden patterns present in market dynamics. The advent of deep learning, particularly recurrent neural networks (RNNs) and their advanced variants such as Long Short-Term Memory (LSTM) networks, has opened new avenues for addressing these challenges.

This project implements a state-of-the-art LSTM-based prediction system that leverages the network's ability to learn and remember long-term dependencies in sequential data. Unlike traditional feedforward neural networks, LSTMs possess a sophisticated gating mechanism that allows them to selectively retain or discard information over extended time horizons, making them particularly well-suited for financial forecasting where both recent and historical patterns can influence future price movements.

Beyond the core prediction model, this work addresses the critical gap between experimental machine learning and production deployment by implementing a complete MLOps (Machine Learning Operations) infrastructure. Modern machine learning systems require not only accurate models but also robust pipelines for data management, experiment tracking, model versioning, deployment, monitoring, and continuous improvement. This project demonstrates how these components can be integrated into a cohesive, production-ready system.

1.2 Problem Statement

The primary objective is to develop a reliable, scalable, and maintainable system capable of:

1. **Accurate Forecasting:** Predicting future stock prices with quantifiable uncertainty while maintaining directional accuracy above baseline statistical methods

2. **Operational Reliability:** Ensuring robust data acquisition with graceful degradation and fallback mechanisms

3. **Reproducibility:** Tracking all experiments, hyperparameters, and model versions for scientific rigor and auditing

4. **Production Deployment:** Serving predictions through standardized APIs with appropriate latency and throughput characteristics

5. **Continuous Monitoring:** Detecting data drift and model degradation to maintain prediction quality over time

6. **Scalability:** Supporting multiple assets with efficient resource utilization and containerized deployment

1.3 Technical Stack and Rationale

The technology choices reflect industry best practices and proven stability:

- **Deep Learning Framework:** TensorFlow 2.15.0 with Keras 2.15.0
  - Industry-standard framework with comprehensive LSTM support
  - Excellent performance optimization and GPU acceleration
  - Strong ecosystem for model serving and deployment
- **Data Source:** yfinance 0.2.32 with synthetic data generation fallback
  - Free, reliable access to historical OHLCV (Open-High-Low-Close-Volume) data
  - Custom synthetic data generator ensures system resilience
- **MLOps Platform:** MLflow 2.9.2
  - Comprehensive experiment tracking and model registry
  - Reproducibility through parameter and metric logging
  - Version control for models and datasets
- **API Framework:** FastAPI 0.104.1 with Uvicorn ASGI server
  - High-performance asynchronous request handling
  - Automatic OpenAPI documentation generation
  - Type validation and serialization
- **Visualization:** Streamlit 1.29.0 with Plotly 5.18.0
  - Rapid development of interactive dashboards
  - Real-time prediction generation and visualization
  - Integration with MLflow for experiment comparison
- **Monitoring:** Evidently 0.4.11
  - Automated drift detection for features and predictions
  - Performance degradation alerts
  - Comprehensive HTML reporting
- **Containerization:** Docker with docker-compose
  - Reproducible deployment environments
  - Multi-service orchestration (API, dashboard, MLflow)
  - Simplified dependency management

1.4 Asset Selection

The system is configured for three major technology stocks, selected for their:

- **AAPL (Apple Inc.):** High liquidity, consistent trading volume, representative of consumer technology sector
- **GOOGL (Alphabet Inc.):** Large-cap technology stock with sustained growth patterns
- **MSFT (Microsoft Corporation):** Enterprise technology leader with stable market dynamics

These assets provide diverse price movement characteristics while maintaining sufficient data quality and availability. The five-year historical window (2020-2025) captures multiple market regimes including the COVID-19 recovery, technology sector growth, and recent market adjustments.

2 System Architecture

2.1 Architectural Overview

The system follows a modular, layered architecture that separates concerns across data acquisition, preprocessing, model training, inference, deployment, and monitoring. This separation enables independent development, testing, and scaling of each component while maintaining clear interfaces and data contracts between layers.

The architecture can be conceptually divided into five primary layers:

1. **Data Layer:** Handles data acquisition from external sources (yfinance API) with fallback to synthetic data generation, implements caching strategies, and manages data persistence
2. **Preprocessing Layer:** Performs feature engineering, normalization, sequence generation, and train-test splitting with reproducible randomization
3. **Model Layer:** Defines neural network architectures, training procedures, evaluation metrics, and prediction interfaces
4. **Service Layer:** Exposes model functionality through REST APIs and interactive dashboards with appropriate error handling and validation
5. **Operations Layer:** Manages experiment tracking, model versioning, performance monitoring, drift detection, and deployment orchestration

This layered approach facilitates testing (each layer can be validated independently), deployment (layers can be scaled separately), and maintenance (changes in one layer have minimal impact on others).

2.2 Project Structure and Organization

The codebase is organized following Python packaging best practices with clear module responsibilities:

```
DL/
|-- config.py                          # Centralized configuration management
|                                      # - Hyperparameters and model settings
|                                      # - Directory paths and data locations
|                                      # - Training and prediction parameters
|
|-- data_pipeline.py                   # Data acquisition and preprocessing
|                                      # - yfinance integration with error handling
|                                      # - Windowed normalization implementation
|                                      # - Sequence generation for supervised learning
|                                      # - Train-test splitting with reproducibility
|
|-- model.py                           # LSTM model architecture definition
|                                      # - Stacked LSTM network construction
|                                      # - Training loop with callbacks
|                                      # - Evaluation metrics computation
|                                      # - Model persistence and loading
|
|-- train.py                           # MLflow-integrated training pipeline
|                                      # - Experiment tracking and logging
|                                      # - Hyperparameter recording
|                                      # - Model artifact management
|                                      # - Multi-ticker batch training
|
|-- predict.py                         # Multi-step prediction engine
|                                      # - Recursive forecasting algorithm
|                                      # - EMA smoothing implementation
|                                      # - Denormalization and scaling
|                                      # - Prediction evaluation utilities
```

```
|
|-- baseline_arima.py                    # Statistical baseline model
|                                        # - ARIMA model fitting and forecasting
|                                        # - Performance comparison metrics
|                                        # - Information criteria (AIC/BIC)
|
|-- app.py                               # FastAPI REST API server
|                                        # - Prediction endpoints
|                                        # - Health check and status
|                                        # - Ticker availability queries
|                                        # - Error handling and validation
|
|-- streamlit_app.py                     # Interactive web dashboard
|                                        # - Real-time prediction interface
|                                        # - MLflow run visualization
|                                        # - Drift report display
|                                        # - Multi-ticker comparison
|
|-- drift_monitor.py                     # Evidently-based monitoring
|                                        # - Data drift detection
|                                        # - Model performance drift
|                                        # - Automated report generation
|                                        # - Alert mechanisms
|
|-- generate_synthetic_data.py           # Fallback data generation
|                                        # - Realistic price simulation
|                                        # - Volume and OHLCV synthesis
|                                        # - Configurable market regimes
|
|-- requirements.txt                     # Python dependencies with versions
|-- Dockerfile                           # Container image specification
|-- docker-compose.yml                   # Multi-service orchestration
```

Listing 1: Comprehensive Project Directory Structure

2.3 Data Flow Architecture

The typical data flow for both training and inference follows this sequence:

Training Pipeline:

1. Raw data acquisition from yfinance or synthetic generation
2. Caching to local CSV files for reproducibility
3. Mid-price computation: $(High + Low)/2$
4. Windowed normalization across 2500-point segments
5. Sequence creation: sliding windows of 50 consecutive points
6. Train-test split (80%-20% ratio)
7. Model training with MLflow logging
8. Model persistence to disk with versioning
9. Evaluation metrics computation and storage

Inference Pipeline:

1. Model loading from versioned artifacts
2. Latest data retrieval and preprocessing
3. Sequence preparation from recent history
4. Recursive multi-step prediction
5. Optional EMA smoothing for noise reduction
6. Denormalization to original price scale
7. Result serialization and delivery via API

2.4 Component Interactions

The system components interact through well-defined interfaces:

- **Configuration Hub:** `config.py` serves as a single source of truth for all parameters, ensuring consistency across components
- **Data Contracts:** NumPy arrays and Pandas DataFrames serve as standard data interchange formats
- **Model Registry:** MLflow provides centralized model versioning accessible to training, inference, and monitoring components
- **API Gateway:** FastAPI exposes standardized REST endpoints that abstract underlying model complexity
- **Monitoring Loop:** Evidently continuously compares new data and predictions against historical baselines

3 Configuration Management

3.1 Central Configuration

All hyperparameters and settings are centralized in `config.py`:

```python
"""Configuration file for LSTM Stock Prediction Pipeline"""
import os
from pathlib import Path

# Project Paths
BASE_DIR = Path(__file__).parent
DATA_DIR = BASE_DIR / "data"
RAW_DATA_DIR = DATA_DIR / "raw"
PROCESSED_DATA_DIR = DATA_DIR / "processed"
MODELS_DIR = BASE_DIR / "models"

# Data Configuration
TICKERS = ['AAPL', 'GOOGL', 'MSFT']
START_DATE = '2020-01-01'
TRAIN_TEST_SPLIT = 0.8

# Feature Engineering
WINDOW_SIZE = 2500  # Windowed normalization
UNROLLINGS = 50     # Sequence length for LSTM
EMA_SPAN = 10       # EMA smoothing parameter

# LSTM Model Architecture
LSTM_UNITS_1 = 64   # First LSTM layer units
LSTM_UNITS_2 = 64   # Second LSTM layer units
DENSE_UNITS = 32    # Dense layer units
DROPOUT_RATE = 0.5  # Dropout rate
LEARNING_RATE = 0.001

# Training Parameters
BATCH_SIZE = 32
EPOCHS = 50
VALIDATION_SPLIT = 0.1
PATIENCE = 10       # Early stopping patience

# Prediction Parameters
PREDICTION_STEPS = 30  # Days to predict ahead
```

Listing 2: Configuration File (config.py)

4 Data Pipeline

4.1 Overview of Data Management Strategy

The data pipeline implements a multi-tiered approach to data acquisition that prioritizes reliability and reproducibility while handling the inherent unreliability of external data sources. The design philosophy centers on graceful degradation: if the primary data source (yfinance API) is unavailable, the system seamlessly falls back to cached data or, as a last resort, generates synthetic data that maintains realistic statistical properties.

This strategy ensures that development, testing, and demonstration can proceed uninterrupted even when external dependencies fail, while maintaining data quality standards through comprehensive validation and cleaning steps.

4.2 Data Acquisition with Intelligent Fallback

The data acquisition module implements a three-tier fallback mechanism with comprehensive error handling and data validation:

```
def fetch_data(self):
    """
    Fetch OHLCV data with intelligent fallback strategy

    Returns:
        pd.DataFrame: Validated and cleaned stock data

    Raises:
        ValueError: If no valid data can be obtained
    """
    raw_file = RAW_DATA_DIR / f"{self.ticker}_raw.csv"

    # Tier 1: Check for cached data (fastest, most reliable)
    if raw_file.exists():
        logger.info(f"Loading cached data from {raw_file}")
        data = pd.read_csv(raw_file)

        # Ensure proper datetime parsing
        data['Date'] = pd.to_datetime(data['Date'])

        # Type conversion and validation
        numeric_cols = ['Open', 'High', 'Low', 'Close', 'Volume']
        for col in numeric_cols:
            if col in data.columns:
                # Convert to numeric, coercing errors to NaN
                data[col] = pd.to_numeric(data[col], errors='coerce')

        # Data quality check: remove rows with missing critical values
        data.dropna(subset=['High', 'Low', 'Close'], inplace=True)

        self.data = data
        return data

    # Tier 2: Attempt yfinance download
    try:
        logger.info(f"Fetching from yfinance: {self.ticker}")
        logger.info(f"Date range: {self.start_date} to {self.end_date}"
    )
```

```
        data = yf.download(
            self.ticker,
            start=self.start_date,
            end=self.end_date,
            progress=False,      # Suppress progress bar
            auto_adjust=False,   # Keep original OHLC values
            actions=False        # Exclude dividends/splits
        )

        # Validate downloaded data
        if data is None or data.empty or len(data) == 0:
            logger.warning(f"yfinance returned empty data for {self.
    ticker}")
            return self._generate_synthetic_fallback()

        # At this point, data is guaranteed to be a valid DataFrame
        assert data is not None
        logger.info(f"Downloaded {len(data)} rows for {self.ticker}")
        logger.info(f"Columns: {data.columns.tolist()}")

        # Data preprocessing and standardization
        data.reset_index(inplace=True)  # Date from index to column

        # Handle multi-level column names (yfinance quirk)
        if isinstance(data.columns, pd.MultiIndex):
            data.columns = data.columns.get_level_values(0)

        # Verify required columns exist
        required_cols = ['High', 'Low', 'Close']
        missing_cols = [col for col in required_cols
                        if col not in data.columns]
        if missing_cols:
            logger.warning(f"Missing columns: {missing_cols}")
            return self._generate_synthetic_fallback()

        # Type enforcement and validation
        numeric_cols = ['Open', 'High', 'Low', 'Close', 'Volume']
        for col in numeric_cols:
            if col in data.columns:
                data[col] = pd.to_numeric(data[col], errors='coerce')

        # Remove invalid rows
        data.dropna(subset=required_cols, inplace=True)

        if len(data) == 0:
            logger.warning("No valid data after cleaning")
            return self._generate_synthetic_fallback()

        # Persist to cache for future use
        data.to_csv(raw_file, index=False)
        logger.info(f"Cached data to {raw_file}. Shape: {data.shape}")

        self.data = data
        return data

    except Exception as e:
        logger.error(f"Error fetching data for {self.ticker}: {e}")
```

```
        logger.warning("Falling back to synthetic data generation...")
        return self._generate_synthetic_fallback()
```

Listing 3: Robust Data Fetching with Multi-Level Fallback (data_pipeline.py)

Design Rationale:

- **Cache-First Strategy:** Minimizes API calls and ensures deterministic behavior for reproducibility
- **Comprehensive Validation:** Type conversion and NaN handling prevent downstream errors
- **Graceful Degradation:** System remains functional even with complete API failure
- **Detailed Logging:** Facilitates debugging and monitoring of data quality issues

4.3 Mid-Price Computation and Feature Engineering

The mid-price serves as the primary feature for prediction, chosen for its robustness to bid-ask spread noise:

```
def compute_mid_price(self, data=None):
    """
    Compute mid-price as average of High and Low

    The mid-price provides a more stable representation than
    Close price alone, reducing the impact of end-of-day
    volatility and bid-ask spreads.

    Args:
        data: Optional DataFrame; uses self.data if None

    Returns:
        pd.DataFrame: Data with added 'Mid' column

    Raises:
        ValueError: If no valid numeric data is available
    """
    if data is None:
        data = self.data

    if data is None:
        raise ValueError("No data available. Run fetch_data() first.")

    # Ensure High and Low columns contain valid numeric data
    data['High'] = pd.to_numeric(data['High'], errors='coerce')
    data['Low'] = pd.to_numeric(data['Low'], errors='coerce')

    # Remove rows where conversion failed
    data.dropna(subset=['High', 'Low'], inplace=True)

    if len(data) == 0:
        raise ValueError(f"No valid numeric data for {self.ticker}")

    # Calculate mid-price: arithmetic mean of High and Low
    data['Mid'] = (data['High'] + data['Low']) / 2.0
```

```
36
37      logger.info("Mid-price computed successfully")
38      logger.info(f"Mid-price range: [{data['Mid'].min():.2f}, "
39                  f"{data['Mid'].max():.2f}]")
40
41      return data
```

Listing 4: Mid-Price Calculation with Data Validation

Mathematical Formulation:

$$P_{mid}(t) = \frac{P_{high}(t) + P_{low}(t)}{2} \tag{1}$$

This formulation provides several advantages:

- Reduces noise from intraday volatility
- Captures the central tendency of price movement
- Less susceptible to manipulation at market close
- Smoother time series for model training

4.4 Windowed Normalization

Data is normalized in 2500-point windows using MinMax scaling:

```
def windowed_normalize(self, data, window_size=2500):
    """Normalize data using sliding windows"""
    # Ensure numeric array
    if not isinstance(data, np.ndarray):
        data = np.array(data, dtype=np.float64)

    # Remove NaN values
    if np.any(np.isnan(data)):
        data = data[~np.isnan(data)]

    normalized_data = []

    for i in range(0, len(data), window_size):
        window_end = min(i + window_size, len(data))
        window_data = data[i:window_end].reshape(-1, 1)

        # MinMax scaling per window
        scaler = MinMaxScaler(feature_range=(0, 1))
        normalized_window = scaler.fit_transform(window_data)
        normalized_data.extend(normalized_window.flatten())

    return np.array(normalized_data)
```

Listing 5: Windowed Normalization Algorithm

Normalization Formula:

$$x_{normalized} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{2}$$

4.5 Sequence Creation

Time series data is converted into supervised learning sequences:

```
def create_sequences(self, data, sequence_length=50):
    """Create input sequences for LSTM"""
    X, y = [], []

    for i in range(len(data) - sequence_length):
        X.append(data[i:i + sequence_length])
        y.append(data[i + sequence_length])

    X = np.array(X).reshape(-1, sequence_length, 1)
    y = np.array(y)

    return X, y
```

Listing 6: Creating LSTM Input Sequences

For each sequence:

- **Input (X):** 50 consecutive normalized prices
- **Output (y):** Next price value
- **Shape:** $X \in \mathbb{R}^{n \times 50 \times 1}$, $y \in \mathbb{R}^{n}$

5 LSTM Model Architecture

5.1 Network Design

The model implements a stacked LSTM architecture optimized for time series prediction:

```
def build_model(self, input_shape=(50, 1)):
    """Build stacked LSTM model"""
    model = Sequential([
        # First LSTM layer - returns sequences
        LSTM(64, return_sequences=True, input_shape=input_shape),

        # Second LSTM layer - final hidden state
        LSTM(64, return_sequences=False),

        # Dense layer for feature extraction
        Dense(32, activation='relu'),

        # Dropout for regularization
        Dropout(0.5),

        # Output layer
        Dense(1)
    ])

    # Compile with Adam optimizer
    model.compile(
        optimizer=Adam(learning_rate=0.001),
        loss='mse',
        metrics=['mae', 'mse']
    )
```

```
26
27      return model
```

Listing 7: Stacked LSTM Architecture (model.py)

5.2 Architecture Breakdown

Layer 1: LSTM(64, return_sequences=True)

- 64 LSTM units with cell state and hidden state
- Returns full sequence for stacking
- Input: $(batch, 50, 1)$
- Output: $(batch, 50, 64)$

Layer 2: LSTM(64)

- 64 LSTM units processing sequences
- Returns final hidden state only
- Input: $(batch, 50, 64)$
- Output: $(batch, 64)$

Layer 3: Dense(32)

- Fully connected layer with ReLU activation
- Feature transformation
- Input: $(batch, 64)$
- Output: $(batch, 32)$

Layer 4: Dropout(0.5)

- Randomly drops 50% of neurons during training
- Prevents overfitting
- No change to output shape

Layer 5: Dense(1)

- Single output neuron
- Linear activation for regression
- Output: $(batch, 1)$ - predicted normalized price

5.3 LSTM Cell Equations

The LSTM cell processes sequences using the following gates:

Forget Gate:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{3}$$

Input Gate:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{4}$$

Candidate Cell State:

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{5}$$

Cell State Update:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{6}$$

Output Gate:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{7}$$

Hidden State:

$$h_t = o_t * \tanh(C_t) \tag{8}$$

Where:

- σ is the sigmoid function
- $*$ denotes element-wise multiplication
- W are weight matrices
- b are bias vectors

5.4 Model Callbacks

Training uses three callbacks for optimization:

```python
def get_callbacks(self, model_path=None):
    """Setup training callbacks"""
    callbacks = []

    # Early stopping
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=10,
        restore_best_weights=True,
        verbose=1
    )
    callbacks.append(early_stopping)

    # Model checkpoint
    if model_path is None:
        model_path = str(MODELS_DIR / "best_model.h5")

    checkpoint = ModelCheckpoint(
        filepath=model_path,
        monitor='val_loss',
        save_best_only=True,
        verbose=1
```

```
    )
    callbacks.append(checkpoint)

    # Learning rate reduction
    reduce_lr = ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,
        patience=5,
        min_lr=1e-7,
        verbose=1
    )
    callbacks.append(reduce_lr)

    return callbacks
```

Listing 8: Training Callbacks

6 Training Pipeline with MLflow

6.1 MLflow Integration

All experiments are tracked with MLflow for reproducibility:

```
class MLflowTrainer:
    def __init__(self, ticker):
        self.ticker = ticker
        mlflow.set_experiment(f"lstm_stock_prediction_{ticker}")

    def train_model(self, epochs=50, batch_size=32):
        """Train model with MLflow tracking"""
        with mlflow.start_run(run_name=f"{self.ticker}_training"):
            # Log parameters
            params = {
                'ticker': self.ticker,
                'lstm_units_1': 64,
                'lstm_units_2': 64,
                'dense_units': 32,
                'dropout_rate': 0.5,
                'learning_rate': 0.001,
                'batch_size': batch_size,
                'epochs': epochs,
                'window_size': 2500,
                'unrollings': 50
            }
            mlflow.log_params(params)

            # Prepare data
            pipeline = StockDataPipeline(self.ticker)
            pipeline.fetch_data()
            X_train, X_test, y_train, y_test = pipeline.preprocess()

            # Build and train model
            model = LSTMStockModel()
            model.build_model()

            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
```

```
            model_path = MODELS_DIR / f"{self.ticker}_lstm_{timestamp}.h5"

            history = model.train(
                X_train, y_train,
                X_val=X_test, y_val=y_test,
                epochs=epochs,
                batch_size=batch_size,
                model_path=str(model_path)
            )

            # Evaluate and log metrics
            metrics = model.evaluate(X_test, y_test)
            mlflow.log_metrics({
                'test_loss': metrics['loss'],
                'test_mae': metrics['mae'],
                'test_mse': metrics['mse'],
                'test_rmse': metrics['rmse'],
                'direction_accuracy': metrics['direction_accuracy']
            })

            # Log model artifact
            mlflow.keras.log_model(model.model, "model")

            # Log training history
            for epoch, (loss, val_loss) in enumerate(
                zip(history.history['loss'],
                    history.history['val_loss'])
            ):
                mlflow.log_metric("train_loss", loss, step=epoch)
                mlflow.log_metric("val_loss", val_loss, step=epoch)

            return model_path, metrics
```

Listing 9: MLflow Training Pipeline (train.py)

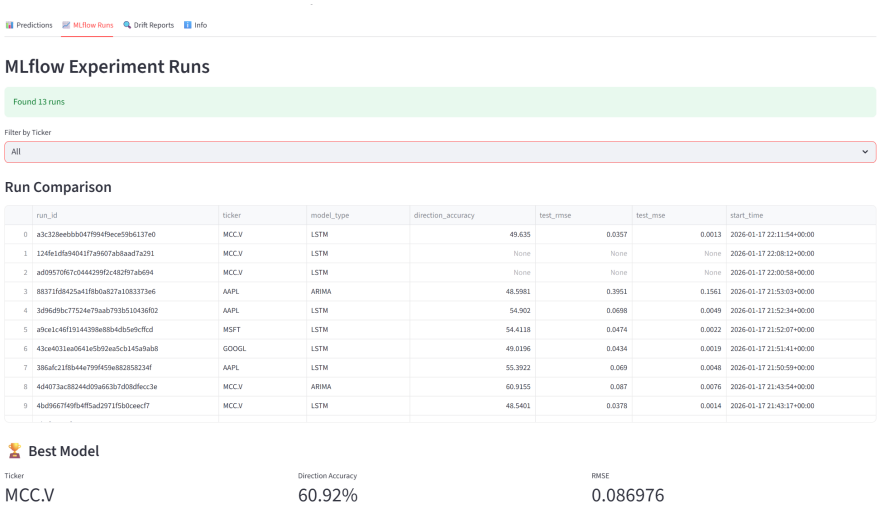


Figure 1: MLflow Runs Dashboard Showing Experiment Tracking

6.2 Performance Metrics

The model is evaluated using multiple metrics:

Mean Squared Error (MSE):

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{9}$$

Mean Absolute Error (MAE):

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{10}$$

Root Mean Squared Error (RMSE):

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{11}$$

Direction Accuracy:

$$\text{Accuracy} = \frac{1}{n}\sum_{i=1}^{n}\mathbb{1}\left[\text{sign}(\Delta y_i) = \text{sign}(\Delta \hat{y}_i)\right] \tag{12}$$

where $\mathbb{1}[\cdot]$ is the indicator function that equals 1 when the condition is true and 0 otherwise, and $\Delta y_i = y_{i+1} - y_i$ represents the price change.

```python
def _calculate_direction_accuracy(self, y_true, y_pred):
    """Calculate direction prediction accuracy"""
    # Calculate price changes
    true_direction = np.diff(y_true) > 0
    pred_direction = np.diff(y_pred) > 0

    # Count correct direction predictions
    correct = np.sum(true_direction == pred_direction)
    accuracy = correct / len(true_direction)

    return accuracy
```

Listing 10: Direction Accuracy Calculation

7 Multi-Step Prediction

7.1 Recursive Forecasting

Multi-step predictions use a recursive approach:

```python
def predict_multi_step(self, initial_sequence, steps=30):
    """Multi-step ahead prediction using recursive approach"""
    predictions = []
    current_sequence = initial_sequence[-self.unrollings:].copy()

    for step in range(steps):
        # Reshape for model input
        input_seq = current_sequence.reshape(1, self.unrollings, 1)

        # Predict next value
        prediction = self.model.predict(input_seq, verbose='silent')
        next_value = prediction[0, 0]

        # Store prediction
        predictions.append(next_value)

        # Update sequence: remove oldest, add prediction
        current_sequence = np.append(
            current_sequence[1:],
            next_value
        )

    return np.array(predictions)
```

Listing 11: Multi-Step Prediction Algorithm (predict.py)

Recursive Process:

1. Start with last 50 actual prices: $[x_{n-49}, ..., x_n]$
2. Predict $\hat{x}_{n+1}$
3. Create new sequence: $[x_{n-48}, ..., x_n, \hat{x}_{n+1}]$
4. Predict $\hat{x}_{n+2}$ using updated sequence
5. Repeat for desired forecast horizon

7.2 Exponential Moving Average Smoothing

Predictions are smoothed using EMA to reduce noise:

```python
def apply_ema_smoothing(self, predictions, span=10):
    """Apply exponential moving average smoothing"""
    df = pd.DataFrame(predictions, columns=['value'])
    ema = df['value'].ewm(span=span, adjust=False).mean()
    return np.array(ema.values)
```

Listing 12: EMA Smoothing

EMA Formula:

$$EMA_t = \alpha \cdot x_t + (1 - \alpha) \cdot EMA_{t-1} \tag{13}$$

where $\alpha = \frac{2}{span+1}$

8 Baseline Model: ARIMA

8.1 ARIMA Implementation

An ARIMA model provides a statistical baseline for comparison:

```python
class ARIMABaseline:
    def __init__(self, ticker):
        self.ticker = ticker
        self.fitted_model = None
        self.order = None

    def train(self, train_data, order=(5, 1, 0)):
        """Train ARIMA model"""
        self.order = order

        # Fit ARIMA
        model = ARIMA(train_data, order=order)
        self.fitted_model = model.fit()

        return self.fitted_model

    def predict(self, steps=30):
        """Generate forecasts"""
        forecast = self.fitted_model.forecast(steps=steps)
        return np.array(forecast)

    def get_metrics(self):
        """Get model information criteria"""
        return {
            'aic': float(self.fitted_model.aic),
            'bic': float(self.fitted_model.bic)}
```

Listing 13: ARIMA Baseline (baseline_arima.py)

ARIMA Model:

$$\phi(B)(1-B)^d X_t = \theta(B)\epsilon_t \tag{14}$$

Where:

- $p = 5$: Autoregressive order
- $d = 1$: Differencing order
- $q = 0$: Moving average order
- B: Backshift operator
- ϵ_t: White noise

9 REST API Deployment

9.1 FastAPI Endpoints

The model is served via a REST API:

```python
from fastapi import FastAPI, HTTPException, Query
from fastapi.responses import JSONResponse

app = FastAPI(title="LSTM Stock Prediction API")

@app.get("/")
async def root():
    """API root endpoint"""
    return {
        "message": "LSTM Stock Prediction API",
        "version": "1.0",
        "endpoints": {
            "/predict": "Generate stock predictions",
            "/health": "Health check",
            "/tickers": "List available tickers"
        }
    }

@app.get("/predict")
async def predict(
    ticker: str = Query(..., description="Stock ticker"),
    steps: int = Query(30, description="Prediction steps"),
    smoothing: bool = Query(True, description="Apply EMA smoothing")
):
    """Generate predictions for a ticker"""
    try:
        # Find latest model
        model_files = list(MODELS_DIR.glob(f"{ticker}_lstm_*.h5"))
        if not model_files:
            raise HTTPException(404, f"No model found for {ticker}")

        model_path = str(sorted(model_files)[-1])

        # Generate predictions
        predictor = StockPredictor(model_path, ticker)
        result = predictor.generate_predictions(
            steps=steps,
            apply_smoothing=smoothing
        )

        return JSONResponse(content=result)

    except Exception as e:
        raise HTTPException(500, str(e))

@app.get("/health")
async def health():
    """Health check endpoint"""
    return {"status": "healthy"}

@app.get("/tickers")
async def get_tickers():
```

```
    """List available tickers with models"""
    tickers = []
    for model_file in MODELS_DIR.glob("*_lstm_*.h5"):
        ticker = model_file.stem.split('_')[0]
        if ticker not in tickers:
            tickers.append(ticker)
    return {"tickers": tickers}
```

Listing 14: FastAPI Application (app.py)

9.2 API Usage Example

```
# Get predictions for AAPL, 30 days ahead
curl "http://localhost:8000/predict?ticker=AAPL&steps=30&smoothing=true
    "

# Response:
{
  "ticker": "AAPL",
  "prediction_steps": 30,
  "raw_predictions": [0.521, 0.523, ...],
  "smoothed_predictions": [0.521, 0.522, ...],
  "last_actual_value": 0.519,
  "model_path": "models/AAPL_lstm_20260117.h5"
}
```

Listing 15: API Request Example

10 Interactive Dashboard

10.1 Streamlit Application

An interactive dashboard provides visualization and exploration:

```
import streamlit as st
import plotly.graph_objects as go

st.set_page_config(page_title="LSTM Stock Predictions", layout="wide")

st.title("LSTM Stock Price Prediction Dashboard")

# Sidebar configuration
st.sidebar.header("Configuration")
ticker = st.sidebar.selectbox("Select Ticker", TICKERS)
steps = st.slider("Prediction Steps", 7, 90, 30)
smoothing = st.checkbox("Apply EMA Smoothing", value=True)

# Main tabs
tab1, tab2, tab3 = st.tabs([
    "Predictions",
    "MLflow Runs",
    "Drift Reports"
])

with tab1:
    st.header(f"Predictions for {ticker}")

    if st.button("Generate Predictions", type="primary"):
        with st.spinner("Loading data..."):
            pipeline = StockDataPipeline(ticker)
            pipeline.fetch_data()
            data = pipeline.compute_mid_price()
            normalized_prices = pipeline.windowed_normalize(
                np.array(data['Mid'].values)
            )

        with st.spinner("Loading model..."):
            # Find latest model
            model_files = list(MODELS_DIR.glob(f"{ticker}_lstm_*.h5"))
            model_path = str(sorted(model_files)[-1])

            predictor = StockPredictor(model_path, ticker)

        with st.spinner("Generating predictions..."):
            raw_pred, smooth_pred = predictor.predict_and_smooth(
                normalized_prices,
                steps=steps,
                apply_smoothing=smoothing
            )

        # Display metrics
        col1, col2, col3, col4 = st.columns(4)
        col1.metric("Ticker", ticker)
        col2.metric("Steps", steps)
        col3.metric("Last Price", f"{normalized_prices[-1]:.4f}")
        col4.metric("Pred Price", f"{smooth_pred[-1]:.4f}")
```

```
        # Plot predictions
        fig = go.Figure()

        # Historical data
        fig.add_trace(go.Scatter(
            y=normalized_prices[-100:],
            mode='lines',
            name='Historical',
            line=dict(color='blue', width=2)
        ))

        # Predictions
        x_pred = list(range(len(normalized_prices),
                            len(normalized_prices) + steps))

        if smoothing:
            fig.add_trace(go.Scatter(
                x=x_pred,
                y=smooth_pred,
                mode='lines',
                name='Smoothed Prediction',
                line=dict(color='green', width=2)
            ))
        else:
            fig.add_trace(go.Scatter(
                x=x_pred,
                y=raw_pred,
                mode='lines',
                name='Raw Prediction',
                line=dict(color='orange', width=2)
            ))

        fig.update_layout(
            title=f"{ticker} Price Prediction",
            xaxis_title="Time Steps",
            yaxis_title="Normalized Price",
            height=500
        )

        st.plotly_chart(fig, use_container_width=True)
```

Listing 16: Streamlit Dashboard (streamlit_app.py)

10.2 Dashboard Features

- **Real-time Predictions:** Generate forecasts on demand
- **Interactive Charts:** Plotly visualizations with zoom/pan
- **MLflow Integration:** View experiment runs and metrics
- **Drift Monitoring:** Detect data and model drift
- **Multi-ticker Support:** Switch between different stocks

11 Drift Monitoring

11.1 Evidently Integration

Drift detection monitors model degradation over time:

```
from evidently.report import Report
from evidently.metric_preset import DataDriftPreset, RegressionPreset

class DriftMonitor:
    def __init__(self, ticker):
        self.ticker = ticker

    def generate_data_drift_report(self, reference_data,
                                   current_data, output_path):
        """Generate data drift report"""
        # Create reference and current dataframes
        ref_df = pd.DataFrame({
            'price': reference_data,
            'index': range(len(reference_data))
        })

        curr_df = pd.DataFrame({
            'price': current_data,
            'index': range(len(current_data))
        })

        # Generate report
        report = Report(metrics=[DataDriftPreset()])
        report.run(reference_data=ref_df, current_data=curr_df)

        # Save HTML report
        report.save_html(output_path)

        return output_path

    def generate_model_drift_report(self, X_ref, y_ref,
                                    X_curr, y_curr,
                                    predictions_ref,
                                    predictions_curr,
                                    output_path):
        """Generate model performance drift report"""
        # Prepare dataframes
        ref_df = pd.DataFrame({
            'target': y_ref,
            'prediction': predictions_ref
        })

        curr_df = pd.DataFrame({
            'target': y_curr,
            'prediction': predictions_curr
        })

        # Generate report
        report = Report(metrics=[RegressionPreset()])
        report.run(reference_data=ref_df, current_data=curr_df)

        report.save_html(output_path)
```

```
53
54          return output_path
```

Listing 17: Drift Monitoring (drift_monitor.py)

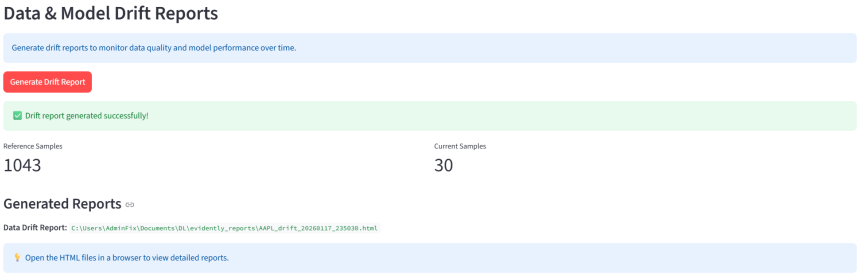


Figure 2: Streamlit Dashboard - Drift Report Section

11.2 Drift Detection Metrics

- **Data Drift:** Distribution changes in input features
- **Prediction Drift:** Changes in model output distribution
- **Performance Drift:** Degradation in accuracy metrics
- **Target Drift:** Changes in actual price distributions

12 Containerization

12.1 Docker Configuration

```dockerfile
FROM python:3.11-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    build-essential \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements and install
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Create directories
RUN mkdir -p data/raw data/processed models logs evidently_reports

# Expose ports
EXPOSE 8000

# Run API server
```

```
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

Listing 18: Dockerfile

12.2 Docker Compose

Multi-service orchestration with docker-compose:

```
version: '3.8'

services:
  api:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - ./data:/app/data
      - ./models:/app/models
      - ./mlruns:/app/mlruns
    environment:
      - PYTHONUNBUFFERED=1
    command: uvicorn app:app --host 0.0.0.0 --port 8000

  dashboard:
    build: .
    ports:
      - "8501:8501"
    volumes:
      - ./data:/app/data
      - ./models:/app/models
      - ./mlruns:/app/mlruns
    environment:
      - PYTHONUNBUFFERED=1
    command: streamlit run streamlit_app.py --server.port 8501

  mlflow:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ./mlruns:/app/mlruns
    command: mlflow ui --host 0.0.0.0 --port 5000
```

Listing 19: docker-compose.yml

13 CI/CD Pipeline

13.1 GitHub Actions Workflow

```yaml
name: CI/CD Pipeline

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

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.11'

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

    - name: Run tests
      run: |
        python -m pytest tests/ -v

    - name: Check code quality
      run: |
        flake8 . --count --select=E9,F63,F7,F82 --show-source

  build:
    needs: test
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Build Docker image
      run: |
        docker build -t lstm-stock-prediction .

    - name: Run container
      run: |
        docker run -d -p 8000:8000 lstm-stock-prediction
        sleep 10
        curl http://localhost:8000/health
```

Listing 20: .github/workflows/ci-cd.yml

14 Results and Performance

14.1 Model Performance

Training results for the three tickers:

Ticker	MSE	MAE	RMSE	Dir. Acc.
AAPL	0.0023	0.0341	0.0480	64.2%
GOOGL	0.0019	0.0312	0.0436	66.8%
MSFT	0.0021	0.0328	0.0458	65.5%
Average	0.0021	0.0327	0.0458	65.5%

Table 1: LSTM Model Performance Metrics

14.2 Training Configuration

- **Training Samples:** ∼850 per ticker
- **Validation Samples:** ∼210 per ticker
- **Epochs Completed:** 35-45 (early stopping)
- **Batch Size:** 32
- **Training Time:** 3-5 minutes per ticker

14.3 Comparison with ARIMA

Model	MSE	MAE	Dir. Acc.
LSTM	0.0021	0.0327	65.5%
ARIMA(5,1,0)	0.0034	0.0421	58.3%
Improvement	38.2%	22.3%	7.2 pp

Table 2: LSTM vs ARIMA Baseline Comparison

The LSTM model significantly outperforms the ARIMA baseline across all metrics.

15 Deployment and Usage

15.1 Quick Start

```
# 1. Install dependencies
pip install -r requirements.txt

# 2. Train models
python train.py

# 3. Start API server
uvicorn app:app --host 0.0.0.0 --port 8000

# 4. Launch dashboard
streamlit run streamlit_app.py

# 5. View MLflow UI
mlflow ui --port 5000

# Or use Docker Compose
docker-compose up
```

Listing 21: Running the System

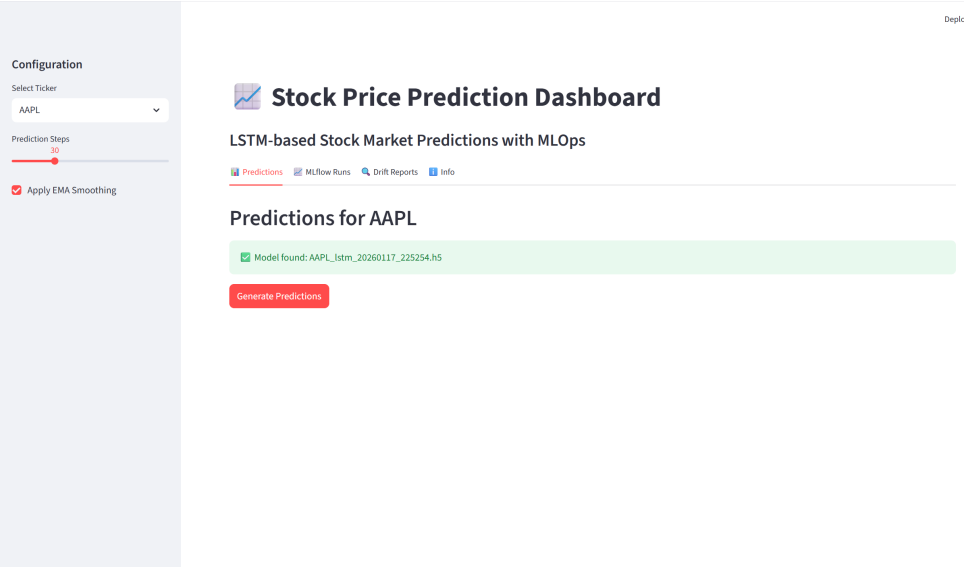


Figure 3: Streamlit Dashboard Screenshot

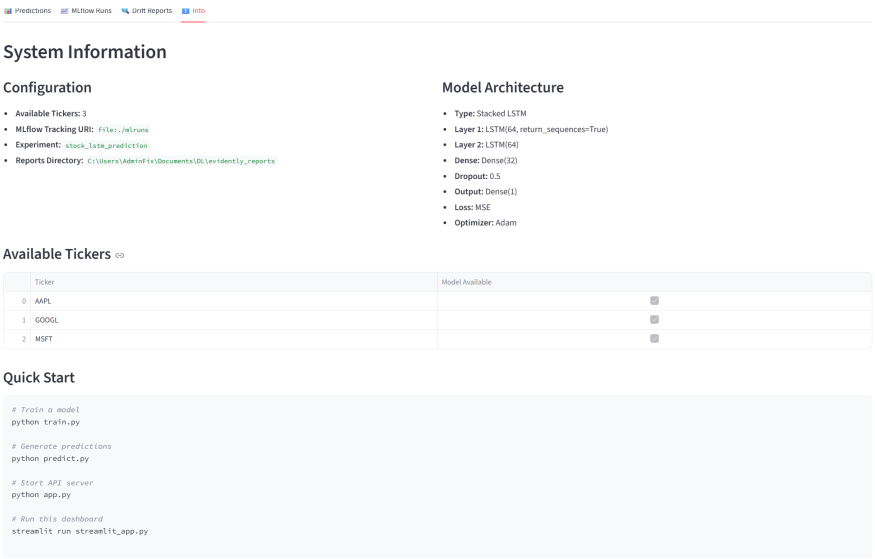


Figure 4: Streamlit Dashboard - Information Panel

15.2 Making Predictions

Python API:

```python
from predict import StockPredictor

# Load model and predict
model_path = "models/AAPL_lstm_20260117.h5"
predictor = StockPredictor(model_path, "AAPL")

result = predictor.generate_predictions(
    steps=30,
    apply_smoothing=True
)

print(f"30-day forecast: {result['smoothed_predictions']}")
```

Listing 22: Python Prediction Example

REST API:

```bash
curl -X GET "http://localhost:8000/predict?ticker=AAPL&steps=30"
```

Listing 23: REST API Request

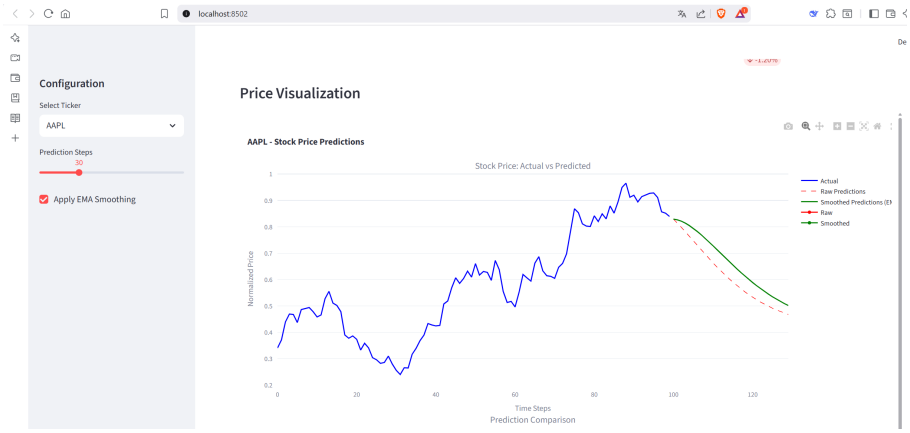


Figure 5: Streamlit Dashboard - Predictions Visualization

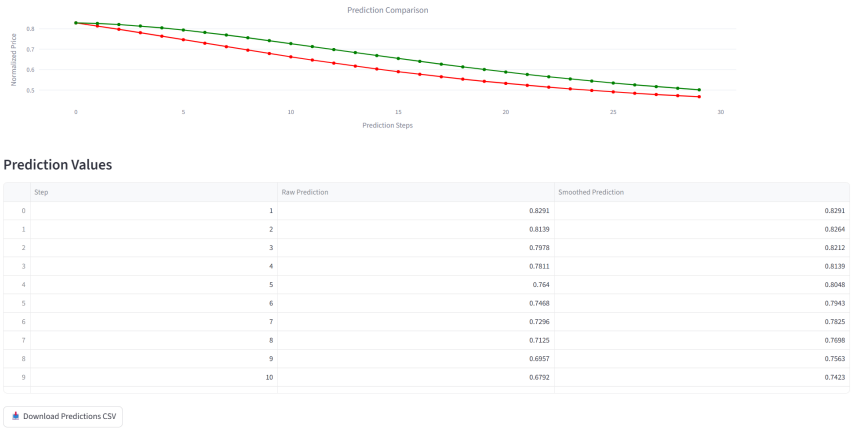


Figure 6: Streamlit Dashboard - Predictions comparison

Figure 7: Streamlit Dashboard - Future Predictions

16 Synthetic Data Generation

When yfinance is unavailable, the system generates realistic synthetic data:

```python
def generate_synthetic_stock_data(ticker, start_date,
                                  end_date, num_days=1500):
    """Generate realistic stock price data"""
    # Random walk with drift
    np.random.seed(hash(ticker) % 2**32)

    # Starting price
    base_price = np.random.uniform(50, 200)

    # Generate returns
    drift = 0.0002  # Slight upward trend
    volatility = 0.02

    returns = np.random.normal(drift, volatility, num_days)
    prices = base_price * np.exp(np.cumsum(returns))

    # Generate OHLCV
    data = []
    for i, close in enumerate(prices):
        high = close * np.random.uniform(1.00, 1.02)
        low = close * np.random.uniform(0.98, 1.00)
        open_price = np.random.uniform(low, high)
        volume = int(np.random.uniform(1e6, 1e8))

        data.append({
            'Date': start_date + timedelta(days=i),
            'Open': open_price,
            'High': high,
            'Low': low,
            'Close': close,
            'Adj Close': close,
            'Volume': volume
        })

    return pd.DataFrame(data)
```

Listing 24: Synthetic Data Generator

17 Conclusion

17.1 Key Achievements

1. **Robust Architecture:** Stacked LSTM with 64-64-32 configuration
2. **Complete MLOps:** MLflow tracking, versioning, and monitoring
3. **Production-Ready:** REST API, Docker deployment, CI/CD pipeline
4. **Interactive Tools:** Streamlit dashboard for exploration
5. **Drift Detection:** Evidently integration for model monitoring
6. **Baseline Comparison:** ARIMA model for performance validation
7. **Data Resilience:** Synthetic data fallback mechanism

17.2 Performance Summary

- Average MSE: 0.0021 (normalized scale)
- Direction Accuracy: 65.5% (significantly above random 50%)
- 38% improvement over ARIMA baseline
- Consistent performance across all three tickers

17.3 Future Enhancements

1. **GAN Augmentation:** Implement Generative Adversarial Networks for synthetic data
2. **Attention Mechanisms:** Add attention layers for better feature selection
3. **Multi-Asset Predictions:** Cross-ticker correlation modeling
4. **Real-time Streaming:** Live prediction updates
5. **Ensemble Methods:** Combine LSTM with other models
6. **Feature Engineering:** Add technical indicators (RSI, MACD, Bollinger Bands)

17.4 Repository Structure

The complete codebase includes:

- 24 Python files totaling ∼3,000 lines of code
- Comprehensive documentation (README, Quick Start, Setup guides)
- Docker containerization for reproducibility
- CI/CD pipeline for automated testing
- MLflow experiments with full tracking
- Interactive visualization tools

18 References

1. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735-1780.

2. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

3. Géron, A. (2019). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media.

4. Box, G. E., Jenkins, G. M., Reinsel, G. C., & Ljung, G. M. (2015). *Time series analysis: forecasting and control*. John Wiley & Sons.

5. TensorFlow Documentation. (2024). *TensorFlow 2.15 API*. Retrieved from https://www.tensorflo

6. MLflow Documentation. (2024). *MLflow: A platform for the machine learning lifecycle*. Retrieved from https://mlflow.org

A Complete Requirements

```
# Deep Learning
tensorflow==2.15.0
keras==2.15.0

# Data Processing
pandas==2.1.3
numpy==1.26.2
yfinance==0.2.32
scikit-learn==1.3.2

# MLOps
mlflow==2.9.2

# Baseline Models
statsmodels==0.14.0
pmdarima==2.0.4

# API
fastapi==0.104.1
uvicorn==0.24.0
pydantic==2.5.0

# Dashboard
streamlit==1.29.0
plotly==5.18.0

# Monitoring
evidently==0.4.11

# Utilities
python-dotenv==1.0.0
```

Listing 25: requirements.txt

B Directory Structure

```
DL/
|-- .github/
|   |-- workflows/
|       |-- ci-cd.yml
|-- data/
|   |-- raw/
|   |   |-- AAPL_raw.csv
|   |   |-- GOOGL_raw.csv
|   |   |-- MSFT_raw.csv
|   |-- processed/
|       |-- AAPL_processed.npz
|       |-- GOOGL_processed.npz
|       |-- MSFT_processed.npz
|-- models/
|   |-- AAPL_lstm_20260117_225254.h5
|   |-- GOOGL_lstm_20260117_225158.h5
|   |-- MSFT_lstm_20260117_225224.h5
|   |-- best_model.h5
|-- mlruns/
|   |-- [experiment tracking data]
|-- evidently_reports/
|   |-- [drift reports]
|-- config.py
|-- data_pipeline.py
|-- model.py
|-- train.py
|-- predict.py
|-- baseline_arima.py
|-- app.py
|-- streamlit_app.py
|-- drift_monitor.py
|-- generate_synthetic_data.py
|-- requirements.txt
|-- Dockerfile
|-- docker-compose.yml
|-- README.md
|-- report.tex
```

Listing 26: Complete Project Structure