Data 226 Lab 1 Report

Matthew Leffler, Daniel Kim

Department of Data Analytics

1. Problem Statement:

Stock trading is inherently volatile, with market conditions fluctuating rapidly. This unpredictability presents a challenge to investors who strive to forecast daily market movements to optimize their investment strategies. Leveraging machine learning for forecasting can significantly enhance predictability by analyzing historical stock data to project future closing prices.

The development of a data pipeline for this purpose addresses a critical need for continuous and automatic data acquisition. Given that stock prices are constantly changing, a data pipeline enables the systematic collection of stock information without manual intervention. This automation not only ensures that data used for forecasting is current, but also increases the efficiency and accuracy of these predictions.

Therefore, the purpose of this project is to implement a data pipeline that supports real-time data updates and facilitates more reliable stock market forecasts. This approach not only aids investors in making informed decisions but also contributes to a more structured method of navigating stock market volatility.

1. Solution Requirements

To build a robust data pipeline for stock prediction, our workflow is divided into three main stages, each requiring different tools.

1. *Data Extraction and Transformation*

The first stage involves retrieving and transforming stock market data. For this, we will utilize the yfinance API to collect daily stock information. Python will be used to extract this data in the form of a Pandas DataFrame and perform any necessary transformations before storing it. This approach ensures that the data is clean and structured before being stored in a database.

1. *Data Storage and Machine Learning Implementation*

The processed data will be stored in Snowflake, which is well-suited for structured, readily accessible datasets. Unlike traditional databases, Snowflake efficiently handles analytical workloads and does not require storing large raw data files, making it ideal for our use case. Additionally, to securely manage database credentials and API keys, we will use the os and dotenv libraries to store and load environment variables from a .env file, ensuring sensitive information is not hardcoded within the scripts.

1. *Automation of Data Pipeline*

Running the data pipeline manually daily is inefficient. To automate this process, we will leverage Apache Airflow, which will schedule and manage the data ingestion workflow. Airflow will execute Python scripts to extract data, transform it, and load it into Snowflake via the Snowflake connector. SQL commands will be executed on Snowflake to ensure that the database is updated daily, providing the latest stock data for prediction models.

By integrating Python, Snowflake, and Airflow, this solution ensures a scalable, secure, and automated data pipeline for stock prediction.

1. Functional Analysis:
2. *Connecting to Snowflake*

To establish a functional data pipeline, a secure and automated connection to Snowflake is required. This is achieved using Python’s os, dotenv, and snowflake-connector libraries, which store authentication credentials in an environment file and retrieve them dynamically to establish a connection. This approach enhances security by preventing hardcoded credentials while ensuring efficiency by automating database authentication. The connection serves as the foundation for the pipeline, enabling seamless data extraction, transformation, storage, and automation as part of the overall data ingestion and processing workflow.

A computer screen shot of a computer code

Description automatically generated

Fig. 1. Using Snowflake Connector to Connect to Snowflake

1. *Obtaining and Transforming Data*

With the Snowflake connector infrastructure in place, we can now proceed with the extraction and transformation phases of our ETL pipeline. To obtain stock data, we will utilize the yfinance API to retrieve historical stock data for the past 180 days. The yfinance library is particularly advantageous for data analysis, as it seamlessly integrates with Pandas, returning stock data in a DataFrame format. This is beneficial since SQL operates most efficiently with table-like structures. Additionally, yfinance offers flexible parameters that allow for easy customization of the data retrieval period. By setting the period parameter to 180 days, we can efficiently extract and analyze stock performance over the desired timeframe.

A screenshot of a computer

Description automatically generated

Fig. 2. Pandas Data Frame Returned Using yfinance

Although the raw data structure is nearly suitable for analysis, several adjustments are required before loading it into Snowflake. As shown in Figure 2, the yfinance download function returns a multi-indexed DataFrame, which could complicate analysis when imported into Snowflake. While it is essential to retain information about which stock each row corresponds to, it is more practical to store this detail as a separate column within the DataFrame. This ensures compatibility when importing data into a single Snowflake database, where information for multiple stocks will be stored together. SQL’s capabilities allow analysts to efficiently group and analyze the data for statistical modeling and forecasting as needed. Figure 3 illustrates the transformed DataFrame after these adjustments, demonstrating that the data is now properly structured and ready for storage in Snowflake.

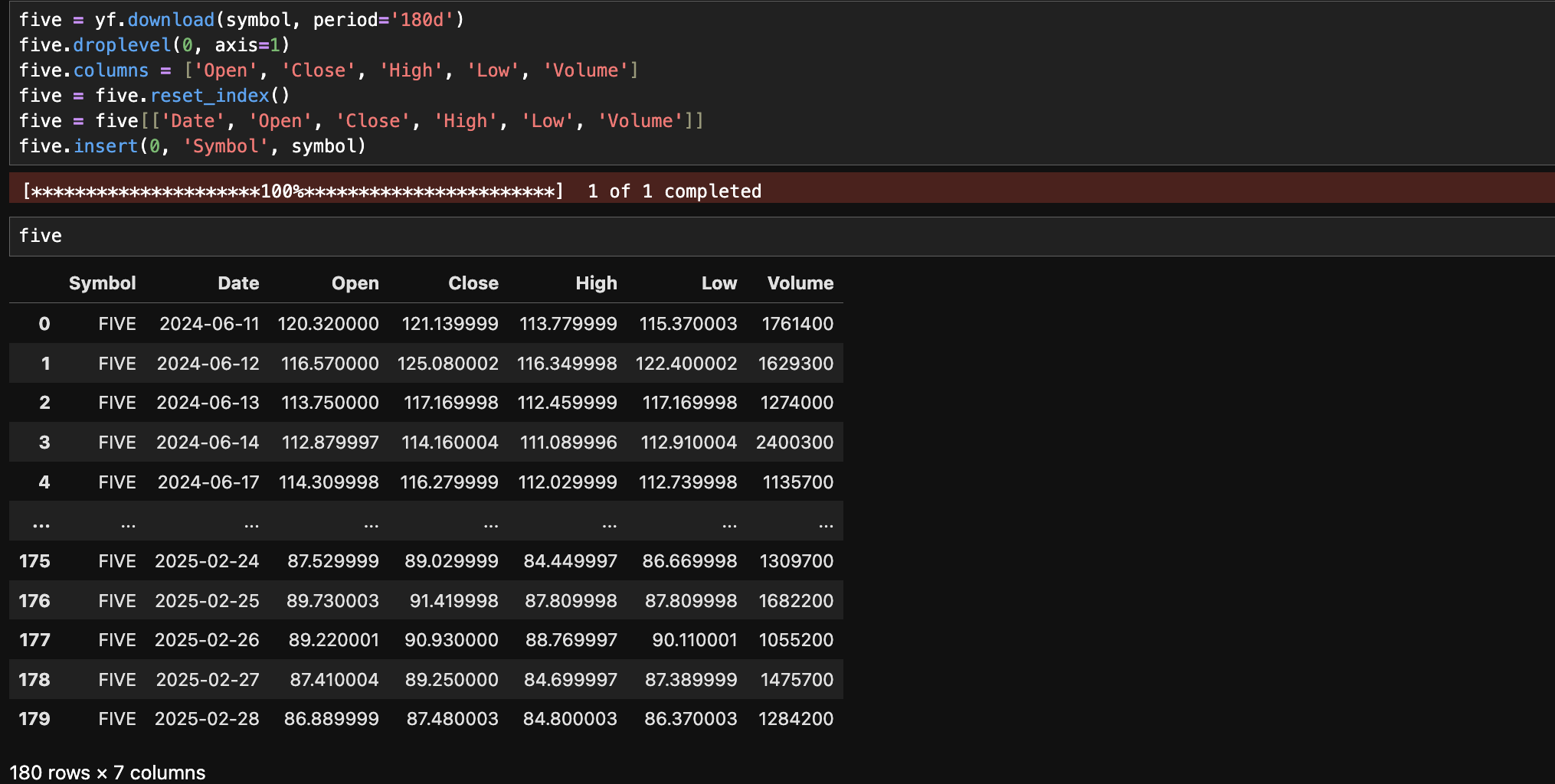


Fig. 3. Transforming Data Frame for Snowflake

1. *Loading Data into Snowflake*

With the tables now properly formatted, the next step is to load the data into Snowflake. As shown in Figure 1, we have already established a connection between Python and Snowflake, allowing us to execute SQL commands within Python to accurately insert data into the data warehouse. To manage the data effectively, we will utilize two tables: a staging table to temporarily hold incoming data and a historical table to store finalized records. Typically, staging tables serve as buffers to prevent incomplete or corrupted data from entering the database. However, since our data originates from a single source and is transformed within Python before loading, along with SQL transactions ensuring data integrity, a staging table may not be essential for this implementation. Nonetheless, incorporating a staging table could be beneficial for scalability if the project expands.

To maintain data integrity, we will wrap the data loading process within SQL transactions, ensuring that all commands execute successfully before committing changes. If an error occurs at any point in the pipeline, the transaction will roll back the tables to their previous state, preventing corrupted or incomplete data from being inserted. This safeguards the final table from skewed results caused by programming or API failures. Within the pipeline, we will first create the historical data table if it does not exist, replace the staging table to ensure it only holds new records, and then execute a bulk update to insert or update values in the historical table. Additionally, when defining the table schema, we must ensure that the Date column is cast as a TIMESTAMP\_NTZ type, as Snowflake’s forecasting functions require datetime-formatted data for proper analysis.

A screenshot of a computer

Description automatically generated

Fig. 4. Data Loaded into Snowflake

1. *Forecasting Stock Prices with Machine Learning*

With the data successfully loaded into Snowflake, we create two separate views based on the stock symbol—Apple (AAPL) and Five Below (FIVE). This separation is necessary because combining both stocks in a single forecast would result in predicting the average price of the two, which is more suitable for analyzing overall market trends rather than individual stock performance. While such an approach could be useful for broader industry-level analysis (e.g., comparing multiple tech stocks to assess market trends), our focus is on forecasting individual stock prices.

Since the stock symbol was stored as a separate column in the DataFrame, we can efficiently filter records using a WHERE clause in SQL to ensure that each view contains data for only one stock. Additionally, Snowflake’s forecasting function requires users to specify both a timestamp column and a target variable for analysis. Extracting and structuring these columns properly is essential to ensure the data is optimized for forecasting and predictive analytics.

A screenshot of a computer

Description automatically generated

Fig. 5. Creating View in Snowflake

After creating the views for the program, we can begin forecasting with Snowflake using its built-in forecasting function. By providing the function with the target column names and timestamps, Snowflake is automatically able to determine the intervals at which data appears in the dataframe. Not only that, but Snowflake will run multiple machine learning models on the data, and will automatically choose the best model at choosing the data internally, taking the manual labor out of the question.

A screenshot of a computer

Description automatically generated

Fig. 6. Creating Forecasting Model 1

As we can see from Figure 6, the forecasting model predicts that the price of the Apple stock will steadily increase over the duration of the next seven days, so as investors, it may be worth looking into purchasing stocks now while the prive is lower (although, this model is not sophisticated enough to account for external factors causing price fluctuations such as negative PR about the company).