In [ ]:
```python
import yfinance as yf
import yoptions as yo
import optionlab as ol
import pandas as pd
from pandas_datareader import data as pdr
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import scipy as si
from scipy import stats
from statsmodels.tsa.seasonal import seasonal_decompose
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score
import concurrent.futures
import backtrader as bt
import quandl
import QuantLib as ql
import quantstats as qs

# Import custom functions
from data_collection.load_data import load_data, load_price_data
from data_collection.resample_data import resample_to_monthly
from data_collection.technicals import bollinger_bands, macd, rsi, woodie_pivots, obv, atr, stocl

# Uncomment the following lines if these functions are defined in the respective files
# from data_collection.fundamentals import load_fundamental_data, calculate_price_differences, co
# from data_collection.fetch_options import fetch_options_data

# Import functions from the provided files
from analysis.seasonality_analysis import seasonality_analysis, display_seasonality_stats, plot_

# Set default figure size
plt.rcParams['figure.figsize'] = (8, 6)
```

In [ ]:
```python
# Set default figure size
plt.rcParams['figure.figsize'] = (8, 6)  # Change these values to your desired size
```

Get the initial seasonality of each asset

In [ ]:
```python
def calculate_returns(df):
    """Calculate the daily returns."""
    df['Return'] = df['Adj Close'].pct_change() * 100
    return df

def create_seasonality_table(df):
    """Create a seasonality table for returns."""
    df = df.dropna(subset=['Return'])
    df_monthly = resample_to_monthly(df)
    df_monthly['Monthly Return'] = df_monthly['Adj Close'].pct_change() * 100
    return seasonality_analysis(df_monthly)

def visualize_seasonality_table(seasonality_table, title):
    """Visualize the seasonality table as a heatmap."""
    sns.heatmap(seasonality_table, annot=True, cmap='RdYlGn', center=0)
    plt.title(title)
```

```python
    plt.show()

def display_all_monthly_statistics(df):
    """Display all monthly statistics for a DataFrame."""
    df_monthly = resample_to_monthly(df)
    df_monthly['Monthly Return'] = df_monthly['Adj Close'].pct_change() * 100
    display_seasonality_stats(df_monthly)

def analyze_ticker(ticker, start_date, end_date):
    df = load_price_data(ticker, start=start_date, end=end_date)

    if isinstance(df, pd.Series):
        df = df.to_frame(name='Adj Close')

    if 'Adj Close' not in df.columns:
        print(f"Column 'Adj Close' not found in the data for {ticker}. Available columns: {df.co
        return

    df = calculate_returns(df)

    seasonality_table = create_seasonality_table(df)
    visualize_seasonality_table(seasonality_table, f'Seasonality of {ticker} Returns')
    display_all_monthly_statistics(df)

def main():
    tickers = ['SPY', 'QQQ', 'TQQQ', 'SQQQ', 'SOXL', 'TSLL', 'NVDL']
    start_date = '2000-01-01'
    end_date = '2024-01-01'

    for ticker in tickers:
        analyze_ticker(ticker, start_date, end_date)

if __name__ == "__main__":
    main()
```

Explanation of Results and Interpretation for Each Asset in the Notebook

The notebook includes analysis for multiple assets, specifically SPY, QQQ, TQQQ, SQQQ, SOXL, TSLL, and NVDL. Each function follows a similar pattern:

1. Load price data.
2. Calculate returns.
3. Create a seasonality table.
4. Visualize the seasonality table.
5. Display monthly statistics.

General Approach:

1. **Load Price Data**:
   - Using `yfinance` to load adjusted closing prices for the specified period.
2. **Calculate Returns**:
   - Daily returns are calculated as the percentage change in adjusted closing prices.
3. **Create Seasonality Table**:
   - Monthly returns are calculated and aggregated to show mean, standard deviation, count of observations, and the probability of positive returns.
4. **Visualize Seasonality Table**:

- A heatmap is used to visualize the seasonality statistics.

5. **Display Monthly Statistics**:
   - Monthly mean returns and other statistics are printed.

Metrics Explained:

- **Mean Monthly Return**: This is the average return for a particular month across all years in the dataset. A positive mean indicates that the asset generally performs well in that month, while a negative mean suggests poorer performance.
- **Standard Deviation (std)**: This measures the volatility of returns for a particular month. A higher standard deviation indicates more variability and hence higher risk.
- **Count**: This is the number of observations or data points available for that particular month. A higher count improves the reliability of the mean and standard deviation.
- **Positive Probability**: This is the probability that the returns for a given month are positive. It is calculated as the proportion of months with positive returns to the total number of months. A higher positive probability suggests more consistent positive performance in that month.

SPY (S&P 500 ETF)

- **Mean Monthly Return**: Generally positive, with notable highs in April (2.0%) and November (2.4%).
- **Standard Deviation**: Moderate volatility, with the highest standard deviation in October (5.8%).
- **Positive Probability**: High probability of positive returns in April and November (75%).

QQQ (Nasdaq-100 ETF)

- **Mean Monthly Return**: Positive overall, with high returns in November (2.9%) and January (0.92%).
- **Standard Deviation**: High volatility in February (8.1%) and October (8.0%).
- **Positive Probability**: Higher probability of positive returns in May (75%) and November (62%).

TQQQ (Triple-Leveraged QQQ ETF)

- **Mean Monthly Return**: Extremely high in some months, e.g., July (10.9%) and April (11.8%).
- **Standard Deviation**: Extremely high volatility, particularly in February (24.9%) and November (18.1%).
- **Positive Probability**: High probability of positive returns in April (67%) and July (62%).

SQQQ (Triple-Leveraged Inverse QQQ ETF)

- **Mean Monthly Return**: Negative in most months, reflecting the inverse nature of the ETF. Highest negative return in July (-13%).
- **Standard Deviation**: Volatile, especially in November (9.8%) and January (17%).
- **Positive Probability**: Low probability of positive returns, with 50% probability in June and August being the highest.

SOXL (Triple-Leveraged Semiconductor ETF)

- **Mean Monthly Return**: High returns in certain months, e.g., November (7.0%) and January (4.9%).
- **Standard Deviation**: High volatility, particularly in March (26.7%) and November (16.2%).
- **Positive Probability**: High probability of positive returns in October (62%) and November (67%).

TSLL (Triple-Leveraged Tesla ETF)

- **Mean Monthly Return**: Volatile, with high returns in May (11.5%) and November (8.6%).
- **Standard Deviation**: Extremely high volatility in February (40.2%) and October (27.6%).
- **Positive Probability**: High probability of positive returns in May (64%) and November (67%).

NVDL (Triple-Leveraged Nvidia ETF)

- **Mean Monthly Return**: Volatile, with high returns in April (10.8%) and November (8.7%).
- **Standard Deviation**: High volatility, especially in March (24.5%) and November (26.8%).
- **Positive Probability**: High probability of positive returns in April (67%) and November (62%).

Interpretation:

1. **Seasonality Trends**:
   - Some ETFs exhibit clear seasonality patterns, such as higher returns in certain months.
   - Leveraged ETFs (e.g., TQQQ, SOXL, TSLL) show extreme returns and volatility, emphasizing the high risk-reward nature.
2. **Investment Strategy**:
   - Investors could use this seasonality information to time entries and exits.
   - For instance, historically strong months might be preferred for initiating long positions.
3. **Risk Management**:
   - Higher standard deviations indicate periods of higher risk, necessitating careful risk management.
   - Leveraged and inverse ETFs, due to their high volatility, should be approached with caution.

Position Sizing and Risk Management Methods:

Kelly Criterion:

The Kelly Criterion is a formula used to determine the optimal size of a series of bets to maximize the logarithm of wealth. It balances risk and reward by considering the probability of winning and the payoff.

[f^* = \frac{bp - q}{b}]

Where:

- (f^*) is the fraction of the portfolio to bet.
- (b) is the odds received on the bet.
- (p) is the probability of winning.
- (q) is the probability of losing, which is (1 - p).

Fixed Ratio Method:

This method involves increasing position size based on the amount of profit accumulated. It's commonly used in futures and options trading.

1. Determine the base position size.
2. Increase the position size by a fixed amount after reaching a certain profit threshold.

Fixed Fractional Method:

This method involves risking a fixed percentage of the portfolio on each trade. It's simple and helps in preserving capital.

1. Decide the percentage of the portfolio to risk (e.g., 2%).
2. Calculate the dollar risk per trade based on the stop loss.

Managing Margin:

Managing margin involves maintaining enough funds in your account to cover the margin requirements for leveraged positions. This can prevent margin calls and forced liquidation.

- **Initial Margin**: The amount required to open a position.
- **Maintenance Margin**: The minimum amount that must be maintained in the account.

Hedging:

Hedging involves taking an offsetting position in a related security to mitigate risk. Common hedging strategies include using options and futures.

- **Options**: Buying puts to protect against downside risk.
- **Futures**: Shorting futures contracts to hedge against a potential decline in the asset's price.

Technical Analysis:

Technical analysis involves using historical price data and technical indicators to forecast future price movements. Common tools include:

- **Moving Averages**: Used to smooth out price data to identify trends.
- **Relative Strength Index (RSI)**: Measures the speed and change of price movements.
- **Bollinger Bands**: Provides a relative definition of high and low prices.

Fundamental Analysis:

Fundamental analysis involves evaluating an asset's intrinsic value based on economic and financial factors. Key elements include:

- **Earnings Reports**: Assessing a company's profitability.
- **Economic Indicators**: Analyzing GDP growth, unemployment rates, etc.
- **Valuation Ratios**: Using P/E ratio, P/B ratio, etc., to determine if an asset is overvalued or undervalued.

By understanding these trends and applying appropriate position sizing and risk management strategies, investors can make more informed decisions, potentially leveraging seasonal patterns to optimize returns and manage risks.

In [ ]:

```
# import pandas as pd
# import matplotlib.pyplot as plt
# import seaborn as sns
# from sklearn.model_selection import train_test_split
# from sklearn.ensemble import RandomForestClassifier
# from sklearn.metrics import accuracy_score

# # Import custom functions
# from data_collection.load_data import load_data, load_price_data
# from data_collection.resample_data import resample_to_monthly
# from data_collection.technicals import bollinger_bands, macd, rsi, woodie_pivots, obv, atr, st
```

```
# # Import functions from the provided files
# from analysis.seasonality_analysis import seasonality_analysis, display_seasonality_stats, plot

# Set default figure size
plt.rcParams['figure.figsize'] = (8, 6)

def calculate_returns(df):
    """Calculate the daily returns."""
    df['Return'] = df['Adj Close'].pct_change() * 100
    return df

def create_seasonality_table(df):
    """Create a seasonality table for returns."""
    df = df.dropna(subset=['Return'])
    df_monthly = resample_to_monthly(df)
    df_monthly['Monthly Return'] = df_monthly['Adj Close'].pct_change() * 100
    return seasonality_analysis(df_monthly)

def visualize_seasonality_table(seasonality_table, title):
    """Visualize the seasonality table as a heatmap."""
    sns.heatmap(seasonality_table, annot=True, cmap='RdYlGn', center=0)
    plt.title(title)
    plt.show()

def apply_kelly_method(mean_return, std_dev, win_prob):
    """Calculate the Kelly criterion for position sizing."""
    b = mean_return / std_dev  # Assuming b is the edge ratio
    kelly_fraction = win_prob - ((1 - win_prob) / b)
    return kelly_fraction

def display_all_monthly_statistics_with_kelly(df):
    """Display all monthly statistics for a DataFrame with Kelly position size."""
    df_monthly = resample_to_monthly(df)
    df_monthly['Monthly Return'] = df_monthly['Adj Close'].pct_change() * 100
    stats = df_monthly.groupby(df_monthly.index.month)['Monthly Return'].agg(['mean', 'std', 'cou
    stats['positive_prob'] = df_monthly.groupby(df_monthly.index.month)['Monthly Return'].apply(
    stats['kelly_size'] = stats.apply(lambda row: apply_kelly_method(row['mean'], row['std'], row
    stats.index = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov',

    for month, row in stats.iterrows():
        print(f"{month}: Mean = {row['mean']:.2f}, Std Dev = {row['std']:.2f}, Count = {row['cou

    return stats

def machine_learning_analysis(df):
    """Perform machine learning analysis using RandomForest and return the model and accuracy."""
    df['Target'] = (df['Return'] > 0).astype(int)  # Binary classification: 1 if return is posit
    features = ['Adj Close', 'Return']  # Example features; you can add more technical indicator

    X = df[features].shift(1)  # Shift features to avoid look-ahead bias
    y = df['Target'].shift(1)

    # Drop rows with NaN values to ensure consistent lengths
    X, y = X.dropna(), y.dropna()
    X, y = X.align(y, join='inner', axis=0)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
    model = RandomForestClassifier(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
```

```
    print(f"Model Accuracy: {accuracy:.2f}")
    return model, accuracy

def create_summary_csv(tickers, start_date, end_date, filename='summary.csv'):
    """Create a CSV file with mean, std, count, positive_prob, and Kelly size for all assets."""
    summary_data = []

    for ticker in tickers:
        df = load_price_data(ticker, start=start_date, end=end_date)

        if isinstance(df, pd.Series):
            df = df.to_frame(name='Adj Close')

        if 'Adj Close' not in df.columns:
            print(f"Column 'Adj Close' not found in the data for {ticker}. Available columns: {d
            continue

        df = calculate_returns(df)
        seasonality_table = create_seasonality_table(df)

        for month, stats in seasonality_table.iterrows():
            mean_return = stats['mean']
            std_dev = stats['std']
            count = stats['count']
            positive_prob = stats['positive_prob']
            kelly_size = apply_kelly_method(mean_return, std_dev, positive_prob)

            summary_data.append({
                'Ticker': ticker,
                'Month': month,
                'Mean Return': mean_return,
                'Standard Deviation': std_dev,
                'Count': count,
                'Positive Probability': positive_prob,
                'Kelly Size': kelly_size
            })

    summary_df = pd.DataFrame(summary_data)
    summary_df.to_csv(filename, index=False)
    print(f"Summary CSV created: {filename}")

def analyze_ticker(ticker, start_date, end_date):
    df = load_price_data(ticker, start=start_date, end=end_date)

    if isinstance(df, pd.Series):
        df = df.to_frame(name='Adj Close')

    if 'Adj Close' not in df.columns:
        print(f"Column 'Adj Close' not found in the data for {ticker}. Available columns: {df.co
        return

    df = calculate_returns(df)

    seasonality_table = create_seasonality_table(df)
    visualize_seasonality_table(seasonality_table, f'Seasonality of {ticker} Returns')
    display_all_monthly_statistics_with_kelly(df)

    # Machine Learning Analysis
    model, accuracy = machine_learning_analysis(df)

def main():
```

```python
    tickers = ['SPY', 'QQQ', 'TQQQ', 'SQQQ', 'SOXL', 'TSLL', 'NVDL']
    start_date = '2000-01-01'
    end_date = '2024-01-01'

    for ticker in tickers:
        analyze_ticker(ticker, start_date, end_date)

    # Create summary CSV
    create_summary_csv(tickers, start_date, end_date)

if __name__ == "__main__":
    main()
```

In [18]:

```python
import yfinance as yf
import yoptions as yo
import optionlab as ol
import pandas as pd
from pandas_datareader import data as pdr
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import scipy as si
from scipy import stats
from statsmodels.tsa.seasonal import seasonal_decompose
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score
import concurrent.futures
import backtrader as bt
import quandl
import QuantLib as ql
import quantstats as qs

# Import custom functions
from data_collection.load_data import load_price_data
from data_collection.resample_data import resample_to_monthly
from data_collection.technicals import bollinger_bands, macd, rsi, woodie_pivots, obv, atr, stocl

# Import functions from the provided files
from analysis.seasonality_analysis import seasonality_analysis, display_seasonality_stats, plot_s

# Set default figure size
plt.rcParams['figure.figsize'] = (8, 6)

# Technical Indicators
def ichimoku_cloud(df):
    if 'High' not in df.columns or 'Low' not in df.columns:
        print("Data does not contain 'High' or 'Low' columns necessary for Ichimoku Cloud.")
        return df

    high_9 = df['High'].rolling(window=9).max()
    low_9 = df['Low'].rolling(window=9).min()
    df['tenkan_sen'] = (high_9 + low_9) / 2

    high_26 = df['High'].rolling(window=26).max()
    low_26 = df['Low'].rolling(window=26).min()
    df['kijun_sen'] = (high_26 + low_26) / 2

    df['senkou_span_a'] = ((df['tenkan_sen'] + df['kijun_sen']) / 2).shift(26)
    high_52 = df['High'].rolling(window=52).max()
```

```
    low_52 = df['Low'].rolling(window=52).min()
    df['senkou_span_b'] = ((high_52 + low_52) / 2).shift(26)

    df['chikou_span'] = df['Adj Close'].shift(-26)

    return df

def add_technical_indicators(df):
    try:
        df = bollinger_bands(df)
        df = macd(df)
        df = rsi(df)
        df = woodie_pivots(df)
        df = obv(df)
        df = atr(df)
        df = stochastic_oscillator(df)
    except KeyError as e:
        print(f"Missing column for technical indicator calculation: {e}")
    return df

# Advanced Statistical Models
def arima_forecast(df, column='Adj Close', order=(5, 1, 0)):
    model = ARIMA(df[column], order=order)
    model_fit = model.fit()
    forecast = model_fit.forecast(steps=10)
    return forecast

def garch_forecast(df, column='Adj Close'):
    model = arch_model(df[column], vol='Garch', p=1, q=1)
    model_fit = model.fit()
    forecast = model_fit.forecast(horizon=10)
    return forecast

# Fundamental Analysis
def get_fundamental_ratios(ticker):
    stock = yf.Ticker(ticker)
    pe_ratio = stock.info.get('trailingPE', None)
    pb_ratio = stock.info.get('priceToBook', None)
    debt_to_equity = stock.info.get('debtToEquity', None)
    return pe_ratio, pb_ratio, debt_to_equity

# Hyperparameter Optimization
def optimize_model_hyperparameters(X, y):
    param_grid = {
        'n_estimators': [50, 100, 200],
        'max_depth': [None, 10, 20, 30],
        'min_samples_split': [2, 5, 10]
    }
    model = RandomForestClassifier(random_state=42)
    grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5)
    grid_search.fit(X, y)
    return grid_search.best_estimator_

# Backtesting Framework
class MyStrategy(bt.Strategy):
    params = (
        ('maperiod', 15),
    )

    def __init__(self):
        self.dataclose = self.datas[0].close
        self.order = None
```

```
        self.sma = bt.indicators.SimpleMovingAverage(self.datas[0], period=self.params.maperiod)

    def next(self):
        if self.order:
            return

        if not self.position:
            if self.dataclose[0] > self.sma[0]:
                self.order = self.buy()
        else:
            if self.dataclose[0] < self.sma[0]:
                self.order = self.sell()

def backtest_strategy(df):
    cerebro = bt.Cerebro()
    cerebro.addstrategy(MyStrategy)
    data = bt.feeds.PandasData(dataname=df)
    cerebro.adddata(data)
    cerebro.run()
    cerebro.plot()

# Main Analysis Function
def analyze_ticker(ticker, start_date, end_date):
    df = load_price_data(ticker, start=start_date, end=end_date)

    if isinstance(df, pd.Series):
        df = df.to_frame(name='Adj Close')

    if 'Adj Close' not in df.columns:
        print(f"Column 'Adj Close' not found in the data for {ticker}. Available columns: {df.co
        return

    if 'Close' not in df.columns or 'High' not in df.columns or 'Low' not in df.columns:
        print(f"Columns 'Close', 'High', and 'Low' are required. Available columns: {df.columns}
        return

    df['Return'] = df['Adj Close'].pct_change() * 100
    df = ichimoku_cloud(df)
    df = add_technical_indicators(df)

    # ARIMA and GARCH Forecasts
    arima_forecast(df)
    garch_forecast(df)

    # Fundamental Ratios
    pe_ratio, pb_ratio, debt_to_equity = get_fundamental_ratios(ticker)
    print(f"P/E Ratio: {pe_ratio}, P/B Ratio: {pb_ratio}, Debt to Equity: {debt_to_equity}")

    # Machine Learning with Hyperparameter Optimization
    df['Target'] = (df['Return'] > 0).astype(int)
    features = ['Adj Close', 'Return']
    X = df[features].shift(1).dropna()
    y = df['Target'].shift(1).dropna()
    X, y = X.align(y, join='inner')
    best_model = optimize_model_hyperparameters(X, y)
    y_pred = best_model.predict(X)
    accuracy = accuracy_score(y, y_pred)
    print(f"Optimized Model Accuracy: {accuracy:.2f}")

    # Backtest Strategy
    backtest_strategy(df)
```

```python
def main():
    tickers = ['SPY', 'QQQ', 'TQQQ', 'SQQQ', 'SOXL', 'TSLL', 'NVDL']
    start_date = '2000-01-01'
    end_date = '2024-01-01'

    for ticker in tickers:
        analyze_ticker(ticker, start_date, end_date)

if __name__ == "__main__":
    main()
```

```
c:\ProgramData\anaconda3\envs\jupyter-ai\Lib\site-packages\yfinance\utils.py:775: FutureWarning:
The 'unit' keyword in TimedeltaIndex construction is deprecated and will be removed in a future v
ersion. Use pd.to_timedelta instead.
  df.index += _pd.TimedeltaIndex(dst_error_hours, 'h')
[*********************100%***********************]  1 of 1 completed
c:\ProgramData\anaconda3\envs\jupyter-ai\Lib\site-packages\yfinance\utils.py:775: FutureWarning:
The 'unit' keyword in TimedeltaIndex construction is deprecated and will be removed in a future v
ersion. Use pd.to_timedelta instead.
  df.index += _pd.TimedeltaIndex(dst_error_hours, 'h')
[*********************100%***********************]  1 of 1 completed
c:\ProgramData\anaconda3\envs\jupyter-ai\Lib\site-packages\yfinance\utils.py:775: FutureWarning:
The 'unit' keyword in TimedeltaIndex construction is deprecated and will be removed in a future v
ersion. Use pd.to_timedelta instead.
  df.index += _pd.TimedeltaIndex(dst_error_hours, 'h')
[*********************100%***********************]  1 of 1 completed
c:\ProgramData\anaconda3\envs\jupyter-ai\Lib\site-packages\yfinance\utils.py:775: FutureWarning:
The 'unit' keyword in TimedeltaIndex construction is deprecated and will be removed in a future v
ersion. Use pd.to_timedelta instead.
  df.index += _pd.TimedeltaIndex(dst_error_hours, 'h')
[*********************100%***********************]  1 of 1 completed
c:\ProgramData\anaconda3\envs\jupyter-ai\Lib\site-packages\yfinance\utils.py:775: FutureWarning:
The 'unit' keyword in TimedeltaIndex construction is deprecated and will be removed in a future v
ersion. Use pd.to_timedelta instead.
  df.index += _pd.TimedeltaIndex(dst_error_hours, 'h')
Columns 'Close', 'High', and 'Low' are required. Available columns: Index(['Adj Close'], dtype='o
bject')
Columns 'Close', 'High', and 'Low' are required. Available columns: Index(['Adj Close'], dtype='o
bject')
Columns 'Close', 'High', and 'Low' are required. Available columns: Index(['Adj Close'], dtype='o
bject')
Columns 'Close', 'High', and 'Low' are required. Available columns: Index(['Adj Close'], dtype='o
bject')
[*********************100%***********************]  1 of 1 completed
c:\ProgramData\anaconda3\envs\jupyter-ai\Lib\site-packages\yfinance\utils.py:775: FutureWarning:
The 'unit' keyword in TimedeltaIndex construction is deprecated and will be removed in a future v
ersion. Use pd.to_timedelta instead.
  df.index += _pd.TimedeltaIndex(dst_error_hours, 'h')
[*********************100%***********************]  1 of 1 completed
c:\ProgramData\anaconda3\envs\jupyter-ai\Lib\site-packages\yfinance\utils.py:775: FutureWarning:
The 'unit' keyword in TimedeltaIndex construction is deprecated and will be removed in a future v
ersion. Use pd.to_timedelta instead.
  df.index += _pd.TimedeltaIndex(dst_error_hours, 'h')
[*********************100%***********************]  1 of 1 completed
Columns 'Close', 'High', and 'Low' are required. Available columns: Index(['Adj Close'], dtype='o
bject')
Columns 'Close', 'High', and 'Low' are required. Available columns: Index(['Adj Close'], dtype='o
bject')
Columns 'Close', 'High', and 'Low' are required. Available columns: Index(['Adj Close'], dtype='o
bject')
```

In [24]:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import LSTM, Dense
from sklearn.metrics import mean_absolute_error, mean_squared_error
from datetime import datetime
import yfinance as yf
from analysis.forecasting import (forecast_future, scale_data, create_sequences,
                                  build_and_train_model, calculate_metrics,
                                  plot_forecasts, plot_ghost_candles, download_stock_data,foreca

# Define the load_price_data function
def load_price_data(tickers, start='2000-01-01', end=None):
    """
    Load historical data for given tickers using yfinance.

    Parameters:
    tickers (str or list): A single ticker symbol or a list of ticker symbols.
    start (str): The start date for fetching data in 'YYYY-MM-DD' format.
    end (str): The end date for fetching data in 'YYYY-MM-DD' format. Defaults to today's date i

    Returns:
    pd.DataFrame: DataFrame containing historical price data.
    """
    if end is None:
        end = datetime.now().strftime('%Y-%m-%d')

    data = yf.download(tickers, start=start, end=end, auto_adjust=False)

    if data.empty:
        print(f"No data found for {tickers}")
        return pd.DataFrame()

    # Ensure all necessary columns are included
    required_columns = ['Open', 'High', 'Low', 'Close', 'Adj Close']

    if isinstance(tickers, str):
        if not all(col in data.columns for col in required_columns):
            print(f"Missing required columns in data for {tickers}. Available columns: {data.colu
            return pd.DataFrame()
    else:
        if not all(col in data.columns.get_level_values(0) for col in required_columns):
            print(f"Missing required columns in data for {tickers}. Available columns: {data.colu
            return pd.DataFrame()

    return data

# Example usage: Load data, perform analysis, forecast, and plot results
def main():
    tickers = ['AAPL']
    start_date = '2020-01-01'
    end_date = '2023-01-01'
    seq_length = 60
    future_days = 10

    # Load data
    stock_data = load_price_data(tickers, start=start_date, end=end_date)
```

```python
    if stock_data.empty:
        print("No data to analyze.")
        return

    # Perform forecasting and plotting
    features = ['Open', 'High', 'Low', 'Close']
    forecast_and_plot_complete(tickers[0], features, start_date, end_date, seq_length, future_day

if __name__ == "__main__":
    main()
```

```
c:\ProgramData\anaconda3\envs\jupyter-ai\Lib\site-packages\yfinance\utils.py:775: FutureWarning:
The 'unit' keyword in TimedeltaIndex construction is deprecated and will be removed in a future v
ersion. Use pd.to_timedelta instead.
  df.index += _pd.TimedeltaIndex(dst_error_hours, 'h')
[*********************100%%**********************]  1 of 1 completed

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[24], line 71
     68     forecast_and_plot_complete(tickers[0], features, start_date, end_date, seq_length, fu
ture_days)
     70 if __name__ == "__main__":
---> 71     main()

Cell In[24], line 68, in main()
     66 # Perform forecasting and plotting
     67 features = ['Open', 'High', 'Low', 'Close']
---> 68 forecast_and_plot_complete(tickers[0], features, start_date, end_date, seq_length, future
_days)

File c:\Users\Administrator\Desktop\DataSciencePortfolio\QuantitativeFinance\Studies\Seasonality
\analysis\forecasting.py:110, in forecast_and_plot_complete(ticker, features, start_date, end_dat
e, seq_length, future_days)
    109 def forecast_and_plot_complete(ticker, features, start_date, end_date, seq_length, future
_days=10):
--> 110     stock_data = download_stock_data(ticker, start_date, end_date)
    111 #     scaler, scaled_data = scale_data(stock_data[features].values)
    112     scaler, scaled_data = scale_data(stock_data[features].values, features)

NameError: name 'download_stock_data' is not defined
```

In [ ]: