

Data 226 Lab 1 Report

Matthew Leffler, Daniel Kim

Department of Data Analytics, San Jose State University

I. 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.

II. Solution Requirements

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

A. 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.

B. 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.

C. 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.

III. Functional Analysis:

A. 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.

```
def return_snowflake_conn():  
    # Manually specify the .env file path  
    load_dotenv(os.path.join(os.getcwd(), "Snowflake_Credentials.env"))  
  
    # Connect to Snowflake  
    conn = snowflake.connector.connect(  
        user = os.environ.get("SNOWFLAKE_USERNAME"),  
        password = os.environ.get("SNOWFLAKE_PASSWORD"),  
        account= os.environ.get("SNOWFLAKE_ACCOUNT"),  
        warehouse='COMPUTE_WH',  
        database='DEV',  
        schema='RAW'  
    )  
  
    return conn.cursor()
```

Fig. 1. Using Snowflake Connector to Connect to Snowflake

B. 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.

yf.download("FIVE", period='180d')

[*****100%*****] 1 of 1 completed

Price	Close	High	Low	Open	Volume
Ticker	FIVE	FIVE	FIVE	FIVE	FIVE
Date					
2024-06-11	120.320000	121.139999	113.779999	115.370003	1761400
2024-06-12	116.570000	125.080002	116.349998	122.400002	1629300
2024-06-13	113.750000	117.169998	112.459999	117.169998	1274000
2024-06-14	112.879997	114.160004	111.089996	112.910004	2400300
2024-06-17	114.309998	116.279999	112.029999	112.739998	1135700
...
2025-02-24	87.529999	89.029999	84.449997	86.669998	1309700
2025-02-25	89.730003	91.419998	87.809998	87.809998	1682200
2025-02-26	89.220001	90.930000	88.769997	90.110001	1055200
2025-02-27	87.410004	89.250000	84.699997	87.389999	1475700
2025-02-28	86.889999	87.480003	84.800003	86.370003	1284200

180 rows x 5 columns

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.

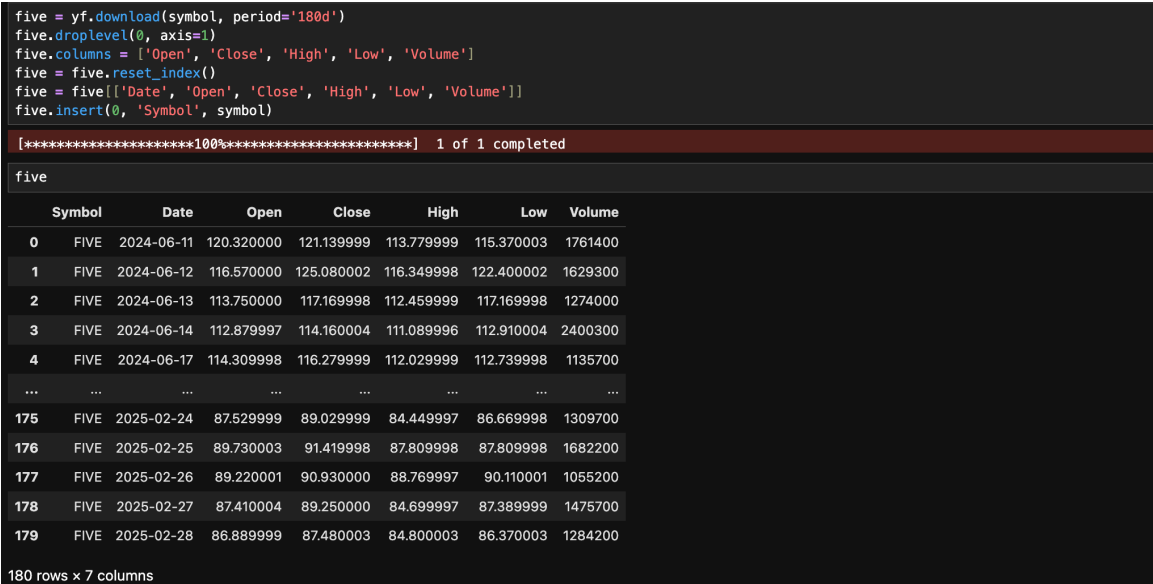


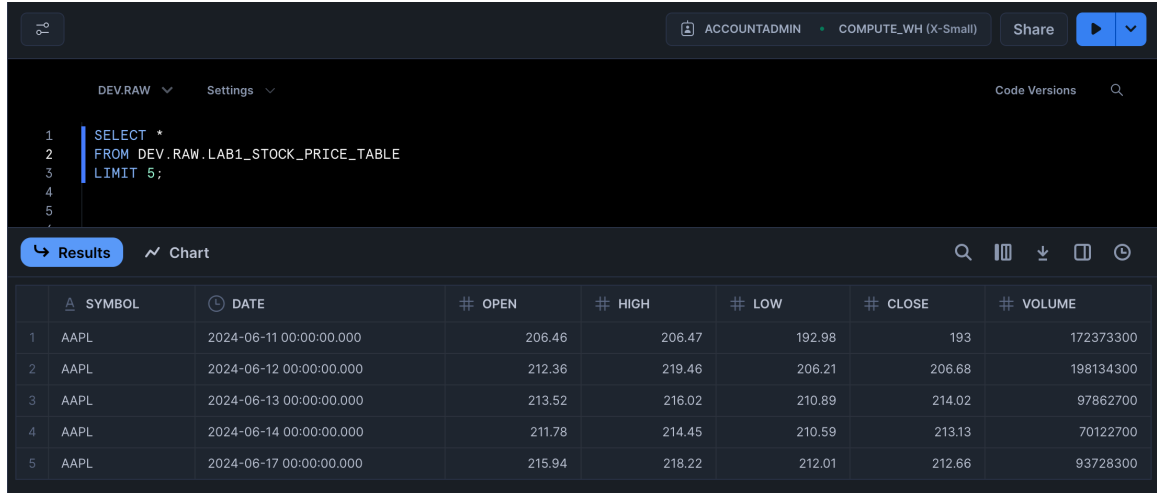
Fig. 3. Transforming Data Frame for Snowflake

C. 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.



The screenshot shows the Snowflake web interface. At the top, there's a header with 'ACCOUNTADMIN' and 'COMPUTE_WH (X-Small)'. Below that, a SQL query is entered in the editor: `SELECT * FROM DEV_RAW.LAB1_STOCK_PRICE_TABLE LIMIT 5;`. The 'Results' tab is selected, displaying a table with 5 rows of stock price data for AAPL. The table has columns: SYMBOL, DATE, OPEN, HIGH, LOW, CLOSE, and VOLUME. The data is as follows:

	SYMBOL	DATE	OPEN	HIGH	LOW	CLOSE	VOLUME
1	AAPL	2024-06-11 00:00:00.000	206.46	206.47	192.98	193	172373300
2	AAPL	2024-06-12 00:00:00.000	212.36	219.46	206.21	206.68	198134300
3	AAPL	2024-06-13 00:00:00.000	213.52	216.02	210.89	214.02	97862700
4	AAPL	2024-06-14 00:00:00.000	211.78	214.45	210.59	213.13	70122700
5	AAPL	2024-06-17 00:00:00.000	215.94	218.22	212.01	212.66	93728300

Fig. 4. Data Loaded into Snowflake

D. 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.

The screenshot shows the Snowflake SQL Editor interface. The top bar includes the user 'ACCOUNTADMIN' and the warehouse 'COMPUTE_WH (X-Small)'. The SQL query in the editor is as follows:

```

9  CREATE OR REPLACE VIEW aapl_view
10 AS
11 SELECT
12     DATE,
13     CLOSE
14 FROM DEV_RAW.LAB1_STOCK_PRICE_TABLE
15 WHERE SYMBOL = 'AAPL'
16
17
18 SELECT * FROM aapl_view
19 LIMIT 5;

```

Below the query, the 'Results' tab is active, displaying a table with 5 rows and 2 columns: 'DATE' and 'CLOSE'.

	DATE	CLOSE
1	2024-06-11 00:00:00.000	193
2	2024-06-12 00:00:00.000	206.68
3	2024-06-13 00:00:00.000	214.02
4	2024-06-14 00:00:00.000	213.13
5	2024-06-17 00:00:00.000	212.66

Fig. 5. Creating View in Snowflake

After creating the necessary views, we can utilize Snowflake’s built-in forecasting function to generate stock price predictions. By specifying the target column and timestamp, Snowflake automatically detects the data intervals within the DataFrame. Additionally, Snowflake applies multiple machine learning models to the dataset, selecting the most effective model internally. This automated approach eliminates the need for manual model selection and optimization, streamlining the forecasting process.

The screenshot shows the Snowflake SQL Editor interface. The top bar includes the user 'ACCOUNTADMIN' and the warehouse 'COMPUTE_WH (X-Small)'. The SQL query in the editor is as follows:

```

35 CREATE OR REPLACE SNOWFLAKE.ML.FORECAST "appl_md1"(
36     INPUT_DATA => TABLE(aapl_view),
37     TIMESTAMP_COLNAME => 'DATE',
38     TARGET_COLNAME => 'CLOSE',
39     CONFIG_OBJECT => {'ON_ERROR': 'SKIP'}
40 );
41
42 CALL "appl_md1"!FORECAST(FORECASTING_PERIODS => 7);

```

Below the query, the 'Results' tab is active, displaying a table with 7 rows and 5 columns: 'SERIES', 'TS', 'FORECAST', 'LOWER_BOUND', and 'UPPER_BOUND'.

	SERIES	TS	FORECAST	LOWER_BOUND	UPPER_BOUND
1	null	2025-02-28 00:00:00.000	239.482371962	231.829157148	247.468356144
2	null	2025-03-03 00:00:00.000	239.536737709	228.219325053	250.694853198
3	null	2025-03-04 00:00:00.000	239.591103456	225.673793452	253.031181595
4	null	2025-03-05 00:00:00.000	239.645469203	223.66995251	255.910911798
5	null	2025-03-06 00:00:00.000	239.69983495	221.987391081	257.233627974
6	null	2025-03-07 00:00:00.000	239.754200696	220.860783141	258.39452694
7	null	2025-03-10 00:00:00.000	239.808566443	218.876468265	260.626468509

Fig. 6. Creating Forecasting Model 1

As shown in Figure 6, the forecasting model predicts a steady increase in Apple’s stock price over the next seven days. From an investment perspective, this suggests that purchasing shares now, while the price is lower, may be advantageous. However, it is important to note that this model does not account for external factors such as market sentiment, economic events, or negative publicity, which could impact stock price fluctuations.