Introduction

The Trading System project was developed to design a trading system that leverages machine learning to predict stock market movements and integrates these predictions into a web-based application. Using this web-application users can analyze market trends, visualize predictive analytics, and make trading decisions based on real-time and historical data that is sourced by SimFin.
The project consists of two main components:

1. Data Analytics Module – This involves constructing an ML model capable of forecasting stock price movements. By analyzing historical stock market data of US companies, the model identifies patterns and trends that inform trading decisions. This module includes data extraction, preprocessing, feature engineering, model selection, and evaluation.

2. Web-Based Application – Developed using Streamlit, the web app provides an interactive interface that allows users to explore market trends, view ML-generated predictions, and understand the methodologies behind the trading system. The app integrates with SimFin, a financial data platform, to retrieve real-time stock data, enabling users to monitor market conditions dynamically.

The project utilizes Python as the primary programming language, with Pandas for data processing and scikit-learnor TensorFlow for ML modelling. This report summarizes how each part of the project was implemented, the challenges encountered during development, and key conclusions drawn from the experience.

Data Analysis Module

The Data Analytics Module serves as the backbone of the Automated Daily Trading System, designed to process financial market data and construct a machine learning model capable of predicting stock price movements. This module follows a structured pipeline, beginning with the ETL process, advancing to machine learning model development, and incorporating a trading strategy to guide decision-making.

ETL Process

The ETL process was designed and implemented as a modular and reusable Python class, making it easier to manage data extraction, filtering, merging, and storage. The data used in this project consists of two key datasets:

1. Companies Data – Contains essential details about publicly traded companies, including ticker symbols and industry classifications.

2. Share Prices Data – Provides historical daily stock price movements, which serve as the primary input for market trend analysis.

The extraction phase loads the data from CSV files, ensuring compatibility with downstream processes. The transformation step includes filtering specific stock tickers (if provided), removing unnecessary columns such as dividends, and converting date

fields to a proper datetime format. Finally, the merge operation combines share prices with company information, creating a unified dataset. The final structured data is then saved as a CSV file, ready for machine learning analysis.

This ETL process was designed with flexibility in mind, allowing the system to process either all available stocks or a subset of selected tickers. Error handling mechanisms ensure smooth execution, and the modular nature of the ETL class allows for easy integration into other components of the trading system.

Machine Learning Model for Stock Prediction

The next step involved building a simple predictive model using Logistic Regression. The goal was to classify whether a stock's price would increase (buy) or decrease (sell) in the next trading session based on recent historical data.

To achieve this, we first engineered a target variable, where 1 represented an expected price increase and 0 indicated a price decrease. The selected features for the model included key trading indicators such as Open, High, Low, Close, and Volume prices.

Using Logistic Regression, we trained the model to recognize patterns in stock price fluctuations. The model was then evaluated on the test set, achieving a measurable 53% level of accuracy in predicting whether prices would rise or fall. After training, both the model and scaler were saved using **joblib** to enable future predictions without retraining.

To facilitate real-time decision-making, we implemented a predictive function that takes the most recent stock data and determines whether the model suggests buying or selling. By inputting the latest market conditions, the system returns a simple recommendation:

- "Price will rise" → Buy
- "Price will fall" → Sell

API Wrapper for SimFin Data Retrieval

To streamline data extraction from SimFin, an API wrapper was implemented to facilitate the retrieval of stock price and financial statement data. This wrapper serves as a bridge between the SimFin API and the trading system, ensuring seamless and structured access to essential financial data.

The API wrapper class (api_wrapper) is designed to fetch historical share prices and financial statements by specifying a stock ticker and date range. It first initializes the connection by loading the API key from environment variables and configuring the SimFin data directory.

Error handling and logging mechanisms are incorporated to ensure reliable data retrieval. If an issue occurs (e.g., invalid API key or missing data), the system logs the error and returns None, preventing unnecessary failures in downstream processes. By encapsulating SimFin API interactions within this wrapper, the trading system maintains a modular structure, allowing for future extensions such as fetching additional financial metrics or integrating real-time data.

Trading Strategy
To demonstrate the potential effectiveness of our stock price prediction model, we implemented a trading strategy simulation that allows users to see what would happen if they had followed the model's buy and sell signals with an initial investment of €1000. The strategy aims to provide insight into how the predictive model could be used in a real-world trading scenario.

Trading Strategy Logic
The trading strategy follows a rule-based system where stock price predictions generated by the machine learning model determine buy and sell decisions:

1. Buy Signal (Prediction = 1): If the model predicts a price increase, the system invests as much of the available cash as possible by buying shares at the current market price.

2. Sell Signal (Prediction = 0): If the model predicts a price decrease and the trader holds shares, all shares are sold, converting them back into cash.

3. Portfolio Value Calculation: The system tracks remaining cash, owned shares, and trade history over the simulation period. At the end, the total portfolio value is calculated as cash + (number of shares × latest stock price).

This strategy was applied to historical stock data retrieved via the api_wrapper, allowing users to simulate past trades for a selected stock (e.g., AAPL in 2023). By analyzing the trade log and final portfolio value, users can evaluate how well the model would have performed in a real market scenario.

Simulation Results and Insights
The trading strategy successfully integrates with the predictive model to automate trade decisions. However, like any model-based trading system, its effectiveness depends on market volatility, model accuracy, and external factors that influence stock prices. Future improvements could incorporate stop-loss mechanisms, moving average-based confirmations, or reinforcement learning techniques to refine the strategy's performance.

With this end-to-end analytics module, our system not only forecasts stock price movements but also provides a realistic trading simulation, helping users understand the financial impact of using algorithmic trading strategies. The next phase of the project focuses on integrating these insights into an interactive web-based trading application for user-friendly accessibility.

Web-Based Application

The Web-Based Application provides an interactive platform for users to analyze stock data, view model-generated trading signals, and test a trading strategy. Developed using Streamlit, the application is structured into three main sections:

1. Home Page, which provides an overview of the system and its objectives.

2. Go Live Page, where users can explore real-time and historical stock data, visualize trends, and receive trading signals.

3. Trading Strategy Page, which simulates trading scenarios based on the model's predictions.

Navigation and User Interface
To enhance user experience, the application incorporates a sidebar navigation system that allows users to switch between pages easily. The Home Page introduces the system, explaining its purpose, features, and development team. It provides users with a clear understanding of the system's capabilities before they interact with the data and predictions.

The Go Live Page serves as the central hub for stock analysis. Users can select a stock ticker, define a date range, and view a closing price graph generated from historical market data. This feature helps in understanding past trends before making investment decisions. The page also integrates a predictive model that applies a logistic regression algorithm to recent stock data, providing a buy or sell signal based on the model's forecast.

Real-Time Data Analysis and Prediction
The Go Live Page retrieves stock data from the ETL output file and filters it based on user input. A time-series graph plots closing prices over the selected period, offering insights into market movements. The machine learning model is then applied to the latest available data to predict whether the stock price will increase or decrease. If the prediction indicates a price increase, a buy signal is displayed, while a price decrease results in a sell signal.

Several measures were taken to ensure the robustness of this feature. If the required dataset is missing, an error message is displayed to prevent system failure. If the model file is not found or cannot be loaded, the application notifies the user instead of breaking. Additional validation ensures that all required features exist before making predictions, reducing the risk of incorrect outputs.

Trading Strategy Simulation
The Trading Strategy Page allows users to simulate an automated trading strategy based on the machine learning model's predictions. Users enter a stock ticker and date range, and the system fetches historical data using the API wrapper. The trading strategy is then applied to execute simulated trades, where the model determines when to buy and sell shares based on its predictions.

The page provides a summary of all trades executed during the simulation, displaying a trade log with details of each buy and sell action. At the end of the simulation, the system calculates the final portfolio value, considering both cash balance and the value of any

remaining shares. A graph visualizing stock price movements over the selected period is also included to help users understand market trends during the trading window.

Error Handling and System Reliability
To ensure smooth operation, several error-handling mechanisms were integrated into the application. If users select an invalid date range where the start date is later than the end date, an error message is displayed to prevent misconfigurations. When stock data is unavailable for a selected ticker or date range, the system informs the user instead of proceeding with incomplete data. If the machine learning model file is missing or corrupted, the application prevents predictions from running and provides a notification.

These validation and error-handling steps contribute to a more stable and user-friendly experience, ensuring that users receive accurate information without unexpected failures.

Challenges and Solutions

One of the main challenges in developing the Automated Trading System was integrating the user interface with the backend. Ensuring seamless communication between the Streamlit web application, the machine learning model, and the API wrapper proved difficult, particularly in handling real-time user inputs and dynamically updating predictions. Initially, there were issues in passing data correctly between components, leading to inconsistencies in displayed results. This was resolved by restructuring the navigation system, optimizing data flow, and implementing error handling to prevent crashes and ensure a smoother user experience.

Working with large datasets also presented significant challenges. The historical stock data contained thousands of records, which caused performance bottlenecks when processed using Pandas. To address this, we optimized data handling with Polars, which allowed for faster data manipulation. Additionally, we refined the ETL pipeline to load only necessary data subsets into memory, reducing resource consumption and improving system responsiveness. Ensuring that all classes and functions worked correctly also required extensive testing, as incorrect data filtering and missing attributes sometimes caused unexpected errors. Implementing structured error handling and validation checks helped to improve the reliability of the system.

Another difficulty involved managing large files and version control. The trained model, processed datasets, and dependencies resulted in large file sizes that exceeded GitHub's limits, making it challenging to store and share files efficiently. Using Git Large File Storage (LFS) helped manage these large files while keeping the repository lightweight. Additionally, restructuring the repository to exclude unnecessary large files improved storage efficiency. Despite these challenges, the project was successfully completed through structured problem-solving, performance optimizations, and iterative improvements, resulting in a scalable and user-friendly system.

Conclusion

The development of the Automated Trading System successfully combined machine learning, financial data processing, and an interactive web-based interface to provide a functional stock market analysis tool. Through a structured ETL pipeline, the system efficiently processed large-scale stock price data, enabling the machine learning model to generate actionable trading predictions. The Streamlit web application provided a user-friendly platform for interacting with the system, allowing users to visualize stock trends, receive trading signals, and simulate investment strategies based on historical data.

Throughout the project, challenges such as UI-backend integration, handling large datasets, implementing complex functionalities, and managing large files in version control were addressed through systematic optimizations. By refining data processing techniques, improving navigation and error handling, and leveraging Git LFS for large file management, we ensured the system remained efficient, scalable, and easy to use. These improvements contributed to a more seamless experience for users interacting with the application.

While the current implementation provides a solid foundation, future enhancements could further improve the system's accuracy, performance, and real-time capabilities. Potential advancements include incorporating additional financial indicators, refining the predictive model with more sophisticated machine learning techniques, and integrating live market data for real-time decision-making. Overall, this project demonstrates how machine learning and automation can be applied to financial markets, offering valuable insights for traders and investors looking to make data-driven decisions.