| Task | PySpark Usage | Statistics Usage |
|---|---|---|
| **Data Loading and Cleaning** | Large-scale data handling and cleaning | Basic summary statistics |
| **Time-Series Forecasting** | Time-series transformations | ARIMA, moving averages, statistical interpretation |
| **Volatility Analysis** | Rolling calculations using window functions | Variance, standard deviation calculations |
| **Event-Driven Analysis** | Filtering and aggregations | Hypothesis testing (e.g., t-test) |
| **Volume-Price Correlation** | Aggregation of volume and price data | Correlation coefficients, interpreting relationships |
| **Seasonal Trends** | Grouping by time periods (months/quarters) | Seasonal decomposition and trend analysis |
| **Moving Average Crossover Strategy** | Calculating moving averages | Performance metrics like Sharpe ratio |
| **Bollinger Bands** | Moving average and standard deviation calculations | Overbought/oversold analysis |
| **Daily Returns** | Calculating daily returns | Distributional analysis |
| **Price Alerts** | Streaming/real-time checks on price changes | Threshold setting based on statistical analysis |

---

**Project Title: Stock Market Data Analysis and Trend Forecasting using PySpark**

**Step 1: Project Setup and Data Understanding**

**Phase 1: Project Setup and Data Understanding**

1. **Objective**: Set up the environment and understand the data structure.
2. **Tasks**:
    - Set up a PySpark environment (using Jupyter, Databricks, or local PySpark installation).

    - Load the stock market data (e.g., Apollo.csv).
    - Use .show(), .printSchema(), and .describe() to explore the dataset.
3. **Key PySpark Functions**: spark.read.csv(), df.show(), df.printSchema(), df.describe().
4. **Deliverable**: Initial report on data structure, column types, and basic statistics.

### 1.1 Initialize the Spark Session

First, we need to start a Spark session, which allows us to use PySpark and SparkSQL.

```
from pyspark.sql import SparkSession

# Initialize Spark session
spark = SparkSession.builder \
   .appName("ApolloStockAnalysis") \
   .getOrCreate()
```

---

### 1.2 Load the Data into a DataFrame

Next, load the apollo.csv file into a Spark DataFrame. This step will read the data and help us perform SQL-based analysis later.

```
# Load the Apollo stock data CSV file into a DataFrame
file_path = "/path/to/apollo.csv"  # Update this path with the actual file location
df = spark.read.csv(file_path, header=True, inferSchema=True)
```

- **header=True**: Indicates that the first row of the CSV file contains column headers.
- **inferSchema=True**: Automatically infers the column data types.

---

### 1.3 Basic Data Exploration

#### a) Check the Schema

After loading, check the schema to understand the data types of each column. This is important because PySparkSQL will rely on correct data types for querying.

```
df.printSchema()
```

This will output the data types of each column, e.g., Date (string), Close (float), etc. If any columns have incorrect types, we'll need to convert them.

**b) Show Initial Rows**

View the first few rows to get a sense of the data and identify any obvious issues, like extra headers or nulls.

```
df.show(5)
```

This step will give an overview of the dataset, showing the values in columns like Date, Open, High, Low, Close, Volume, etc.

**c) Summary Statistics**

Get basic summary statistics to understand the distribution of numerical columns. This is useful for spotting potential outliers or inconsistencies.

```
df.describe().show()
```

This command provides descriptive statistics, such as mean, stddev, min, and max for numerical columns like Open, Close, and Volume.

---

**1.4 SQL-based Data Understanding**

To leverage PySparkSQL, we need to register the DataFrame as a temporary SQL table, which enables querying using SQL syntax.

```
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("apollo_stock")
```

Now, you can query the data directly using SQL syntax.

**a) SQL Query: Check for Null Values**

Query to check for any NULL values across each column. This will help in understanding data completeness and identifying columns that may need cleaning.

```
# SQL query to count NULL values for each column
spark.sql("""
   SELECT
      SUM(CASE WHEN Date IS NULL THEN 1 ELSE 0 END) AS Date_NullCount,
      SUM(CASE WHEN Open IS NULL THEN 1 ELSE 0 END) AS Open_NullCount,
      SUM(CASE WHEN High IS NULL THEN 1 ELSE 0 END) AS High_NullCount,
      SUM(CASE WHEN Low IS NULL THEN 1 ELSE 0 END) AS Low_NullCount,
      SUM(CASE WHEN Close IS NULL THEN 1 ELSE 0 END) AS Close_NullCount,
      SUM(CASE WHEN Volume IS NULL THEN 1 ELSE 0 END) AS Volume_NullCount
   FROM apollo_stock
""").show()
```

This query provides the count of NULL values in each column, which is helpful in determining whether any columns require filling or dropping missing data.

**b) SQL Query: Summary Statistics by SQL**

You can also compute summary statistics for specific columns using SQL, which might be useful if you're focusing on certain columns or aggregating statistics by time intervals.

```
# Summary statistics for the 'Close' column
spark.sql("""
   SELECT
      MIN(Close) AS Min_Close,
      MAX(Close) AS Max_Close,
      AVG(Close) AS Avg_Close,
      STDDEV(Close) AS StdDev_Close
   FROM apollo_stock
""").show()
```

This provides a quick statistical overview of the Close prices for Apollo stock, including the minimum, maximum, average, and standard deviation, helping to understand price ranges and volatility.

---

**1.5 Data Quality Checks**

To ensure the data is ready for analysis, perform additional quality checks:

**a) Check for Duplicates**

Duplicates in stock data can skew analysis, so it's essential to identify and handle them.

```
# Count duplicate rows
duplicate_count = df.count() - df.dropDuplicates().count()
print(f"Number of duplicate rows: {duplicate_count}")
```

If duplicates exist, you may choose to remove them using .dropDuplicates().

**b) Date Consistency Check**

Verify that dates are sequential with no gaps, as missing dates could affect time-series analysis.

```
# Check for date consistency
date_check_df = spark.sql("""
    SELECT Date
    FROM apollo_stock
    ORDER BY Date
""")
date_check_df.show(10)  # Display first 10 dates to inspect
```

---

**Phase 2: Data Cleaning and Preparation**

1. **Objective**: Clean and prepare the data for analysis.
2. **Tasks**:
   - Remove or handle any rows with missing data.
   - Convert columns like Date into the appropriate format and remove unnecessary rows or headers.
   - Add new columns for Daily Returns (percentage change) and moving averages (e.g., 50-day and 200-day).
3. **Key PySpark Functions**: .dropna(), .withColumn(), .cast(), window().
4. **Deliverable**: Cleaned dataset with added columns for daily returns and moving averages.

**Step 2: Data Cleaning and Preparation**

In this step, we will clean the data, handle any missing values, transform data types as needed, and prepare additional columns for analysis. Here's a step-by-step guide to performing these tasks on the Apollo stock data using PySpark.

---

**2.1 Handle Missing Values**

Based on the analysis from Step 1, we should handle any missing values to ensure data integrity. There are several options for dealing with missing values, such as dropping rows, filling with a specific value, or forward filling (for time series).

**a) Drop Rows with Missing Values in Essential Columns**

If columns like Date, Close, or Volume are essential for your analysis, you may want to drop rows where these values are missing.

```
# Drop rows where essential columns have null values
df_cleaned = df.dropna(subset=["Date", "Close", "Volume"])
```

**b) Fill Missing Values with Forward Fill (if applicable)**

For time-series data, it's common to fill missing values by carrying forward the last known value. This can be done by creating a window function.

```
from pyspark.sql.window import Window
from pyspark.sql.functions import last

# Define a window for forward fill (based on Date ordering)
window_spec = Window.orderBy("Date").rowsBetween(Window.unboundedPreceding, 0)

# Forward fill missing values in 'Close' column
df_cleaned = df_cleaned.withColumn("Close_filled", last("Close",
ignorenulls=True).over(window_spec))
```

**Note**: You can apply the same process to other columns like Open, High, Low, etc., as needed.

---

**2.2 Convert Data Types**

Check if the Date column is in the correct format. If it's a string, convert it to a date type to facilitate time-based operations and analyses.

```
from pyspark.sql.functions import to_date

# Convert 'Date' column to DateType
df_cleaned = df_cleaned.withColumn("Date", to_date("Date", "yyyy-MM-dd"))
```

---

### 2.3 Add New Columns for Analysis

To gain insights into stock trends, add calculated fields such as daily returns and moving averages.

#### a) Calculate Daily Returns

Daily returns are the percentage change in the closing price from one day to the next. This metric helps understand price fluctuations.

```
from pyspark.sql.functions import lag, col

# Create a window to calculate the previous day's Close price
window_spec = Window.orderBy("Date")

# Calculate daily returns as a percentage change from the previous day
df_cleaned = df_cleaned.withColumn("Prev_Close", lag("Close").over(window_spec))
df_cleaned = df_cleaned.withColumn("Daily_Return", ((col("Close") - col("Prev_Close")) /
col("Prev_Close")) * 100)
```

#### b) Calculate Moving Averages

Moving averages smooth out price data, making trends more visible. You can calculate a 50-day and 200-day moving average.

```
from pyspark.sql.functions import avg

# Define windows for 50-day and 200-day moving averages
window_50 = Window.orderBy("Date").rowsBetween(-49, 0)  # 50-day window
window_200 = Window.orderBy("Date").rowsBetween(-199, 0)  # 200-day window

# Calculate 50-day and 200-day moving averages for 'Close' prices
```

```
df_cleaned = df_cleaned.withColumn("SMA_50", avg("Close").over(window_50))
df_cleaned = df_cleaned.withColumn("SMA_200", avg("Close").over(window_200))
```

---

### 2.4 Remove Unwanted Columns

If columns like Ticker or others are not needed, you can drop them to streamline the dataset.

```
# Drop unnecessary columns
df_cleaned = df_cleaned.drop("Ticker", "Prev_Close")  # Adjust as needed
```

---

### 2.5 Final Data Check

After cleaning and preparation, it's essential to perform a final check on the data. This includes verifying data types, checking for any remaining null values, and confirming that the new columns were added correctly.

```
# Verify schema and show final data preview
df_cleaned.printSchema()
df_cleaned.show(5)
```

---

### Summary of Step 2

After completing Step 2, you should have a cleaned and prepared DataFrame that includes:

1. **Date column in the correct format**.
2. **Filled or removed missing values**.
3. **New columns** for Daily_Return, SMA_50, and SMA_200 for trend analysis.

### Deliverables for Step 1

1. **Data Schema**: Understand the data types and columns.
2. **Initial Data Overview**: Show the first few rows and check for any immediate issues.
3. **Summary Statistics**: Basic statistics for numerical columns.
4. **Missing Data Analysis**: Null counts for each column.
5. **Data Quality Report**: Check for duplicates and date consistency.

---

**Phase 3: Exploratory Data Analysis (EDA)**

1. **Objective**: Explore historical stock trends to derive initial insights.
2. **Tasks**:
   - Calculate and visualize descriptive statistics for Open, Close, Volume, etc.
   - Plot daily returns to understand price fluctuation.
   - Calculate and analyze volatility (standard deviation of daily returns) over different periods.
3. **Key PySpark Functions**: .groupBy(), .agg(), window() for rolling calculations.
4. **Deliverable**: EDA report with descriptive statistics, volatility analysis, and initial visualizations.

**Step 3: Exploratory Data Analysis (EDA)**

In this step, we'll explore the cleaned **Apollo stock data** to understand patterns, trends, and other insights. This process involves summary statistics, visualizing data distributions, and looking at stock price trends over time. We'll use both **PySpark DataFrame API** and **PySparkSQL** for EDA tasks.

---

**3.1 Summary Statistics**

Calculate basic statistics for key columns to understand the distribution of values and spot any outliers or anomalies.

```
# Summary statistics for numerical columns

df_cleaned.describe(["Open", "High", "Low", "Close", "Volume", "Daily_Return"]).show()
```

This provides statistics such as the mean, standard deviation, min, and max for columns like Close, Volume, and Daily_Return.

**Using PySparkSQL:**

```
# Register DataFrame as a temporary SQL view

df_cleaned.createOrReplaceTempView("apollo_stock_cleaned")
```

```
# SQL query to calculate summary statistics for the 'Close' and 'Volume' columns

spark.sql("""

    SELECT

        MIN(Close) AS Min_Close,

        MAX(Close) AS Max_Close,

        AVG(Close) AS Avg_Close,

        STDDEV(Close) AS StdDev_Close,

        MIN(Volume) AS Min_Volume,

        MAX(Volume) AS Max_Volume,

        AVG(Volume) AS Avg_Volume,

        STDDEV(Volume) AS StdDev_Volume

    FROM apollo_stock_cleaned

""").show()
```

---

**3.2 Analyze Daily Returns**

Daily returns show the percentage change in the closing price from one day to the next. Analyzing the mean and standard deviation of daily returns helps assess volatility and average movement.

```
# Summary of daily returns

spark.sql("""

    SELECT

        AVG(Daily_Return) AS Avg_Daily_Return,
        STDDEV(Daily_Return) AS Daily_Return_StdDev,

        MIN(Daily_Return) AS Min_Daily_Return,

        MAX(Daily_Return) AS Max_Daily_Return

    FROM apollo_stock_cleaned

""").show()
```

**Interpretation**:

- High standard deviation in daily returns indicates high volatility.
- The average daily return gives an idea of the typical daily price change.

---

**3.3 Trend Analysis with Moving Averages**

Moving averages help smooth out stock price fluctuations to reveal underlying trends.

**a) Calculate Rolling Averages**

Using SQL, you can calculate averages over different timeframes and compare them.

```
spark.sql("""

    SELECT

        Date,

        Close,

        SMA_50,

        SMA_200,

        CASE WHEN SMA_50 > SMA_200 THEN 'Uptrend' ELSE 'Downtrend' END AS Trend

    FROM apollo_stock_cleaned

    ORDER BY Date

""").show(10)
```

**Interpretation**:

- When the 50-day moving average (SMA_50) is above the 200-day moving average (SMA_200), it generally indicates an upward trend, and vice versa.

---

**3.4 Volume Analysis**

Analyze trading volume trends to see if higher volumes correlate with significant price movements, which could indicate periods of high investor interest or major events.

```
# SQL query to find days with the highest volume and corresponding price changes

spark.sql("""

   SELECT

      Date,

      Volume,

      Close,

      Daily_Return

   FROM apollo_stock_cleaned

   ORDER BY Volume DESC

   LIMIT 10

""").show()
```

This query highlights the days with the highest trading volumes and their corresponding daily returns. Spikes in volume often accompany big price moves, either up or down.

---

**3.5 Seasonal Patterns Analysis**

Check if there are any seasonal trends by looking at monthly or quarterly performance.

**Monthly Average Close Price and Volume**

```
from pyspark.sql.functions import month


# Extract month from the Date column and calculate average Close price and Volume by month
df_monthly = df_cleaned.withColumn("Month", month("Date"))
df_monthly.groupBy("Month").avg("Close", "Volume").orderBy("Month").show()
```

**Using SQL**:

```
# Calculate average close price and volume by month using SQL
spark.sql("""
   SELECT
      MONTH(Date) AS Month,
      AVG(Close) AS Avg_Monthly_Close,
      AVG(Volume) AS Avg_Monthly_Volume
   FROM apollo_stock_cleaned
   GROUP BY MONTH(Date)
   ORDER BY Month
""").show()
```

**Interpretation**:

- Monthly averages can reveal if certain months have higher or lower average closing prices and trading volumes, which could indicate seasonal trends.

---

## 3.6 Identify High and Low Volatility Periods

High volatility periods often indicate periods of uncertainty or high trading activity. Use the Daily_Return column's standard deviation to analyze volatility.

**Identify Volatile Days**

```
# Query to find the most volatile days based on absolute daily returns

spark.sql("""

    SELECT

        Date,

        Close,

        Daily_Return

    FROM apollo_stock_cleaned

    ORDER BY ABS(Daily_Return) DESC

    LIMIT 10

""").show()
```

**Interpretation**:

- This lists days with the largest price swings, which could correlate with market events, earnings announcements, or other impactful news.

---

**Phase 4: Correlation and Trend Analysis**

1. **Objective**: Investigate relationships between variables and detect trends.
2. **Tasks**:
    - Calculate the correlation between Volume and Price.
    - Identify seasonal trends by grouping data by month or quarter.
    - Use moving averages and Bollinger Bands to observe buy/sell signals.
3. **Key PySpark Functions**: .corr(), window(), .groupBy().

4. **Deliverable**: Report on key correlations, trends, and any patterns observed using moving averages.

**Step 4: Correlation and Trend Analysis**

In this phase, we'll investigate relationships between key variables and analyze long-term trends in the stock price of Apollo. This involves calculating correlation coefficients, analyzing moving averages, and detecting patterns in price movements.

---

**4.1 Correlation Analysis**

Understanding the correlation between variables like Volume, Close, High, and Low can help reveal relationships in the data. For instance, high trading volumes might correlate with price volatility.

**a) Correlation between Close Price and Volume**

Calculate the correlation coefficient to measure the strength of the relationship between the Close price and Volume. High correlation might indicate that as trading volume increases, the price tends to move in a particular direction.

```
# Calculate correlation between Close and Volume

close_volume_corr = df_cleaned.stat.corr("Close", "Volume")

print(f"Correlation between Close and Volume: {close_volume_corr}")
```

**Using PySparkSQL:**

```
# SQL query to calculate correlation between Close price and Volume

spark.sql("""

    SELECT corr(Close, Volume) AS Close_Volume_Correlation

    FROM apollo_stock_cleaned

""").show()
```

**Interpretation**:

- A positive correlation suggests that higher volumes are associated with higher closing prices, while a negative correlation suggests the opposite. Near-zero correlation would indicate no strong relationship.

---

**b) Correlation Among Other Variables**

Analyze correlations between other variables like Open, High, Low, and Close.

```
# SQL query for correlation matrix between key stock metrics

spark.sql("""

   SELECT

      corr(Open, Close) AS Open_Close_Correlation,

      corr(High, Close) AS High_Close_Correlation,

      corr(Low, Close) AS Low_Close_Correlation,

      corr(Volume, Close) AS Volume_Close_Correlation

   FROM apollo_stock_cleaned

""").show()
```

**Interpretation**:

- Correlation values closer to ±1 indicate a strong relationship, while values near 0 indicate weak or no relationship.

---

**4.2 Trend Analysis with Moving Averages**

Moving averages help identify long-term trends and reduce the impact of short-term fluctuations. We already calculated the 50-day and 200-day moving averages (SMA_50 and SMA_200) in previous steps.

**a) Crossovers in Moving Averages**

A common trend analysis technique is to identify "crossovers" between short-term and long-term moving averages. When the short-term average crosses above the long-term average, it may indicate a bullish trend, and vice versa.

```
# Identify crossovers between SMA_50 and SMA_200

spark.sql("""

  SELECT

    Date,

    Close,

    SMA_50,

    SMA_200,

    CASE

      WHEN SMA_50 > SMA_200 THEN 'Uptrend'

      ELSE 'Downtrend'

    END AS Trend

  FROM apollo_stock_cleaned

  ORDER BY Date

""").show(10)
```

**Interpretation**:

- Uptrend indicates a potential bullish phase, while Downtrend indicates a bearish phase. This helps in understanding long-term market trends.

---

**4.3 Price Volatility Analysis**

Volatility indicates how much the price fluctuates over a given period. High volatility often signals periods of high uncertainty, which might be important for risk management or short-term trading.

**a) Calculate Rolling Standard Deviation (Volatility)**

Calculate the rolling standard deviation of the daily returns as a measure of volatility. This will help determine periods when the stock was more volatile.

```
from pyspark.sql.functions import stddev



# Define a 30-day window for volatility calculation

window_30 = Window.orderBy("Date").rowsBetween(-29, 0)



# Calculate 30-day rolling standard deviation of Daily_Return

df_cleaned = df_cleaned.withColumn("Volatility_30",
stddev("Daily_Return").over(window_30))
```

**Using SQL**:

```
# SQL query to calculate 30-day rolling volatility for Daily_Return

spark.sql("""

   SELECT

      Date,

      Close,

      Daily_Return,

      stddev(Daily_Return) OVER (ORDER BY Date ROWS BETWEEN 29 PRECEDING
AND CURRENT ROW) AS Volatility_30

   FROM apollo_stock_cleaned
```

```
    ORDER BY Date
""").show(10)
```

**Interpretation**:

- High Volatility_30 values indicate periods of significant price swings, which might correlate with market events or economic conditions.

---

### 4.4 Detecting Seasonal Patterns

If stock prices exhibit seasonal patterns (e.g., specific months or quarters), understanding these can be beneficial for making strategic decisions.

#### a) Monthly Trends in Closing Prices

Identify average closing prices by month to see if any months consistently show higher or lower averages, which might indicate a seasonal effect.

```
# Calculate average close price by month
spark.sql("""
    SELECT
        MONTH(Date) AS Month,
        AVG(Close) AS Avg_Close
    FROM apollo_stock_cleaned
    GROUP BY MONTH(Date)
    ORDER BY Month
""").show()
```

**Interpretation**:

- If certain months have consistently higher or lower prices, it may indicate a seasonal trend, such as annual increases in healthcare demand affecting Apollo's stock.

---

**4.5 Detect Anomalous Trends**

Identify outliers in daily returns, which may point to unusual events or errors in the data.

```
# Identify dates with unusually high or low daily returns

spark.sql("""

   SELECT

      Date,

      Close,

      Daily_Return

   FROM apollo_stock_cleaned

   WHERE ABS(Daily_Return) > (SELECT AVG(Daily_Return) + 3 * STDDEV(Daily_Return)
FROM apollo_stock_cleaned)

   ORDER BY ABS(Daily_Return) DESC

""").show(10)
```

**Interpretation**:

- High or low returns (outliers) could signal impactful events such as earnings releases or other news that significantly moved the stock price.

---

**Summary of Step 4 Deliverables**

After completing Step 4, you should have:

1. **Correlation Analysis**: Understanding of relationships between variables such as Volume, Close, Open, etc.

2. **Moving Average Crossovers**: Insights into bullish and bearish trends using 50-day and 200-day moving averages.
3. **Volatility Analysis**: Rolling volatility analysis to detect high-risk periods.
4. **Seasonal Trends**: Monthly or quarterly trends in closing prices to identify any seasonality.
5. **Outlier Detection**: Identification of dates with anomalous returns to investigate potential significant events.

---

**Phase 5: Time-Series Forecasting (Optional Advanced)**

1. **Objective**: Develop a time-series model to forecast future prices.
2. **Tasks**:
   - Choose a model, such as moving average, exponential smoothing, or ARIMA.
   - Split data into training and testing sets.
   - Train the model on historical data and evaluate on test data.
3. **Key PySpark Libraries**: MLlib (for time-series model implementation).
4. **Deliverable**: Forecast model with accuracy metrics and visualizations comparing predicted vs. actual prices.

---

**Phase 6: Strategic Insights and Recommendations**

1. **Objective**: Derive actionable insights from the analysis.
2. **Tasks**:
   - Summarize key findings, including trends, volatility insights, and potential buy/sell indicators.
   - Formulate strategic recommendations based on observed trends (e.g., periods of high volatility, ideal entry/exit points).
3. **Deliverable**: A strategic report detailing insights and investment recommendations.

**Step 5: Strategic Insights and Recommendations**

In this phase, we synthesize the findings from previous analyses to derive actionable insights and strategic recommendations. This step is essential as it bridges the gap between raw data analysis and real-world decision-making, providing Apollo stock investors or business strategists with informed perspectives.

## 5.1 Key Findings

Based on our analysis of the Apollo stock data, here are some potential insights and interpretations:

1. **Price Trends (Moving Average Crossovers)**
   - **Insight**: The analysis of moving average crossovers (e.g., 50-day vs. 200-day) reveals patterns in price trends. Bullish crossovers (where the short-term average exceeds the long-term average) may indicate upward momentum, while bearish crossovers suggest downward trends.
   - **Recommendation**: Monitor moving average crossovers to signal potential buy or sell opportunities. Investors might consider buying when a bullish crossover occurs and selling during a bearish crossover.
2. **Volatility Analysis**
   - **Insight**: Periods with high volatility (identified via rolling standard deviation of daily returns) often coincide with significant events or announcements, potentially leading to rapid price changes.
   - **Recommendation**: High volatility days can indicate both risk and opportunity. For risk-averse investors, it may be best to avoid trading during high volatility. Conversely, risk-tolerant traders could explore short-term trading strategies to capitalize on these swings.
3. **Trading Volume and Price Correlation**
   - **Insight**: We observed that trading volume correlates moderately with price fluctuations. Large volume spikes tend to coincide with significant price movements, suggesting investor interest during these periods.
   - **Recommendation**: Use volume as a secondary indicator for confirming price trend signals. For example, if a bullish price trend coincides with high trading volume, it may indicate stronger market confidence in the upward movement.
4. **Seasonal Patterns**
   - **Insight**: Seasonal analysis shows that certain months or quarters might exhibit consistent patterns in average closing prices and trading volume. For instance, there may be price increases in quarters where Apollo sees increased healthcare demand.
   - **Recommendation**: Leverage these seasonal patterns for strategic entry and exit points. Investors can plan their investments to align with periods of historically higher returns, maximizing their chances of favorable outcomes.
5. **Outliers and Event-Based Insights**

    - **Insight**: Significant outliers in daily returns (both positive and negative) typically correspond to impactful events, such as earnings announcements, new product launches, or industry news.
    - **Recommendation**: Monitor external events closely, as they significantly impact stock price. Using event-based alert systems (e.g., notifications on earnings announcements), investors can stay informed and make timely trading decisions.

---

### 5.2 Strategic Recommendations for Investors and Analysts

Based on the insights, here are some tailored recommendations for different types of investors and analysts:

1. **For Long-Term Investors**:
    - Focus on observing moving average crossovers and volatility trends to identify long-term entry or exit points.
    - Use the 200-day moving average as a support or resistance indicator to guide decisions on holding or selling.
    - Avoid reacting to short-term fluctuations and high volatility periods unless they indicate substantial, sustained changes.
2. **For Short-Term Traders**:
    - Leverage high-volume days and short-term volatility as trading opportunities.
    - Pay close attention to the 50-day moving average and recent price trends for identifying entry and exit points.
    - Consider using automated alerts based on volume spikes or unusual daily returns to capitalize on short-term trends.
3. **For Analysts and Portfolio Managers**:
    - Integrate seasonal trends into market forecasting models, aligning portfolios with known seasonal patterns.
    - Use correlation insights to balance portfolios based on relationships between Volume, Close, and Daily_Return, understanding the interdependencies in price movement.
    - Track industry-specific and macroeconomic events as they may significantly impact Apollo's stock performance.

---

### 5.3 Summary of Recommendations

- **Buy Signals**: Use bullish moving average crossovers, high trading volume, and seasonal trends as potential buy signals.

- **Sell Signals**: Monitor bearish crossovers, high volatility, and volume drop-offs as possible indicators for selling or reducing holdings.
- **Risk Management**: Be cautious during high volatility periods and watch for outliers or extreme daily returns that may indicate market overreaction or instability.
- **Event-Based Strategy**: Track quarterly reports, regulatory changes, and major healthcare announcements to inform timely trades.

---

**Deliverable: Strategic Insights Report**

Prepare a report summarizing:

1. **Findings and Insights**: Key patterns, trends, and correlations from the data analysis.
2. **Investment Recommendations**: Actionable strategies tailored to different investor profiles.
3. **Supporting Visualizations**: Graphs and tables from previous steps that substantiate recommendations, such as moving average trends and volume-price correlation charts.

---

**Phase 7: Presentation and Documentation**

1. **Objective**: Compile and present findings in a structured report or presentation.
2. **Tasks**:
    - Document the code, including explanations of key functions and methods.
    - Prepare visualizations and graphs summarizing key points from each phase.
    - Present findings and insights as a PowerPoint or written report.
3. **Deliverable**: Final presentation with visual summaries, documentation, and project conclusions.

**Phase 7: Presentation and Documentation**

In this final phase, we consolidate the project findings into a clear and structured presentation and document the entire process. This helps communicate insights effectively and provides a reference for future analysis.

---

**7.1 Structure the Presentation**

A well-organized presentation is essential for conveying the analysis results, key findings, and strategic recommendations. Here's an outline to structure your presentation:

1. **Introduction**:
   - **Project Objectives**: Briefly explain the purpose of the project, such as analyzing Apollo's stock data to identify trends, correlations, and make investment recommendations.
   - **Data Overview**: Describe the dataset (columns, date range, and any key features) and data sources.
2. **Data Preparation and Cleaning**:
   - Summarize the data cleaning process, including how missing values were handled and data types converted.
   - Include visuals like a summary of missing values before and after cleaning, or examples of calculated columns (e.g., Daily Returns, Moving Averages).
3. **Exploratory Data Analysis (EDA)**:
   - Present key findings from EDA, such as:
     - Summary statistics (mean, standard deviation) of key columns.
     - Monthly or seasonal trends observed in the Close prices and volume.
     - High volatility periods and high-volume days.
   - Use visuals (e.g., line plots for monthly trends, bar charts for volume analysis).
4. **Correlation and Trend Analysis**:
   - Show correlation results between variables (e.g., Close and Volume).
   - Present moving average crossover trends with graphs depicting 50-day and 200-day moving averages and the crossovers as buy/sell signals.
   - Include charts for volatility trends and any outliers or unusual return events.
5. **Strategic Insights and Recommendations**:
   - Summarize insights from previous phases, highlighting:
     - Key trends (bullish/bearish indicators from moving averages).
     - Seasonal patterns that investors can leverage.
     - Recommendations for different types of investors (e.g., long-term vs. short-term).
   - Use a concise format with bullet points to communicate actionable insights effectively.
6. **Conclusion**:
   - Recap the main findings and how they support the strategic recommendations.
   - Emphasize the potential impact of the insights on investment decisions.

---

**7.2 Documentation of the Analysis Process**

Document the analysis in a report or notebook that includes code, explanations, and outputs for reproducibility. A well-documented notebook or report allows others to understand the steps taken and follow the logic of the analysis.

**Sections to Include in Documentation:**

1. **Introduction**:
   - Outline the project goals and scope.
2. **Data Loading and Preparation**:
   - Include code snippets for loading, cleaning, and transforming the data.
   - Document each data cleaning decision, such as how missing values were handled and why.
3. **Exploratory Data Analysis (EDA)**:
   - Provide code and output summaries for initial data exploration.
   - Include key visualizations created during EDA, with explanations.
4. **Correlation and Trend Analysis**:
   - Document the code and methodology for calculating correlations and moving averages.
   - Provide clear explanations of each trend or insight, supported by visuals.
5. **Strategic Insights and Recommendations**:
   - Summarize the main insights, referencing specific analysis results that support each recommendation.
6. **Conclusion**:
   - Conclude with a summary of findings, insights, and recommendations.

---

### 7.3 Visualization Tips

Visualizations are essential to make the findings more understandable. Here are some tips:

- **Use Line Charts** for time-series data, such as stock price and moving averages.
- **Bar Charts** work well for comparing data across categories, such as average monthly volumes.
- **Scatter Plots** can illustrate correlations between variables, like Volume and Close.
- **Annotations**: Label significant events, such as earnings reports or buy/sell signals, to give context to trends.

---

### 7.4 Final Presentation

Using PowerPoint, Google Slides, or Jupyter Notebooks (for an interactive option), assemble the presentation with a focus on clarity and conciseness. Use each slide to highlight specific findings and supplement with visuals.

---

### 7.5 Final Report

The final report should serve as a reference document and include both technical and strategic sections. The report can be delivered in PDF format or as a Jupyter Notebook with Markdown cells documenting the analysis.

---

### Summary of Key Project Deliverables

1. **Data Understanding Report**: Summary of the dataset structure.
2. **Cleaned Data File**: Prepared dataset with calculated columns (returns, moving averages).
3. **EDA Report**: Descriptive statistics, volatility analysis, and initial visualizations.
4. **Trend and Correlation Report**: Analysis of correlations and detected trends.
5. **Forecasting Model Output (Optional)**: Predictive model and its performance metrics.
6. **Strategic Insights Report**: Summary of insights and strategic recommendations.
7. **Final Presentation**: Documented findings with visuals, code explanations, and recommendations.

---

### 1. Identify Missing Data

Before handling missing data, identify where and how much missing data exists. This helps in choosing the most suitable approach for filling or dropping the missing values.

```
# Count missing values in each column
from pyspark.sql.functions import col, sum

# Check for missing values (nulls) in each column
df.select([sum(col(column).isNull().cast("int")).alias(column) for column in df.columns]).show()
```

This code snippet shows the count of null values in each column, giving an overview of missing data in your DataFrame.

---

**2. Dropping Missing Data**

PySpark allows you to drop rows with missing values using the .dropna() function. You can specify certain conditions for dropping rows:

- **Drop rows with any missing values**: Removes rows where any column has a NULL.
- **Drop rows with all missing values**: Removes rows where all columns are NULL.
- **Drop rows based on a subset of columns**: Drops rows based on missing values in specific columns.
- **Specify a threshold**: Drop rows if they have fewer than a certain number of non-null values.

**Examples:**

```
# Drop rows where any column has a NULL value
df_cleaned = df.dropna("any")

# Drop rows where all columns are NULL
df_cleaned = df.dropna("all")

# Drop rows based on missing values in specific columns (e.g., "Close" and "Volume")
df_cleaned = df.dropna(subset=["Close", "Volume"])

# Drop rows with fewer than a specified number of non-null values (e.g., require at least 3
non-null values)
df_cleaned = df.dropna(thresh=3)
```

---

**3. Filling Missing Data**

Filling missing values can be useful when you want to maintain the dataset's structure without removing rows. PySpark's .fillna() function allows you to fill missing values in different ways.

**Fill with a Constant Value**

You can replace NULL values in a column with a specified constant. This is useful for categorical data or setting missing values to a default, like 0.

```
# Fill missing values with a constant value (e.g., 0)
```

```
df_filled = df.fillna(0)

# Fill specific columns with constant values
df_filled = df.fillna({"Close": 0, "Volume": 1})
```

**Fill with the Mean, Median, or Mode**

For numerical data, filling with the mean, median, or mode can be effective. Since PySpark doesn't have a direct function to fill missing values with these statistics, you can calculate them separately and use .fillna().

```
# Calculate the mean of the "Close" column
from pyspark.sql.functions import mean

mean_close = df.select(mean(col("Close"))).collect()[0][0]
df_filled = df.fillna({"Close": mean_close})
```

**Forward and Backward Fill**

For time-series data, such as stock prices, forward fill (also known as "carry forward") or backward fill is often more appropriate. However, PySpark lacks direct support for these. You can implement forward/backward fill by creating a window function.

**Example of Forward Fill**

```
from pyspark.sql.window import Window
from pyspark.sql.functions import last

# Define a window and perform forward fill
window_spec = Window.orderBy("Date").rowsBetween(Window.unboundedPreceding, 0)
df_filled = df.withColumn("Close_filled", last("Close", ignorenulls=True).over(window_spec))
```

This code carries forward the last known non-null value within a given time window.

**Interpolation**

Interpolation involves estimating missing values based on surrounding values. PySpark does not support direct interpolation, but linear interpolation can be approximated by calculating averages between known values. For complex interpolations, consider moving the data to Pandas or using Spark in combination with Pandas.

---

**5. Using Conditional Filling**

Conditional filling allows you to replace missing values based on other columns or conditions. This approach is useful when domain-specific rules define how missing values should be treated.

```
from pyspark.sql.functions import when

# Example: If "Close" is NULL, set it to 0; otherwise, leave it unchanged
df_filled = df.withColumn("Close", when(col("Close").isNull(), 0).otherwise(col("Close")))
```

---

**Summary of Methods**

| Method | Description | PySpark Function |
|---|---|---|
| **Drop rows** | Remove rows with missing values | df.dropna() |
| **Fill with constant** | Replace missing values with a constant (e.g., 0 or "unknown") | df.fillna() |
| **Fill with mean/median** | Replace missing values with the column mean or median | df.fillna() + manual |
| **Forward/backward fill** | Carry forward the last known value | Window + last() |
| **Conditional filling** | Replace values based on conditions | when() |

**Step 1: Calculate Missing Data Counts in PySpark**

Start by calculating the missing values for each column or within specific groups, then collect the results to visualize in Pandas or directly in Python.

```
from pyspark.sql.functions import col, sum

# Calculate the number of missing values for each column
```

```
missing_counts = df.select([sum(col(column).isNull().cast("int")).alias(column) for column in df.columns])
missing_counts.show()
```

If you want to view missing values as percentages:

```
total_rows = df.count()
missing_percentages = missing_counts.select([(col(column) / total_rows * 100).alias(column) for column in df.columns])
missing_percentages.show()
```

After calculating missing values, you can convert the data into a Pandas DataFrame to facilitate visualization.

---

**Step 2: Convert Results to Pandas for Visualization**

Once you have the counts or percentages of missing values, you can convert the PySpark DataFrame into a Pandas DataFrame:

```
# Collect the missing data into a Pandas DataFrame
missing_counts_pd = missing_counts.toPandas()
missing_percentages_pd = missing_percentages.toPandas()
```

---

**Step 3: Visualize Missing Data with Matplotlib and Seaborn**

Using Pandas, Matplotlib, and Seaborn, you can create visualizations that help illustrate missing data patterns.

**1. Bar Plot of Missing Values per Column**

A bar plot shows the number of missing values for each column, giving a quick overview of the data completeness.

```
import matplotlib.pyplot as plt

# Bar plot of missing counts
```

```
missing_counts_pd.T.plot(kind='bar', legend=False)
plt.ylabel("Count of Missing Values")
plt.title("Missing Values per Column")
plt.show()
```

**2. Heatmap of Missing Data**

A heatmap shows the presence of missing values across a sample of rows, with each cell indicating if data is missing for that entry.

```
import seaborn as sns
# Convert the PySpark DataFrame to Pandas (small sample for visualization)
df_sample = df.limit(100).toPandas()

# Use a heatmap to show missing values
plt.figure(figsize=(10, 6))
sns.heatmap(df_sample.isnull(), cbar=False, cmap="viridis")
plt.title("Missing Data Heatmap")
plt.show()
```

**3. Missing Data Percentage Plot**

A bar plot of the percentage of missing data provides insight into the extent of data absence in each column.

```
# Bar plot of missing percentages
missing_percentages_pd.T.plot(kind='bar', legend=False)
plt.ylabel("Percentage of Missing Values")
plt.title("Percentage of Missing Values per Column")
plt.show()
```

**4. Time-Series Analysis of Missing Data (For Time-Series Data)**

If you're working with time-series data, you can visualize missing data over time to identify trends (e.g., more missing data on weekends or specific time periods).

```
# For time-series, you may need a column like 'Date' to analyze missing data trends over time
df_missing_by_date = df.groupBy("Date").agg(
    *[sum(col(c).isNull().cast("int")).alias(c + "_missing") for c in df.columns]
)
df_missing_by_date_pd = df_missing_by_date.toPandas()

# Plot missing values over time for each column
plt.figure(figsize=(12, 6))
for column in df_missing_by_date_pd.columns[1:]:
    plt.plot(df_missing_by_date_pd['Date'], df_missing_by_date_pd[column], label=column)
plt.xlabel("Date")
plt.ylabel("Count of Missing Values")
plt.title("Missing Data Over Time")
plt.legend()
plt.show()
```