A Time Series Forecast of Stock Prices Using Twitter Sentiment and Financial Data

Ronan Downes SBA22447

September 14, 2024

Contents

1 Data Collection Processing

1.1 Data Loading

To begin the analysis, we load the datasets provided in the ZIP file. This includes tweet data and stock price data for multiple companies, including Apple (AAPL), Amazon (AMZN), Google (GOOG), Microsoft (MSFT), and Tesla (TSLA). The tweet data captures the sentiment expressed in tweets, while the stock price data provides daily stock prices for each company.

1.2 Loading Stock Price Data

The next step involved loading the stock price data for the selected companies: Apple (AAPL), Amazon (AMZN), Google (GOOG), Microsoft (MSFT), and Tesla (TSLA). This data spans the period from January 2020 to December 2020 and contains information on daily stock prices, including fields such as date, open, high, low, close, adjusted close, and trading volume.

The data for each company was stored in separate CSV files, which were imported using Python's `pandas` library.

1.3 Date Conversion

To ensure consistency in our time series analysis, we converted the date columns in both the tweet and stock price datasets to `datetime` format. This step is crucial as it allows us to properly align the data during analysis. Invalid date formats were handled using the `errors='coerce'` option, converting any problematic entries into `NaT` (Not a Time) values for easier identification and cleaning.

1.4 Sentiment Analysis

Sentiment analysis was performed on the tweet data using the `TextBlob` library. Each tweet's text was processed to calculate a sentiment polarity score, which ranges from -1 (negative) to 1 (positive). This score helps identify the overall sentiment of the tweets related to the selected stocks.

The sentiment scores were then aggregated by date to align with the daily stock price data. This aggregation allowed us to assess the average sentiment for each day, providing a temporal dimension that can be used in the forecasting analysis.

1.5 Merging Sentiment Data with Stock Prices

In this step, we merge the daily sentiment data with the stock price data for each selected company (AAPL, AMZN, GOOG, MSFT, TSLA). The sentiment data is joined on the date, allowing us to analyze how daily sentiment might correlate with stock price changes. After merging, we drop redundant columns to clean up the data.

1.6 Storing Processed Data for VM Integration

The processed datasets, including the merged stock prices and sentiment scores, are stored in the `processed_data` directory. This storage step is essential for facilitating integration with the virtual machines (VMs) in the subsequent phases of the analysis. Each VM is configured to access the `processed_data` folder, enabling the following actions:

- **SQL Database VM:** The stock price data is imported into a MySQL database running on the SQL VM. This structured data storage allows for efficient querying and time-series analysis using SQL commands.
- **NoSQL Database VM:** The tweet data and sentiment scores are stored in a MongoDB database on the NoSQL VM. The flexible schema of MongoDB is well-suited for the unstructured nature of tweet text data, making it easier to perform sentiment-related queries.
- **Big Data Processing VM (Hadoop/Spark):** Both the stock price and tweet data are accessed by the Hadoop/Spark VM for distributed processing. Spark jobs are executed to analyze the data at scale, including further feature engineering and aggregation necessary for the forecasting models.

Storing the processed data in the `processed_data` folder creates a unified data source accessible by the various VMs, supporting the subsequent stages of analysis, storage, and processing.

1.7 Feature Engineering with Lag Variables

To capture temporal dependencies within the stock prices and sentiment scores, lag features were engineered for 1, 3, and 7-day intervals. This feature engineering step allows the model to use historical data for future predictions, enabling more robust time-series forecasting. Specifically, lagged features for both the closing prices and the aggregated sentiment scores were generated for each company. Rows with missing values, resulting from the lagging process, were subsequently removed to maintain data integrity.

References

A Appendix Data Collection Processing

A.1 Loading Tweet Dataset

The following Python code snippet demonstrates how the tweet dataset was loaded into the environment:

```
import pandas as pd
import os
import zipfile
import warnings

warnings.filterwarnings(’ignore’)

# Unzip the file directly in the current directory
with zipfile.ZipFile(’stock-tweet-and-price.zip’, ’r’) as zip_ref:
    zip_ref.extractall()

# Load the tweet dataset
tweets_df = pd.read_csv(’stock-tweet-and-price/stocktweet/stocktweet.csv

# List all unique tickers in the ’ticker’ column
unique_tickers = tweets_df[’ticker’].unique()
print(f"Unique tickers in the dataset: {unique_tickers}")

# Display the first few rows of the tweet dataset
tweets_df.head()
```

A.2 Loading Stock Price Data

The following Python code snippet demonstrates how the stock price data for the selected companies was loaded:

```
# Choose companies
companies = [’AAPL’, ’AMZN’, ’GOOG’, ’MSFT’, ’TSLA’]
stock_data = {}
```

```
# Load stock data for each company
for company in companies:
    stock_data[company] = pd.read_csv(f'stock-tweet-and-price/stockprice

# Display the first few rows of one of the company's stock data
stock_data['AAPL'].head()
```

A.3 Date Conversion

The following Python code snippet demonstrates how the date columns in the tweet and stock price datasets were converted to `datetime` format:

```
# Convert tweet dates to datetime and handle any invalid date formats
tweets_df['date'] = pd.to_datetime(tweets_df['date'], format='%d/%m/%Y',

# Convert stock prices 'Date' columns to datetime and handle invalid fo
for company in companies:
    stock_data[company]['Date'] = pd.to_datetime(stock_data[company]['Da
```

A.4 Sentiment Analysis

The following Python code snippet demonstrates how sentiment analysis was performed on the tweets using the `TextBlob` library:

```
from textblob import TextBlob

# Apply sentiment analysis
tweets_df['sentiment'] = tweets_df['tweet'].apply(lambda tweet: TextBlob

# Aggregate sentiment by date
daily_sentiment = tweets_df.groupby('date')['sentiment'].mean().reset_in

# Display the first few rows of the aggregated sentiment data
daily_sentiment.head()
```

A.5 Merging Sentiment and Stock Price Data

The following Python code snippet demonstrates how we merged the daily sentiment data with the stock price data for each selected company:

```
# Merge sentiment data with stock price data for each selected company
for company in companies:
    df = stock_data[company]
    df = df.merge(daily_sentiment, left_on='Date', right_on='date', how=
    df.drop(columns=['date'], inplace=True)
    stock_data[company] = df

# Display the first few rows of the merged data for one company (e.g., A
stock_data['AAPL'].head()
```

B Appendix - Storing Processed Data for VM Integration

The datasets, after undergoing cleaning and preprocessing, were organized into a dedicated folder named `processed_data`. This centralization facilitates easy access and integration with the virtual machines (VMs) set up for the project.

B.1 Data Storage Overview

The `processed_data` folder contains the following:

- Cleaned stock price data for selected companies (e.g., AAPL, AMZN, GOOG, MSFT, TSLA).
- Aggregated daily sentiment scores derived from the tweet data.

B.2 VM Integration

Each VM accesses the `processed_data` folder for subsequent tasks:

- **SQL Database VM:** This VM imports stock price data into a MySQL database for structured storage and analysis.

- **NoSQL Database VM:** The tweet data and sentiment scores are stored in a MongoDB database on the NoSQL VM, allowing for flexible and efficient querying.
- **Big Data Processing VM (Hadoop/Spark):** Both the stock price data and sentiment scores are utilized in distributed processing tasks within the Big Data VM to handle large-scale computations.

By storing processed data centrally, we streamline data access across the VMs, ensuring consistent and efficient integration throughout the project's analysis phases.

B.3 Appendix: Code for Creating Lag Features

The following code snippet demonstrates how lag features were created for both the closing prices and sentiment scores in the dataset. Lag features were introduced for 1, 3, and 7-day intervals to incorporate temporal patterns into the forecasting models.

```
# Function to create lag features for the target column
def create_lag_features(df, lags, target_col):
    """
    Creates lagged features for a specified column in the dataframe.

    Parameters:
    df (pd.DataFrame): The dataframe containing the data.
    lags (list): A list of integers indicating the number of lags to cre
    target_col (str): The column for which lag features are to be create

    Returns:
    pd.DataFrame: The dataframe with new lagged features.
    """
    for lag in lags:
        # Creating lag features for the specified column
        df[f'{target_col}_lag_{lag}'] = df[target_col].shift(lag)
    return df
```

```
# Define the lag intervals to create
lags = [1, 3, 7]

# Loop through each company to create lag features for 'Close' and 'sent
for company in companies:
    df = stock_data[company]
    # Creating lag features for the 'Close' price
    df = create_lag_features(df, lags, 'Close')
    # Creating lag features for the 'sentiment' scores
    df = create_lag_features(df, lags, 'sentiment')

    # Drop rows with NaN values introduced by the lagging process
    df.dropna(inplace=True)

    # Store the updated dataframe back to the stock_data dictionary
    stock_data[company] = df
```