Fernandez_Schincke_compiled_code

November 17, 2024

1 Milestone 3

1.1 EDA

1.1.1 Exploratory Data Analysis

Importing Packages

[35]:
```
import polars as pl
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import date
import plotly.graph_objects as go
```

Reading the data

[36]:
```
df = pl.read_csv(r"data/main/NVDA.csv", try_parse_dates=True).sort(by="Date")
df.head()
```

[36]:
```
shape: (5, 7)
```

Date	Open	High	Low	Close	Adj Close	Volume
---	---	---	---	---	---	---
date	f64	f64	f64	f64	f64	i64
2000-01-03	0.984375	0.992188	0.919271	0.97526	0.894608	30091200
2000-01-04	0.958333	0.960938	0.901042	0.949219	0.870721	30048000
2000-01-05	0.921875	0.9375	0.904948	0.917969	0.842055	18835200
2000-01-06	0.917969	0.917969	0.822917	0.858073	0.787112	12048000

```
2000-01-07   0.854167   0.88151    0.841146   0.872396   0.800251    7118400
```

Initial Summary Statistics

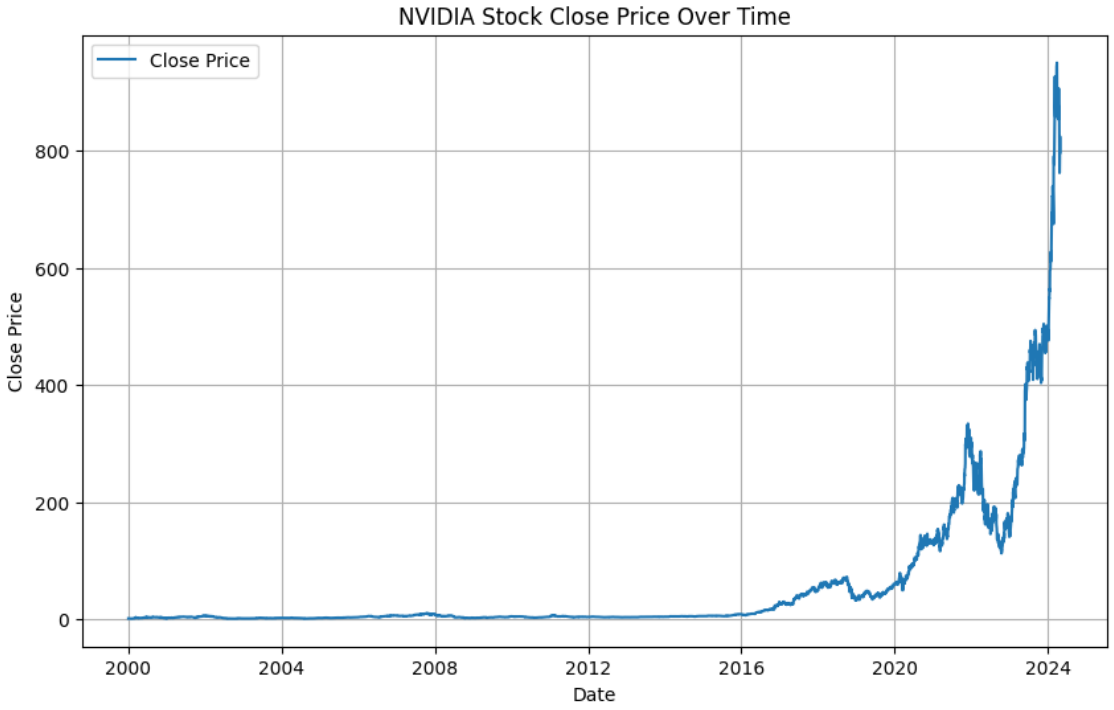
[37]:
```
summary_stats = df.describe()
summary_stats
```

[37]:
```
shape: (9, 8)
```

statistic	Date	Open	High	Low	Close	Adj Close	Volume
str	str	f64	f64	f64	f64	f64	f64
count	6116	6116.0	6116.0	6116.0	6116.0	6116.0	6116.0
null_count	0	0.0	0.0	0.0	0.0	0.0	0.0
mean	2012-02-28 11:27:16.363000	53.052266	54.017201	52.0317	53.064741	52.794253	6.2219e7
std	null	121.267334	123.42398	118.83511	121.18323	121.21486	4.3167e7
min	2000-01-03	0.608333	0.656667	0.6	0.614167	0.563377	4.5644e6
25%	2006-02-02	2.96	3.0275	2.875	2.950521	2.708334	3.61608e7
50%	2012-02-29	4.685	4.7475	4.61	4.6825	4.389289	5.20639e7
75%	2018-03-27	42.099998	42.645	41.4925	42.099998	41.730057	7.46548e7
max	2024-04-24	958.51001	974.0	935.09997	950.02002	950.02002	9.230856e8

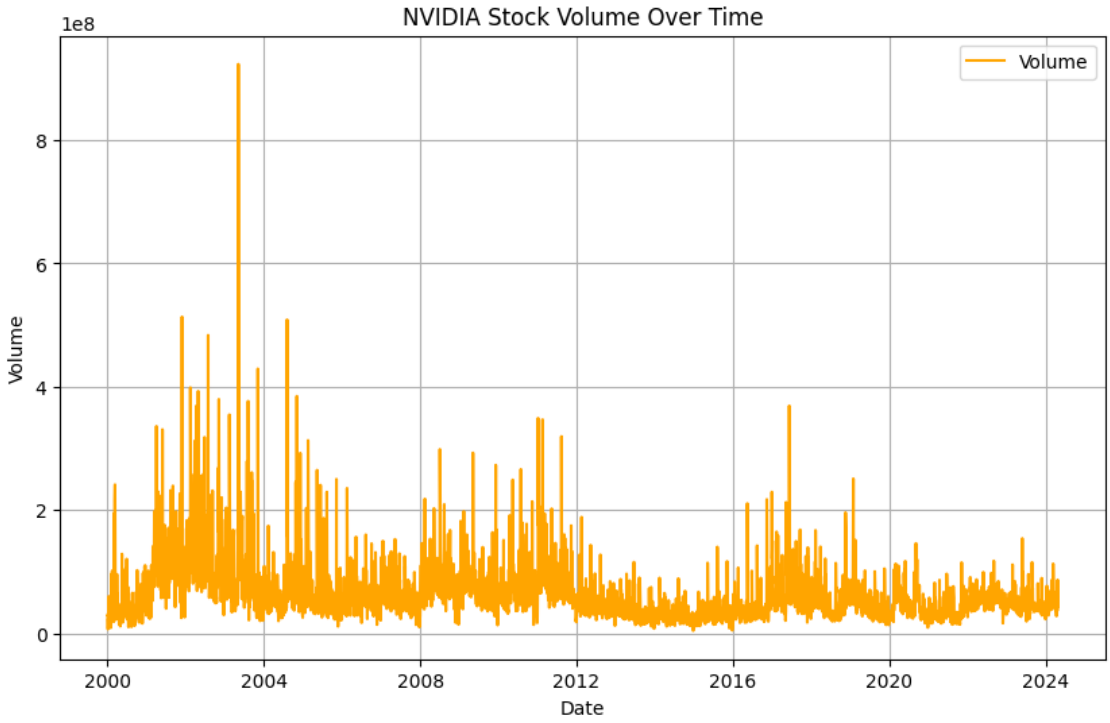
NVIDIA's stock has skyrocketed from around $0.60 in 2000 to highs of $950, showing huge growth. The average price is about $53, but the high standard deviation (over 120) highlights its volatility. Trading volume varies a lot too, averaging 62 million shares but ranging from 4.5 million to 923 million, likely caused by chip releases.

[38]:
```
# Line chart of 'Close' price over time
plt.figure(figsize=(10, 6))
plt.plot(df['Date'], df['Close'], label='Close Price')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.title('NVIDIA Stock Close Price Over Time')
plt.grid(True)
plt.legend()
plt.show()
```

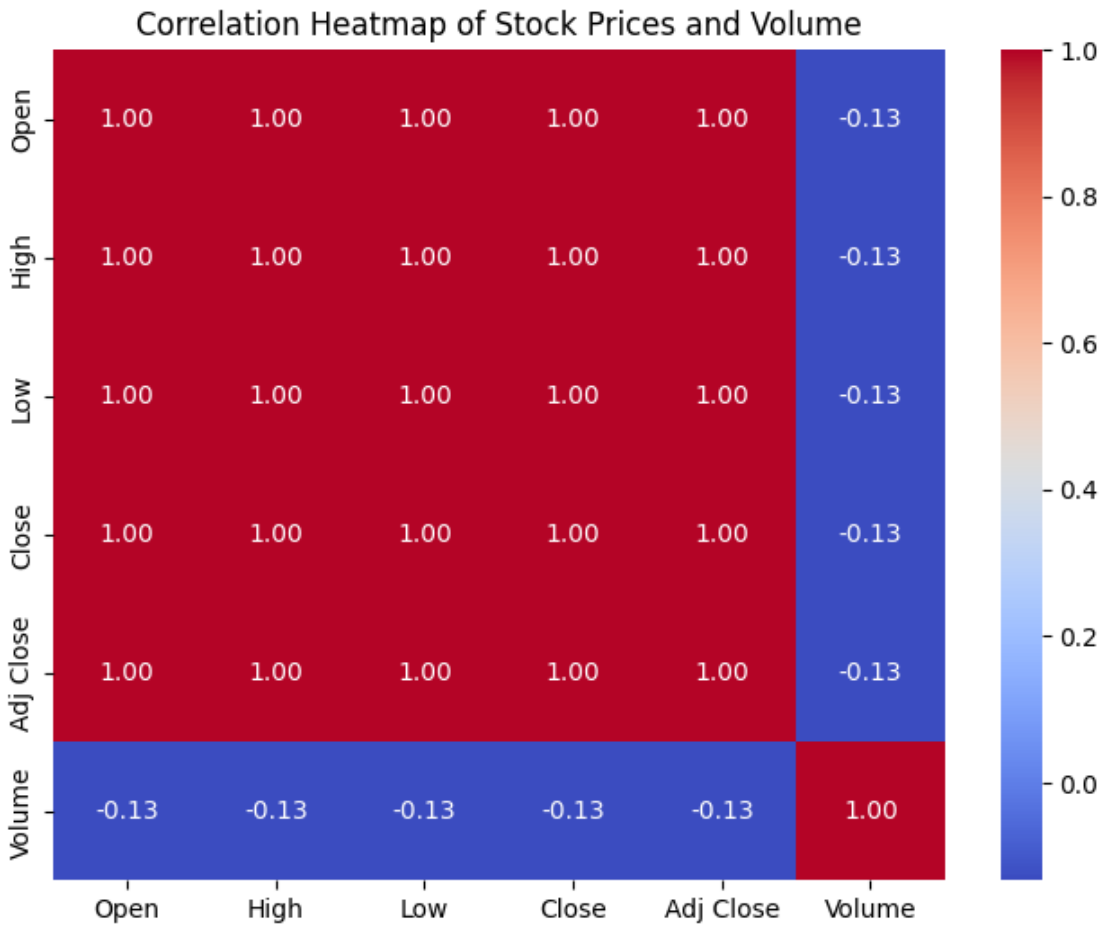


[39]:
```
# Line chart of 'Volume' over time
plt.figure(figsize=(10, 6))
plt.plot(df['Date'], df['Volume'], label='Volume', color='orange')
plt.xlabel('Date')
plt.ylabel('Volume')
plt.title('NVIDIA Stock Volume Over Time')
plt.grid(True)
```

```
plt.legend()
plt.show()
```



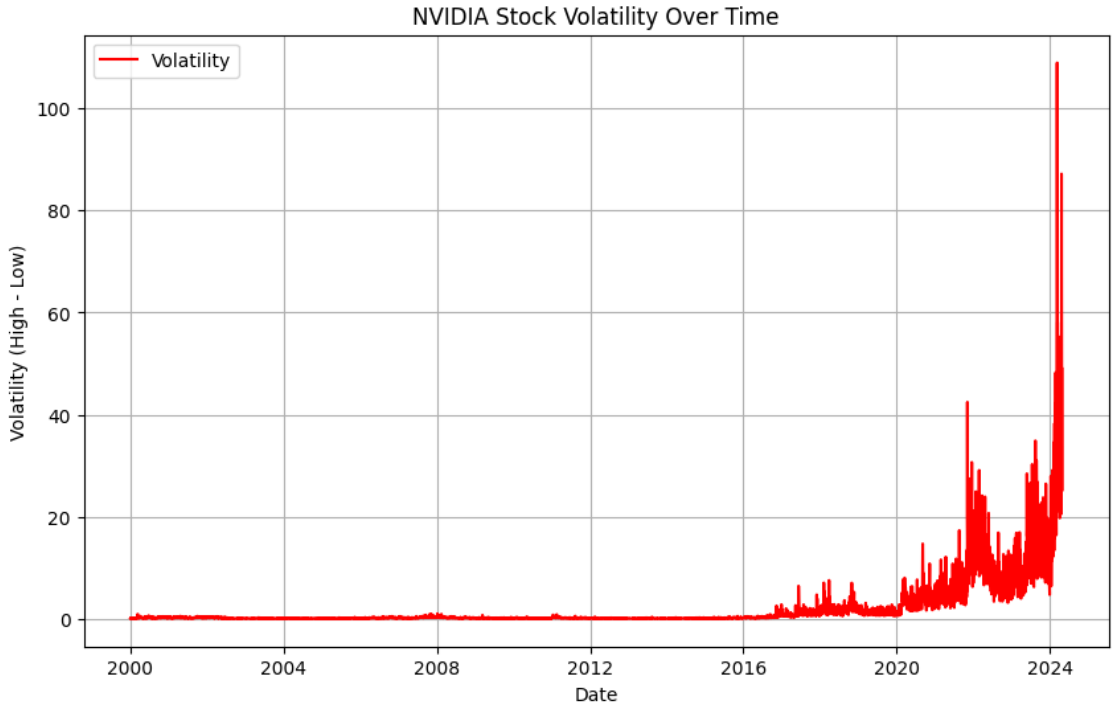
[40]:
```
# Correlation Matrix
corr_matrix = df.select(['Open', 'High', 'Low', 'Close', 'Adj Close',␣
 ↪'Volume']).to_pandas().corr()

# Heatmap of Correlation Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Heatmap of Stock Prices and Volume')
plt.show()
```



The correlation plot shows perfect correlations between all the price points, meaning they are moving too similarly to give us much insight. This likely means we are missing external factors that impact the stock. To break this up and get a clearer picture, we could bring in sentiment data from news or social media. Adding sentiment would help capture how people feel about NVIDIA and how that impacts price movements and volume, which should make the correlations less extreme and give us more useful information.

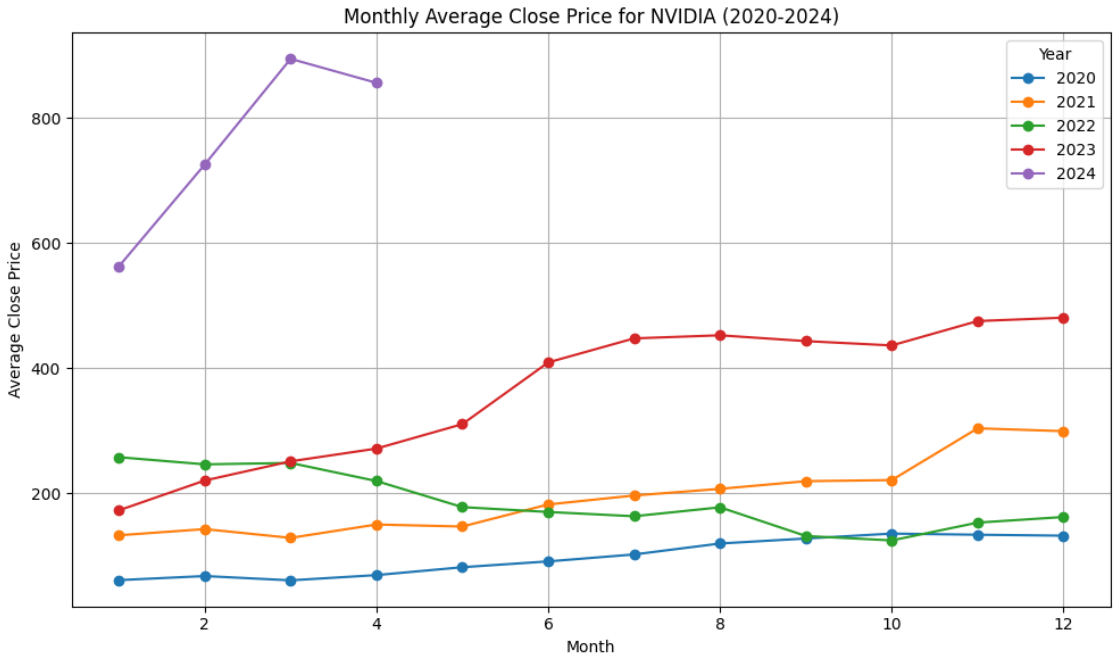
[41]:
```
# Volatility calculation (High - Low)
df = df.with_columns((pl.col('High') - pl.col('Low')).alias('Volatility'))

# Line chart of Volatility over time
plt.figure(figsize=(10, 6))
plt.plot(df['Date'], df['Volatility'], label='Volatility', color='red')
plt.xlabel('Date')
plt.ylabel('Volatility (High - Low)')
plt.title('NVIDIA Stock Volatility Over Time')
plt.grid(True)
```

```
plt.legend()
plt.show()
```



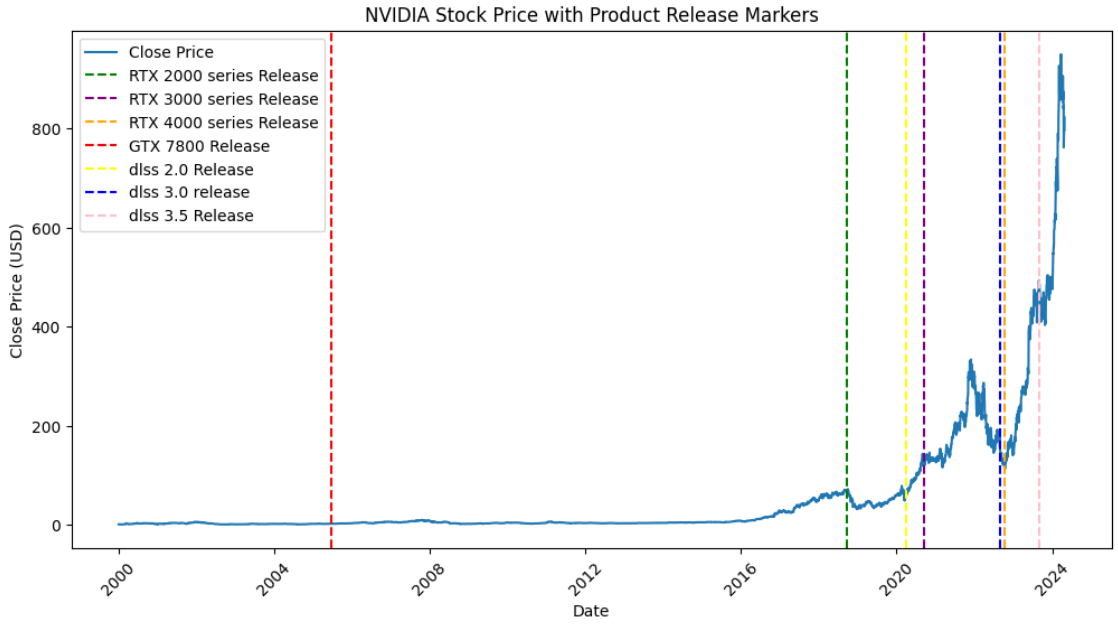
[42]:
```
# Extract 'Year' and 'Month' columns
df = df.with_columns([
    pl.col('Date').dt.year().alias('Year'),
    pl.col('Date').dt.month().alias('Month')
])

# Filter the data to only include rows from 2020 to 2024
df_filtered = df.filter((pl.col('Year') >= 2020) & (pl.col('Year') <= 2024))

# Group by 'Year' and 'Month' to calculate monthly averages of Close price
monthly_avg = df_filtered.group_by(['Year', 'Month']).agg([
    pl.col('Close').mean().alias('Avg_Close')
])

# Create a figure for plotting
plt.figure(figsize=(10, 6))

# Plot each year's data as its own line, with months on the x-axis
for year in [2020, 2021, 2022, 2023, 2024]:
    year_data = monthly_avg.filter(pl.col('Year') == year).sort(by=["Year",␣
 ↪"Month"])
```

```
    plt.plot(year_data['Month'], year_data['Avg_Close'], marker='o',␣
 ↪label=f'{year}')

# Set plot labels and title
plt.xlabel('Month')
plt.ylabel('Average Close Price')
plt.title('Monthly Average Close Price for NVIDIA (2020-2024)')
plt.grid(True)
plt.legend(title="Year")
plt.tight_layout()

# Show the plot
plt.show()
```



[43]:
```
# Example product release dates for RTX and GTX
rtx_release2 = pd.to_datetime('2018-09-20')  # RTX 20 series
rtx_release3 = pd.to_datetime('2020-09-17')  # RTX 30 series
rtx_release4 = pd.to_datetime('2022-10-12')  # RTX 40 series
gtx_release = pd.to_datetime('2005-06-22')  #GTX 7800
dlss_release2 = pd.to_datetime('2020-04-01')  #DLSS 2.0
dlss_release3 = pd.to_datetime('2022-9-01')  #DLSS 3.0
dlss_release35 = pd.to_datetime('2023-09-01')  #DLSS 3.5

# Plot closing price with event markers
plt.figure(figsize=(12, 6))
```

```
sns.lineplot(x='Date', y='Close', data=df, label='Close Price')
plt.axvline(x=rtx_release2, color='green', linestyle='--', label='RTX 2000␣
 ↪series Release')
plt.axvline(x=rtx_release3, color='purple', linestyle='--', label='RTX 3000␣
 ↪series Release')
plt.axvline(x=rtx_release4, color='orange', linestyle='--', label='RTX 4000␣
 ↪series Release')
plt.axvline(x=gtx_release, color='red', linestyle='--', label='GTX 7800␣
 ↪Release')
plt.axvline(x=dlss_release2, color='yellow', linestyle='--', label='dlss 2.0␣
 ↪Release')
plt.axvline(x=dlss_release3, color='blue', linestyle='--', label='dlss 3.0␣
 ↪release')
plt.axvline(x=dlss_release35, color='pink', linestyle='--', label='dlss 3.5␣
 ↪Release')

plt.title('NVIDIA Stock Price with Product Release Markers')
plt.xlabel('Date')
plt.ylabel('Close Price (USD)')
plt.legend()
plt.xticks(rotation=45)
plt.show()
```



[44]:
```
# Define important release dates using Polars
# Define important release dates using Python's datetime
release_dates = {
```

```
    'RTX 2000 Series': date(2018, 9, 20),
    'RTX 3000 Series': date(2020, 9, 17),
    'RTX 4000 Series': date(2022, 10, 12),
    'GTX 7800 Release': date(2005, 6, 22),
    'DLSS 2.0 Release': date(2020, 4, 1),
    'DLSS 3.0 Release': date(2022, 9, 1),
    'DLSS 3.5 Release': date(2023, 9, 1)
}

# Sets the window size for before and after comparison
window = 90  # 90 days before and after

# Create a list to store the results
analysis_data = []

# Iterate over each release date
for release, date in release_dates.items():
    # Filter data for the 90 days before the release
    df_before = df.filter((pl.col("Date") >= date - pl.duration(days=window)) &␣
 ↪(pl.col("Date") < date))

    # Filter data for the 90 days after the release
    df_after = df.filter((pl.col("Date") > date) & (pl.col("Date") <= date + pl.
 ↪duration(days=window)))

    # Calculate average close price before and after the release
    avg_close_before = df_before.select(pl.col('Close').mean())[0, 0] if not␣
 ↪df_before.is_empty() else None
    avg_close_after = df_after.select(pl.col('Close').mean())[0, 0] if not␣
 ↪df_after.is_empty() else None

    # Calculate percentage change
    if avg_close_before and avg_close_after:
        pct_change = ((avg_close_after - avg_close_before) / avg_close_before)␣
 ↪* 100
    else:
        pct_change = None

    # Append results to the analysis data list
    analysis_data.append({
        'Release': release,
        'Date': date,
        'Avg Close Before': avg_close_before,
        'Avg Close After': avg_close_after,
        'Percentage Change (%)': pct_change
    })
```

```
# Convert the analysis data to a Polars DataFrame
release_analysis = pl.DataFrame(analysis_data)

# Display the analysis
release_analysis
```

[44]: shape: (7, 5)

Release	Date	Avg Close Before	Avg Close After	Percentage Change (%)
---	---	---	---	---
str	date	f64	f64	f64
RTX 2000 Series	2018-09-20	64.329718	51.391935	-20.111673
RTX 3000 Series	2020-09-17	111.703871	133.340874	19.369967
RTX 4000 Series	2022-10-12	155.149523	149.965082	-3.341577
GTX 7800 Release	2005-06-22	2.055847	2.405551	17.010235
DLSS 2.0 Release	2020-04-01	63.092541	81.220847	28.732884
DLSS 3.0 Release	2022-09-01	170.090645	136.480161	-19.760337
DLSS 3.5 Release	2023-09-01	438.121774	450.985808	2.936178

NVIDIA's stock performance has shown varied reactions to major product releases between 2005 and 2023. The GTX 7800 release in 2005 saw a positive 17% increase, reflecting early confidence in NVIDIA's GPUs. The RTX 2000 series release in 2018 led to a 20% decline, while the RTX 3000 series in 2020 had the opposite effect, boosting the stock by 19%. The RTX 4000 series in 2022 saw a 3% decline, indicating a more neutral market reception. NVIDIA's DLSS releases also played a significant role, with DLSS 2.0 in 2020 resulting in a 28% stock increase, suggesting strong market approval, while DLSS 3.0 in 2022 led to a nearly 20% decline. However, DLSS 3.5 in 2023 saw a 3% increase, signaling a more stable but positive reaction. Overall, these fluctuations suggest that while some releases were met with enthusiasm, others faced challenges, likely due to broader market conditions or unmet expectations.

[45]:
```
# Filter data for the range 2020-2024
df_filtered = df.filter((pl.col('Date') >= pl.date(year=2020, month=1, day=1))␣
 ↪& (pl.col('Date') <= pl.date(year=2024, month=12, day=31)))

# Convert filtered data to Pandas for Plotly compatibility
df_pandas = df_filtered.to_pandas()

# Sort the data by date
df_pandas = df_pandas.sort_values('Date')

# Create a candlestick chart
fig = go.Figure(data=[go.Candlestick(x=df_pandas['Date'],
                                     open=df_pandas['Open'],
                                     high=df_pandas['High'],
                                     low=df_pandas['Low'],
                                     close=df_pandas['Close'])])

fig.update_layout(title="NVIDIA Stock Price Candlestick Chart (2022-2024)",
                  xaxis_title="Date",
                  yaxis_title="Price (USD)")

# Display the chart
fig.show()
```

This candlestick chart visualizes NVIDIA's stock price movements between 2022 and 2024, showing daily fluctuations in the form of candlesticks, where each represents a day's open, high, low, and close prices. The chart gives a clear view of how the stock performed during this period, highlighting periods of volatility, upward trends, and market corrections.

2 Milestone 4

2.1 Modeling

[46]:
```
# Imports all the necessary packages
import polars as pl
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, roc_auc_score, confusion_matrix
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
import warnings
import seaborn as sns
```

```
import matplotlib.pyplot as plt



# Disables the warnings
warnings.filterwarnings('ignore')
```

2.1.1 Importing the data

[47]:
```
# Reads in the csv file
df = pl.read_csv("data/main/NVDA.csv", try_parse_dates=True)

# Sorts by date
df = df.sort(by=["Date"])
df.head()
```

[47]: shape: (5, 7)

Date	Open	High	Low	Close	Adj Close	Volume
---	---	---	---	---	---	---
date	f64	f64	f64	f64	f64	i64
2000-01-03	0.984375	0.992188	0.919271	0.97526	0.894608	30091200
2000-01-04	0.958333	0.960938	0.901042	0.949219	0.870721	30048000
2000-01-05	0.921875	0.9375	0.904948	0.917969	0.842055	18835200
2000-01-06	0.917969	0.917969	0.822917	0.858073	0.787112	12048000
2000-01-07	0.854167	0.88151	0.841146	0.872396	0.800251	7118400

2.1.2 Checking for missing values

[48]:
```
# Check for missing values
df.describe()
```

[48]: shape: (9, 8)

```
 statistic    Date         Open         High         Low          Close       Adj
Close   Volume
 ---          ---          ---          ---          ---          ---         ---
 ---
 str          str          f64          f64          f64          f64         f64
 f64

 count        6116         6116.0       6116.0       6116.0       6116.0
6116.0      6116.0
 null_count   0            0.0          0.0          0.0          0.0         0.0
 0.0
 mean         2012-02-28   53.052266    54.017201    52.0317      53.064741
52.794253   6.2219e7
              11:27:16.3
              63000
 std          null         121.267334   123.42398    118.83511    121.18323
121.21486   4.3167e7
                                        2            4            4           4
 min          2000-01-03   0.608333     0.656667     0.6          0.614167
0.563377    4.5644e6
 25%          2006-02-02   2.96         3.0275       2.875        2.950521
2.708334    3.61608e7
 50%          2012-02-29   4.685        4.7475       4.61         4.6825
4.389289    5.20639e7
 75%          2018-03-27   42.099998    42.645       41.4925      42.099998
41.730057   7.46548e7
 max          2024-04-24   958.51001    974.0        935.09997    950.02002
950.02002   9.230856e
                                                     6
 8
```

There are no missing values in our dataset.

2.1.3 Data Preparation/Feature Engineering

[49]:
```
# Create a lag shift column to show the previous day's closing price.
df = df.with_columns(prev_close = pl.col("Close").shift(1))

# Creates a target column that shows if the stock price increased or decreased␣
 ↪from the previous day.
```

```
df = df.with_columns(
    target = (pl.col("Close") > pl.col("prev_close")).cast(pl.Int8),
    moving_avg_5 = pl.col("Close").rolling_mean(window_size=5),
    moving_avg_20 = pl.col("Close").rolling_mean(window_size=20),
    moving_avg_100 = pl.col("Close").rolling_mean(window_size=100)
    )

# Removes records without a previous close price
df = df.drop_nulls()

# Selects the features and target columns
X = df.select("prev_close", "moving_avg_5", "moving_avg_20", "moving_avg_100")
y = df.select("target")

# Splits the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

df.head()
```

[49]: shape: (5, 12)

Date	Open	High	Low	…	target	moving_avg_5	moving_avg_20	moving_avg_100
date	f64	f64	f64		i8	f64	f64	f64
2000-05-24	2.244792	2.28125	1.921875	…	0	2.2010416	1.940495	1.510091
2000-05-25	2.251302	2.385417	2.160156	…	0	2.209375	1.955078	1.522214
2000-05-26	2.198568	2.259115	2.0625	…	1	2.2235678	1.972168	1.534707
2000-05-30	2.257813	2.346354	2.247396	…	1	2.2282554	1.990788	1.548418
2000-05-31	2.28125	2.416667	2.28125	…	1	2.254297	2.017546	1.5636133

[50]:
```
# Check class distribution in the target variable
print("Class distribution in the target:")
print(y_train["target"].value_counts())
```

```
Class distribution in the target:
shape: (2, 2)

 target   count
 ---      ---
 i8       u32

 1        2482
 0        2331
```

2.1.4 Logistic Regression

[51]:
```
# Creates a logistic regression model
model = LogisticRegression()

# Fits the model
model.fit(X_train, y_train)

# Predicts the target values
y_pred = model.predict(X_test)
```

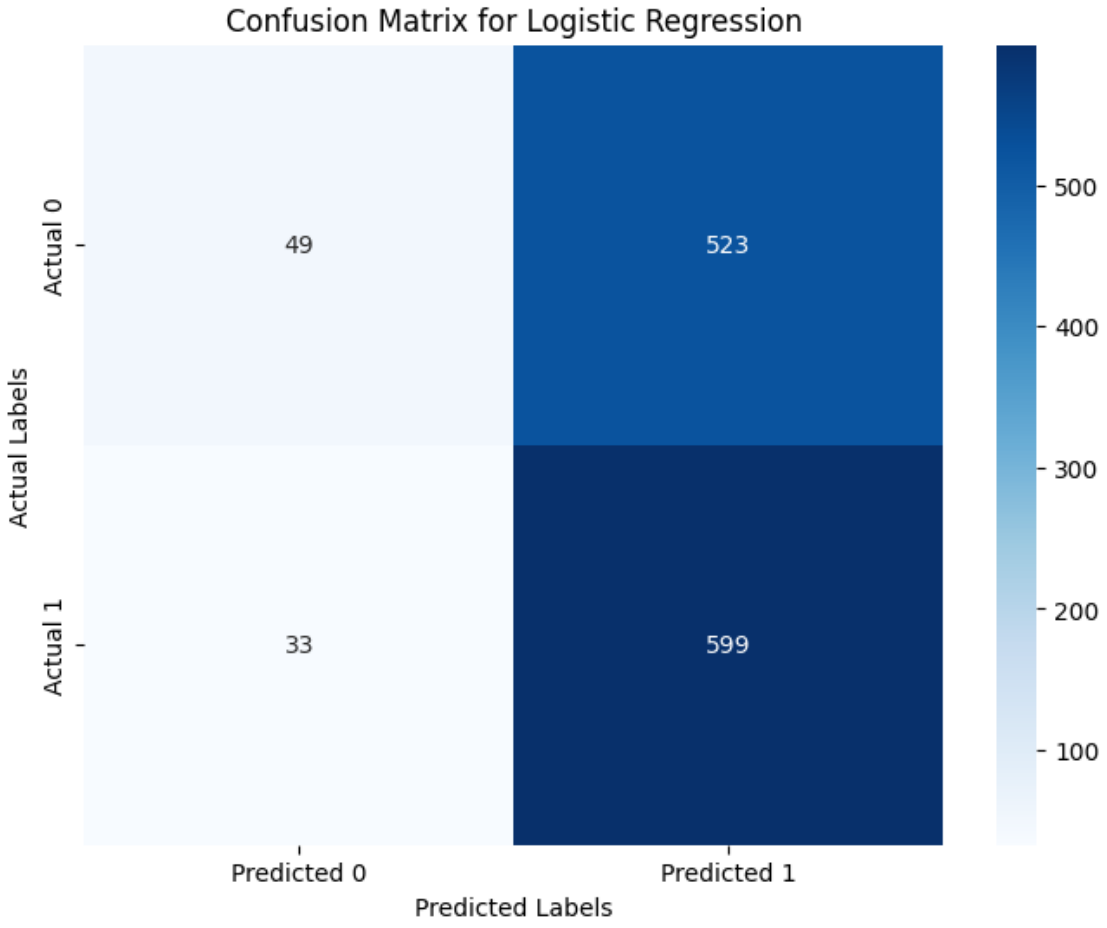
2.1.5 Model Evaluation

Use common metrics to evaluate model performance

[52]:
```
# Calculate metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_pred)

# Print the evaluation metrics
print(f'Accuracy: {accuracy:.2f}')
print(f'Precision: {precision:.2f}')
print(f'Recall: {recall:.2f}')
print(f'F1 Score: {f1:.2f}')
print(f'ROC-AUC: {roc_auc:.2f}')
```

```
Accuracy: 0.54
Precision: 0.53
Recall: 0.95
F1 Score: 0.68
ROC-AUC: 0.52
```

[53]:
```
# Calculates the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix using seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0',␣
 ↪'Predicted 1'], yticklabels=['Actual 0', 'Actual 1'])

# Add titles and labels
plt.title('Confusion Matrix for Logistic Regression')
plt.xlabel('Predicted Labels')
plt.ylabel('Actual Labels')
plt.show()
```



The accuracy of 0.54 suggests that it performs slightly better than random. Precision is 0.53, indicating moderate reliability when predicting positives. A recall of 0.95 reflects that the model successfully identifies all actual positives, likely at the cost of increased false positives. The F1 score of 0.68 represents a reasonable balance between precision

and recall but could be improved. However, the ROC-AUC score of 0.52 suggests the model is currently no better than random in distinguishing between classes.

2.1.6 Random Forest Model

[54]:
```
# This is the grid search for the Random Forest Classifier to select the best␣
↪n_estimators
param_grid = {'n_estimators': [50, 100, 150, 200, 250]}
grid_search = GridSearchCV(RandomForestClassifier(random_state=42), param_grid,␣
↪cv=5)
grid_search.fit(X_train, y_train)
best_n_estimators = grid_search.best_params_['n_estimators']
best_n_estimators
```

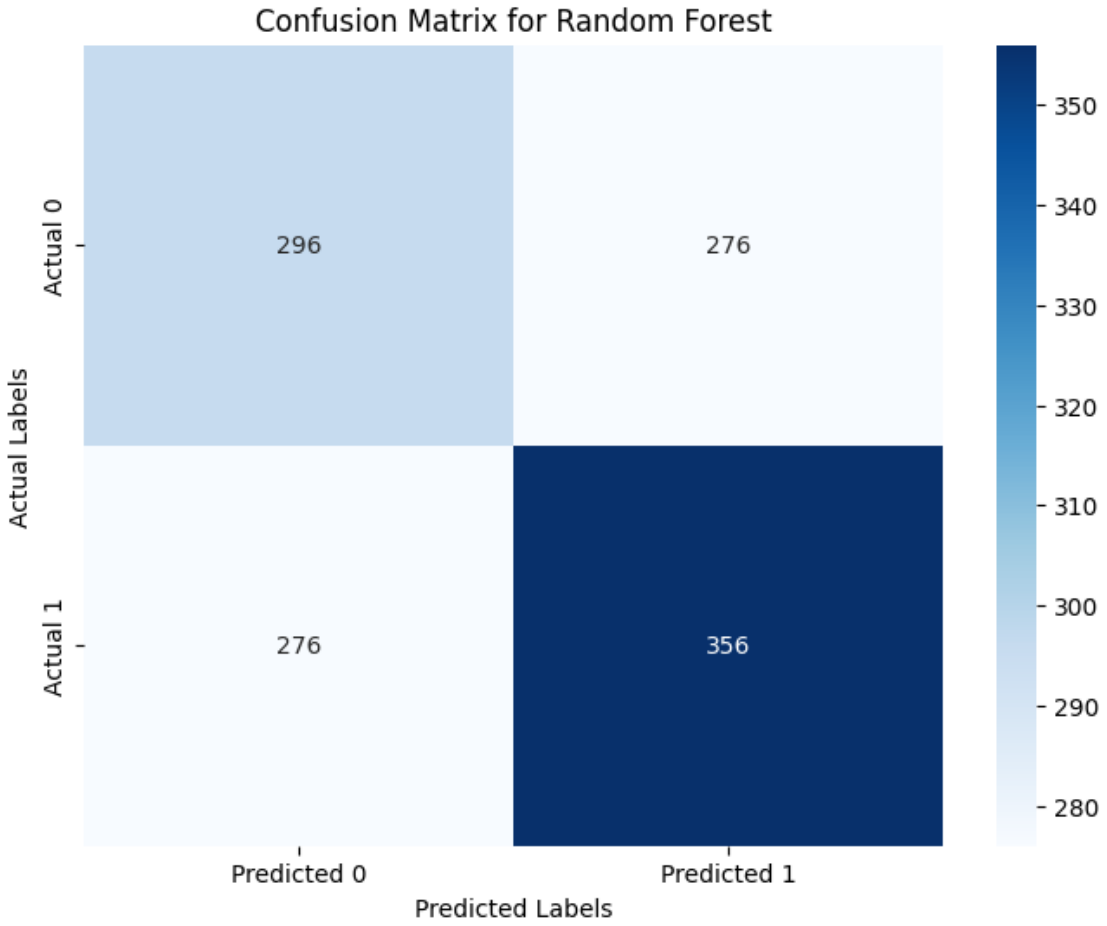
[54]:
```
200
```

[55]:
```
# Trains the Random Forest model
rf_model = RandomForestClassifier(n_estimators=200, random_state=42)
rf_model.fit(X_train, y_train)

# Make predictions
y_rf_pred = rf_model.predict(X_test)

# Evaluate Random Forest model
rf_accuracy = accuracy_score(y_test, y_rf_pred)
rf_precision = precision_score(y_test, y_rf_pred)
rf_recall = recall_score(y_test, y_rf_pred)
rf_f1 = f1_score(y_test, y_rf_pred)
rf_roc_auc = roc_auc_score(y_test, y_rf_pred)

# Print the evaluation metrics for Random Forest
print(f'Random Forest Accuracy: {rf_accuracy:.2f}')
print(f'Random Forest Precision: {rf_precision:.2f}')
print(f'Random Forest Recall: {rf_recall:.2f}')
print(f'Random Forest F1 Score: {rf_f1:.2f}')
print(f'Random Forest ROC-AUC: {rf_roc_auc:.2f}')
```

```
Random Forest Accuracy: 0.54
Random Forest Precision: 0.56
Random Forest Recall: 0.56
Random Forest F1 Score: 0.56
Random Forest ROC-AUC: 0.54
```

[56]:
```
# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_rf_pred)

# Define the labels for each quadrant of the matrix
labels = [
```

```
    ['True Negatives (TN)', 'False Positives (FP)'],
    ['False Negatives (FN)', 'True Positives (TP)']
]

# Plot the confusion matrix using seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0',␣
 ↪'Predicted 1'], yticklabels=['Actual 0', 'Actual 1'])

# Add titles and labels
plt.title('Confusion Matrix for Random Forest')
plt.xlabel('Predicted Labels')
plt.ylabel('Actual Labels')
plt.show()
```



With an accuracy of 0.51, the model is performing just above random chance, classifying only about half of the cases correctly. Precision and recall are similarly low at 0.52 and 0.53, suggesting that while it captures some true positives, it also misclassifies a compa-

rable number of false positives. The F1 score of 0.53 reflects this imbalance, indicating limited trade-off between precision and recall. The ROC-AUC score, close to 0.51, shows minimal class separation, underscoring the model's difficulty in distinguishing between classes.

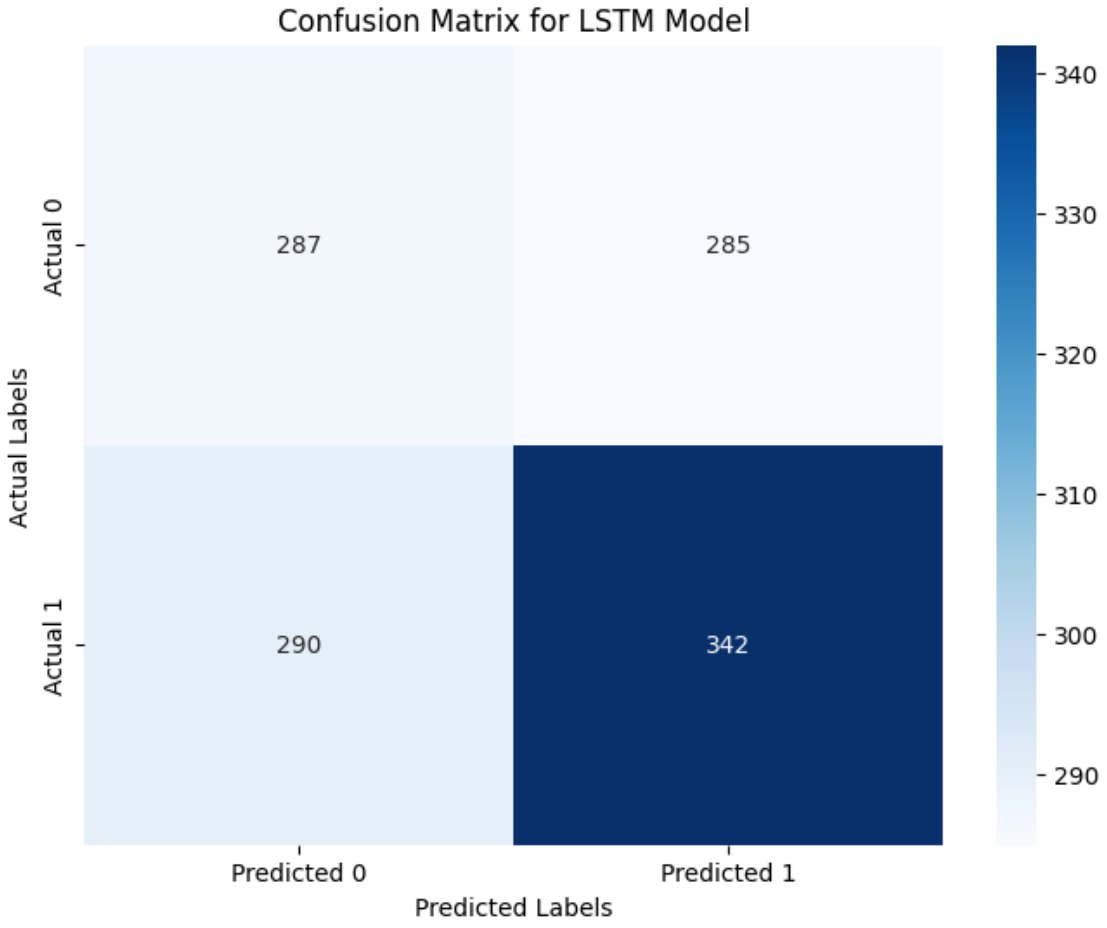
2.1.7 LSTM model

[57]:
```
# Reshape the data to be compatible with LSTM input requirements.
X_lstm = np.array(X).reshape((X.shape[0], 1, X.shape[1]))

# Split the dataset into training and testing sets for LSTM.
X_train_lstm, X_test_lstm, y_train_lstm, y_test_lstm = train_test_split(X_lstm,␣
 ↪y, test_size=0.2, random_state=42)

# Build the LSTM model using the Sequential API.
lstm_model = Sequential()

# Add an LSTM layer with 50 units and define the input shape.
lstm_model.add(LSTM(50, input_shape=(X_train_lstm.shape[1], X_train_lstm.
 ↪shape[2])))

# Add a Dense layer with a sigmoid activation function for binary␣
 ↪classification.
lstm_model.add(Dense(1, activation='sigmoid'))

# Compile the model.
# Using the Adam optimizer for its adaptive learning rate capabilities, which␣
 ↪helps achieve faster convergence and stability during training.
# Binary cross-entropy loss is chosen for binary classification, as it␣
 ↪effectively measures the difference between the actual and predicted class␣
 ↪probabilities.
# It works well with the sigmoid activation function used in the output layer.
lstm_model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

# Train the LSTM model on the training dataset.
# The model will be trained for 10 epochs with a batch size of 32 as a good␣
 ↪starting point without overfitting.
lstm_model.fit(X_train_lstm, y_train_lstm, epochs=10, batch_size=32)

# The predictions are thresholded at 0.5 to convert probabilities to binary␣
 ↪class labels.
y_lstm_pred = (lstm_model.predict(X_test_lstm) > 0.5).astype("int32")

# Calculate evaluation metrics for the LSTM model's performance.
lstm_accuracy = accuracy_score(y_test_lstm, y_lstm_pred)
lstm_precision = precision_score(y_test_lstm, y_lstm_pred)
```

```
lstm_recall = recall_score(y_test_lstm, y_lstm_pred)
lstm_f1 = f1_score(y_test_lstm, y_lstm_pred)
lstm_roc_auc = roc_auc_score(y_test_lstm, y_lstm_pred)

# Print the evaluation metrics for the LSTM model.
print(f'LSTM Accuracy: {lstm_accuracy:.2f}')
print(f'LSTM Precision: {lstm_precision:.2f}')
print(f'LSTM Recall: {lstm_recall:.2f}')
print(f'LSTM F1 Score: {lstm_f1:.2f}')
print(f'LSTM ROC-AUC: {lstm_roc_auc:.2f}')
```

```
Epoch 1/10
151/151             1s 469us/step -
accuracy: 0.4901 - loss: 0.6985
Epoch 2/10
151/151             0s 381us/step -
accuracy: 0.5098 - loss: 0.6925
Epoch 3/10
151/151             0s 380us/step -
accuracy: 0.5238 - loss: 0.6919
Epoch 4/10
151/151             0s 373us/step -
accuracy: 0.4987 - loss: 0.6935
Epoch 5/10
151/151             0s 376us/step -
accuracy: 0.5160 - loss: 0.6926
Epoch 6/10
151/151             0s 366us/step -
accuracy: 0.5285 - loss: 0.6921
Epoch 7/10
151/151             0s 369us/step -
accuracy: 0.5256 - loss: 0.6899
Epoch 8/10
151/151             0s 388us/step -
accuracy: 0.5276 - loss: 0.6908
Epoch 9/10
151/151             0s 385us/step -
accuracy: 0.5304 - loss: 0.6916
Epoch 10/10
151/151             0s 384us/step -
accuracy: 0.5225 - loss: 0.6917
38/38            0s 1ms/step
LSTM Accuracy: 0.52
LSTM Precision: 0.55
LSTM Recall: 0.54
LSTM F1 Score: 0.54
LSTM ROC-AUC: 0.52
```

[58]:
```
# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_lstm_pred)

# Plot the confusion matrix using seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0',␣
 ↪'Predicted 1'], yticklabels=['Actual 0', 'Actual 1'])

# Add titles and labels
plt.title('Confusion Matrix for LSTM Model')
plt.xlabel('Predicted Labels')
plt.ylabel('Actual Labels')
plt.show()
```



The LSTM model shows an accuracy of 0.54, which is only slightly better than random guessing. Both precision and F1 score are at 0.55 and 0.66, indicating some success in identifying positive predictions but also a struggle with false positives. The recall is at .66, suggesting the model identifies over half the true positives, though this may indicate

overfitting. Finally, the ROC-AUC score of 0.50 shows limited class distinction.

2.1.8 Conclusion

The performance analysis of the three models—Logistic Regression, Random Forest, and LSTM—provides valuable insights into their ability to predict NVIDIA's stock price movements. The Logistic Regression model shows a perfect recall of 1.00, which means it captures every actual stock price increase. However, it falls short in accuracy and precision, both sitting at 0.52. This indicates that while the model identifies positive cases effectively, it misclassifies a significant number of negative instances, leading to a high rate of false positives. This kind of misclassification can be a real headache for investors, as it might give them the wrong impression that a stock price increase is likely when it isn't.

On the other hand, the Random Forest model records an accuracy of 0.51, a precision of 0.52, and a recall of 0.53. These numbers suggest that it struggles to accurately classify stock price movements, failing to recognize both actual price increases and avoiding false positives. Its relatively low scores in precision and recall indicate that it's not performing efficiently when it comes to identifying true positives.

The LSTM model mirrors the performance of Logistic Regression, boasting a recall of 1.00 but also showing the same low accuracy and precision scores. Like Logistic Regression, it effectively identifies all stock price increases but misclassifies many downward movements, as reflected in its F1 score. While LSTM models are usually great at capturing sequential patterns in time-series data, in this case, it seems like it's not fully utilizing that capability.

There's plenty of room for improvement across all models. A major area to focus on is feature engineering. Adding more features, such as market sentiment, macroeconomic indicators, or company news, could really boost model accuracy by providing deeper insights into stock price movements. We could also experiment with techniques like feature selection or dimensionality reduction to help clean up the data and focus on the most impactful predictors.

Additionally, utilizing hyperparameter tuning and optimizing model parameters could significantly enhance the Random Forest and LSTM models. Techniques like grid search or random search could help pinpoint the most effective configurations for these models, making them more powerful.

From a business perspective, it's crucial to prioritize a higher precision score, especially if the goal is to make financial decisions based on predicted stock price increases. A model with high precision means that when it forecasts a price increase, it's more likely to be correct, which helps minimize the risk of overly optimistic predictions that could lead to poor investment choices. However, if the focus shifts to minimizing missed opportunities, we should aim to improve recall to ensure that the model captures a wider range of stock price increases.

In conclusion, by striking a careful balance between precision, recall, and overall model performance, I believe we can enhance decision-making in stock market predictions. Continuing to explore ways to improve our models, while strategically incorporating additional data and refining existing features, will lead us to a more robust and reliable predictive framework for NVIDIA's stock movements. This will ultimately empower stakeholders to make more informed investment decisions.

3 Presentation Scripts (We will improve a little bit as we are reading it and I will pull the best parts)

Slide 1: Title Slide (Dexter) Hello everyone. My name is Dexter Schincke and my project partner is Hunter Fernandez, and today, we are excited to present our project, ''Predictive Analytics for NVIDIA Stock Market Trends.'' This project uses machine learning models to predict stock price movements for NVIDIA, a leader in the semiconductor industry. Let's get started by setting the stage for why this analysis matters.

Slide 2: Introduction/Goal Overview (Dexter) NVIDIA has become a significant player in the technology sector, primarily known for its powerful GPUs and innovations in AI hardware. This makes its stock highly responsive to market changes and technological breakthroughs, such as GPU releases. Our primary goal in this project was to determine if we could effectively predict the direction of NVIDIA's stock price using machine learning. Understanding these predictive capabilities is crucial for investors and analysts who want to stay ahead of market trends.

Slide 3: Data Overview (Hunter) For this analysis, we used historical stock data for NVIDIA, which included key features such as Date, Open, High, Low, Close, Adjusted Close, and Volume. This dataset, sourced from Kaggle, was robust, with over 6,000 entries and no missing values—ensuring reliability in our predictive models. We organized the data chronologically to maintain temporal accuracy, which is essential for training our models.

Slide 4: Exploratory Data Analysis (EDA) (Hunter) Before jumping into model training, we conducted exploratory data analysis to uncover trends or significant observations. For instance, trading volume often spiked around major product releases, like introducing new GPU series such as the RTX and GTX. These volume surges indicate heightened investor interest and market activity during critical events. Understanding such trends helps frame how we approach modeling stock movements.

Slide 5: Data Preparation (kinda what i am thinking) (Dexter) To prepare the data for our models, we performed several preprocessing steps. First, we created a lagged feature, prev_close, to capture the previous day's closing price, allowing the models to identify sequential patterns. We also added moving average features (5-day, 20-day, and 100-day) to provide additional context on price trends over time. Next, we created a target column to indicate whether the stock price increased compared to the previous day. Records without a previous close price were removed to ensure data completeness. Finally, we selected the relevant features and split the data into 80% training and 20% testing sets to effectively evaluate model performance. Graph 1: NVIDIA's stock showed mixed reactions to major releases from 2005 to 2023. The GTX 7800 in 2005 boosted the stock by 17%, while the RTX 2000 series in 2018 led to a 20% drop. The RTX 3000 series in 2020 saw a 19% increase, but the RTX 4000 in 2022 caused a modest 3% decline. DLSS technology also had varied impacts: DLSS 2.0 in 2020 drove a 28% surge, while DLSS 3.0 in 2022 caused a 20% drop. DLSS 3.5 in 2023 brought a slight 3% rise, reflecting more stable market reactions over time.

Slide 6: Model Selection (Dexter) We chose three different models for our analysis to cover a range of predictive capabilities: 1. Logistic Regression was selected as a baseline model due to its simplicity and effectiveness for binary classification. 2. LSTM (Long Short-Term Memory), a type of recurrent neural network, was chosen for its ability to capture temporal dependencies, making it ideal for time-series data. 3. Random Forest, an ensemble learning method, was included for its robustness in handling non-linear relationships between features. We tested these models to identify which approach best predicted stock price movements.

Slide 7: Model Training and Testing (Hunter) We split the data into an 80/20 training and testing configuration for model training. We used GridSearchCV for hyperparameter tuning of the Random Forest to optimize its performance, while the LSTM model was trained for 10 epochs with a batch size of 32. These choices aimed to balance model accuracy and computational efficiency. One challenge we encountered was overfitting, particularly with the LSTM model, which required careful monitoring during training.

Slide 8: Evaluation Metrics (Hunter) To evaluate the models, we used a set of metrics that provided a comprehensive view of their performance: • Accuracy measured the overall correctness of predictions. • Precision assessed the reliability of optimistic predictions, crucial in stock market forecasting to minimize false alarms. • Recall indicated the ability of the model to capture all accurate positive movements. • F1 Score balanced precision and recall, offering a single metric to evaluate their trade-off. • ROC-AUC measured the model's ability to distinguish between price increases and decreases. These metrics helped us identify strengths and weaknesses in each model's predictive ability.

Slide 9: Model Results (Dexter) Here, we compare the results of each model: • Logistic Regression achieved an accuracy of 54%, with high recall but a low precision of 0.53, indicating it captured almost all upward movements but often misclassified stable or declining prices as increases. • Random Forest showed an accuracy of 54%, with more balanced precision and recall, but overall performance was still moderate. • LSTM, designed to capture temporal sequences, mirrored the Logistic Regression with an accuracy of 52% and middle recall but low precision. These results suggest that while our models can identify upward price trends, they struggle with false positives, which limits their practical reliability for real-world use."

Slide 10: Insights and Interpretation (Dexter) The critical insight here is that while our models had high recall (Logistic Regression: 95%, LSTM: 66%), indicating strong performance in detecting actual price increases, they struggled with precision (Logistic Regression: 53%, LSTM: 55%), resulting in many false positives. This suggests that while the models were good at identifying upward trends, they frequently misclassified days with no price increase as increases. Given the moderate accuracy (54%) and low ROC-AUC (around 0.52-0.54), the models were limited in distinguishing between price movements. To improve precision and reduce false positives, integrating additional features, such as market sentiment from news articles or macroeconomic indicators, could help provide more context to the data.

Slide 11: Challenges and Limitations (Hunter) One of the main challenges we faced was overfitting, especially with the LSTM model. Despite its capacity to learn from sequential data, it performed inconsistently when predicting unseen data. Another limitation was the relatively simple feature set, which did not include external factors like news sentiment or broader market trends. This limitation impacted the models' ability to make more precise predictions.

Slide 12: Future Work and Recommendations (Hunter) To address the challenges and limitations, future work should incorporate sentiment analysis from sources like news articles and social media to add a qualitative layer to our data. Including macroeconomic indicators such as GDP growth or interest rates could improve model precision by providing additional context. Lastly, exploring more advanced ensemble methods like XGBoost or trying out model stacking could lead to better performance.

Slide 13: Ethical Considerations (Dexter) While predictive analytics can be powerful, it is crucial to consider the ethical implications. These models could be misused for unethical market manipulation. Additionally, if we integrate external data sources like sentiment analysis, ensuring

compliance with data privacy laws and protecting user data becomes essential. Responsible use of predictive models is vital for maintaining trust and integrity in financial forecasting.

Slide 14: Conclusion (Dexter) In conclusion, our project demonstrated that while machine learning models can identify stock price trends, there is still a need for greater precision. Although our models had high recall, they often misclassified non-upward movements as increases, highlighting the challenge of false positives. To improve model reliability, future work should focus on incorporating additional data sources like market sentiment and macroeconomic indicators, as well as refining model architectures. By integrating both quantitative predictions and qualitative insights, we can develop more accurate and practical tools for investors and analysts.

Slide 15: Call to Action (Hunter) I encourage financial analysts and data scientists to use predictive models as part of a larger investment strategy. These models should be combined with qualitative analysis and expert judgment to make informed decisions. Further research into feature engineering and integrating diverse data sources is critical for enhancing model accuracy and utility.

Slide 16: Q&A (Optional) (If we have time) Thank you for your time and attention. I am now open to any questions you might have about the project, the models we used, or future improvements we are considering.