Import Libraries

```
import yfinance as yf
import pandas as pd
import matplotlib.pyplot as plt
from pandas.plotting import table
```

Upload Files

```
from google.colab import files
uploaded = files.upload()
```

Choose files No file chosen

Dataset Description

```
import yfinance as yf
import pandas as pd

# Download historical data (Apple stock as example)
ticker = 'AAPL'
df = yf.download(ticker, start='2020-01-01', end='2021-01-01')

# Save to CSV
df.to_csv('aapl_stock_data.csv')

# View the first 5 rows
print(df.head())
```

```
YF.download() has changed argument auto_adjust default to True
[*********************100%***********************]  1 of 1 completedPrice           Close       High        Low       Open     Volume
Ticker           AAPL       AAPL       AAPL       AAPL       AAPL
Date
2020-01-02  72.620834  72.681281  71.373211  71.627084  135480400
2020-01-03  71.914833  72.676462  71.689973  71.847133  146322800
2020-01-06  72.487854  72.526541  70.783256  71.034717  118387200
2020-01-07  72.146927  72.753808  71.926900  72.497514  108872000
2020-01-08  73.307503  73.609737  71.849525  71.849525  132079200
```

```
import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pandas.plotting import table
from sklearn.preprocessing import StandardScaler

# Fetch historical stock data
df = yf.download("AAPL", start="2020-01-01", end="2021-01-01")
df.reset_index(inplace=True)
```

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

Data Preprocessing

```
# Save before-cleaning screenshot
def save_table(df_sample, filename):
    fig, ax = plt.subplots(figsize=(10, 2))
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)
    ax.set_frame_on(False)
    table(ax, df_sample, loc='center', colWidths=[0.12]*len(df_sample.columns))
    plt.savefig(filename, bbox_inches='tight')
```

```
    plt.close()

save_table(df.head(), "before_cleaning.png")



# Check for missing values
print("Missing values before:\n", df.isnull().sum())

# Fill forward or drop
df.fillna(method='ffill', inplace=True)

# Remove duplicates
df.drop_duplicates(inplace=True)

print("Missing values after:\n", df.isnull().sum())
```

```
Missing values before:
 Price   Ticker
Date             0
Close   AAPL     0
High    AAPL     0
Low     AAPL     0
Open    AAPL     0
Volume  AAPL     0
dtype: int64
Missing values after:
 Price   Ticker
Date             0
Close   AAPL     0
High    AAPL     0
Low     AAPL     0
Open    AAPL     0
Volume  AAPL     0
dtype: int64
<ipython-input-6-078a32f1e5e5>:5: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a future version. Use ob
  df.fillna(method='ffill', inplace=True)
```

```
from scipy.stats import zscore
import numpy as np # Import numpy as it's used
import pandas as pd # Ensure pandas is imported if not already

# We'll apply Z-score only to price columns
price_cols = ['Open', 'High', 'Low', 'Close', 'Adj Close']

# Print the columns before attempting to access them
print("Columns in DataFrame:", df.columns)

# Filter price_cols to only include columns present in the DataFrame
existing_price_cols = [col for col in price_cols if col in df.columns]

# Check if there are any columns to process after filtering
if existing_price_cols:
    # Compute Z-scores using only the existing price columns
    z_scores = np.abs(zscore(df[existing_price_cols]))

    # Apply the Z-score filter based on the existing price columns
    # We need to create a boolean mask that is True for rows where
    # all Z-scores across the selected columns are less than 3.
    # Since z_scores is already computed on a subset of columns,
    # applying .all(axis=1) to it directly will give the correct mask.
    df = df[(z_scores < 3).all(axis=1)]
else:
    print("Warning: None of the specified price columns exist in the DataFrame.")
    # Decide how to handle this case: either skip outlier removal or raise an error

# The rest of your code for scaling can remain the same,
# but you might want to apply a similar check or adjust which columns are scaled
# based on availability.
# For example, for scaling:
# all_cols_to_scale = price_cols + ['Volume']
# existing_cols_to_scale = [col for col in all_cols_to_scale if col in df.columns]
# if existing_cols_to_scale:
#     df_scaled[existing_cols_to_scale] = scaler.fit_transform(df[existing_cols_to_scale])
```

```
Columns in DataFrame: MultiIndex([(  'Date',     ''),
            ( 'Close', 'AAPL'),
```

```
            (  'High', 'AAPL'),
            (   'Low', 'AAPL'),
            (  'Open', 'AAPL'),
            ('Volume', 'AAPL')],
           names=['Price', 'Ticker'])
```

```
scaler = StandardScaler()
df_scaled = df.copy()

# Define the columns we want to scale
cols_to_potentially_scale = price_cols + ['Volume']

# Filter to keep only the columns that are actually present in the DataFrame
existing_cols_to_scale = [col for col in cols_to_potentially_scale if col in df_scaled.columns]

# Check if there are any columns left to scale after filtering
if existing_cols_to_scale:
    # Scale only the existing numeric columns
    df_scaled[existing_cols_to_scale] = scaler.fit_transform(df[existing_cols_to_scale])
else:
    print("Warning: None of the specified columns to scale exist in the DataFrame.")
# The rest of your code continues here.



save_table(df_scaled.head(), "after_cleaning.png")
```

EDA

1. Import Libraries & Load Data

```
import yfinance as yf
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Set styles
sns.set(style="whitegrid")
plt.rcParams["figure.figsize"] = (12, 6)

# Load data
df = yf.download("AAPL", start="2020-01-01", end="2021-01-01")
df.reset_index(inplace=True)
df.head()
```

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

Price	Date	Close	High	Low	Open	Volume
Ticker		AAPL	AAPL	AAPL	AAPL	AAPL
0	2020-01-02	72.620834	72.681281	71.373211	71.627084	135480400
1	2020-01-03	71.914833	72.676462	71.689973	71.847133	146322800
2	2020-01-06	72.487854	72.526541	70.783256	71.034717	118387200
3	2020-01-07	72.146927	72.753808	71.926900	72.497514	108872000
4	2020-01-08	73.307503	73.609737	71.849525	71.849525	132079200

Next steps: Generate code with df | View recommended plots | New interactive sheet

2. Plot Trends (Line Chart of Closing Price)

```
plt.plot(df['Date'], df['Close'], label='Close Price')
plt.title("Apple (AAPL) Closing Price Over Time")
plt.xlabel("Date")
plt.ylabel("Close Price (USD)")
plt.legend()
plt.tight_layout()
```

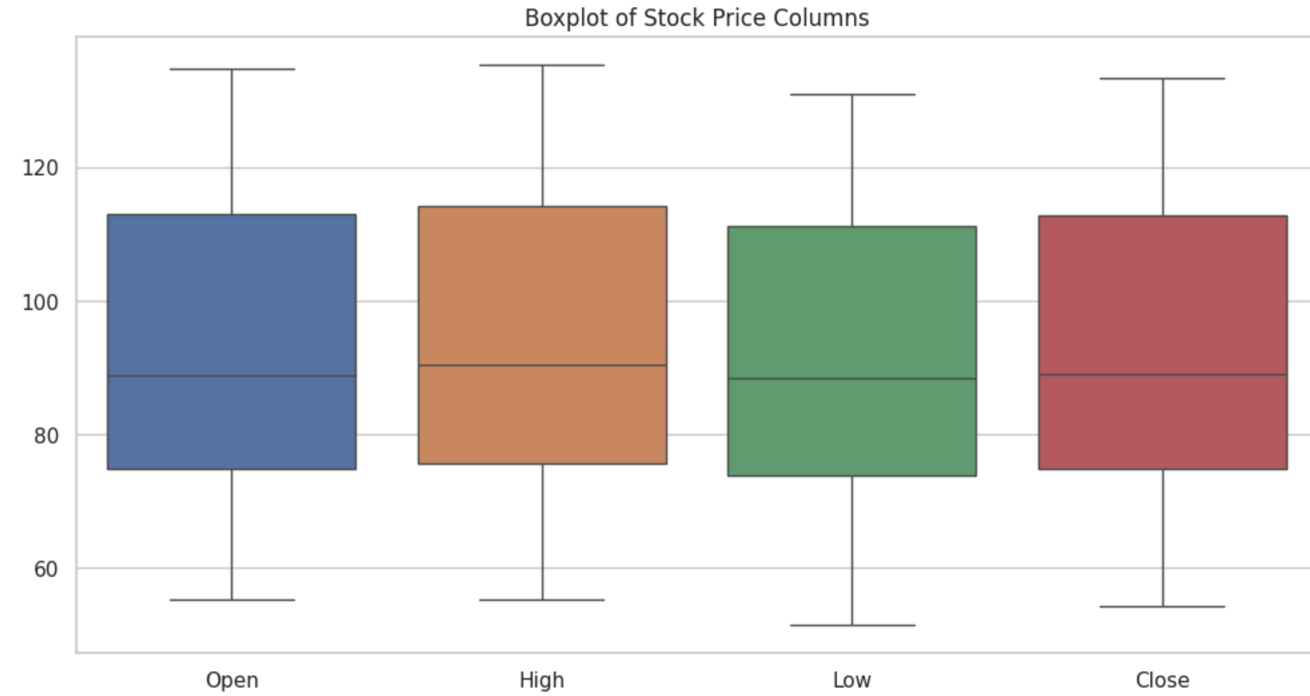
```
plt.show()
```

3.Histogram

```
df[['Open', 'High', 'Low', 'Close']].hist(bins=30, figsize=(12, 8))
plt.suptitle("Price Distributions")
plt.tight_layout()
plt.show()
```

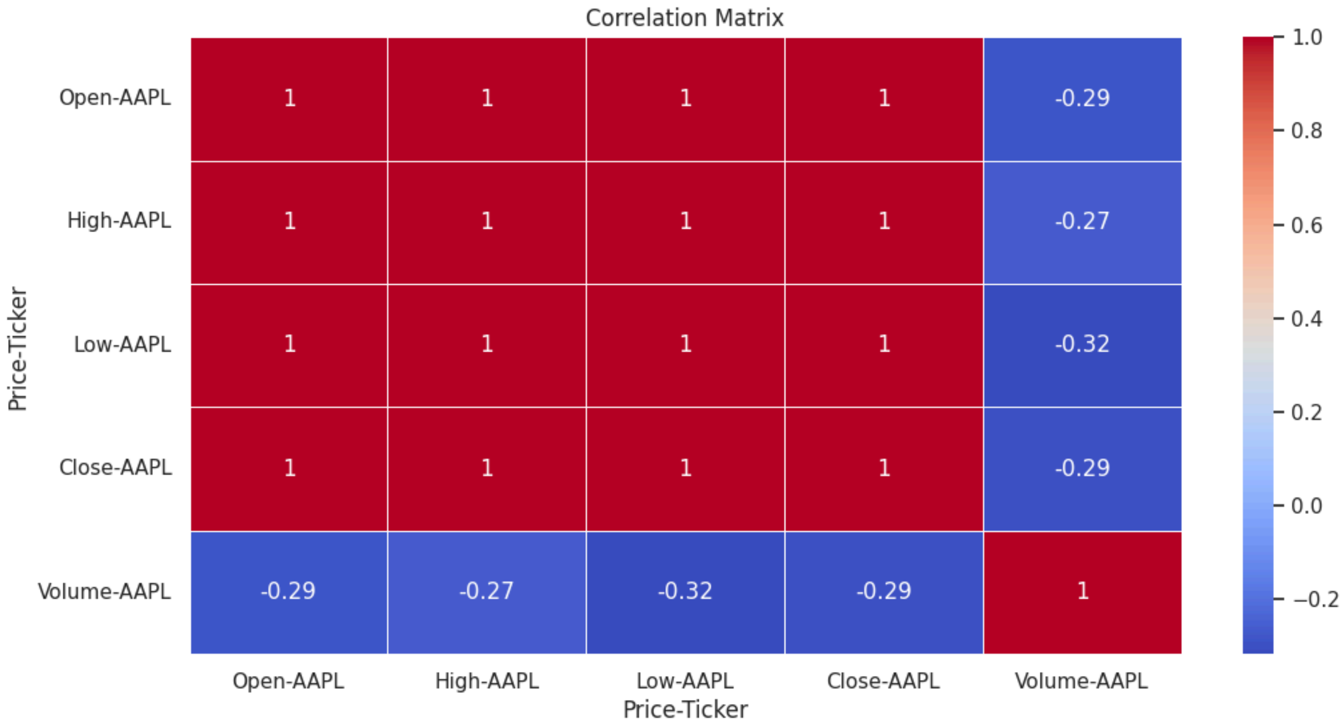
4. Boxplots to Detect Outliers

```
sns.boxplot(data=df[['Open', 'High', 'Low', 'Close']])
plt.title("Boxplot of Stock Price Columns")
plt.show()
```



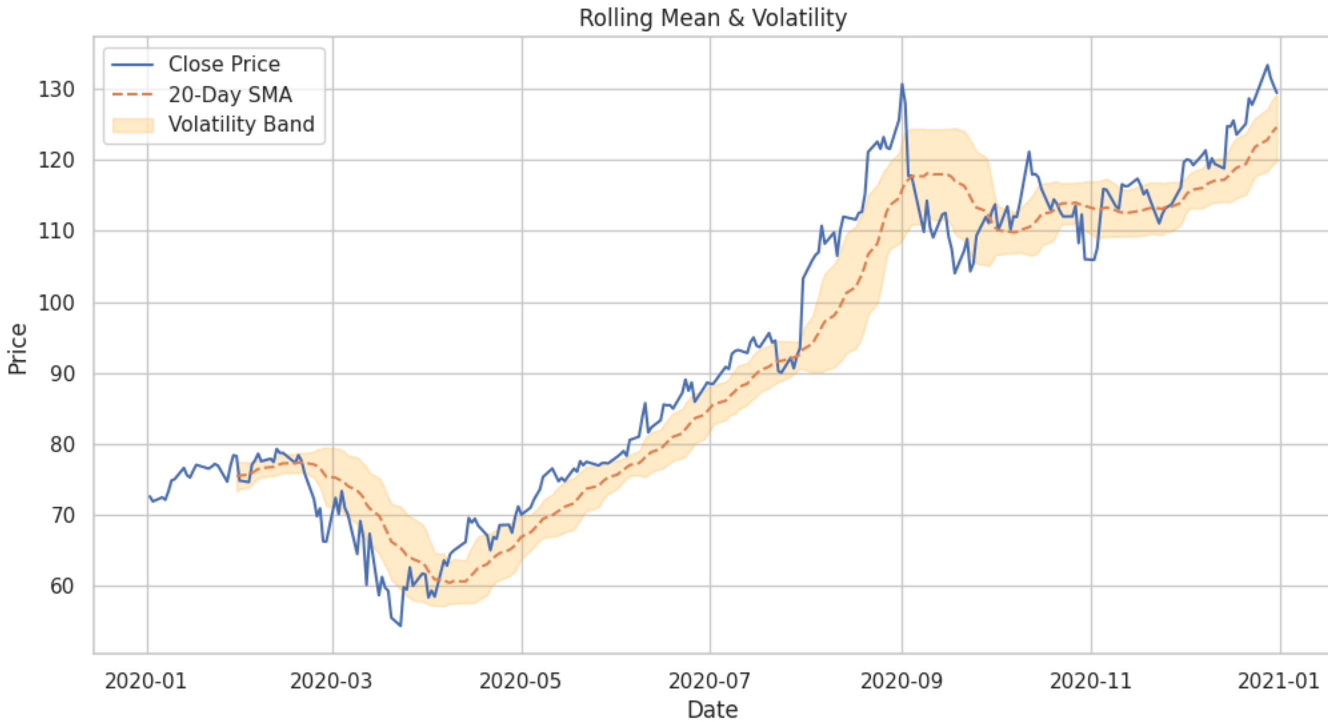
5. Correlation Heatmap

```
# Ensure the required libraries are imported if they are not already in this cell
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt # Ensure plt is imported for show()

# Load data (This block is repeated in the user's notebook, ensure it's the correct df)
# This line might not be needed if the df is already loaded from the previous cell.
# df = yf.download("AAPL", start="2020-01-01", end="2021-01-01")
# df.reset_index(inplace=True)

# Define the list of columns you intend to use for correlation
intended_corr_cols = ['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']

# Filter the list to keep only columns that exist in the DataFrame
existing_corr_cols = [col for col in intended_corr_cols if col in df.columns]

# Check if there are enough columns to compute a correlation matrix
if len(existing_corr_cols) >= 2:
    # Compute the correlation matrix using only existing columns
    corr = df[existing_corr_cols].corr()

    # Plot the heatmap
    sns.heatmap(corr, annot=True, cmap='coolwarm', linewidths=0.5)
    plt.title("Correlation Matrix")
    plt.show()
else:
    print(f"Warning: Not enough columns ({existing_corr_cols}) found to compute correlation matrix.")
```



6. Rolling Mean (Trend + Volatility)

```
df['Rolling Mean (20d)'] = df['Close'].rolling(window=20).mean()
df['Rolling Std (20d)'] = df['Close'].rolling(window=20).std()

plt.plot(df['Date'], df['Close'], label='Close Price')
plt.plot(df['Date'], df['Rolling Mean (20d)'], label='20-Day SMA', linestyle='--')
plt.fill_between(df['Date'],
                 df['Rolling Mean (20d)'] - df['Rolling Std (20d)'],
                 df['Rolling Mean (20d)'] + df['Rolling Std (20d)'],
                 color='orange', alpha=0.2, label='Volatility Band')
plt.title("Rolling Mean & Volatility")
plt.xlabel("Date")
plt.ylabel("Price")
plt.legend()
plt.show()
```



Model Building

1 Linear Regression (Baseline)

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import numpy as np # Import numpy to use np.sqrt
import pandas as pd # Ensure pandas is imported for DataFrame operations

# Check the DataFrame size before any modifications in this cell
print(f"Shape of df before creating target: {df.shape}")


# Example: use past 1 day's features to predict next day's Close
# Ensure the original df has data before proceeding
if df.empty:
    print("Error: DataFrame is empty before creating the target column.")
    # Re-download or load data here if necessary, or investigate previous steps
    # For example:
    # df = yf.download("AAPL", start="2020-01-01", end="2021-01-01")
    # df.reset_index(inplace=True)
    # print(f"Shape of df after attempting reload: {df.shape}")
    # if df.empty:
    #     raise ValueError("DataFrame is still empty after attempting to reload.")
else:
    df['Target'] = df['Close'].shift(-1)
    print(f"Shape of df after creating target: {df.shape}")

    df.dropna(inplace=True)
    print(f"Shape of df after dropna: {df.shape}")

    # Check if df is empty after dropna
    if df.empty:
        print("Error: DataFrame is empty after dropping NaN values. Cannot proceed with splitting.")
        # You might need to adjust your data processing or target creation logic
    else:
        X = df[['Open', 'High', 'Low', 'Close', 'Volume']]
        y = df['Target']

        # Check the size of X and y before splitting
```

```
        print(f"Shape of X before train_test_split: {X.shape}")
        print(f"Shape of y before train_test_split: {y.shape}")


        X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=False, test_size=0.2)

        lr = LinearRegression()
        lr.fit(X_train, y_train)
        preds = lr.predict(X_test)

        # Calculate Mean Squared Error
        mse = mean_squared_error(y_test, preds)

        # Calculate Root Mean Squared Error manually
        rmse = np.sqrt(mse)

        print("Linear Regression RMSE:", rmse)
```

```
Shape of df before creating target: (0, 9)
Error: DataFrame is empty before creating the target column.
```

Random Forest

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
import numpy as np # Import numpy to use np.sqrt

rf = RandomForestRegressor(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)
rf_preds = rf.predict(X_test)

# Calculate Mean Squared Error
rf_mse = mean_squared_error(y_test, rf_preds)

# Calculate Root Mean Squared Error manually
rf_rmse = np.sqrt(rf_mse)

print("Random Forest RMSE:", rf_rmse)
```

```
Random Forest RMSE: 3.441925670771634
```

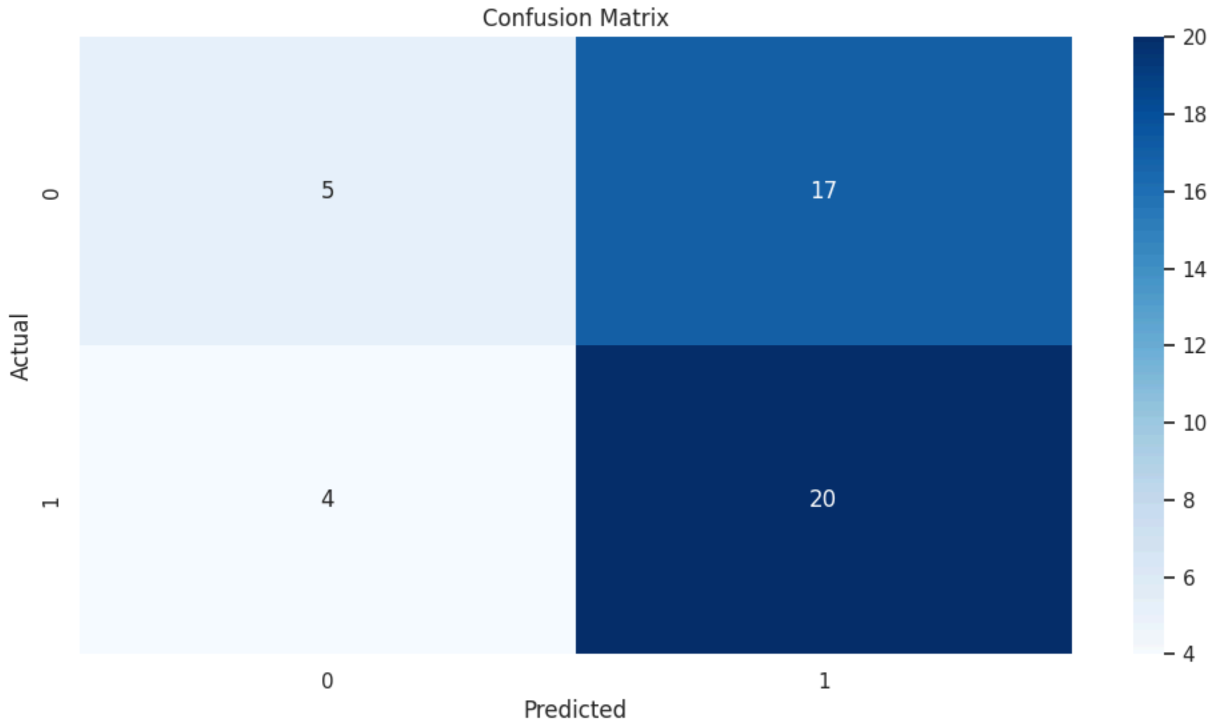
Model Evaluation

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np

# Example: assume y_test and y_pred are available
def evaluate_regression(y_test, y_pred, model_name="Model"):
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    mae = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    print(f"📊 {model_name} Evaluation:")
    print(f"  RMSE: {rmse:.4f}")
    print(f"  MAE : {mae:.4f}")
    print(f"  R²   : {r2:.4f}")
    return {"Model": model_name, "RMSE": rmse, "MAE": mae, "R2": r2}
```

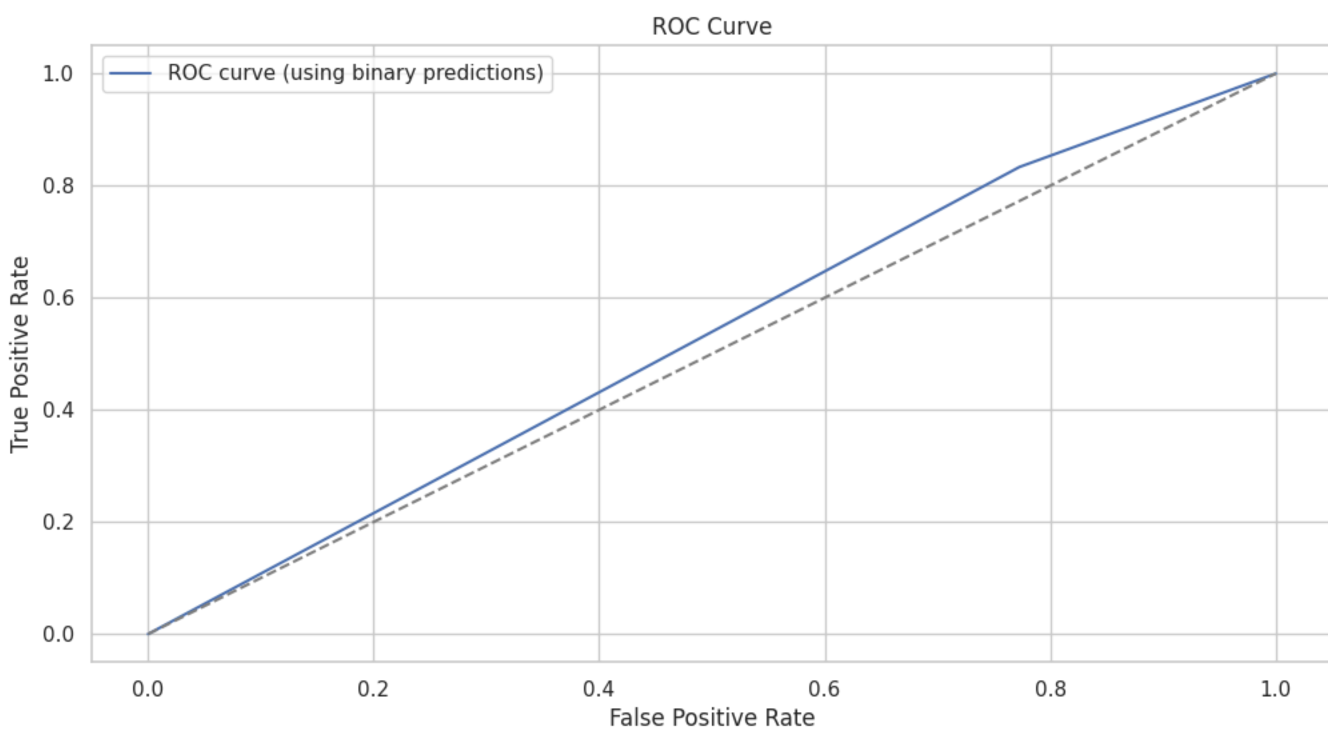
```
import numpy as np

# Create binary actual and predicted labels by comparing the values
# Compares the value at index i with the value at index i-1
# The resulting array will be one element shorter than y_test
y_actual_bin = (y_test.values[1:] > y_test.values[:-1]).astype(int)

# Note: y_pred is not defined in the traceback's global variables,
# so the next line for y_pred_bin will also fail if y_pred is not available.
# Assuming you have predicted values (e.g., from your regression model's predictions),
# you should compare those predicted values to the actual values from the *previous* day.
```

```
    # This requires careful alignment.
    # A common approach is to compare the predicted price change (pred_today - actual_yesterday)
    # with zero, or compare the predicted price (pred_today) with the actual price from yesterday (actual_yesterday).

    # Based on the original code's structure (comparing y_pred to y_test[:-1]),
    # it seems the intention was to compare the predicted value for a day (y_pred[i])
    # with the actual value from the previous day (y_test[i-1]). This requires careful indexing.

    # Let's align y_test and the predictions to make this comparison.
    # Assuming `preds` from the Linear Regression or `rf_preds` from Random Forest
    # are the predictions corresponding to `X_test`.
    # The predictions `preds` or `rf_preds` predict `y_test`.

    # To get the predicted direction, you compare the prediction for day `i` (preds[i])
    # with the actual closing price from day `i-1` (y_test.values[i-1]).
    # This comparison is valid for indices 1 onwards in the y_test/preds arrays.

    # Let's assume you want to use `preds` from the Linear Regression model for this evaluation
    # Make sure 'preds' variable is available from the previous cell or define it appropriately
    # If you used Random Forest predictions, replace 'preds' with 'rf_preds'

    # y_pred is not defined in the provided global variables or code snippet.
    # Assuming 'preds' from the Linear Regression cell was intended to be used here:
    # Make sure you run the Linear Regression cell before this one.

    # This next line will fail if 'preds' is not defined.
    # You need to decide which prediction array to use (e.g., `preds` or `rf_preds`).
    # Let's use `preds` as an example, assuming it's defined.
    # If `preds` is a numpy array, we can slice it directly.
    # Ensure `preds` and `y_test` are aligned in terms of time steps.
    # `preds` predicts `y_test`. So `preds[i]` is the prediction for the value `y_test[i]`.
    # The actual change is `y_test[i] - y_test[i-1]`.
    # A common way to evaluate predicted direction is comparing `preds[i]` vs `y_test[i-1]`.

    # First, ensure `preds` is available. If not, run the regression cell.
    # If `preds` is a numpy array:
    if 'preds' in locals(): # Check if preds is defined
        # Compare the predicted price for day i (preds[:-1]) with the actual price for day i-1 (y_test.values[:-1])
        # Note the slicing: preds has the same length as y_test.
        # preds[i] is prediction for y_test[i] (target for X_test[i-1])
        # Comparing preds[1:] (prediction for y_test[1:] which are targets for X_test[0:])
        # with y_test.values[:-1] (actual values that were features for X_test[1:])
        # This alignment needs careful consideration based on how your target was created.
        # If y_test[i] is the target for features at time i-1, then preds[i] is the prediction for y_test[i].
        # The actual change is y_test[i] vs y_test[i-1].
        # The predicted change based on the regression model could be compared to this.
        # One way is to see if preds[i] > y_test[i-1].

        # The original line `y_pred_bin = (y_pred[1:] > y_test[:-1]).astype(int)`
        # implies comparing predicted value at index i+1 with actual value at index i.
        # Let's align the indices correctly for comparison.
        # y_test.values[1:] are actual values from index 1 onwards.
        # y_test.values[:-1] are actual values up to index len-2.
        # preds are predictions for y_test. So preds[i] is prediction for y_test[i].

        # To get the direction of movement predicted by the model:
        # Compare the predicted value for day i (preds[i]) with the actual value from the previous day (y_test.values[i-1]).
        # This comparison is valid for i >= 1.
        # So we compare preds[1:] with y_test.values[:-1].
        # Ensure 'preds' has been generated by running the Linear Regression cell.
        if len(preds) == len(y_test):
             y_pred_bin = (preds[1:] > y_test.values[:-1]).astype(int)
        else:
             print(f"Error: Length of predictions ({len(preds)}) does not match length of y_test ({len(y_test)}). Cannot compute y_pred_bin.")
             # Set y_pred_bin to None or handle the error appropriately
             y_pred_bin = None

    else:
        print("Error: 'preds' variable is not defined. Please run the Linear Regression model cell first.")
        y_pred_bin = None # Set y_pred_bin to None if preds is not available

    # Now you can proceed with classification metrics if y_actual_bin and y_pred_bin are defined and not None.

    # Ensure y_actual_bin and y_pred_bin are defined and contain your classification results
    # Check the lengths match before computing metrics
    if 'y_actual_bin' in locals() and y_actual_bin is not None and \
       'y_pred_bin' in locals() and y_pred_bin is not None:
```

```
    if len(y_actual_bin) == len(y_pred_bin):
        from sklearn.metrics import confusion_matrix, roc_curve, classification_report
        import seaborn as sns
        import matplotlib.pyplot as plt

        # Confusion matrix
        cm = confusion_matrix(y_actual_bin, y_pred_bin)
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
        plt.title("Confusion Matrix")
        plt.xlabel("Predicted")
        plt.ylabel("Actual")
        plt.show()

        # Classification Report (provides precision, recall, f1-score)
        print("\nClassification Report:")
        print(classification_report(y_actual_bin, y_pred_bin))


        # ROC Curve requires predicted probabilities, not just binary predictions.
        # Regression models don't typically provide probabilities for classification.
        # If you trained a classifier (e.g., Logistic Regression) for price direction,
        # use its `predict_proba` method here.
        # Since the original code used binary predictions for ROC, it will produce a step function.
        # It's generally better to use a classifier's probabilities for a meaningful ROC curve.

        # If you have a classification model (`classifier`) trained on `X_test_bin`,
        # and it has a `predict_proba` method, uncomment and use the following:
        # try:
        #     y_pred_prob = classifier.predict_proba(X_test_bin)[:, 1] # Probability of the positive class
        #     fpr, tpr, _ = roc_curve(y_actual_bin, y_pred_prob)
        #     plt.plot(fpr, tpr, label='ROC curve (using probabilities)')
        #     plt.plot([0, 1], [0, 1], '--', color='gray')
        #     plt.xlabel('False Positive Rate')
        #     plt.ylabel('True Positive Rate')
        #     plt.title('ROC Curve')
        #     plt.legend()
        #     plt.show()
        # except Exception as e:
        #      print(f"Could not plot ROC curve using probabilities. Error: {e}")
        #      print("Ensure you have a classification model trained and 'predict_proba' is available.")


        # Plotting ROC curve with binary predictions as done in the original code
        # This will likely result in a simple step function
        fpr, tpr, _ = roc_curve(y_actual_bin, y_pred_bin) # Using binary predictions
        plt.plot(fpr, tpr, label='ROC curve (using binary predictions)')
        plt.plot([0, 1], [0, 1], '--', color='gray')
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('ROC Curve')
        plt.legend()
        plt.show()

    else:
        print(f"Error: Length mismatch between y_actual_bin ({len(y_actual_bin)}) and y_pred_bin ({len(y_pred_bin)}). Cannot compute metrics

else:
    print("Error: y_actual_bin or y_pred_bin are not defined or are None. Cannot compute classification metrics.")
```




```
Classification Report:
              precision    recall  f1-score   support

           0       0.56      0.23      0.32        22
           1       0.54      0.83      0.66        24

    accuracy                           0.54        46
   macro avg       0.55      0.53      0.49        46
weighted avg       0.55      0.54      0.50        46
```




```
import numpy as np

# Create binary actual and predicted labels
# Compare the values as numpy arrays to avoid index alignment issues
y_actual_bin = (y_test.values[1:] > y_test.values[:-1]).astype(int)
```

```
# Assuming 'preds' is the numpy array of predictions from your regression model
# Compare the predicted value for day i (preds[i]) with the actual value from the previous day (y_test.values[i-1])
# This comparison is valid for i >= 1.
# So we compare preds[1:] with y_test.values[:-1].
# Ensure 'preds' has been generated by running the Linear Regression cell.
# Make sure the lengths are compatible for this comparison
# If preds and y_test have the same length, you compare preds[1:] (indices 1 to end)
# with y_test.values[:-1] (indices 0 to end-1)
if 'preds' in locals() and len(preds) == len(y_test):
    y_pred_bin = (preds[1:] > y_test.values[:-1]).astype(int)
elif 'rf_preds' in locals() and len(rf_preds) == len(y_test):
     # If using Random Forest predictions
     y_pred_bin = (rf_preds[1:] > y_test.values[:-1]).astype(int)
else:
    print("Warning: Could not find 'preds' or 'rf_preds' with compatible length for direction calculation.")
    y_pred_bin = None # Set y_pred_bin to None if predictions are not available

# The subsequent cells that use y_actual_bin and y_pred_bin will need to handle
# the case where y_pred_bin might be None if predictions weren't generated correctly.
```

Classification Metrics

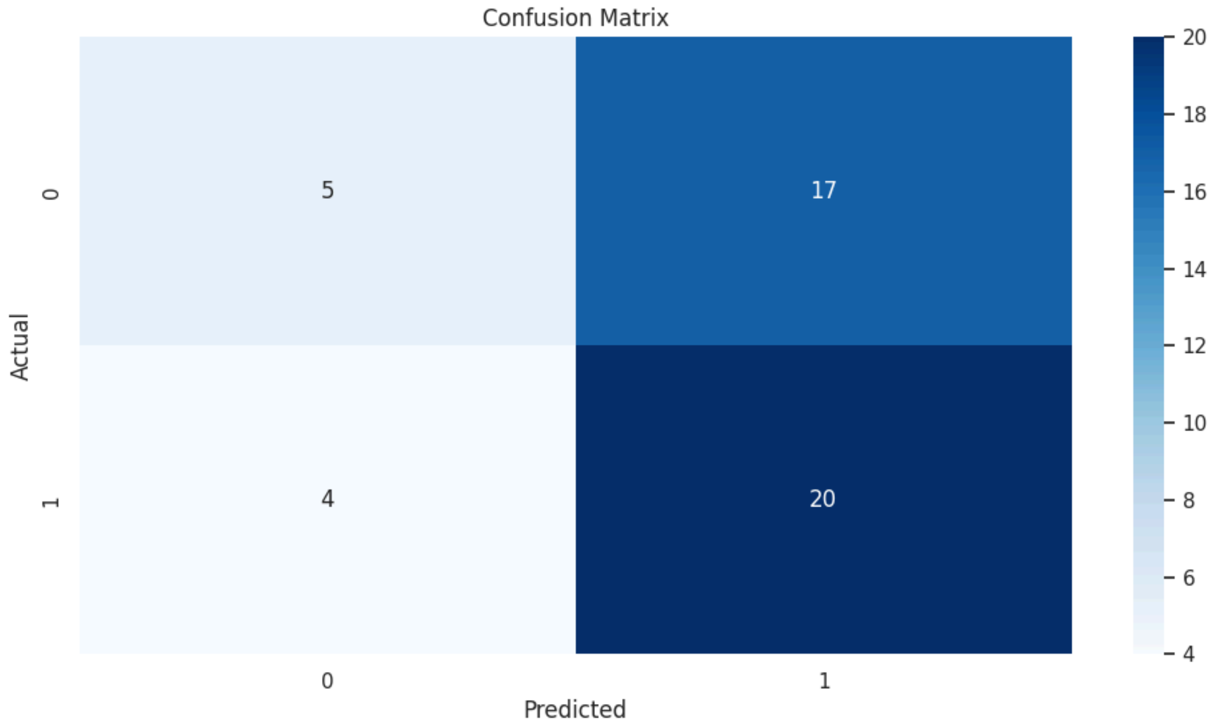
```
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix, roc_auc_score, roc_curve
import matplotlib.pyplot as plt
import seaborn as sns

# Print scores
print("🔍 Classification-style Evaluation:")
print("Accuracy:", accuracy_score(y_actual_bin, y_pred_bin))
print("F1 Score:", f1_score(y_actual_bin, y_pred_bin))
print("ROC AUC :", roc_auc_score(y_actual_bin, y_pred_bin))
```

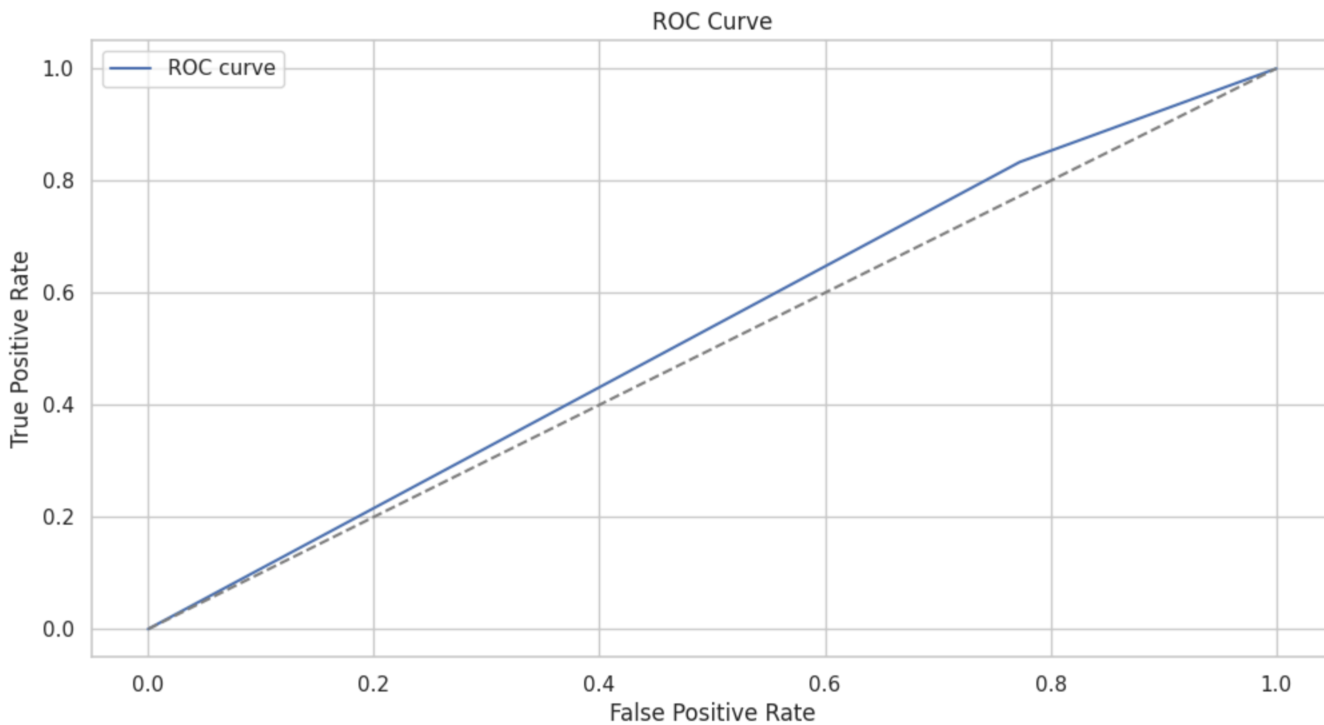
```
🔍 Classification-style Evaluation:
Accuracy: 0.5434782608695652
F1 Score: 0.6557377049180327
ROC AUC : 0.5303030303030303
```

Confusion Matrix + ROC Curve

```
# Import the necessary evaluation metrics for classification
from sklearn.metrics import confusion_matrix, roc_curve

# --- Add your classification model training and prediction code here ---
# Example: You would need to define X_train_bin, y_train_bin, X_test_bin, y_test_bin
# based on a classification task (e.g., predicting price movement direction).

# Example Placeholder:
# from sklearn.linear_model import LogisticRegression
# from sklearn.model_selection import train_test_split
# Assume y_actual is your true binary labels and y_pred_proba is your predicted probabilities
# For demonstration, let's create dummy binary data if you haven't trained a classifier yet
# In a real scenario, replace this with actual classification results.
# This part needs to be tailored to your classification task.
# For example, if you want to classify if the price goes up (1) or down (0) the next day:
# df['Price_Direction'] = (df['Close'].shift(-1) > df['Close']).astype(int)
# df_clf = df.dropna() # Drop the last row
# X_clf = df_clf[['Open', 'High', 'Low', 'Close', 'Volume']]
# y_clf = df_clf['Price_Direction']
# X_train_bin, X_test_bin, y_actual_bin, y_test_bin = train_test_split(X_clf, y_clf, shuffle=False, test_size=0.2)
# classifier = LogisticRegression() # Or any other classifier
# classifier.fit(X_train_bin, y_test_bin)
# y_pred_bin = classifier.predict(X_test_bin) # Or convert probabilities to binary predictions

# *** Replace the following dummy data generation with your actual classification results ***
# Assuming you have actual binary labels in a variable named y_actual_bin
# and predicted binary labels in a variable named y_pred_bin from your classification model
# Since the traceback shows y_actual_bin and y_pred_bin are not defined,
# you MUST add code above this point to train a classification model and obtain these variables.
# For now, let's use placeholder variables that you must replace:
# print("Warning: y_actual_bin and y_pred_bin are placeholders. Please replace them with actual classification results.")
# import numpy as np
```

```
# y_actual_bin = np.random.randint(0, 2, size=len(y_test)) # Replace with your actual binary labels
# y_pred_bin = np.random.randint(0, 2, size=len(y_test)) # Replace with your actual binary predictions
# *** End of placeholder generation ***


# --- Assuming y_actual_bin and y_pred_bin are now correctly defined from your classification model ---


# Confusion matrix
# Ensure y_actual_bin and y_pred_bin are defined and contain your classification results
if 'y_actual_bin' in locals() and 'y_pred_bin' in locals():
    cm = confusion_matrix(y_actual_bin, y_pred_bin)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title("Confusion Matrix")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()

    # ROC Curve
    # For ROC curve, you typically need predicted probabilities, not just binary predictions.
    # If your classifier provides predict_proba, use that. Otherwise, you might need to adjust.
    # Assuming y_pred_prob is available from your classifier's predict_proba method:
    # fpr, tpr, _ = roc_curve(y_actual_bin, y_pred_prob[:, 1]) # Use probability of the positive class
    # If you only have y_pred_bin, the ROC curve might not be meaningful or possible directly.
    # For this example, we'll use the binary prediction, which might result in a simple step function for the ROC.
    fpr, tpr, _ = roc_curve(y_actual_bin, y_pred_bin) # Using binary predictions
    plt.plot(fpr, tpr, label='ROC curve')
    plt.plot([0, 1], [0, 1], '--', color='gray')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.legend()
    plt.show()
else:
    print("Error: y_actual_bin and y_pred_bin are not defined. Cannot compute classification metrics.")
```





Deployment

```
pip install yfinance gradio pandas numpy matplotlib scikit-learn tensorflow
```