High Level Design (HLD)

Investment Prediction

Revision Number: 1.1
Last date of revision: 11/06/2024

Sudip Joshi

Document Version Control

Date Issued	Version	Description	Author
7th Jan 2024	1.0	Abstract, Introduction and general description	Sudip Joshi
8th March 2024	1.1	Design detail, and deployment	Sudip Joshi
9th May2024	1.2	Final Version of Complete HLD	Sudip Joshi

Table of Contents

Abstract

This document provides a high-level design (HLD) for a stock price prediction application. The primary objective is to predict future stock prices using the XGBoost machine learning algorithm and present these predictions through an interactive web application built with Streamlit. This document outlines the overall architecture, tools used, design details, and deployment strategy for the application.

1. Introduction

Why this High-Level Design Document?

A High-Level Document (HLD) is essential in IT projects as it serves as a blueprint that outlines the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements Helps in planning and allocating resources efficiently, ensuring that the project has the necessary assets to proceed smoothly

The scope of this document includes:

- Real Time access to market Data
- Enhance Investment Knowledge
- User friendly Interface
- Customeeizable Alerts and Notifications

2. General Description

Product Perspective & Problem Statement

The stock market is an area characterized by significant volatility and rapid changes. Traditional investment models often struggle to keep up with these fluctuations. By employing machine learning techniques, we can identify market changes earlier and automate the investment prediction process, thereby enhancing decision-making and potentially improving investment outcomes.Provides a high-level architectural blueprint of the system, including the main components, their relationships, and interactions.

Tools Used

- **Python**: Programming language for developing the application. Known for its simplicity and extensive libraries for data science.
- **Streamlit**: Framework for building the interactive web interface. Facilitates rapid development and deployment of data-driven applications.
- **yfinance**: Library for retrieving historical stock data from Yahoo Finance. Provides easy access to financial data.
- **pandas**: Library for data manipulation and preprocessing. Offers powerful data structures and data analysis tools.
- **xgboost**: Library for training the regression model to predict stock prices. Known for its efficiency and accuracy.
- **matplotlib**: Library for creating visualizations. Supports various types of plots and charts for data representation.

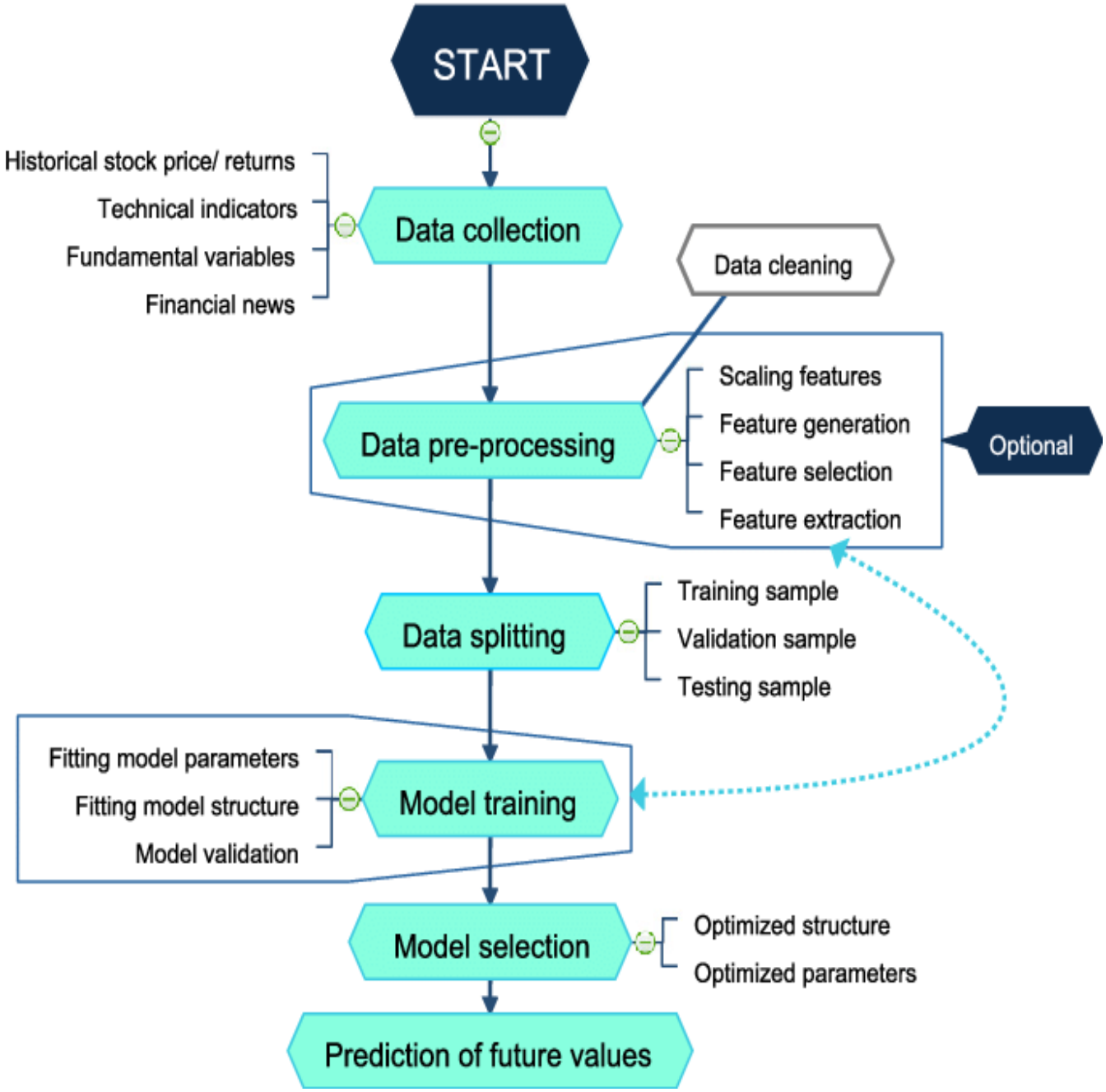
3. Design Details

Functional Architecture

The application consists of the following components:

1. **Data Collection**
2. **Data Preprocessing**
3. **Model Training**
4. **Prediction**
5. **Web Interface**

Functional Architecture:



Detailed Design

Data Collection

- **Objective: Retrieve historical stock price data for the specified stock symbol and date range.**
- **Implementation: Use the `yfinance` library to fetch data including open, high, low, close prices, and trading volume.**

Code Snippet:
python
Copy code

```python
def fetch_stock_data(symbol, start_date, end_date):

    data = yf.download(symbol, start=start_date, end=end_date)

    return data
```

Data Preprocessing

- Objective: Clean, normalize, and augment data with technical indicators such as Simple Moving Average (SMA) and Relative Strength Index (RSI).
- Implementation:
    - Handle missing values using forward-fill or interpolation methods.
    - Normalize features to ensure comparability.
    - Calculate technical indicators to capture trends and momentum.

code

```python
def create_features(data):

    data['SMA'] = data['Close'].rolling(window=20).mean()
```

```
    data['RSI'] = calculate_RSI(data['Close'])

    return data


def calculate_RSI(close_prices, window=14):

    delta = close_prices.diff()

    gain = (delta.where(delta > 0, 0)).rolling(window=window).mean()

    loss = (-delta.where(delta < 0, 0)).rolling(window=window).mean()

    rs = gain / loss

    rsi = 100 - (100 / (1 + rs))

    return rsi
```

Model Training

- Objective: Train the XGBoost model on the processed data to predict future stock prices.
- Implementation:
    - Split data into training and testing sets.
    - Train the model using the training set.

Code Snippet:
python
Copy code

```
def train_model(X_train, y_train):

    model = XGBRegressor()

    model.fit(X_train, y_train)

    return model
```

Prediction

- **Objective: Generate future stock price predictions for the next 30 business days.**
- **Implementation:**
    - **Use the trained model to make iterative predictions.**

- Each prediction informs the next.

Code Snippet:
python

```python
def predict_future_prices(model, data):

    future_dates = pd.date_range(data.index[-1] + timedelta(days=1), periods=30, freq='B')

    future_prices = []

    for date in future_dates:

        X_pred = prepare_data_for_prediction(data)

        future_price = predict_stock_price(model, X_pred)

        future_prices.append(future_price[0])

        data.loc[date] = future_price[0]  # Append predicted price to data for next prediction

    return future_dates, future_prices
```

Web Interface

- **Objective: Allow users to interact with the application via a web interface built with Streamlit.**
- **Implementation:**
  - **Users can view historical data, predictions, and related visualizations.**
  - **Fetch and display detailed stock information.**

Code Snippet:
python

```python
def main():

    st.title("Stock Price Prediction App")



    # Page navigation
```

```
    page = st.sidebar.selectbox("Page", ["Actual Value", "Prediction", "Ticker Information"])


    # Actual Value page
    if page == "Actual Value":
        st.header("Actual Value")
        symbol = st.text_input("Enter the stock symbol (e.g., AAPL):").upper()


        # Filter selection
        filter_option = st.selectbox("Select time period for the data:", ["1 Week", "1 Month", "3 Months", "6 Months", "1 Year", "5 Years", "Max"])


        # Set date range based on filter
        start_date = get_start_date(filter_option)
        end_date = datetime.now()


        if st.button("Fetch Data"):
            data = fetch_stock_data(symbol, start_date, end_date)
            if not data.empty:
                st.write(data.tail())
                plt.figure(figsize=(10, 6))
                plt.plot(data.index, data['Close'], label='Actual Closing Prices')
                plt.xlabel('Date')
                plt.ylabel('Price')
```

```
                plt.title(f'Actual Closing Prices for {symbol}')

                plt.legend()

                plt.grid(True)

                st.pyplot(plt)

            else:

                st.error("No data found for the given symbol and date range.")


    # Prediction page

    elif page == "Prediction":

        st.header("Prediction")

        symbol = st.text_input("Enter the stock symbol (e.g., AAPL):").upper()

        start_date = datetime.now() - timedelta(days=180)

        end_date = datetime.now()

        data = fetch_stock_data(symbol, start_date, end_date)

        X, y = prepare_data(data)

        model = train_model(X, y)

        future_dates, future_prices = predict_future_prices(model, data)

        predicted_data = pd.DataFrame({'Date': future_dates, 'Predicted Close': future_prices})

        st.write(predicted_data)

        plt.figure(figsize=(10, 6))

        plt.plot(data.index, data['Close'], label='Actual Closing Prices (Last 6 Months)', color='blue')

        plt.plot(predicted_data['Date'], predicted_data['Predicted Close'], label='Predicted Closing Prices (Next 30 Days)', color='red', linestyle='--
```

```
')
        plt.xlabel('Date')
        plt.ylabel('Price')
        plt.title(f'Actual vs Predicted Closing Prices for {symbol}')
        plt.legend()
        plt.grid(True)
        st.pyplot(plt)


    # Ticker Information page
    elif page == "Ticker Information":
        st.header("Ticker Information")
        symbol = st.text_input("Enter the stock symbol (e.g., AAPL):").upper()
        if st.button("Fetch Info"):
            ticker = yf.Ticker(symbol)
            info = ticker.info
            st.subheader(f"Information about {symbol}")
            st.write(f"**Name:** {info.get('longName', 'N/A')}")
            st.write(f"**Sector:** {info.get('sector', 'N/A')}")
            st.write(f"**Industry:** {info.get('industry', 'N/A')}")
            st.write(f"**Country:** {info.get('country', 'N/A')}")
            st.write(f"**Full Time Employees:** {info.get('fullTimeEmployees', 'N/A')}")
            st.write(f"**Business Summary:** {info.get('longBusinessSummary', 'N/A')}")
            st.write(f"**Market Cap:** {info.get('marketCap', 'N/A')}")
```

```
            st.write(f"**Enterprise Value:** {info.get('enterpriseValue',
'N/A')}")

            st.write(f"**Trailing P/E:** {info.get('trailingPE', 'N/A')}")

            st.write(f"**Forward P/E:** {info.get('forwardPE', 'N/A')}")
st.write(f"**Price to Book:** {info.get('priceToBook', 'N/A')}")
st.write(f"**PEG Ratio:** {info.get('pegRatio', 'N/A')}") st.write(f"**Price
to Sales:** {info.get('priceToSalesTrailing12Months', 'N/A')}")
st.write(f"**50-Day Moving Average:** {info.get('fiftyDayAverage', 'N/A')}")
st.write(f"**200-Day Moving Average:** {info.get('twoHundredDayAverage',
'N/A')}") st.write(f"**Website:** {info.get('website', 'N/A')}")
st.write(f"**Address:** {info.get('address1', 'N/A')}, {info.get('city',
'N/A')}, {info.get('state', 'N/A')}, {info.get('zip', 'N/A')}")

if __name__ == "__main__": main()
```

KPIs (Key Performance Indicators)

- **Accuracy**: The mean absolute error (MAE) between predicted and actual stock prices.
- **Performance**: Time taken to fetch data, preprocess, train the model, and make predictions.
- **User Engagement**: Number of users accessing the app and their interaction patterns.
- **Model Efficiency**: Training and inference time for the XGBoost model.

4. Deployment

Hosting

- The application will be hosted on a cloud platform such as AWS, GCP, or Heroku to ensure scalability and availability.
- **Streamlit Deployment**: The Streamlit app will be deployed as a web service, with the backend running on a cloud instance.

CI/CD Pipeline

- **Version Control**: The codebase will be maintained in a Git repository.
- **Continuous Integration**: Automated tests and builds will be configured using GitHub Actions or a similar CI tool.
- **Continuous Deployment**: Successful builds will be automatically deployed to the hosting environment.

Monitoring

- Application performance and user activity will be monitored using tools like Google Analytics and AWS CloudWatch.
- Alerts will be configured for any critical issues or downtime.

5. References

[1] https://pypi.org/project/yfinance/

[2] **https://xgboost.readthedocs.io/en/stable/**

[3] **https://docs.streamlit.io/**

[4] **https://numpy.org/doc/**

[5] **https://pandas.pydata.org/docs/**

[6] **https://matplotlib.org/stable/index.html**

[7] **https://scikit-learn.org/stable/**