Stock Recommendation System

Rhythm Balooni
rbalooni@uwaterloo.ca

Fadil Meraj
fmeraj@uwaterloo.ca

Ekjot Sandhu
e3sandhu@uwaterloo.ca

Manish
mna@uwaterloo.ca

1. Introduction

The prediction of the stock market has always been fascinating to researchers and investors because they believe it can lead to high financial profits. The dynamic nature of the stock market allows it to be influenced by many factors like historical data, market sentiment, and external events on prices. Most of the basic strategies of stock analysis involve the use of fundamental and technical analyses. While fundamental analysis incorporates the study of the financial position of a company and its relation to the economic environment, technical analysis incorporates the study of price and traded amounts of securities with the intention of determining future prices. However, with the emergence of big data and improved methods of machine learning, there are discussions of potential usage of non-traditional data such as the sentiment analysis of financial news and social media for predicting the stock prices.[2] Not only does such a shift improve the spatial and temporal resolutions of the model's predictions, but it also enables the use of data that are obtained in real-time and thus can reflect newly emerging news events on social media. Our project aims to enhance the performance of stock predictions by creating a multifaceted recommendation system where we use sentiment analysis of financial news and technical indicators together to recommend short-term decisions while leveraging the power of Long Short-Term Memory (LSTM) networks for long-term predictions. This comprehensive system (Fig.1.1) will tackle both short- and long-term stock matters and will be an essential contribution to the investors' decision-making process. In such a way, we try to obtain a better workable scenario of market conditions which encompasses both the quantitative aspects and the qualitative aspects driving stock prices. As a result, including both the experience of assigning stocks based on historical indicators and the use of modern methodologies for machine learning and artificial intelligence, the presented research can be regarded as a progressive step in the further advancement of the tools for stock market prediction.

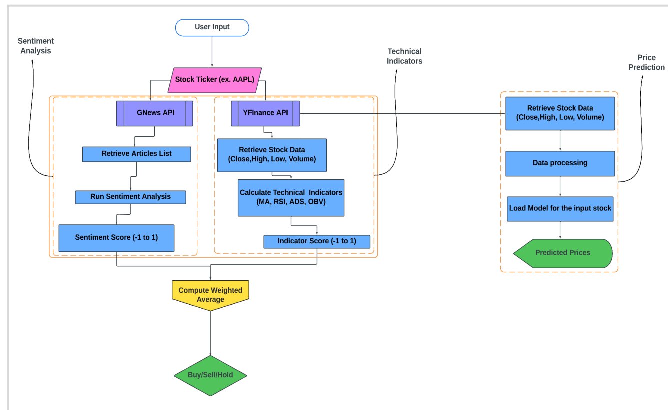

Fig.1.1 Project Framework

2. Literature Review

This literature review covers key studies that have contributed to the development of sentiment analysis, technical indicators, and machine learning models in our project. Tetlock (2007) conducted a seminal study on the role of media in financial markets, demonstrating that negative news sentiment can predict declines in stock prices. The study utilized content analysis to measure the sentiment of news articles and correlated these measures with market movements. Sharma et al. (2019) combined sentiment analysis with time series analysis for stock price prediction. They used the FinancialPhraseBank dataset to train their sentiment analysis model and applied it to news articles to predict stock price movements. Bollen, Mao, and Zeng (2011) explored the relationship between Twitter mood and stock prices. They discovered that it is possible to predict the DJIA using the outcome of public mood states that are extracted from the analysis of posts on the Twitter group. Our project is based on these foundations, and sentiment analysis along with the technical indicators are used combined with LSTM to have more efficient predictions on stock prices.

3. Data

3.1. YFinance API

The YFinance API was used to collect historical stock data from the previous 5 years. The data includes the Open, Close, Adjusted Close, Low, High, and Volume for each day for a particular stock. Visualizing closing price trends and moving averages helped identify underlying patterns. The closing price trend over the five years shows the overall movement and volatility of the stock (Fig.3.1.1) while the moving average helps smooth out price data to identify trends over a specified period. (Fig.3.1.2)

Fig.3.1.1 Five-Year Closing Price Trend of ZS

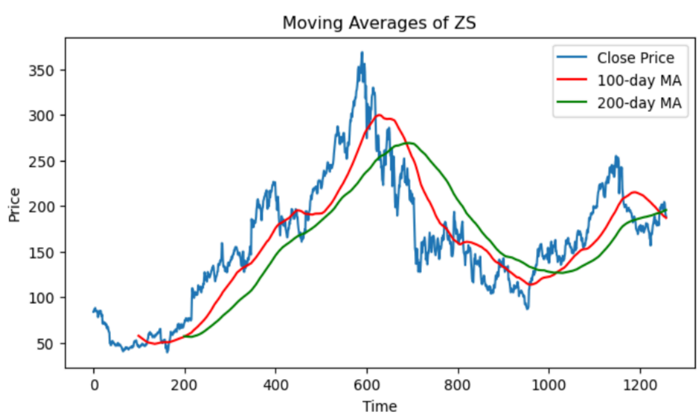

Fig.3.1.2 Moving Averages of ZS (100-day and 200-day)

3.2. GNews API

The GNews API gathers news articles from prominent online news outlets like Yahoo News and Google News. This API takes the company name as the input (Apple Inc. for example) and then outputs all the relevant news articles from the past month sorted by popularity. This helps understand the financial environment, related to the user input stock. [7]

3.3. Financial Phrase Bank (Hugging Face)

The Financial Phrase bank was used for the news sentiment detection model. This dataset consists of 4,840 sentences that have been collected from English financial news and each of them has been classified by the sentiment. The attitudes were divided into positive, negative, and neutral. The classification was done based on the level of certainty of 5-8 individuals There is high credibility with such classification. Thus, using this dataset, it is possible to analyze the sentiment of the financial section to investigate the market sentiment and the influence of news on stocks. [8]

4. Methodology

4.1. Long-Term Price Prediction Model

The price prediction model employs a Long Short-Term Memory (LSTM) that is specifically appropriate for time series data since it has great capabilities in discovering long-term dependencies and sequential patterns in data. The model was designed with the following architecture:

Layers: Four LSTM layers all of which are followed by a Dropout layer to avoid over-fitting. A dense layer with one neuron provides the predicted closing stock price value.

Parameter Settings: Optimizer: Adam, Loss Function: Mean Squared Error (MSE), Epochs: 50, Batch Size: 32

The dataset was split into training (70%) and testing (30%). It was that the model was not overfitting by closely monitoring the training and validation loss. The trained model was used to predict stock prices on the test set. The Evaluation Metrics were Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R-squared. The predicted vs. actual prices were plotted together to visually assess the model's performance.

4.2. Short Term Recommendation Model

The short-term model aims to provide users with a definitive recommendation based on the Technical Indicators and the News Sentiment model recommendations. The final output is one of ‘Buy’, ‘Sell’ or ‘Hold’ and is defined to be valid for one month.

4.2.1. News Sentiment Model

The News Sentiment model is trained by the best-performing model among Multinomial Naive Bayes Classifier (MNB), Support Vector Machine (SVM), and Random Forest (RF). The ‘Financial Phrase Bank’ dataset is used to train and test these models. The dataset is split into train(80%), validation(10%), and test(10%). The hyperparameters are fine-tuned to maximize the validation accuracy. The hyperparameters for MNB were the alpha value, for SVM was the ‘C’ value for

SVM, and depth 'n' for RF. While scoring, all the related articles are collected and then scored. Their average is then passed on to the Integrated model along with the Technical Indicator model.

4.2.2. Technical Indicators Model

We employed four essential technical indicators that give us an all-around view of how the stock has performed and consequently assist in making trading decisions. Detailed interpretation of each key technical indicator is given below:

The 50-day Moving Average (MA) is used to smoothen out short-term fluctuations in price, help identify the overall direction of a trend by averaging the closing prices of a stock over the past fifty days.

- ➔ *Buy Signal:* If the current closing price is above the 50-day MA, it indicates an upward trend, suggesting a buy.
- ➔ *Sell Signal:* If the current closing price is below the 50-day MA, it indicates a downward trend, suggesting a sell.

Relative Strength Index (RSI) between 0 and 100 measures how fast prices move, indicating potential overbought or oversold conditions when it crosses certain thresholds.

- ➔ *Buy Signal:* An RSI value below 30 suggests the stock is oversold and may be undervalued, indicating a buy opportunity.
- ➔ *Sell Signal:* An RSI value above 70 indicates the stock is overbought and may be overvalued, suggesting it might be time to sell.

Average Directional Index (ADX) looks at the strength of the trend (whatever it's direction may be)

- ➔ *Buy Signal:* A trend is considered to be strong and upward if ADX is above 25, combined with a positive directional index (PosDI) greater than the negative directional index (NegDI).
- ➔ *Sell Signal:* A trend is considered to be strong and downward if ADX above 25, with the negative directional index (NegDI) greater than the positive directional index (PosDI).

On-Balance Volume (OBV) is the flow of volume used to gauge buying and selling pressure that will indicate if there is sufficient volume behind a price movement.

- ➔ *Buy Signal:* An increasing OBV suggests buying pressure, confirming the validity of a buy signal.
- ➔ *Sell Signal:* A decreasing OBV suggests selling pressure, confirming the validity of a sell signal.[6]

4.2.3. Integration

A weighted average of the Sentiment Analysis Model score and the Technical Indicators Model is used to calculate the final recommendation score. We have assigned a weight of 0.40 to the sentiment score and 0.60 to the technical score, reflecting a slightly higher emphasis on technical indicators. The final score is used to generate actionable recommendations:

- ➔ Buy: If the final score > 0.33
- ➔ Sell: If the final score < -0.33
- ➔ Hold: If the final score is between -0.33 and 0.33

These two models are integrated to obtain a balance between the market view and the technical position on a particular stock thus making the recommendations more accurate and dependable.

5. Results

5.1. Long-Term Price Prediction Model

The Evaluation Metrics used to assess the accuracy of the model were MSE, RMSE, MAE, and R-squared. Plots of Predicted vs. actual prices for the test set showed the model's effectiveness in capturing trends.

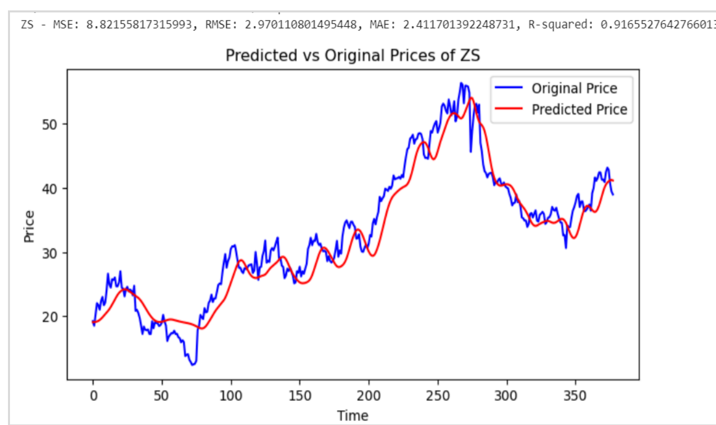

Fig.5.1.1 Actual vs Predicted Plot for ZS

5.2. Short Term Recommendation Model

5.2.1. Technical Indicators Model (TIM)

The accuracy of recommendations (Buy, Sell, Hold) was evaluated by comparing them to actual price movements over the next month. Fig.5.2.1.1. and Fig.5.2.1.2. illustrate the confusion matrix of the recommendations provided by the Technical Indicator Model (TIM).

```
correct_predictions:  385
total_predictions:  574
Overall Accuracy: 67.07%
```

Fig.5.2.1.2. Technical indicators model accuracy

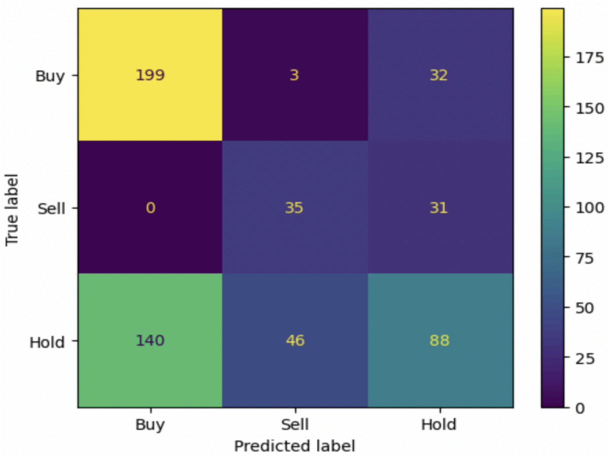

Fig.5.2.1.1. Technical indicators model confusion matrix

While the model effectively avoided opposite signals (e.g., predicting 'Buy' for 'Sell' or vice versa), there is room for improvement in its performance. The overall accuracy of the TIM was 67.07%, indicating that further refinement is necessary.

5.2.2. News Sentiment Model (NSM)

The Support Vector Machine (SVM) model performed the best out of all the models trained for the News Sentiment Model (NSM), obtaining an F1 Score of 0.79 and an accuracy of 79.75%. With an accuracy of 68.80%, the Random Forest (RF) model performed the worst in contrast. Based on these results, the SVM model was selected for news prediction.

Results/Model	MNB	RF	SVM
Test Accuracy	76.87	68.80	79.75
F1 Score	0.76	0.61	0.79

5.2.3. Final Weighted Average Model

Combining the two models, TIM and NSM, through the weighted average method and using a grid search method, the maximum test accuracy of 71.95% was achieved, which is 5 percentage points higher than the TIM alone. This optimal accuracy was obtained with weights W1 (NSM) = 0.40 and W2 (TIM) = 0.60 (Fig.5.2.3.1.) Analyzing the new confusion matrix, improvements can be observed in the predictions. The 'Buy' prediction accuracy increased slightly, and the 'Sell' prediction accuracy improved with 10 additional correct 'Sell' predictions. The most significant improvement was observed in the 'Hold' signal, with 163 stocks correctly predicted as 'Hold.' Overall, the false negatives for the 'Hold' signal were reduced (Fig.5.2.3.2.)

```
correct_predictions: 413
total_predictions: 574
Overall Accuracy: 71.95%
```

Fig.5.2.3.1. Combined short-term model accuracy

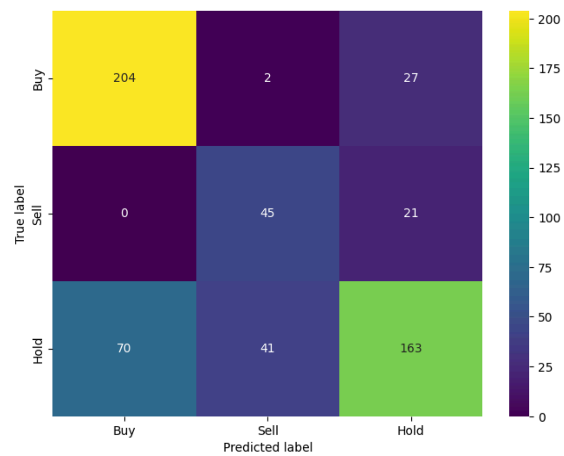

Fig.5.2.3.2. Combined short-term model confusion matrix

6. Conclusions

The extensive setup established with the help of the LSTM model for long-term price forecasting and integration of sentiment analysis along with the technical indicators for short-term recommendation highly improves the decision-making process. The project demonstrates the effectiveness of using advanced machine-learning techniques for financial forecasting. LSTMs have the ability to handle repeated long-term dependencies in numerical sequences, which makes the given model suitable for long-term stock price prediction. While sentiment analysis and technical indicators are a way of analyzing the social impact on the stock market. Another avenue for further research could be the implementation of more sophisticated architectures such as GRU, as well as attention-based models, combining predictions with the help of ensemble methods, the use of more extensive, technical indicators and fundamental analysis metrics. [5] Adding such a filter as 'Budget' could add more objectivity to recommendations.

References

[1] Kaggle. (n.d.). Stock Price Prediction Using yfinance. https://www.kaggle.com/code/sirilpc/stock-price-prediction-using-yfinance/notebook

[2] Sharma V., Khemnar R., Kumari R., & Mohan B. R. (2019). Time Series with Sentiment Analysis for Stock Price Prediction. 2019 2nd International Conference on Intelligent Communication and Computational Techniques (ICCT). doi:10.1109/icct46177.2019.896906

[3] TrendSpider. (n.d.). The Ultimate Guide to Technical Indicators from https://trendspider.com/blog/the-ultimate-guide-to-technical-indicators/

[4] Strike Money. (n.d.). Technical Analysis. https://www.strike.money/technical-analysis

[5] Zhang, G. P. (2003). Time series forecasting using a hybrid ARIMA and neural network model. Neurocomputing, 50, 159-175.

[6] Technical indicators: The tools of the trade accessed June 21 2024, from https://www.britannica.com/money/technical-indicator-types

[7] GNews. (n.d.). GNews API Documentation. Retrieved July 30, 2024, from https://gnews.io/docs/v4#introduction

[8] Takala, P. (n.d.). Financial PhraseBank. Hugging Face. Retrieved July 30, 2024, from https://huggingface.co/datasets/takala/financial_phrasebank

Appendix

7.1 LSTM Price Prediction Model Code:

7.1.1 Train_and_save.py - This Python file trains and saves the model for each stock

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
import os
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from keras.layers import Dense, Dropout, LSTM
from keras.models import Sequential


# Path to desktop
desktop_path = os.path.expanduser("~/Desktop")

def predict_and_plot_stock_prices(tickers, prediction_months=12):
    for stock_symbol in tickers:
        print(f"Processing {stock_symbol}...")

        # Download data
        df = yf.download(tickers=stock_symbol, period='5y', interval='1d')
        if df.empty:
            print(f"No data found for {stock_symbol}")
            continue

        # Preprocess data
        df = df.reset_index()
        df = df.drop(['Date', 'Adj Close'], axis=1)

        # Plot the stock closing price
        plt.figure(figsize=(8, 4))
```

```python
# Plot the stock closing price
plt.figure(figsize=(8, 4))
plt.plot(df.Close, label='Close Price')
plt.title(f'Closing Price of {stock_symbol}')
plt.xlabel('Time')
plt.ylabel('Close Price')
plt.legend()
plt.show()

# Calculate and plot moving averages
ma100 = df.Close.rolling(100).mean()
ma200 = df.Close.rolling(200).mean()

plt.figure(figsize=(8, 4))
plt.plot(df.Close, label='Close Price')
plt.plot(ma100, 'r', label='100-day MA')
plt.plot(ma200, 'g', label='200-day MA')
plt.title(f'Moving Averages of {stock_symbol}')
plt.xlabel('Time')
plt.ylabel('Price')
plt.legend()
plt.show()

# Split data into training and testing
data_training = pd.DataFrame(df['Close'][0:int(len(df) * 0.70)])
data_testing = pd.DataFrame(df['Close'][int(len(df) * 0.70):int(len(df))])
```

```python
# Normalize training data
scaler = MinMaxScaler(feature_range=(0, 1))
data_training_array = scaler.fit_transform(data_training)

# Prepare training data for LSTM
x_train = []
y_train = []
for i in range(100, data_training_array.shape[0]):
    x_train.append(data_training_array[i-100:i])
    y_train.append(data_training_array[i, 0])
x_train, y_train = np.array(x_train), np.array(y_train)

# Define the LSTM model
model = Sequential()
model.add(LSTM(units=50, activation='relu', return_sequences=True, input_shape=(x_train.shape[1], 1)))
model.add(Dropout(0.2))
model.add(LSTM(units=60, activation='relu', return_sequences=True))
model.add(Dropout(0.3))
model.add(LSTM(units=80, activation='relu', return_sequences=True))
model.add(Dropout(0.4))
model.add(LSTM(units=120, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(units=1))

model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(x_train, y_train, epochs=50, verbose=1)
```

```python
# Save the model for each ticker
#model.save(f'{stock_symbol}_keras_model.h5')
model.save(os.path.join(desktop_path, f'{stock_symbol}_keras_model.h5'))

# Concatenate the last 100 days of training data with testing data
past_100_days = data_training.tail(100)
final_df = pd.concat([past_100_days, data_testing], ignore_index=True)

# Normalize the combined data
input_data = scaler.transform(final_df)

# Prepare testing data
x_test = []
y_test = []
for i in range(100, input_data.shape[0]):
    x_test.append(input_data[i-100:i])
    y_test.append(input_data[i, 0])
x_test, y_test = np.array(x_test), np.array(y_test)

# Predict stock prices
y_predicted = model.predict(x_test)
scale_factor = 1 / 0.01162115
y_predicted = y_predicted * scale_factor
y_test = y_test * scale_factor
```

```python
        # Calculate and print accuracy metrics
        mse = mean_squared_error(y_test, y_predicted)
        rmse = np.sqrt(mse)
        mae = mean_absolute_error(y_test, y_predicted)
        r2 = r2_score(y_test, y_predicted)

        print(f"{stock_symbol} - MSE: {mse}, RMSE: {rmse}, MAE: {mae}, R-squared: {r2}")

        # Plot the predictions
        plt.figure(figsize=(8, 4))
        plt.plot(y_test, 'b', label='Original Price')
        plt.plot(y_predicted, 'r', label='Predicted Price')
        plt.title(f'Predicted vs Original Prices of {stock_symbol}')
        plt.xlabel('Time')
        plt.ylabel('Price')
        plt.legend()
        plt.show()

if __name__ == '__main__':
    tickers = ['DAY', 'SNOW', 'AAPL', 'GTLB', 'ZS', 'NINOY', 'DBX', 'WIX', 'LOGI', 'IQV', 'TEAM', 'WDAY', 'SPOT', 'OTEX', 'TDC', 'UDMY'] # List of stock
    prediction_months = 12  # prediction period
    predict_and_plot_stock_prices(tickers, prediction_months)
```

7.1.2 Load_and_predict.py - This Python file loads the model and predicts the output (future prices)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
from sklearn.preprocessing import MinMaxScaler
from keras.models import load_model
from datetime import datetime, timedelta
import os

# Path to desktop
desktop_path = os.path.expanduser("~/Desktop")

def predict_future_prices(ticker, prediction_months=12):
    print(f"Processing {ticker}...")

    # Load the model
    model_path = os.path.join(desktop_path, f'{ticker}_keras_model.h5')
    if not os.path.exists(model_path):
        print(f"No saved model found for {ticker}.")
        return

    model = load_model(model_path)

    # Download data for prediction
    df = yf.download(tickers=ticker, period='5y', interval='1d')
    if df.empty:
        print(f"No data found for {ticker}")
        return
```

```
# Reset index and drop unnecessary columns
df = df.reset_index()
df = df.drop(['Adj Close'], axis=1)
df = df.set_index('Date')

# Prepare data for prediction
data_training = pd.DataFrame(df['Close'][0:int(len(df) * 0.70)])
data_testing = pd.DataFrame(df['Close'][int(len(df) * 0.70):int(len(df))])

scaler = MinMaxScaler(feature_range=(0, 1))
data_training_array = scaler.fit_transform(data_training)

past_100_days = data_training.tail(100)
final_df = pd.concat([past_100_days, data_testing], ignore_index=True)

input_data = scaler.transform(final_df)

x_test = []
for i in range(100, input_data.shape[0]):
    x_test.append(input_data[i-100:i])
x_test = np.array(x_test)

# Predict future prices
y_predicted = model.predict(x_test)
scale_factor = 1 / 0.01162115
y_predicted = y_predicted * scale_factor
```

```
    # Get the last date in the DataFrame
    last_date = df.index[-1]

    # Calculate prediction dates
    prediction_days = prediction_months * 30  # Approximate trading days
    if prediction_days > len(y_predicted):
        print(f"Warning: Requested prediction period is longer than available data. Limiting to available predictions.")
        prediction_days = len(y_predicted)

    last_n_predictions = y_predicted[-prediction_days:]

    prediction_start_date = last_date + timedelta(days=1)
    prediction_end_date = prediction_start_date + timedelta(days=prediction_days - 1)

    print(f"Prediction period: {prediction_start_date.date()} to {prediction_end_date.date()}")

    # Plot the predictions
    plt.figure(figsize=(8, 4))
    plt.plot(range(prediction_days), last_n_predictions, 'r', label='Predicted Price')
    plt.title(f'Predicted Prices of {ticker} from {prediction_start_date.date()} to {prediction_end_date.date()}')
    plt.xlabel('Time (Days)')
    plt.ylabel('Price')
    plt.show()

# Example usage
if __name__ == '__main__':
    user_ticker = input("Enter the stock ticker: ")
    prediction_months = 12  # Fixed prediction period
    predict_future_prices(user_ticker, prediction_months)
```

7.2 Short-Term Recommendation Model Code:

7.2.1 model_training.py: Trains the 3 Models (SVM, MNB, and RF) for News Sentiment prediction

```
import os
import random
import pandas as pd
import pickle

from sklearn.feature_extraction.text import CountVectorizer,TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

import numpy as np
from gensim.models import Word2Vec
import torch
from torch.utils.data import DataLoader, Dataset
from transformers import BertTokenizer, BertForSequenceClassification, AdamW
from transformers import get_linear_schedule_with_warmup
```

```python
def splitter(text, delimiters):
    result = [text]

    for delimiter in delimiters:
        temp = []
        for substring in result:
            temp.extend(substring.split(delimiter))
        result = temp
    return result

def split_data(data, train_split=0.8, test_split=0.1, val_split=0.1):
    random.shuffle(data)
    total = len(data)
    train_index = int(total * train_split)
    test_index = train_index + int(total * test_split)

    train_data = data[:train_index]
    test_data = data[train_index:test_index]
    val_data = data[test_index:]

    return train_data, test_data, val_data

def process_file(path, delimiters, stopwords):
    data = []
    with open(path, 'r', encoding='ISO-8859-1') as data_file:
        lines = data_file.readlines()
        for line in lines:
            label, sentence = line.split(',', 1)
            sentence = sentence.strip()
            split_sentence = splitter(sentence, delimiters)
            # filtered_sentence = [word for word in split_sentence if word.lower() not in stopwords]
            data.append((split_sentence, label.strip()))

    return data
```

```python
# CONSTANTS
DELIMITERS = list("!#$%&()*+/:,;<=>@[\\]^`{|}~\t.\"- ")
STOP_WORDS = ["", "'s", "a", "an", "and", "are", "as", "at", "be", "but", "by", "for",
                "if", "in", "into", "is", "it", "no", "not", "of", "on", "or",
                "such", "that", "the", "their", "then", "there", "these", "they",
                "this", "to", "was", "will", "with"]

# import the file and data
path = r'datasets\finance_news_sentiment.csv'

data = process_file(path, DELIMITERS, STOP_WORDS)

# Split data into train, test, and validation sets
train, test, val = split_data(data)


# Example of how to access the data
print("Train data:", train[:1])
print("Test data:", test[:1])
print("Validation data:", val[:1])
```

```python
def load_data(dataset):
    rows = []
    for row in dataset:
        rows.append(row)
    data = pd.DataFrame(rows)
    data.dropna(inplace=True)  # Drop rows with NaN values
    return data[0].apply(lambda x: ' '.join(x)), data[1]

def compute_continuous_output(probs, mapping):
    return np.dot(probs, mapping)

def train_mnb(x_train, y_train, x_val, y_val, vectorizer, modelname):
    vectorizer.fit(x_train)
    x_train_vec = vectorizer.transform(x_train)
    x_val_vec = vectorizer.transform(x_val)

    best_alpha = None
    best_accuracy = 0
    best_model = None

    # Hyperparameter tuning: trying different alpha values
```

```python
def train_svm(x_train, y_train, x_val, y_val, vectorizer, modelname):
    vectorizer.fit(x_train)
    x_train_vec = vectorizer.transform(x_train)
    x_val_vec = vectorizer.transform(x_val)

    svm_model = SVC(probability=True)  # Enable probability estimates
    param_grid_svm = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf']}
    svm_grid = GridSearchCV(svm_model, param_grid_svm, cv=5)
    svm_grid.fit(x_train_vec, y_train)
    best_svm = svm_grid.best_estimator_

    y_val_pred_svm = best_svm.predict(x_val_vec)
    accuracy = accuracy_score(y_val, y_val_pred_svm)
    print(f"{modelname} Validation Accuracy: {accuracy:.4f}")
    print(classification_report(y_val, y_val_pred_svm))

    if not os.path.exists("data"):
        os.makedirs("data")

    with open(f"models/{modelname}.pkl", 'wb') as f:
        pickle.dump((vectorizer, best_svm), f)  # Save the best model

def train_rf(x_train, y_train, x_val, y_val, vectorizer, modelname):
    vectorizer.fit(x_train)
    x_train_vec = vectorizer.transform(x_train)
    x_val_vec = vectorizer.transform(x_val)

    rf_model = RandomForestClassifier()
    param_grid_rf = {'n_estimators': [100, 200], 'max_depth': [10, 20]}
    rf_grid = GridSearchCV(rf_model, param_grid_rf, cv=5)
    rf_grid.fit(x_train_vec, y_train)
    best_rf = rf_grid.best_estimator_

    y_val_pred_rf = best_rf.predict(x_val_vec)
    accuracy = accuracy_score(y_val, y_val_pred_rf)
    print(f"{modelname} Validation Accuracy: {accuracy:.4f}")
    print(classification_report(y_val, y_val_pred_rf))

    if not os.path.exists("data"):
        os.makedirs("data")
```

```python
    with open(f"models/{modelname}.pkl", 'wb') as f:
        pickle.dump((vectorizer, best_rf), f)  # Save the best model

def evaluate_model(x_test, y_test, modelname, mapping):
    with open(f"models/{modelname}.pkl", 'rb') as f:
        vectorizer, model = pickle.load(f)

    x_test_vec = vectorizer.transform(x_test)
    y_test_pred = model.predict(x_test_vec)
    accuracy = accuracy_score(y_test, y_test_pred)
    print(f"{modelname} Test Accuracy: {accuracy:.4f}")
    print(classification_report(y_test, y_test_pred))

    # Get probabilities
    y_test_probs = model.predict_proba(x_test_vec)
    y_test_continuous = compute_continuous_output(y_test_probs, mapping)
    return y_test_continuous

# Load the data
# Assuming train, val, and test are already loaded datasets in the required format
x_train, y_train = load_data(train)
x_val, y_val = load_data(val)
x_test, y_test = load_data(test)

# Define vectorizers
vec = CountVectorizer(ngram_range=(1, 2))

# Define mapping from class probabilities to continuous output
class_mapping = [-1, 0, 1]

# Train the models
print("Training MNB...")
train_mnb(x_train, y_train, x_val, y_val, vec, "mnb_uni_bi")
print("Training SVM...")
train_svm(x_train, y_train, x_val, y_val, vec, "svm_uni_bi")
print("Training Random Forest...")
train_rf(x_train, y_train, x_val, y_val, vec, "rf_uni_bi")
```

```python
# Evaluate on the test set
print("\nEvaluating MNB...")
y_test_continuous_mnb = evaluate_model(x_test, y_test, "mnb_uni_bi", class_mapping)
print("\nEvaluating SVM...")
y_test_continuous_svm = evaluate_model(x_test, y_test, "svm_uni_bi", class_mapping)
print("\nEvaluating Random Forest...")
y_test_continuous_rf = evaluate_model(x_test, y_test, "rf_uni_bi", class_mapping)
```

```python
VECTOR_SIZE = 100

combined = []
for dataset in [train, test, val]:
    for sentence, label in dataset:
        combined.append(sentence)

# Train Word2Vec model
print("Training Word2Vec model...")
model = Word2Vec(combined, vector_size=VECTOR_SIZE, window=5, min_count=1, workers=4)

# Save the model
model_path = r'models/w2v.model'
model.save(model_path)
print(f"Model saved to {model_path}")
```

```python
# Parameters
EPOCHS = 3
BATCH_SIZE = 16
LEARNING_RATE = 2e-3

# Dataset class
class CustomDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_len):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = self.texts[idx]
        label = self.labels[idx]
        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            max_length=self.max_len,
            return_token_type_ids=False,
            padding='max_length',
            truncation=True,
            return_attention_mask=True,
            return_tensors='pt',
        )
        return {
            'text': text,
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'labels': torch.tensor(label, dtype=torch.long)  # Ensure label is an integer
        }
```

```python
# Initialize tokenizer and model
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=3)

# Define label mapping
label_mapping = {'negative': 0, 'neutral': 1, 'positive': 2}

# Convert labels to integers if they are not already
def convert_labels(labels, label_mapping):
    return [label_mapping[label] if isinstance(label, str) else label for label in labels]

y_train = convert_labels(y_train, label_mapping)
y_val = convert_labels(y_val, label_mapping)
y_test = convert_labels(y_test, label_mapping)


# Create datasets
train_dataset = CustomDataset(x_train, y_train, tokenizer, max_len=128)
val_dataset = CustomDataset(x_val, y_val, tokenizer, max_len=128)

# Create dataloaders
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)

# Set up optimizer and scheduler
optimizer = AdamW(model.parameters(), lr=LEARNING_RATE, correct_bias=False)
total_steps = len(train_loader) * EPOCHS
scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps=0,
    num_training_steps=total_steps
)
```

```python
# Training function
def train_epoch(model, data_loader, optimizer, scheduler, device):
    model = model.train()
    total_loss = 0

    for batch in data_loader:
        optimizer.zero_grad()
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        outputs = model(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
        loss = outputs.loss
        total_loss += loss.item()

        loss.backward()
        optimizer.step()
        scheduler.step()

    return total_loss / len(data_loader)

# Evaluation function
def eval_model(model, data_loader, device):
    model = model.eval()
    preds = []
    labels = []

    with torch.no_grad():
        for batch in data_loader:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels.extend(batch['labels'].tolist())

            outputs = model(input_ids=input_ids, attention_mask=attention_mask)
            _, pred = torch.max(outputs.logits, dim=1)
            preds.extend(pred.tolist())

    return accuracy_score(labels, preds), classification_report(labels, preds)
```

```python
# Training loop
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)

for epoch in range(EPOCHS):
    print(f'Epoch {epoch + 1}/{EPOCHS}')
    train_loss = train_epoch(model, train_loader, optimizer, scheduler, device)
    print(f'Train loss: {train_loss}')
    val_accuracy, val_report = eval_model(model, val_loader, device)
    print(f'Validation accuracy: {val_accuracy}')
    print(val_report)
```

7.2.2 sentiment_classifier.py: Provides News sentiment score based on the model trained in model_training.py

```python
import json
import pickle
import numpy as np
import re

# Constants for preprocessing
DELIMITERS = list("!#$%&()*+/:,;<=>@[\\]^`{|}~\t.\"- ")
STOP_WORDS = ["", "'s", "a", "an", "and", "are", "as", "at", "be", "but", "by", "for",
              "if", "in", "into", "is", "it", "no", "not", "of", "on", "or",
              "such", "that", "the", "their", "then", "there", "these", "they",
              "this", "to", "was", "will", "with"]

Codeium: Refactor | Explain | Generate Docstring | X
def load_model(model_path):
    with open(model_path, 'rb') as file:
        vectorizer, model = pickle.load(file)
    return vectorizer, model

Codeium: Refactor | Explain | Generate Docstring | X
def load_articles(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        data = json.load(file)
    contents = [article['content'] for article in data if 'content' in article]
    return contents

Codeium: Refactor | Explain | Generate Docstring | X
def splitter(text, delimiters):
    regex_pattern = '|'.join(map(re.escape, delimiters))
    return re.split(regex_pattern, text)

Codeium: Refactor | Explain | Generate Docstring | X
def preprocess_articles(articles):
    data = []
    for sentence in articles:
        split_sentence = splitter(sentence, DELIMITERS)
        filtered_sentence = [word for word in split_sentence if word.lower() not in STOP_WORDS]
        data.append(' '.join(filtered_sentence))
    return data

Codeium: Refactor | Explain | Generate Docstring | X
def vectorize_articles(vectorizer, articles):
    return vectorizer.transform(articles)
```

```
def classify_sentiment(model, articles_tfidf):
    probabilities = model.predict_proba(articles_tfidf)
    return probabilities

Codeium: Refactor | Explain | Generate Docstring | X
def compute_continuous_score(probabilities, mapping):
    return np.dot(probabilities, mapping)

Codeium: Refactor | Explain | Generate Docstring | X
def interpret_score(score):
    if score <= -0.33:
        return "sell"
    elif score <= 0.33:
        return "hold"
    else:
        return "buy"

Codeium: Refactor | Explain | Generate Docstring | X
def get_news_sentiment(keyword):
    input_file = f"news_sentiment/data/news_{keyword}.json"
    model_file = "news_sentiment/models/svm_uni_bi.pkl"

    vectorizer, model = load_model(model_file)
    articles = load_articles(input_file)

    if len(articles) < 10:
        return 0, "NA", 0, 0, 0

    preprocessed_articles = preprocess_articles(articles)
    articles_tfidf = vectorize_articles(vectorizer, preprocessed_articles)
    probabilities = classify_sentiment(model, articles_tfidf)

    class_mapping = np.array([-1, 0, 1])
    continuous_scores = np.apply_along_axis(lambda prob: compute_continuous_score(prob, class_mapping), 1, probabilities)
    avg_score = np.mean(continuous_scores)

    hold_count = sum(1 for score in continuous_scores if score <= 0.33 and score > -0.33)
    buy_count = sum(1 for score in continuous_scores if score > 0.33)
    sell_count = sum(1 for score in continuous_scores if score <= -0.33)

    recommendation = interpret_score(avg_score)

    return avg_score, recommendation, hold_count, buy_count, sell_count
```

7.2.3 article_generator.py: Compiles all the news articles relevant to the stock input by the user

```python
import json
import os
import urllib.request
from datetime import datetime, timedelta
import time

# Load API keys from the json file
Codeium: Refactor | Explain | Generate Docstring | X
def load_json(file_path, key):
    with open(file_path, 'r', encoding='utf-8') as file:
        data = json.load(file)
        return data[key]

# Function to fetch and append data
Codeium: Refactor | Explain | Generate Docstring | X
def fetch_and_append_data(keyword, start_date, end_date, file_path, apiKey):
    url = f"https://gnews.io/api/v4/search?q={keyword}&lang=en&country=us&from={start_date}T00:00:00Z&to={end_date}T23:59:59Z&sortby=relevance&max=10&apikey={apiKey}"

    try:
        with urllib.request.urlopen(url) as response:
            data = json.loads(response.read().decode("utf-8"))

        new_articles = data.get('articles', [])

        if os.path.exists(file_path):
            with open(file_path, 'r', encoding='utf-8') as file:
                existing_data = json.load(file)
        else:
            existing_data = []

        existing_urls = {article['url'] for article in existing_data}
        filtered_articles = [article for article in new_articles if article['url'] not in existing_urls]

        existing_data.extend(filtered_articles)

        with open(file_path, 'w', encoding='utf-8') as file:
            json.dump(existing_data, file, indent=2)

        return len(filtered_articles)
    except Exception as e:
        print(f"Failed to fetch data using API key: {apiKey}. Error: {e}")
        return -1
```

```python
# Function to remove Unicode characters and newlines
Codeium: Refactor | Explain | Generate Docstring | X
def remove_unicode_and_newlines_from_file(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        data = json.load(file)

    content = json.dumps(data, ensure_ascii=False, indent=2)
    cleaned_content = content.replace('\\n', ' ')

    with open(file_path, 'w', encoding='utf-8') as file:
        file.write(cleaned_content)

    print(f"Unicode characters and new lines replaced, data written to {file_path}")

# Function to partition date range and fetch news
Codeium: Refactor | Explain | Generate Docstring | X
def partition_and_fetch(keyword, start_date, end_date, step_days, api_keys):
    start_date = datetime.strptime(start_date, "%Y-%m-%d")
    end_date = datetime.strptime(end_date, "%Y-%m-%d")
    file_path = f'news_sentiment/data/news_{keyword}.json'
    os.makedirs('data', exist_ok=True)

    total_articles = 0
    current_start = start_date
    api_key_index = 0

    while current_start < end_date:
        current_end = current_start + timedelta(days=step_days - 1)
        if current_end > end_date:
            current_end = end_date

        while api_key_index < len(api_keys):
            articles_fetched = fetch_and_append_data(keyword, current_start.strftime("%Y-%m-%d"), current_end.strftime("%Y-%m-%d"), file_path, api_keys[api_key_index])
            if articles_fetched != -1:
                total_articles += articles_fetched
                print(f"Data updated from {current_start.strftime('%Y-%m-%d')} to {current_end.strftime('%Y-%m-%d')} for keyword '{keyword}'")
                break
            else:
                api_key_index += 1
```

```python
        if api_key_index == len(api_keys):
            print("All API keys have reached their limits.")
            break

        time.sleep(1)
        current_start = current_end + timedelta(days=1)

    print(f"Total number of articles collected for '{keyword}': {total_articles}")
    remove_unicode_and_newlines_from_file(file_path)

# If you want to run this file independently for testing
if __name__ == "__main__":
    apiKeys = load_json(r"news_sentiment\api_keys.json", "api_keys")
    keyword = "Trivago"  # Specify your single keyword here for standalone runs
    start_date = "2024-06-21"
    end_date = "2024-07-21"
    step_days = 2
    partition_and_fetch(keyword, start_date, end_date, step_days, apiKeys)
```

7.2.4 technical_indicators.py: Computes recommendation score based on different technical indicators

```python
import yfinance as yf
import pandas as pd
import numpy as np
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Define the ticker symbols of the stocks you want to analyze
ticker_symbols = [
    "AAPL", "MSFT", "GOOGL", "AMZN", "META", "TSLA", "NVDA", "NFLX", "ADBE", "PYPL"]

# Function to calculate technical indicators
Codeium: Refactor | Explain | Generate Docstring | X
def calculate_indicators(data):
    # Calculate 50-day Moving Average
    data['MA'] = data['Close'].rolling(window=50).mean()

    # Calculate RSI
    delta = data['Close'].diff()  # Calculate the difference in closing prices
    gain = delta.where(delta > 0, 0)  # Positive gains
    loss = -delta.where(delta < 0, 0)  # Negative losses
    avg_gain = gain.rolling(window=14).mean()  # Average gain over 14 days
    avg_loss = loss.rolling(window=14).mean()  # Average loss over 14 days
    rs = avg_gain / avg_loss  # Relative strength
    rsi = 100 - (100 / (1 + rs))  # Relative Strength Index calculation
    data['RSI'] = rsi

    # Calculate ADX
    data['UpMove'] = data['High'] - data['High'].shift(1)  # Upward price movement
    data['DownMove'] = data['Low'].shift(1) - data['Low']  # Downward price movement
    data['PosDM'] = data['UpMove'].where(data['UpMove'] > data['DownMove'], 0)  # Positive directional movement
    data['NegDM'] = data['DownMove'].where(data['DownMove'] > data['UpMove'], 0)  # Negative directional movement
    data['TR'] = np.maximum.reduce([data['High'] - data['Low'], abs(data['High'] - data['Close'].shift(1)), abs(data['Low'] - data['Close'].shift(1))])  # True Range
    data['ATR'] = data['TR'].rolling(window=14).mean()  # Average True Range
    data['PosDI'] = 100 * (data['PosDM'].rolling(window=14).mean() / data['ATR'])  # Positive Directional Index
    data['NegDI'] = 100 * (data['NegDM'].rolling(window=14).mean() / data['ATR'])  # Negative Directional Index
    data['DX'] = 100 * abs(data['PosDI'] - data['NegDI']) / (data['PosDI'] + data['NegDI'])  # Directional Movement Index
    data['ADX'] = data['DX'].rolling(window=14).mean()  # Average Directional Index
```

```python
    # Calculate OBV
    data['OBV'] = (data['Volume'] * (data['Close'] > data['Close'].shift(1)).astype(int) -
                   data['Volume'] * (data['Close'] < data['Close'].shift(1)).astype(int)).cumsum()  # On-Balance Volume

    return data

# Function to predict stock action based on current technical indicators
Codeium: Refactor | Explain | Generate Docstring | X
def predict_action(data):
    last_row = data.iloc[-1]

    ma_buy = int(last_row['Close'] > last_row['MA'])
    ma_sell = int(last_row['Close'] < last_row['MA'])
    rsi_buy = int(last_row['RSI'] < 30)
    rsi_sell = int(last_row['RSI'] > 70)
    adx_buy = int(last_row['ADX'] > 25 and last_row['PosDI'] > last_row['NegDI'])
    adx_sell = int(last_row['ADX'] > 25 and last_row['NegDI'] > last_row['PosDI'])
    obv_buy = int(last_row['OBV'] > data['OBV'].iloc[-2])
    obv_sell = int(last_row['OBV'] < data['OBV'].iloc[-2])

    return ma_buy, rsi_buy, adx_buy, obv_buy, ma_sell, rsi_sell, adx_sell, obv_sell

# Function to calculate the signal score
Codeium: Refactor | Explain | Generate Docstring | X
def calculate_signal(ma_buy, rsi_buy, adx_buy, obv_buy, ma_sell, rsi_sell, adx_sell, obv_sell):
    weights = {
        'ma_buy': 0.4,
        'rsi_buy': 0.2,
        'adx_buy': 0.25,
        'obv_buy': 0.15,
        'ma_sell': -0.4,
        'rsi_sell': -0.2,
        'adx_sell': -0.25,
        'obv_sell': -0.15
    }
```

```python
    score = (
        weights['ma_buy'] * ma_buy +
        weights['rsi_buy'] * rsi_buy +
        weights['adx_buy'] * adx_buy +
        weights['obv_buy'] * obv_buy +
        weights['ma_sell'] * ma_sell +
        weights['rsi_sell'] * rsi_sell +
        weights['adx_sell'] * adx_sell +
        weights['obv_sell'] * obv_sell
    )

    if score > 0.33:
        recommendation = 'Buy'
    elif score < -0.33:
        recommendation = 'Sell'
    else:
        recommendation = 'Hold'

    return score, recommendation

# Function to get technical indicator score
Codeium: Refactor | Explain | Generate Docstring | X
def get_technical_indicator_score(ticker):
    end_date = pd.to_datetime('today')
    start_date = end_date - pd.DateOffset(days=400)
    data = yf.download(ticker, start=start_date, end=end_date)

    if len(data) < 200:
        return 0, "NA"

    data = calculate_indicators(data)

    if len(data.dropna()) < 31:
        return 0, "NA"

    ma_buy, rsi_buy, adx_buy, obv_buy, ma_sell, rsi_sell, adx_sell, obv_sell = predict_action(data)
    score, recommendation = calculate_signal(ma_buy, rsi_buy, adx_buy, obv_buy, ma_sell, rsi_sell, adx_sell, obv_sell)

    return score, recommendation
```

```python
# Function to generate recommendations for the past month
Codeium: Refactor | Explain | Generate Docstring | X
def generate_past_month_recommendations(ticker):
    end_date = pd.to_datetime('today')
    start_date = end_date - pd.DateOffset(days=400)  # Get data for the past 400 days to ensure we have enough for the 200-day MA
    data = yf.download(ticker, start=start_date, end=end_date)

    # Log the length of the downloaded data
    #  print(f"Downloaded {len(data)} data points for {ticker}")

    # Ensure there is sufficient data
    if len(data) < 200:
        print(f"Insufficient data to calculate indicators for {ticker}.")
        return None, None, None

    data = calculate_indicators(data)

    # Ensure we have data for at least one month after the 200-day MA period
    if len(data.dropna()) < 31:
        print(f"Insufficient data after calculating indicators for {ticker}.")
        return None, None, None

    # Get the recommendation for the last date in the data
    recommendation_date = data.dropna().index[-31]  # 31 days back from today to ensure one month ago
    ma_buy, rsi_buy, adx_buy, obv_buy, ma_sell, rsi_sell, adx_sell, obv_sell = predict_action(data)
    score, recommendation = calculate_signal(ma_buy, rsi_buy, adx_buy, obv_buy, ma_sell, rsi_sell, adx_sell, obv_sell)

    return recommendation, recommendation_date, data
```

```python
# Function to evaluate recommendation against actual performance
Codeium: Refactor | Explain | Generate Docstring | X
def evaluate_recommendation(ticker, recommendation, recommendation_date, data):
    actual_price_today = data.loc[data.index[-1], 'Close']
    price_on_recommendation_date = data.loc[recommendation_date, 'Close']

    correct_prediction = False
    if recommendation == 'Buy' and actual_price_today > 1.05 * (price_on_recommendation_date):
        correct_prediction = True
    elif recommendation == 'Sell' and actual_price_today < 0.95 * (price_on_recommendation_date):
        correct_prediction = True
    elif recommendation == 'Hold' and actual_price_today < 1.05 * (price_on_recommendation_date) and actual_price_today > 0.95 * (price_on_recommendation_date):
        correct_prediction = True  # Assume hold is always correct

    return correct_prediction, price_on_recommendation_date, actual_price_today, recommendation
```

7.2.5 main.py: Combines the score of the technical indicators model and the news sentiment model to output a final recommendation to the user

```
import yfinance as yf
import random
from datetime import datetime, timedelta
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
from news_sentiment.article_generator import partition_and_fetch, load_json
from news_sentiment.sentiment_classifier2 import get_news_sentiment
from technical_ind.technical_ind import get_technical_indicator_score

Codeium: Refactor | Explain | Generate Docstring | X
def weighted_average(predictions, weights):
    combined_prediction = sum(pred * weight for pred, weight in zip(predictions, weights))
    total_weight = sum(weights)
    return combined_prediction / total_weight

Codeium: Refactor | Explain | Generate Docstring | X
def get_company_name_from_ticker(ticker):
    try:
        stock_info = yf.Ticker(ticker).info
        return stock_info.get('shortName', None)
    except Exception as e:
        print(f"Error retrieving company name for ticker {ticker}: {e}")
        return None
```

```
Codeium: Refactor | Explain | Generate Docstring | X
def true_value_recommendation(stock, start_date, end_date):
    ticker = yf.Ticker(stock)
    hist = ticker.history(start=start_date, end=end_date)

    if len(hist) == 0:
        return None

    start_price = hist['Close'].iloc[0]
    end_price = hist['Close'].iloc[-1]
    price_change = (end_price - start_price) / start_price

    if price_change >= 0.05:
        return 'Buy'
    elif price_change <= -0.05:
        return 'Sell'
    else:
        return 'Hold'

Codeium: Refactor | Explain | Generate Docstring | X
def main(stock):
    # Calculate the start and end dates
    end_date = (datetime.now() - timedelta(days=1)).strftime('%Y-%m-%d')
    start_date = (datetime.now() - timedelta(days=31)).strftime('%Y-%m-%d')
    two_months_ago_date = (datetime.now() - timedelta(days=62)).strftime('%Y-%m-%d')
    step_days = 2

    # Get the company name from the ticker
    company_name = get_company_name_from_ticker(stock)
    if company_name is None:
        print(f"No company name found for ticker {stock}")
        return None, None

    keyword = company_name.split()[0]  # Use the first word of the company name as the keyword
    sanitized_keyword = sanitize_filename(keyword)

    # Load API keys from the json file
    print("Loading API keys...")
    apiKeys = load_json(r"news_sentiment\api_keys.json", "api_keys")
    print(f"API keys loaded: {apiKeys}")
```

```python
# Fetch and process news sentiment data
print(f"Fetching news data for {sanitized_keyword} from {two_months_ago_date} to {start_date}...")
partition_and_fetch(sanitized_keyword, two_months_ago_date, start_date, step_days, apiKeys)
print(f"News data fetched and stored in data/news_{sanitized_keyword}.json")

print(f"Processing news sentiment for {sanitized_keyword}...")
news_score, news_recommendation, hold_count, buy_count, sell_count = get_news_sentiment(sanitized_keyword)
print(f"News Sentiment Score: {news_score}, Recommendation: {news_recommendation}")
print(f"Hold count: {hold_count}, Buy count: {buy_count}, Sell count: {sell_count}")

# Fetch and process technical indicator data
print(f"Fetching and processing technical indicators for {stock}...")
technical_score, technical_recommendation = get_technical_indicator_score(stock)
print(f"Technical Indicator Score: {technical_score}, Recommendation: {technical_recommendation}")

predictions = []
weights = []

if news_recommendation != "NA":
    print("Including news sentiment in prediction...")
    predictions.append(news_score)
    weights.append(1.0 if technical_recommendation == "NA" else 0.5)

if technical_recommendation != "NA":
    print("Including technical indicators in prediction...")
    predictions.append(technical_score)
    weights.append(1.0 if news_recommendation == "NA" else 0.5)

if not predictions:
    print("Insufficient data to make a recommendation.")
    return None, None

print(f"Predictions: {predictions}")
print(f"Weights: {weights}")

combined_prediction = weighted_average(predictions, weights)
print(f"Combined Prediction: {combined_prediction:.2f}")
```

```
    if combined_prediction >= 0.33:
        final_recommendation = "Buy"
    elif combined_prediction <= -0.33:
        final_recommendation = "Sell"
    else:
        final_recommendation = "Hold"

    print(f"Final Recommendation: {final_recommendation}")

    # Determine the true value of recommendation based on the stock price change
    true_recommendation = true_value_recommendation(stock, start_date, end_date)
    print(f"True Value of Recommendation: {true_recommendation}")

    return final_recommendation, true_recommendation

if __name__ == "__main__":
    # List of famous tickers
    famous_tickers = [
        "AAPL", "MSFT", "GOOGL", "AMZN", "META", "TSLA", "NVDA", "NFLX", "ADBE", "PYPL",
        "ORCL", "IBM", "CSCO", "INTC", "AMD", "AVGO", "QCOM", "TXN", "MU", "LRCX",
        "SHOP", "SQ", "ZM", "CRM", "NOW", "TEAM", "WDAY", "DOCU", "SNOW", "PLTR",
        "BABA", "JD", "PDD", "NIO", "XPEV", "LI", "BIDU", "TCEHY", "NTES", "IQ",
        "BA", "GE", "CAT", "MMM", "HON", "DE", "UPS", "FDX", "LMT", "NOC",
        "DIS", "CMCSA", "NFLX", "T", "VZ", "TMUS", "CHTR", "DISH", "SPOT", "ROKU",
        "PEP", "KO", "MNST", "KDP", "SBUX", "MCD", "YUM", "DPZ", "QSR", "SHAK",
        "JNJ", "PFE", "MRK", "BMY", "LLY", "ABBV", "AMGN", "GILD", "REGN", "VRTX",
        "XOM", "CVX", "COP", "PSX", "VLO", "OXY", "HES", "PXD", "EOG", "DVN"
    ]

    # Select 100 random tickers from the list, or all if the list has less than 100 tickers
    sample_size = min(len(famous_tickers), 100)
    selected_tickers = random.sample(famous_tickers, sample_size)

    predictions = []
    true_values = []
```

```
    for ticker in selected_tickers:
        print(f"\nProcessing ticker: {ticker}")
        prediction, true_value = main(ticker)
        if prediction is not None and true_value is not None:
            predictions.append(prediction)
            true_values.append(true_value)

    # Calculate and display the overall accuracy
    correct_predictions = sum(p == t for p, t in zip(predictions, true_values))
    accuracy = correct_predictions / len(predictions) if predictions else 0
    print(f'Overall Accuracy: {accuracy:.2f}%')

    # Calculate and display the confusion matrix
    labels = ['Buy', 'Sell', 'Hold']
    conf_matrix = confusion_matrix(true_values, predictions, labels=labels)
    cmd = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels=labels)
    cmd.plot()
    plt.show()
```