```
#!pip install openpyxl
#!pip install mplfinance
import mplfinance as mpf
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
import numpy as np
from google.colab import drive
import seaborn as sns
```

```
##DOWNLOADING FROM Y FINANCE:
# NIFTY 50 Yahoo Finance ticker symbol
#nifty_symbol = "^NSEI"  # "^NSEI" is the Yahoo ticker for NIFTY 50

# Download historical data
#df = yf.download(nifty_symbol, start="2007-09-17", end="2025-09-13")
#print(df.head(10))
#print(df.tail(10))
#print(df.size)
#print(df.shape)
#print(df.columns)
##SAVING THE FILE TO LOCAL:
#df.to_excel("data.xlsx", index=True)
#from google.colab import files
#files.download("data.xlsx")
```

```
sta = "2025-09-12"
nif = 25114
```

```
##OPENING FILE FROM DRIVE:
# Mount Google Drive
drive.mount('/content/drive')
# Define file path (update with your actual file name)
file_path = "/content/drive/My Drive/Colab/data-3.xlsx"
# Load the Excel file
df = pd.read_excel(file_path, sheet_name="Sheet1")  # Replace with you
# Display first few rows
print(df.head())
print(df.shape)
print(df.tail())
```

```
Mounted at /content/drive
        Date        Close         High          Low         Open  Volu
0 2007-09-17  4494.649902  4549.049805  4482.850098  4518.450195
1 2007-09-18  4546.200195  4551.799805  4481.549805  4494.100098
2 2007-09-19  4732.350098  4739.000000  4550.250000  4550.250000
3 2007-09-20  4747.549805  4760.850098  4721.149902  4734.850098
4 2007-09-21  4837.549805  4855.700195  4733.700195  4752.950195
(4412, 6)
           Date     Close      High       Low      Open  Volume
4407 2025-09-08  24773.15  24885.50  24751.55  24802.60  213200
4408 2025-09-09  24868.60  24891.80  24814.00  24864.10  226900
4409 2025-09-10  24973.10  25035.70  24915.05  24991.00  244100
4410 2025-09-11  25005.50  25037.30  24940.15  24945.50  224600
4411 2025-09-12  25114.00  25139.45  25038.05  25074.45  225700
```

```
new_df = df[['Date','Open', 'High', 'Low', 'Close', 'Volume']].copy()
new_df.columns = ['Date','Open', 'High', 'Low', 'Close', 'Volume']
new_df['Date'] = pd.to_datetime(df['Date'])
new_df.set_index('Date', inplace=True)
print(new_df.head(1))
dw = new_df.copy(deep = True)
#dw = new_df.resample('W').agg({'Open': 'first',
#                                       'High': 'max',
#                                       'Low': 'min',
#                                       'Close': 'last',
#                                       'Volume': 'sum'})
#print(dw.head(10))
#print(dw.tail(10))
#print(dw.size)
#print(dw.shape)
dw['MA20'] = dw['Close'].rolling(window=20).mean()

# Calculate 2 standard deviation Bollinger Bands
dw['Upper'] = dw['MA20'] + (dw['Close'].rolling(window=20).std() * 2)
dw['Lower'] = dw['MA20'] - (dw['Close'].rolling(window=20).std() * 2)
```

```
# Define additional Bollinger Bands indicators
bollinger_bands = [
    mpf.make_addplot(dw['Upper'], color='red'),
    mpf.make_addplot(dw['Lower'], color='blue'),
    mpf.make_addplot(dw['MA20'], color='green')
]

# Calculate MACD and Signal Line
dw['EMA_12'] = dw['Close'].ewm(span=12, adjust=False).mean()  # 12-da
dw['EMA_26'] = dw['Close'].ewm(span=26, adjust=False).mean()  # 26-da
dw['MACD'] = dw['EMA_12'] - dw['EMA_26']  # MACD Line
dw['Signal'] = dw['MACD'].ewm(span=9, adjust=False).mean()  # Signal 

# Calculate Stochastic Oscillator (14-day period)
low_14 = dw['Low'].rolling(window=14).min()
high_14 = dw['High'].rolling(window=14).max()
dw['%K'] = (dw['Close'] - low_14) / (high_14 - low_14) * 100  # %K Li
dw['%D'] = dw['%K'].rolling(window=3).mean()  # %D (3-day SMA of %K)

# RSI Calculation Function
def compute_rsi(data, period=14):
    delta = data.diff(1)  # Price changes
    gain = (delta.where(delta > 0, 0)).rolling(window=period).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=period).mean()

    rs = gain / loss  # Relative Strength
    rsi = 100 - (100 / (1 + rs))  # RSI Formula
    return rsi

# Compute RSI for 'Close' prices
dw['RSI'] = compute_rsi(dw['Close'])

# DMI Calculation Function
def compute_dmi(data, period=14):

    # Calculate True Range (TR)
    dw['High-Low'] = dw['High'] - dw['Low']
    dw['High-Close'] = (dw['High'] - dw['Close'].shift(1)).abs()
    dw['Low-Close'] = (dw['Low'] - dw['Close'].shift(1)).abs()
    dw['TR'] = dw[['High-Low', 'High-Close', 'Low-Close']].max(axis=1

    # Calculate Directional Movement (DM)
    dw['+DM'] = dw['High'].diff()
    dw['-DM'] = dw['Low'].diff()
    dw['+DM'] = dw['+DM'].where((dw['+DM'] > dw['-DM']) & (dw['+DM'] 
    dw['-DM'] = dw['-DM'].where((dw['-DM'] > dw['+DM']) & (dw['-DM'] 
```

```
        # Smooth TR, +DM, and -DM using EMA
        dw['TR_smooth'] = dw['TR'].rolling(window=period).mean()
        dw['+DM_smooth'] = dw['+DM'].rolling(window=period).mean()
        dw['-DM_smooth'] = dw['-DM'].rolling(window=period).mean()

        # Calculate +DI and -DI
        dw['+DI'] = (dw['+DM_smooth'] / dw['TR_smooth']) * 100
        dw['-DI'] = (dw['-DM_smooth'] / dw['TR_smooth']) * 100

        # Calculate ADX (Average Directional Index)
        dw['DX'] = (abs(dw['+DI'] - dw['-DI']) / (dw['+DI'] + dw['-DI']))
        dw['ADX'] = dw['DX'].rolling(window=period).mean()

        return dw[['+DI', '-DI', 'ADX']]

    # Compute DMI
    dmi_dw = compute_dmi(dw)
    dw['macdco'] = dw.apply(lambda row: 1 if row['MACD'] > row['Signal'] e
    dw['macdzo'] = dw.apply(lambda row: 1 if row['MACD'] > 0 else -1, axis
    dw['macdtick'] = dw['MACD'].diff().apply(lambda x: 1 if x > 0 else (-1
    dw['macddelta'] = (dw['MACD'].diff()/dw['MACD'].shift(1))*100
    dw['macdhist'] = dw['MACD'] - dw['Signal']

    dw["stochco"] = dw.apply(lambda row: 1 if row['%K'] > row['%D'] else -
    dw['stochzo'] = dw.apply(lambda row: 1 if row['%K'] > 80 else (-1 if r
    dw['stochtick'] = dw['%K'].diff().apply(lambda x: 1 if x > 0 else (-1
    dw['stodelta'] = (dw['%K'].diff()/dw['%K'].shift(1))*100
    dw['stohist'] = dw['%K'] - dw['%D']

    dw['rsizo'] = dw.apply(lambda row: 1 if row['RSI'] > 70 else (-1 if ro
    dw['rsitick'] = dw['RSI'].diff().apply(lambda x: 1 if x > 0 else (-1 i
    dw['rsidelta'] = (dw['RSI'].diff()/dw['RSI'].shift(1))*100

    dw["bbzo"] = dw.apply(lambda row: 1 if row['Close'] > row['MA20'] else
    dw['ubbtick'] = dw['Upper'].diff().apply(lambda x: 1 if x > 0 else (-1
    dw['lbbtick'] = dw['Lower'].diff().apply(lambda x: 1 if x > 0 else (-1
    dw['ubbdelta'] = (dw['Upper'].diff()/dw['Upper'].shift(1))*100
    dw['lbbdelta'] = (dw['Lower'].diff()/dw['Lower'].shift(1))*100
    dw['ulhist'] = dw['Upper'] - dw['Lower']
    dw['s20delta'] = (dw['MA20'].diff()/dw['MA20'].shift(1))*100



    dw['didelta'] = dw['+DI'] - dw['-DI']
    dw['dizo'] = dw.apply(lambda row: 1 if row['+DI'] > row['-DI'] else -1
    dw['adxzo'] = dw.apply(lambda row: 1 if row['ADX'] > 15 else -1, axis=
    dw['postick'] = dw['+DI'].diff().apply(lambda x: 1 if x > 0 else (-1 i
    dw['negtick'] = dw['-DI'].diff().apply(lambda x: 1 if x > 0 else (-1 i
    dw['posdelta'] = (dw['+DI'].diff()/dw['+DI'].shift(1))*100
```

```
dw['negdelta'] = (dw['-DI'].diff()/dw['-DI'].shift(1))*100
dw['adxdelta'] = (dw['ADX'].diff()/dw['ADX'].shift(1))*100

dw['return'] = dw['Close'].diff().apply(lambda x: 1 if x > 0 else (-1
dw['returnval'] = (dw['Close'].diff()/dw['Close'].shift(1))*100
dw['openclose'] = ((dw['Close'] - dw['Open'])/dw['Open'])*100

dw['returnval'].fillna(0, inplace=True)
dw['macddelta'].fillna(0, inplace=True)
dw['stodelta'].fillna(0, inplace=True)
dw['rsidelta'].fillna(0, inplace=True)
dw['ubbdelta'].fillna(0, inplace=True)
dw['lbbdelta'].fillna(0, inplace=True)
dw['posdelta'].fillna(0, inplace=True)
dw['negdelta'].fillna(0, inplace=True)
dw.replace([np.inf, -np.inf], 0, inplace=True)
#print(dw.head(30))
#print(dw.shape)
#dwf = dw[['Open', 'High', 'Low', 'Close','macdco','macdzo','macdtick
#dwf = dw[['macddelta','stodelta','rsidelta','ubbdelta','lbbdelta','p
dwf = dw[['macdhist', 'stohist','%K','ulhist','s20delta','didelta','a

#print(dwf.tail(5))
dwf['returnval'] = dwf['returnval'].shift(-1)
#dwf['openclose'] = dwf['openclose'].shift(-1)
#print(dwf.tail(5))
#print(dwf.head(30))
dwf = dwf.drop(dwf.index[:30])
#print(dwf.head(30))
print("*************************new data frame**********************
#print(dwf.head(30))
#print(dwf.tail(5))
#dfinal = dwf.iloc[[-2]]
dfinal = dwf.tail(2)
print(dfinal)
dwf = dwf.iloc[:-2]
#print(dwf.tail(5))
print("*************************new data frame**********************
print(dwf.tail(5))
print(dwf.head(5))
```

```
For example, when doing 'df[col].method(value, inplace=True)', try us


  dw['returnval'].fillna(0, inplace=True)
/tmp/ipython-input-2818557836.py:125: FutureWarning: A value is tryin
The behavior will change in pandas 3.0. This inplace method will neve
```

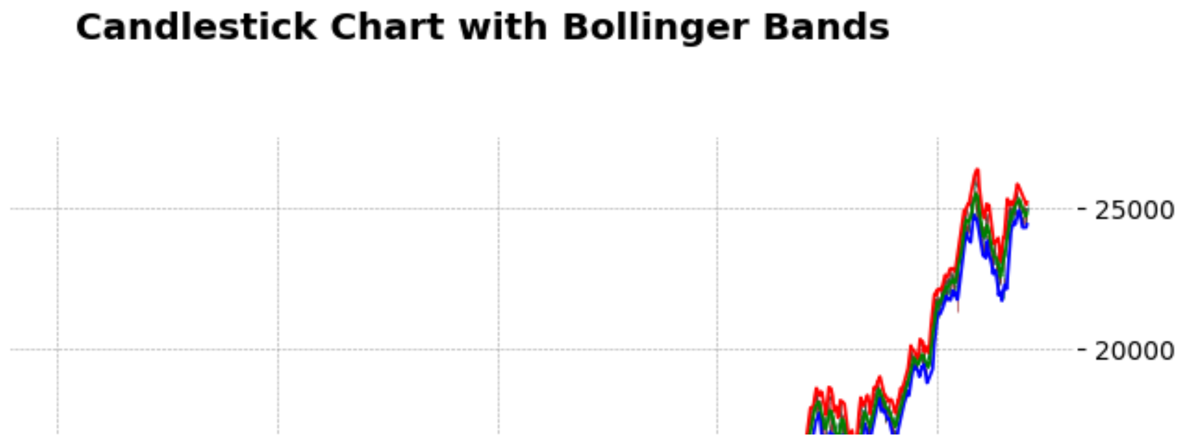
```
For example, when doing 'df[col].method(value, inplace=True)', try us


  dw['macddelta'].fillna(0, inplace=True)
/tmp/ipython-input-2818557836.py:126: FutureWarning: A value is tryin
The behavior will change in pandas 3.0. This inplace method will neve

For example, when doing 'df[col].method(value, inplace=True)', try us


  dw['stodelta'].fillna(0, inplace=True)
/tmp/ipython-input-2818557836.py:127: FutureWarning: A value is tryin
The behavior will change in pandas 3.0. This inplace method will neve

For example, when doing 'df[col].method(value, inplace=True)', try us


  dw['rsidelta'].fillna(0, inplace=True)
/tmp/ipython-input-2818557836.py:128: FutureWarning: A value is tryin
The behavior will change in pandas 3.0. This inplace method will neve

For example, when doing 'df[col].method(value, inplace=True)', try us


  dw['ubbdelta'].fillna(0, inplace=True)
/tmp/ipython-input-2818557836.py:129: FutureWarning: A value is tryin
The behavior will change in pandas 3.0. This inplace method will neve

For example, when doing 'df[col].method(value, inplace=True)', try us


  dw['lbbdelta'].fillna(0, inplace=True)
/tmp/ipython-input-2818557836.py:130: FutureWarning: A value is tryin
The behavior will change in pandas 3.0. This inplace method will neve

For example, when doing 'df[col].method(value, inplace=True)', try us


  dw['posdelta'].fillna(0, inplace=True)
/tmp/ipython-input-2818557836.py:131: FutureWarning: A value is tryin
The behavior will change in pandas 3.0. This inplace method will neve

For example, when doing 'df[col].method(value, inplace=True)', try us


  dw['negdelta'].fillna(0, inplace=True)
/tmp/ipython-input-2818557836.py:140: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/panda
```

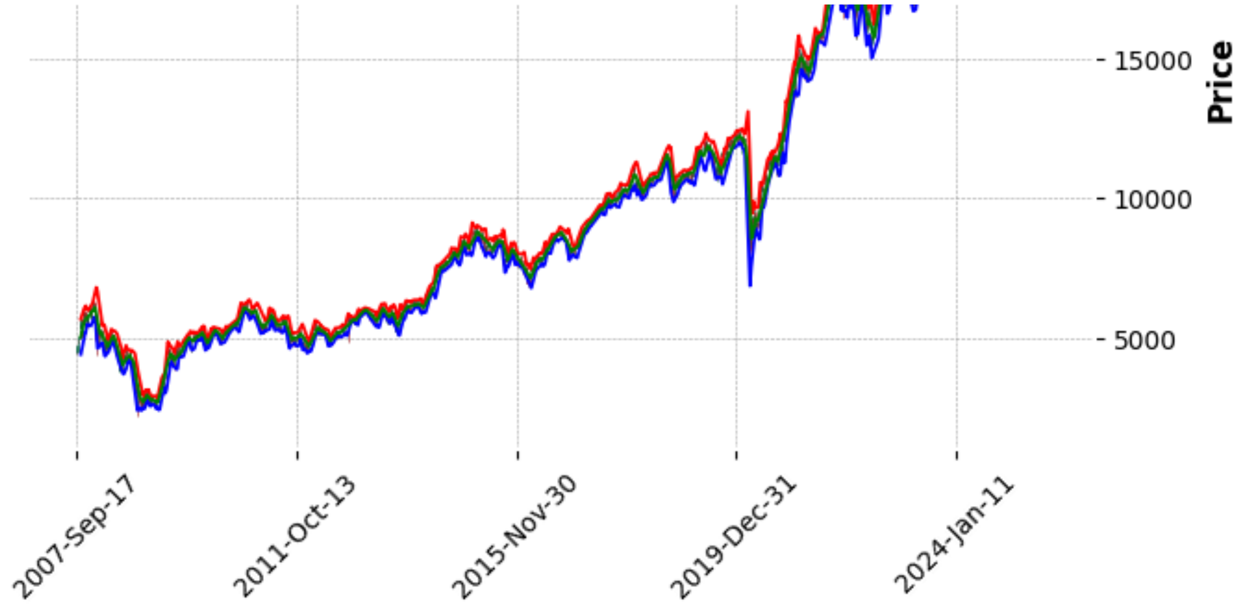
```
#dm = new_df.resample('ME').agg({'Open': 'first',
#                                       'High': 'max',
#                                       'Low': 'min',
#                                       'Close': 'last',
#                                       'Volume': 'sum'})
#print(dm.tail(2))
#mpf.plot(dw, type='candle', style='charles', volume=True, title="Nif
#mpf.plot(dm, type='candle', style='charles', volume=True, title="Nif
```

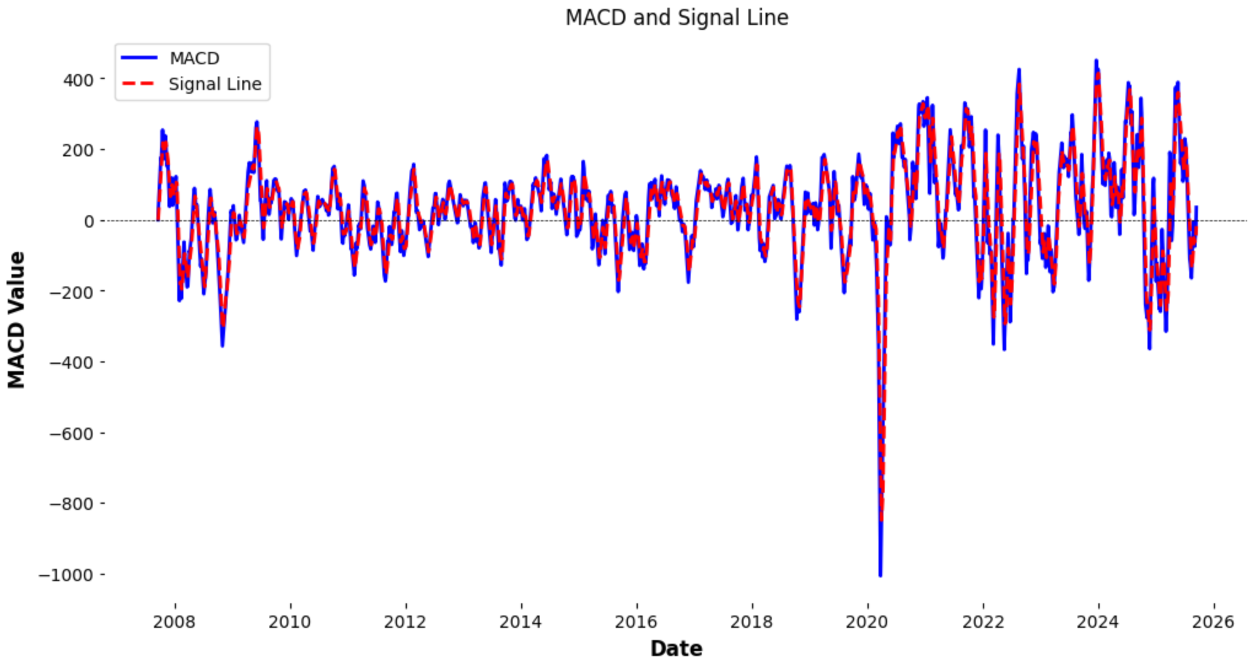
```
# Plot the candlestick chart with Bollinger Bands
mpf.plot(dw, type='candle', style='charles', volume=False,
         title=f"Candlestick Chart with Bollinger Bands",
         addplot=bollinger_bands)
# Plot MACD and Signal Line
plt.figure(figsize=(12, 6))
plt.plot(dw.index, dw['MACD'], label='MACD', color='blue')
plt.plot(dw.index, dw['Signal'], label='Signal Line', color='red', li
plt.axhline(0, color='black', linewidth=0.5, linestyle='dashed')  # Z

# Labels and Title
plt.xlabel('Date')
plt.ylabel('MACD Value')
plt.title('MACD and Signal Line')
plt.legend()
plt.grid()
plt.show()

# Plot Stochastic Indicator
plt.figure(figsize=(12, 6))
plt.plot(dw.index, dw['%K'], label='%K (Fast Line)', color='blue')
plt.plot(dw.index, dw['%D'], label='%D (Signal Line)', color='red', l
plt.axhline(80, color='green', linestyle='dashed', label='Overbought
plt.axhline(20, color='brown', linestyle='dashed', label='Oversold Le

# Labels and Title
plt.xlabel('Date')
plt.ylabel('Stochastic Value')
plt.title('Stochastic Oscillator')
plt.legend()
plt.grid()
plt.show()

## Plot RSI Indicator
plt.figure(figsize=(12, 6))
plt.plot(dw.index, dw['RSI'], label='RSI (14-day)', color='blue')
plt.axhline(70, color='red', linestyle='dashed', label='Overbought Le
plt.axhline(30, color='green', linestyle='dashed', label='Oversold Le
```

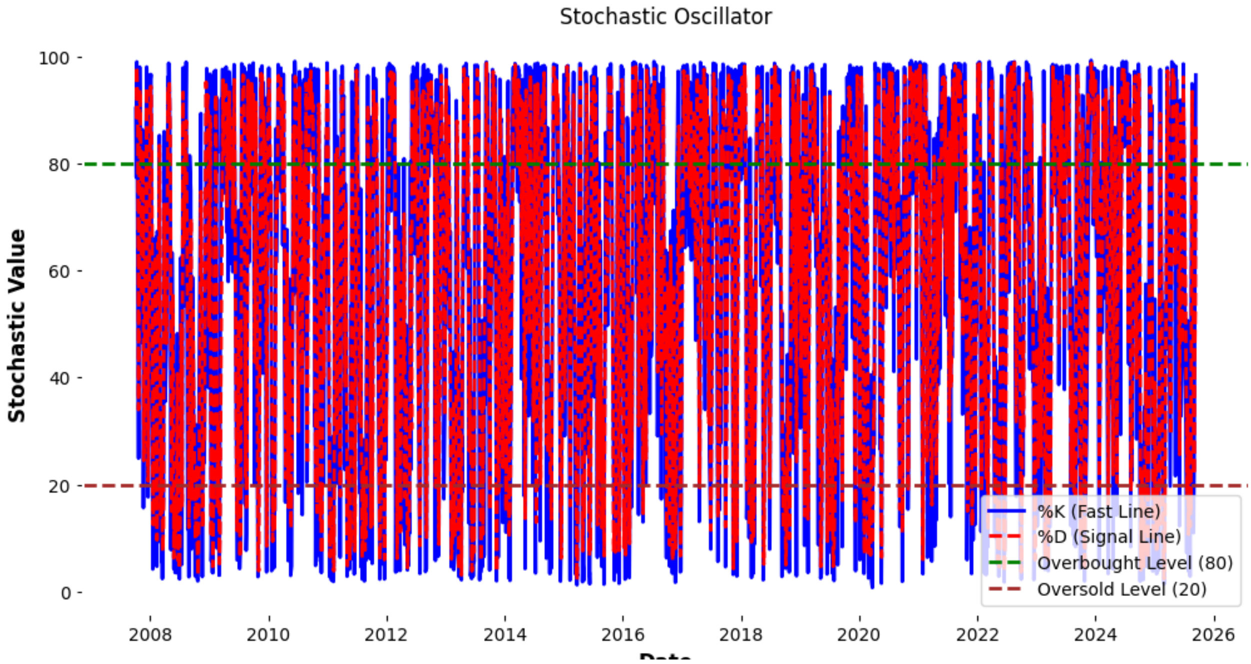
```
## Labels and Title
plt.xlabel('Date')
plt.ylabel('RSI Value')
plt.title('Relative Strength Index (RSI) Indicator')
plt.legend()
plt.grid()
plt.show()

# Plot DMI (DI+ and DI-)
plt.figure(figsize=(12, 6))
plt.plot(dmi_dw.index, dmi_dw['+DI'], label='+DI (Positive Direction)
plt.plot(dmi_dw.index, dmi_dw['-DI'], label='-DI (Negative Direction)
plt.plot(dmi_dw.index, dmi_dw['ADX'], label='ADX (Trend Strength)', co

# Labels and Title
plt.xlabel('Date')
plt.ylabel('Indicator Value')
plt.title('Directional Movement Index (DMI) and ADX')
plt.axhline(25, color='gray', linestyle='dashed', label='Trend Thresho
plt.legend()
plt.grid()
plt.show()
```

```
/usr/local/lib/python3.12/dist-packages/mplfinance/_arg_validators.py:8

 ================================================================

   WARNING: YOU ARE PLOTTING SO MUCH DATA THAT IT MAY NOT BE
            POSSIBLE TO SEE DETAILS (Candles, Ohlc-Bars, Etc.)
   For more information see:
   - https://github.com/matplotlib/mplfinance/wiki/Plotting-Too-Much-Da

   TO SILENCE THIS WARNING, set `type='line'` in `mpf.plot()`
   OR set kwarg `warn_too_much_data=N` where N is an integer
   LARGER than the number of data points you want to plot.

 ================================================================
  warnings.warn('\n\n ================================================
```

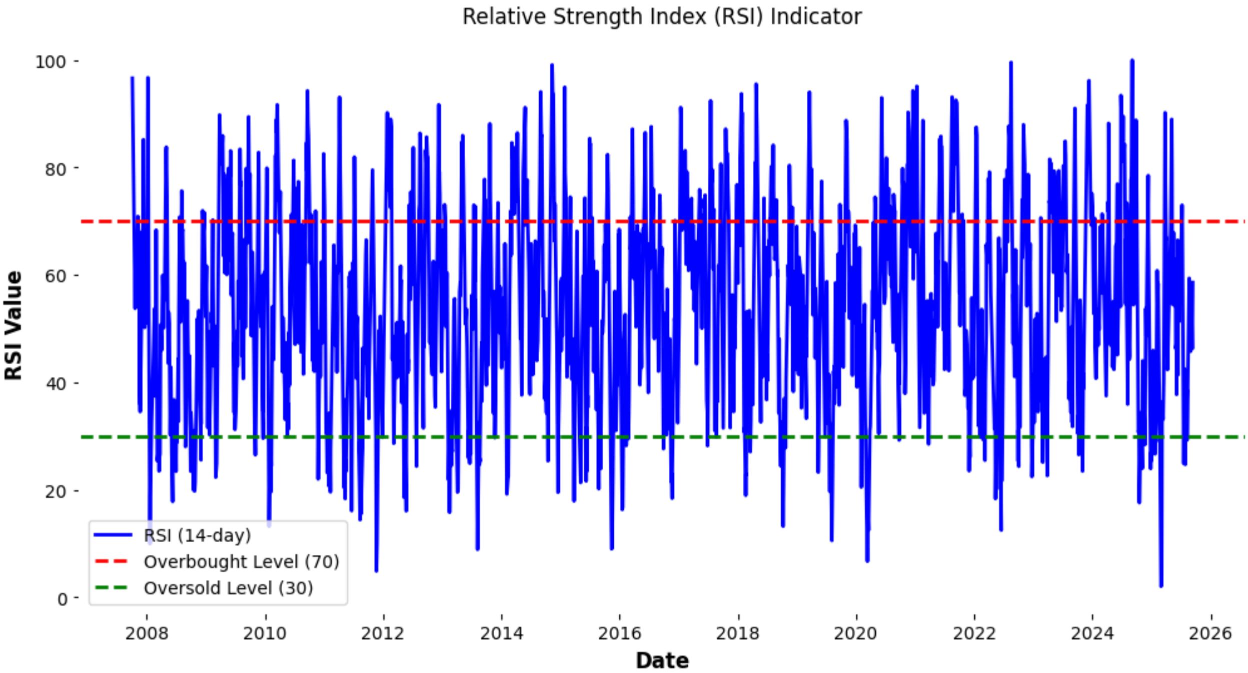


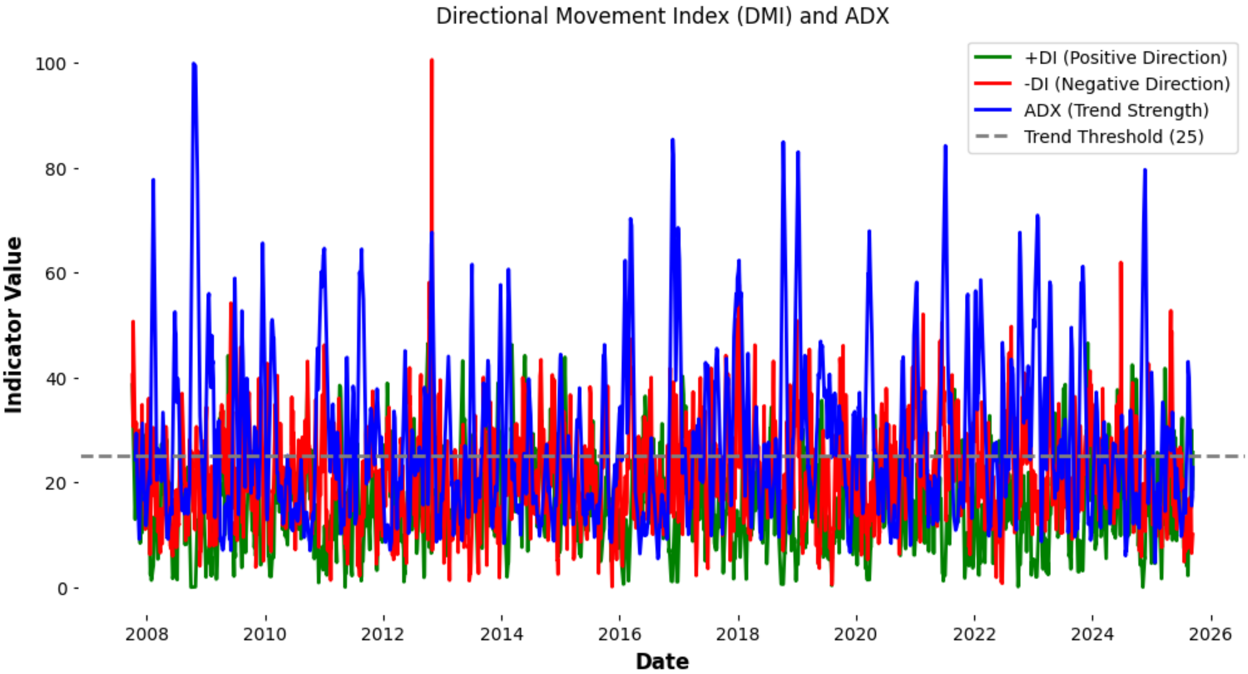

15000
10000
5000
Price
2007-Sep-17
2011-Oct-13
2015-Nov-30
2019-Dec-31
2024-Jan-11


MACD and Signal Line
MACD
Signal Line
400
200
0
−200
−400
−600
−800
−1000
MACD Value
2008
2010
2012
2014
2016
2018
2020
2022
2024
2026
Date


Stochastic Oscillator
100
80
60
40
20
0
Stochastic Value
%K (Fast Line)
%D (Signal Line)
Overbought Level (80)
Oversold Level (20)
2008
2010
2012
2014
2016
2018
2020
2022
2024
2026

Date

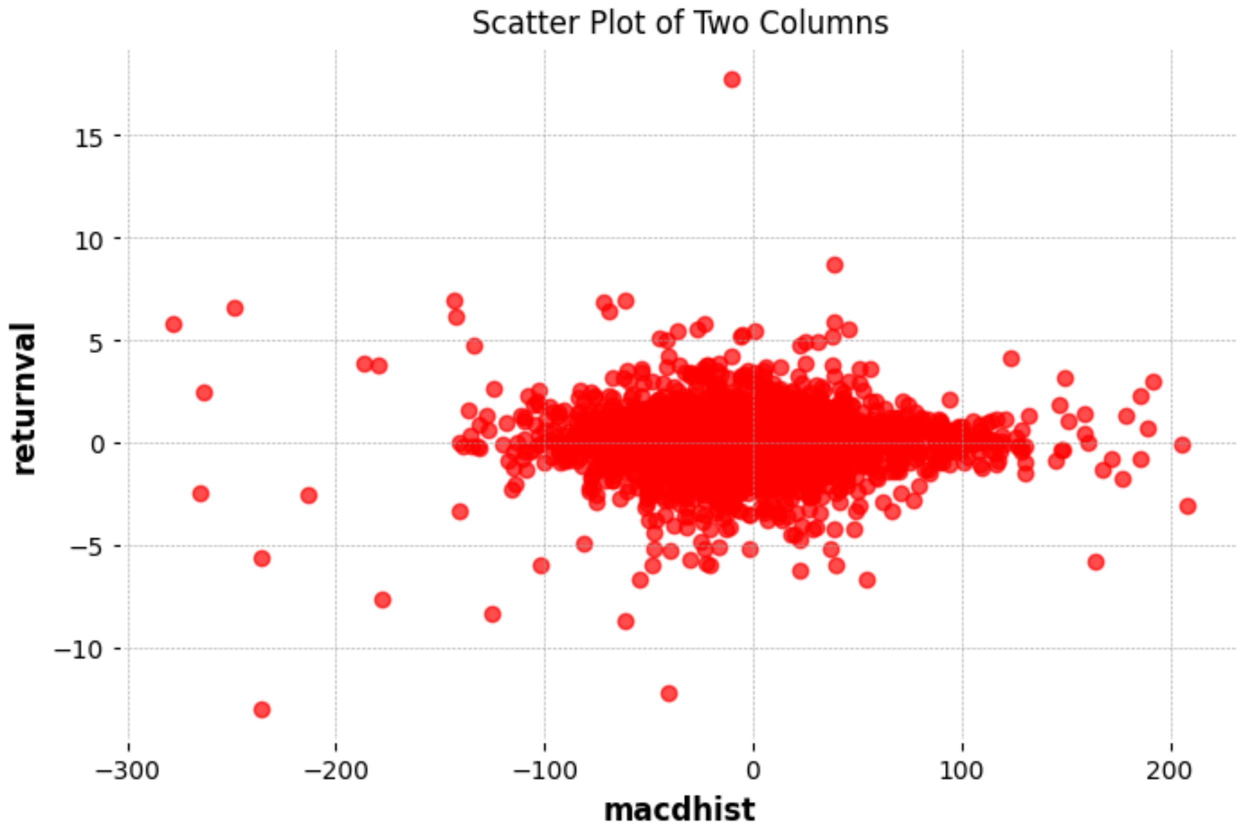

Relative Strength Index (RSI) Indicator
RSI Value
100
80
60
40
20
0
RSI (14-day)
Overbought Level (70)
Oversold Level (30)
2008
2010
2012
2014
2016
2018
2020
2022
2024
2026
Date


Directional Movement Index (DMI) and ADX
Indicator Value
100
80
60
40
20
0
+DI (Positive Direction)
-DI (Negative Direction)
ADX (Trend Strength)
Trend Threshold (25)
2008
2010
2012
2014
2016
2018
2020
2022
2024
2026
Date

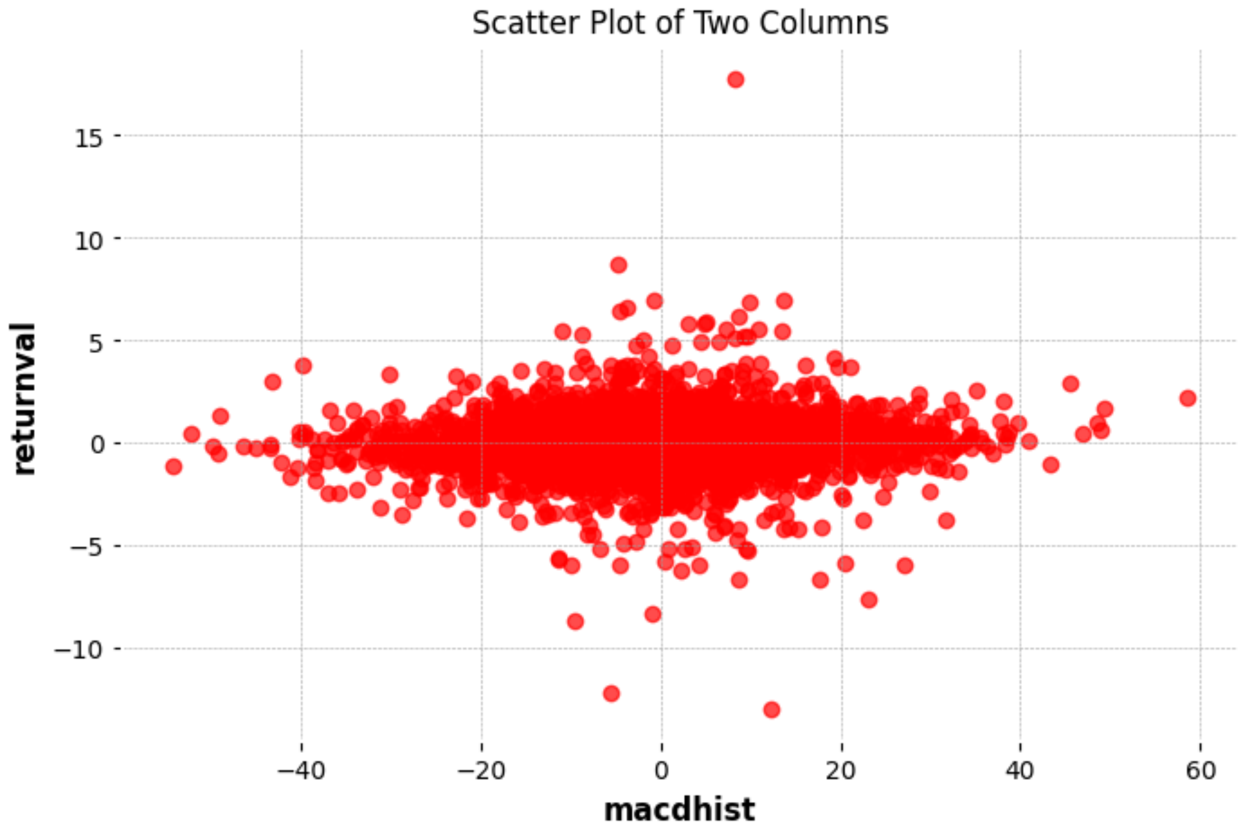
```
dwf3 = dwf.query("returnval > 1")
dwf2 = dwf.query("returnval < 1")
#print(dwf3.shape)
#print(dwf3)
print(dwf3.describe())
#print(dwf2.describe())
```

```
         macdhist    stohist          %K       ulhist    s20delta  \
count  664.000000 664.000000  664.000000   664.000000  664.000000
mean    -7.161371   0.816315   50.674107   796.260311   -0.044662
std     47.660760  12.587738   31.933671   618.610246    0.413454
min   -278.297727 -49.012425    0.700226   100.392060   -2.289764
25%    -29.897861  -6.246885   20.315736   428.023476   -0.222069
50%     -5.996677   0.273087   52.347063   619.027346   -0.006560
75%     19.004366   7.932120   80.821970   961.916752    0.194281
max    191.501328  58.585683   99.046619  5575.686635    1.369179

          didelta   adxdelta         RSI    rsidelta   returnval
count  664.000000 664.000000  664.000000  664.000000  664.000000
mean    -4.789528   0.852474   49.471581    1.830549    1.896643
std     10.420333   8.801471   17.262794   16.081316    1.194550
min    -43.132145 -28.906886    2.881911  -55.483854    1.000121
25%    -11.569181  -5.242472   37.399261   -8.172866    1.212070
50%     -5.084693   0.180942   48.908142    0.643459    1.575067
75%      1.999972   6.483784   62.679611    9.748320    2.112854
max     27.111682  39.924993   92.546362   73.547880   17.744066
```

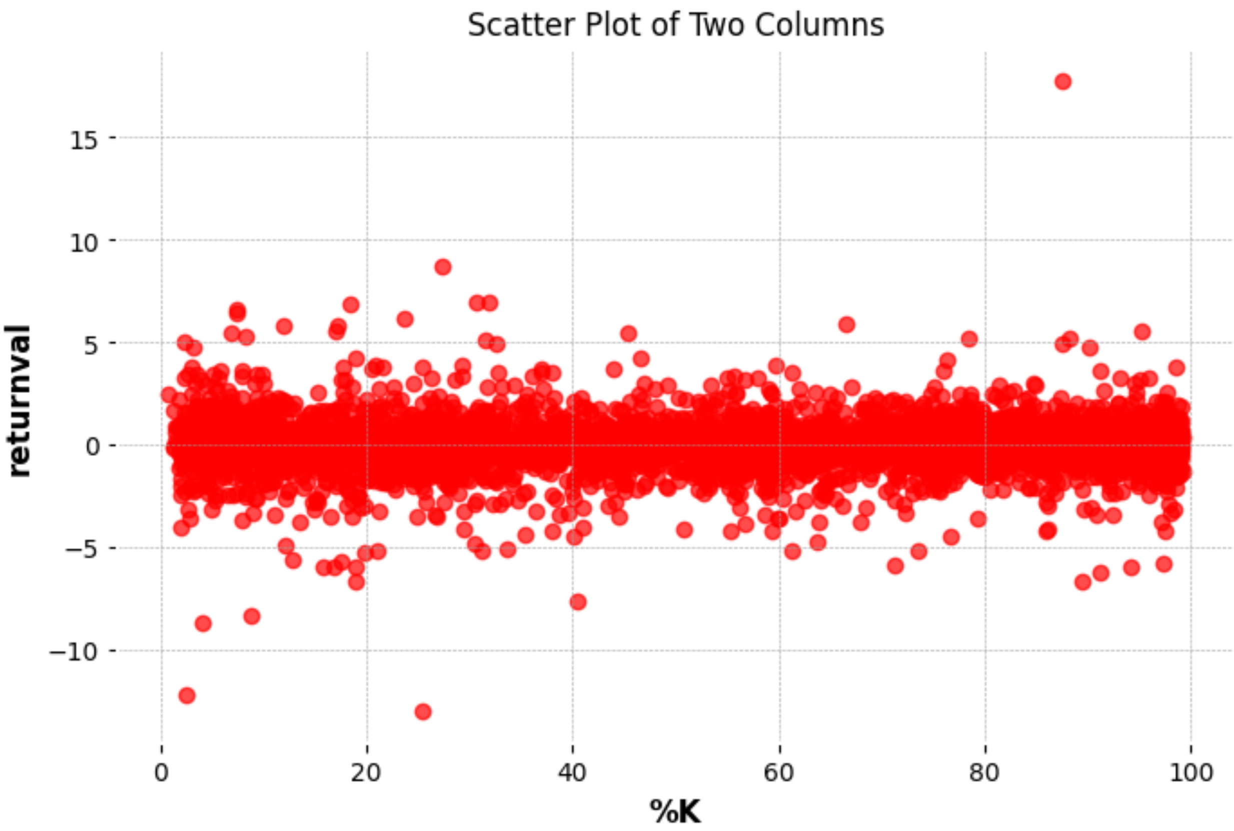
```
plt.figure(figsize=(8, 5))
plt.scatter(dwf['macdhist'], dwf['returnval'], color='red', alpha=0.7
plt.xlabel("macdhist")
plt.ylabel("returnval")
plt.title("Scatter Plot of Two Columns")
plt.grid(True)
plt.show()
```



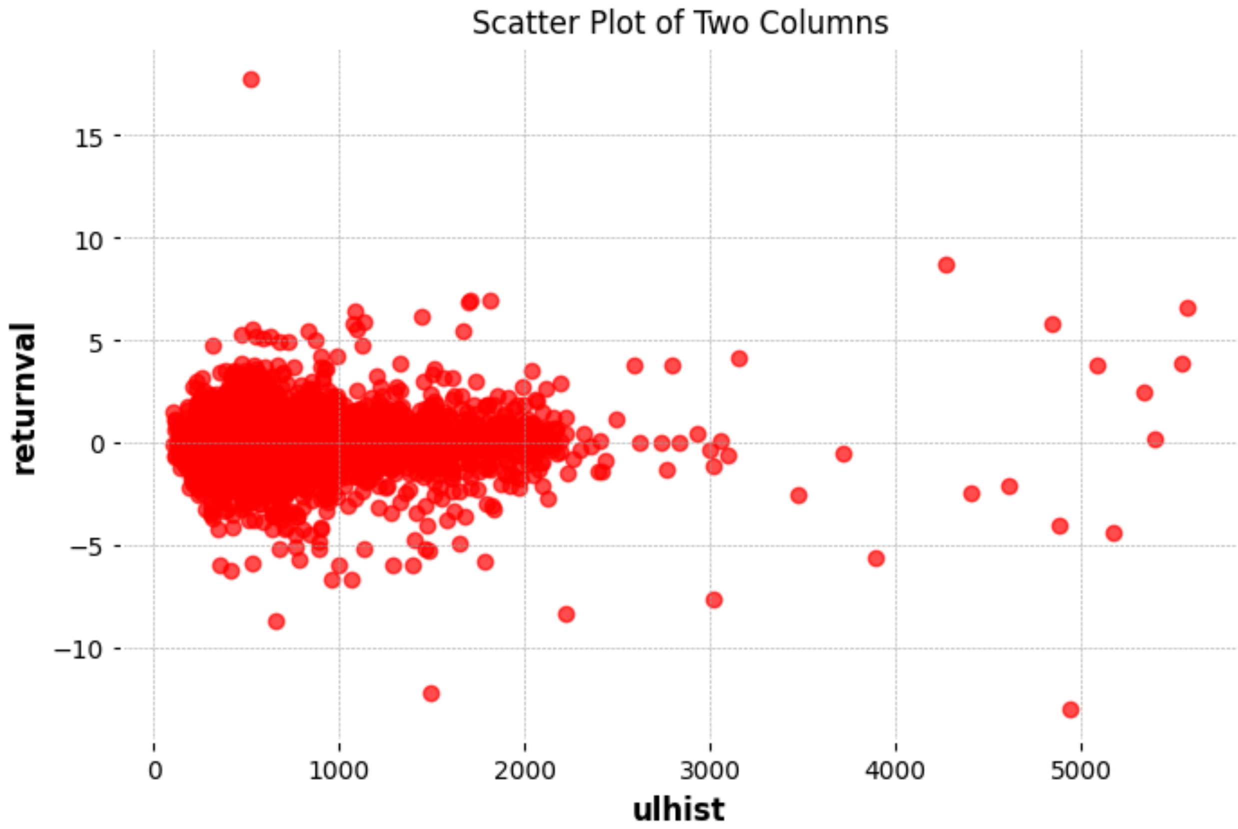
```
plt.figure(figsize=(8, 5))
plt.scatter(dwf['stohist'], dwf['returnval'], color='red', alpha=0.7)
plt.xlabel("macdhist")
plt.ylabel("returnval")
plt.title("Scatter Plot of Two Columns")
plt.grid(True)
plt.show()
```



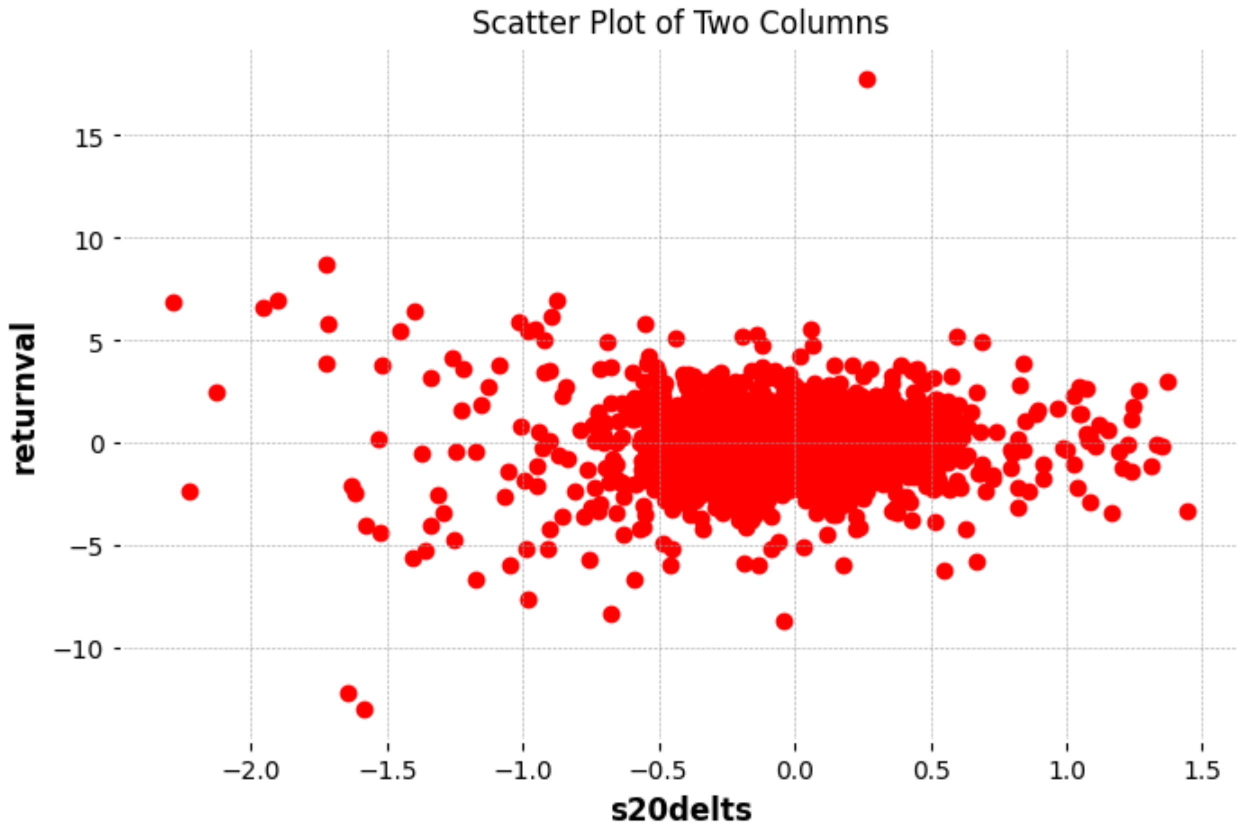
```
plt.figure(figsize=(8, 5))
plt.scatter(dwf['%K'], dwf['returnval'], color='red', alpha=0.7)
plt.xlabel("%K")
plt.ylabel("returnval")
plt.title("Scatter Plot of Two Columns")
plt.grid(True)
plt.show()
```



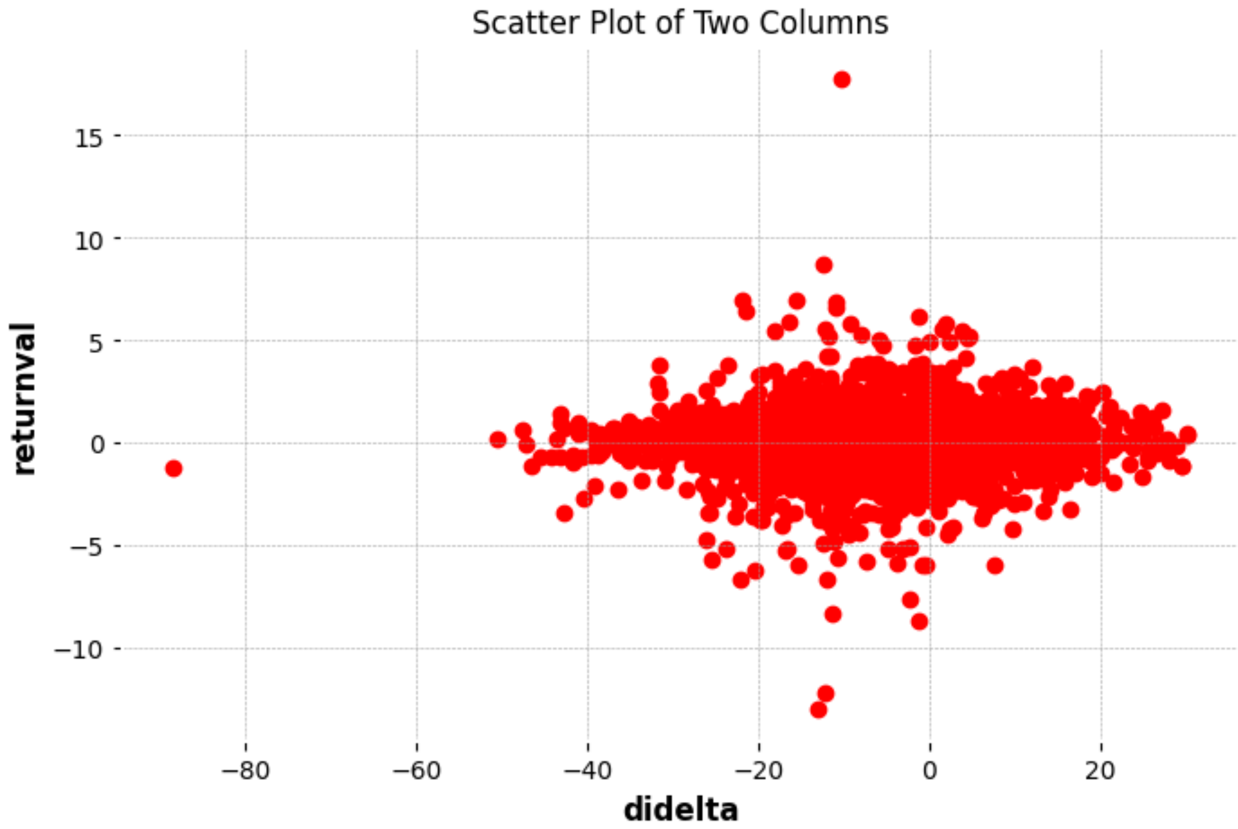
```
plt.figure(figsize=(8, 5))
plt.scatter(dwf['ulhist'], dwf['returnval'], color='red', alpha=0.7)
plt.xlabel("ulhist")
plt.ylabel("returnval")
plt.title("Scatter Plot of Two Columns")
plt.grid(True)
plt.show()
```



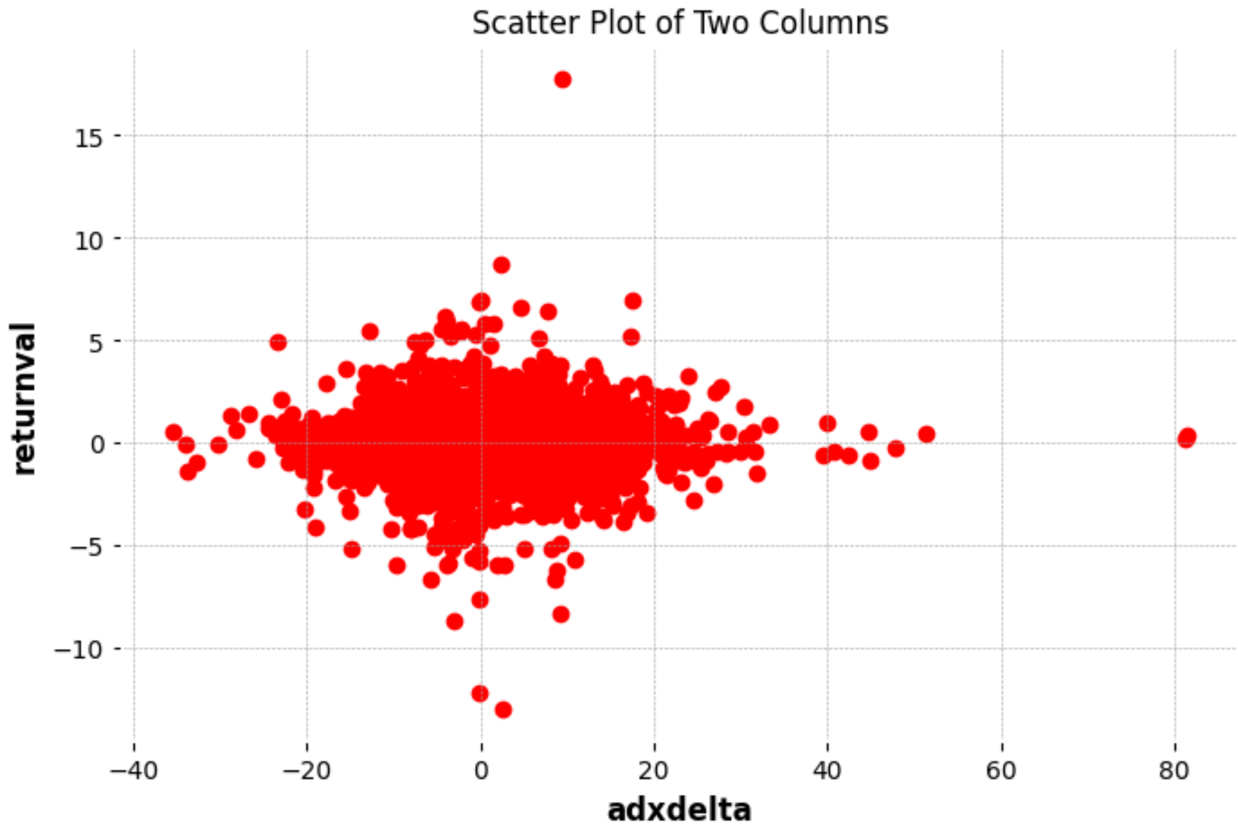
```
plt.figure(figsize=(8, 5))
plt.scatter(dwf['s20delta'], dwf['returnval'], color='red', alpha=1)
plt.xlabel("s20delts")
plt.ylabel("returnval")
plt.title("Scatter Plot of Two Columns")
plt.grid(True)
plt.show()
```



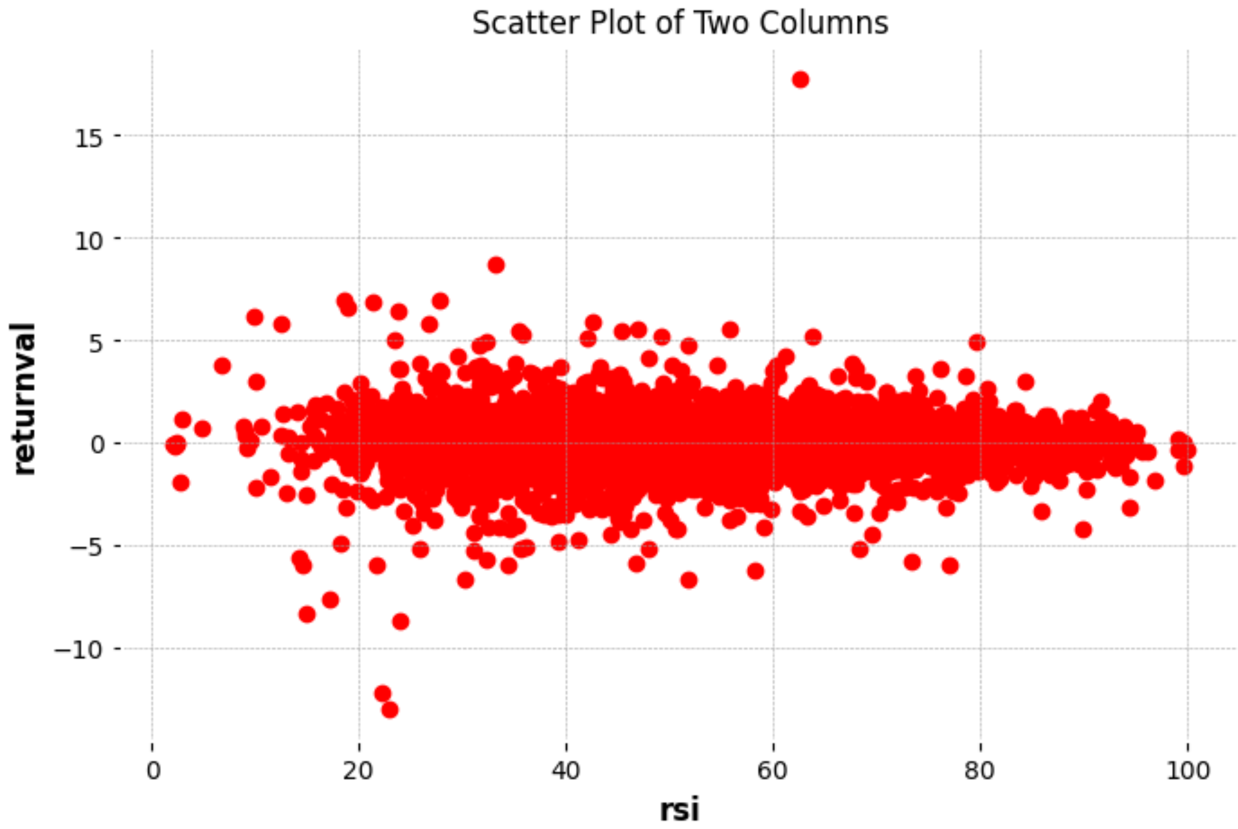
```
plt.figure(figsize=(8, 5))
plt.scatter(dwf['didelta'], dwf['returnval'], color='red', alpha=1)
plt.xlabel("didelta")
plt.ylabel("returnval")
plt.title("Scatter Plot of Two Columns")
plt.grid(True)
plt.show()
```



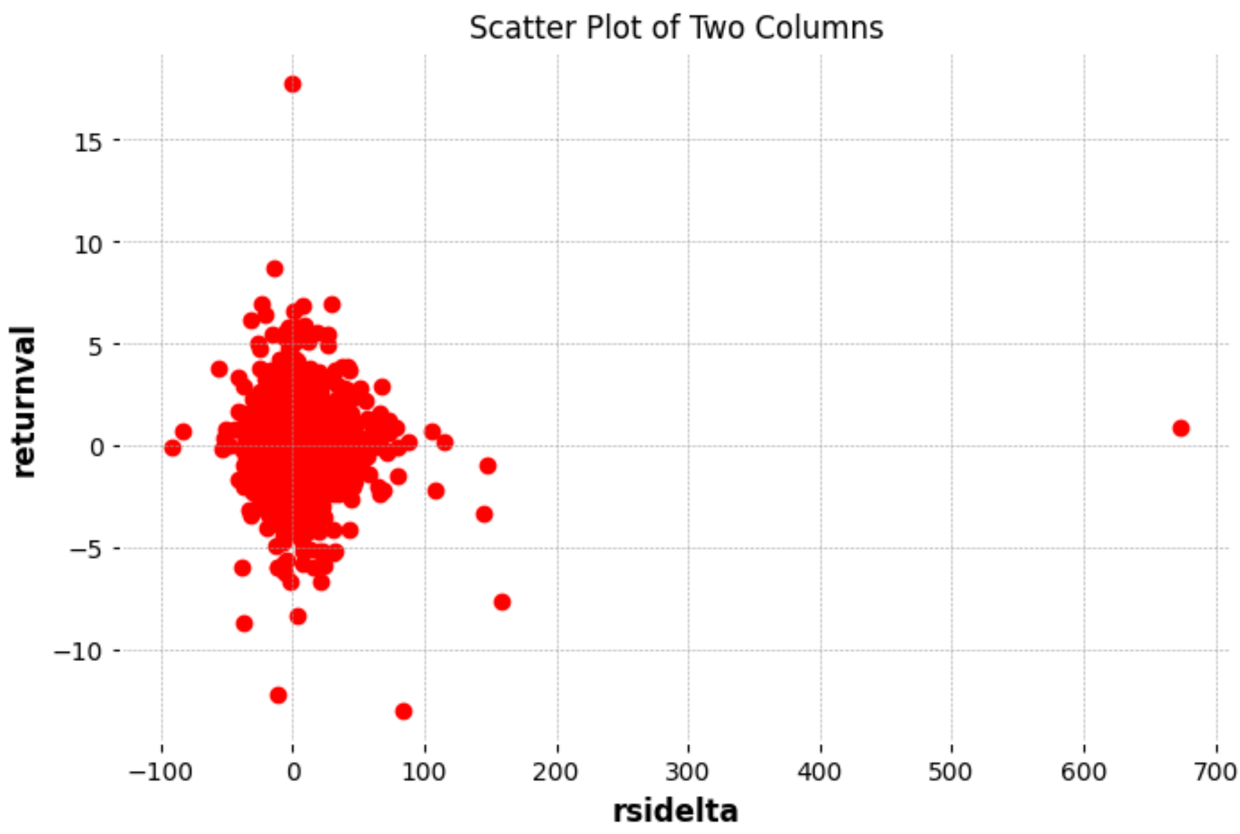
```
plt.figure(figsize=(8, 5))
plt.scatter(dwf['adxdelta'], dwf['returnval'], color='red', alpha=1)
plt.xlabel("adxdelta")
plt.ylabel("returnval")
plt.title("Scatter Plot of Two Columns")
plt.grid(True)
plt.show()
```



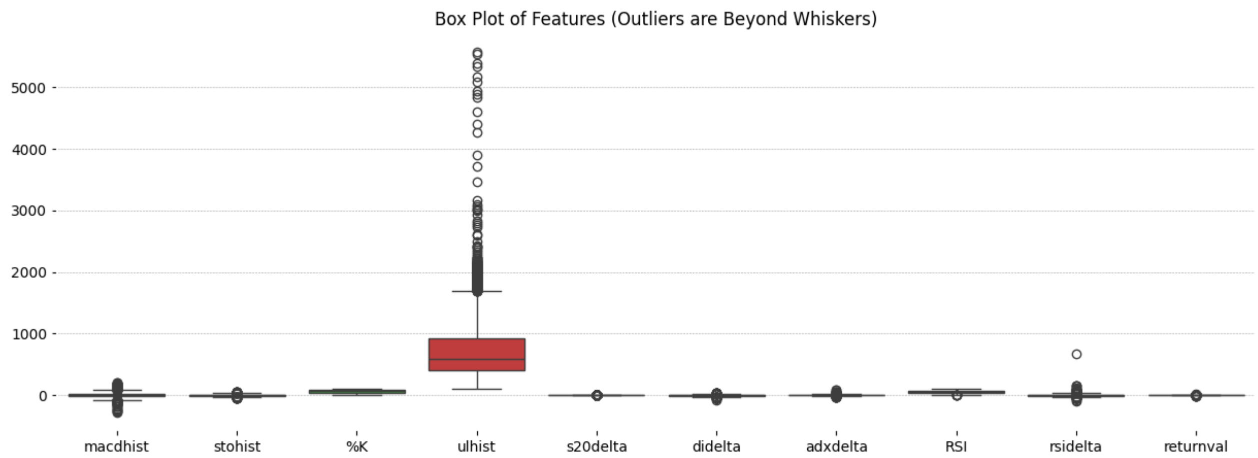
```
plt.figure(figsize=(8, 5))
plt.scatter(dwf['RSI'], dwf['returnval'], color='red', alpha=1)
plt.xlabel("rsi")
plt.ylabel("returnval")
plt.title("Scatter Plot of Two Columns")
plt.grid(True)
plt.show()
```



```
plt.figure(figsize=(8, 5))
plt.scatter(dwf['rsidelta'], dwf['returnval'], color='red', alpha=1)
plt.xlabel("rsidelta")
plt.ylabel("returnval")
plt.title("Scatter Plot of Two Columns")
plt.grid(True)
plt.show()
```



```
plt.figure(figsize=(15, 5))
sns.boxplot(data=dwf)
plt.title("Box Plot of Features (Outliers are Beyond Whiskers)")
plt.show()
```




```
list1 = list(dwf.columns)
print(list1)
lis = np.arange(0,len(list1))
print(lis)
```

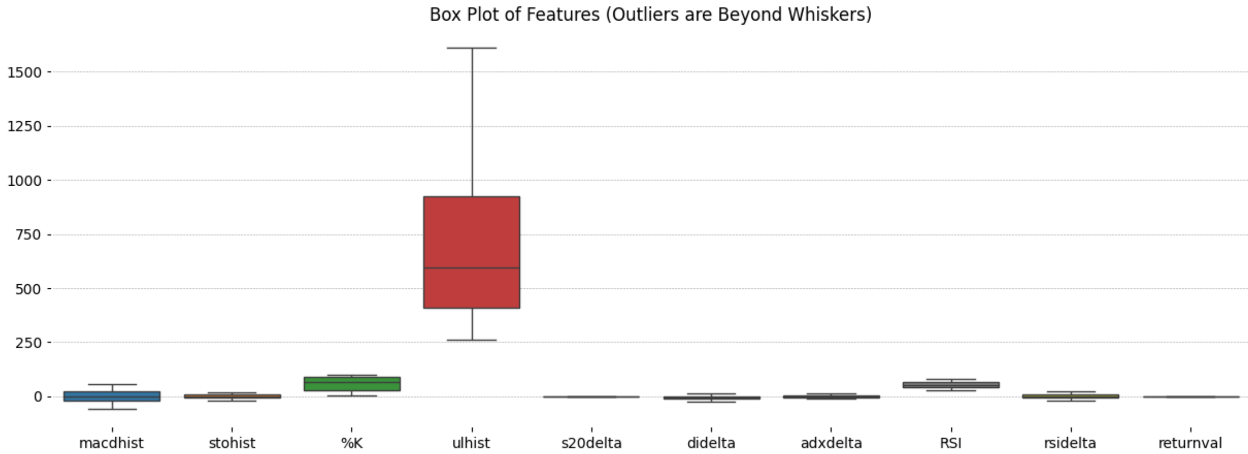
```
['macdhist', 'stohist', '%K', 'ulhist', 's20delta', 'didelta', 'adxdel
[0 1 2 3 4 5 6 7 8 9]
```

```
l = 0.06
h = 0.94

lish = list()
lisl = list()
for x in lis:
  lish.append(dwf[list1[x]].quantile(h))
  lisl.append(dwf[list1[x]].quantile(l))
```

```
print(lish)
print(lisl)
print(dwf.tail(10))
for pos in lis:
  #plt.figure(figsize=(5, 5))
  #sns.boxplot(data=dwf[list1[x]])
  #plt.title("Box Plot of Features (Outliers are Beyond Whiskers)")
  #plt.show()

  dwf[list1[pos]] = np.where(dwf[list1[pos]] > dwf[list1[pos]].quanti
  dwf[list1[pos]] = np.where(dwf[list1[pos]] < dwf[list1[pos]].quanti
  dwf[list1[pos]] = dwf[list1[pos]].astype(float)

  #plt.figure(figsize=(5, 5))
  #sns.boxplot(data=dwf[list1[x]])
  #plt.title("Box Plot of Features (Outliers are Beyond Whiskers)")
  #plt.show()

plt.figure(figsize=(15, 5))
sns.boxplot(data=dwf)
plt.title("Box Plot of Features (Outliers are Beyond Whiskers)")
plt.show()
print(dwf.tail(10))
x = dwf.drop(columns=['returnval'])
y = dwf[['returnval']]
```

```
[np.float64(56.68474054692158), np.float64(19.24734804931645), np.float
[np.float64(-56.59720363595929), np.float64(-19.81546989881195), np.flo
             macdhist    stohist         %K      ulhist  s20delta    di
Date
2025-08-28   9.301231 -27.690975  20.020868  805.011125 -0.064715   0.2
2025-08-29 -12.647863 -14.672621  10.947689  836.369280 -0.086600   0.3
2025-09-01 -12.603292  12.632672  34.433287  836.561509 -0.029006   2.5
2025-09-02 -14.296453   0.441481  23.352709  834.804884  0.002885   6.3
2025-09-03  -5.412682   8.363439  41.438157  834.619009 -0.001559  10.3
2025-09-04   2.231453   7.741990  44.008418  834.193855  0.017159  21.0
2025-09-05   7.891841   1.453074  44.902898  826.482377  0.033767  22.1
2025-09-08  13.618350   3.159940  49.195568  821.575025  0.035819   6.6
2025-09-09  23.094730   9.927216  61.940057  760.864624  0.102220   5.1
2025-09-10  34.949601  13.550060  75.892902  772.755966  0.078421   7.6

             adxdelta        RSI   rsidelta  returnval
Date
2025-08-28 -11.887910  48.119290  -6.968684  -0.302237
2025-08-29 -13.491804  45.769174  -4.883937   0.811407
2025-09-01 -13.574221  56.656402  23.787249  -0.184573
2025-09-02  -3.185070  49.847720 -12.017497   0.551072
2025-09-03 -11.229127  56.227939  12.799420   0.077888
2025-09-04  -7.041717  53.351434  -5.115793   0.027085
2025-09-05  -5.005722  53.208136  -0.268592   0.129946
```

```
2025-09-08   7.007861  46.531249 -12.548620   0.385296
2025-09-09   3.123018  46.234748  -0.637209   0.420209
2025-09-10   0.706698  47.456549   2.642603   0.129740
```




```
              macdhist    stohist        %K      ulhist  s20delta    di
Date
2025-08-28   9.301231 -19.815470  20.020868  805.011125 -0.064715   0.2
2025-08-29 -12.647863 -14.672621  10.947689  836.369280 -0.086600   0.3
2025-09-01 -12.603292  12.632672  34.433287  836.561509 -0.029006   2.5
2025-09-02 -14.296453   0.441481  23.352709  834.804884  0.002885   6.3
2025-09-03  -5.412682   8.363439  41.438157  834.619009 -0.001559  10.3
2025-09-04   2.231453   7.741990  44.008418  834.193855  0.017159  11.5
2025-09-05   7.891841   1.453074  44.902898  826.482377  0.033767  11.5
2025-09-08  13.618350   3.159940  49.195568  821.575025  0.035819   6.6
2025-09-09  23.094730   9.927216  61.940057  760.864624  0.102220   5.1
2025-09-10  34.949601  13.550060  75.892902  772.755966  0.078421   7.6

              adxdelta        RSI   rsidelta  returnval
Date
2025-08-28 -11.887910  48.119290  -6.968684  -0.302237
2025-08-29 -12.227541  45.769174  -4.883937   0.811407
2025-09-01 -12.227541  56.656402  23.350200  -0.184573
2025-09-02  -3.185070  49.847720 -12.017497   0.551072
2025-09-03 -11.229127  56.227939  12.799420   0.077888
2025-09-04  -7.041717  53.351434  -5.115793   0.027085
2025-09-05  -5.005722  53.208136  -0.268592   0.129946
2025-09-08   7.007861  46.531249 -12.548620   0.385296
2025-09-09   3.123018  46.234748  -0.637209   0.420209
2025-09-10   0.706698  47.456549   2.642603   0.129740
```

```
print(dfinal)
l = 0.06
h = 0.94

for pos in lis:
  dfinal[list1[pos]] = np.where(dfinal[list1[pos]] > lish[pos], lish[
  dfinal[list1[pos]] = np.where(dfinal[list1[pos]] < lisl[pos], lisl[
  dfinal[list1[pos]] = dfinal[list1[pos]].astype(float)

#print(dfinal)
print("**-------**")

xfinal = dfinal.drop(columns=['returnval'])
yfinal = dfinal[['returnval']]

print(xfinal)
print("**-------**")
print(yfinal)
```

```
             macdhist    stohist        %K      ulhist  s20delta    di
Date
2025-09-11  42.992900  12.944694  88.33352  757.142450  0.104621  11.3
2025-09-12  52.984073   9.615353  96.53624  793.425713  0.099781  15.6

             adxdelta        RSI   rsidelta  returnval
Date
2025-09-11  14.749225  47.428954  -0.058149   0.433905
2025-09-12  15.339161  58.608633  23.571423        NaN
**-------**
             macdhist    stohist        %K      ulhist  s20delta    di
Date
2025-09-11  42.992900  12.944694  88.33352  757.142450  0.104621  11.3
2025-09-12  52.984073   9.615353  96.53624  793.425713  0.099781  11.5

             adxdelta        RSI   rsidelta
Date
2025-09-11  13.958359  47.428954  -0.058149
2025-09-12  13.958359  58.608633  23.350200
**-------**
            returnval
Date
2025-09-11   0.433905
2025-09-12        NaN
```

LINEAR REGRESSION

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

import numpy as np
from scipy import stats
```

```
# Hyperparameters
train_sizes = [0.6,0.7]  # Different train-test splits
random_states = range(100,500)
#random_states = list(94,)


# Tracking Best Configuration
best_mse = np.inf
best_params = {}

# Loop through train-test splits and random states
for train_size in train_sizes:
    for random_state in random_states:

        # Split Data
        X_train, X_test, y_train, y_test = train_test_split(x, y, tra

        # Train Linear Regression Model
        modellin = LinearRegression()
        modellin.fit(X_train, y_train)

        # Predict and Evaluate using RMSE
        y_pred = modellin.predict(X_test)
        mselin = mean_squared_error(y_test, y_pred)  # RMSE

        # Update Best Configuration if RMSE Improves
        if mselin < best_mse:
            best_mse = mselin
            best_params = {
                'train_size': train_size,
                'random_state': random_state,
                'mse': mselin
            }

print("🎯 Best Configuration Found:")
print(best_params)

ts = best_params['train_size']
rs = best_params['random_state']
print(ts)
print(rs)
```

```
🎯 Best Configuration Found:
{'train_size': 0.7, 'random_state': 365, 'mse': 0.7703089856942993}
0.7
365
```

```
X_train, X_test, y_train, y_test = train_test_split(x, y, train_size=
modelli = LinearRegression()
modelli.fit(X_train, y_train)
y_pred = modelli.predict(X_test)
mselin = mean_squared_error(y_test, y_pred)

print('mse:',mselin)
#print(y_pred)

ypredlog = modellin.predict(xfinal)
print('Y prediction:',ypredlog)
#print(y_test)
print('mse:',mselin)
rmse = np.sqrt(mselin)
print('rmse:',rmse)
```

```
mse: 0.7703089856942993
Y prediction: [[0.19810367]
 [0.16150061]]
mse: 0.7703089856942993
rmse: 0.877672482019517
```

```
print(ypredlog)
print(yfinal)
```

```
[[0.19810367]
 [0.16150061]]
            returnval
Date
2025-09-11   0.433905
2025-09-12        NaN
```

```
# Last observed closing NIFTY value
last_close = nif  # <-- replace with your actual last NIFTY close

predictions = []

for step, pred in enumerate(ypredlog, start=1):
    # Convert percentage return to decimal
    r = pred[0] / 100.0

    # Forecasted close
    forecast_close = last_close * (1 + r)

    # Confidence intervals
    lower_68 = last_close * (1 + r - rmse/100)
    upper_68 = last_close * (1 + r + rmse/100)

    lower_95 = last_close * (1 + r - 1.96*rmse/100)
    upper_95 = last_close * (1 + r + 1.96*rmse/100)

    predictions.append([step, forecast_close, lower_68, upper_68, low

# Put results in a DataFrame
dflog = pd.DataFrame(predictions,
                       columns=["Horizon", "Forecasted Close", "68% L
                                "95% Lower", "95% Upper"])

dflog = dflog.iloc[1:].reset_index(drop=True)
print(dflog)
```

```
   Horizon  Forecasted Close     68% Lower    68% Upper     95% Lower
0        2      25154.559263  24934.140596  25374.97793  24722.538675

     95% Upper
0  25586.57985
```

LASSO RIDGE REGRESSION

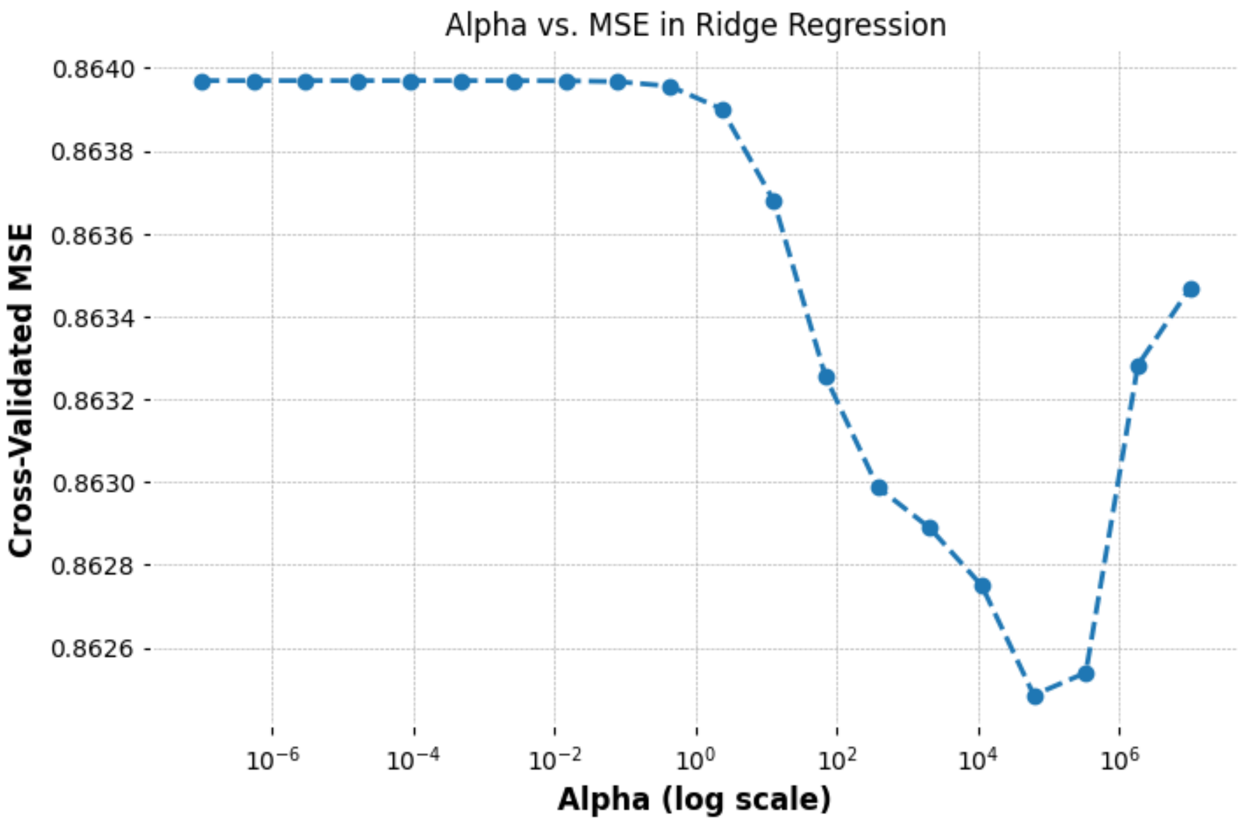
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
```

```
X_train, X_test, y_train, y_test = train_test_split(x, y, train_size=
```

```
# 🔷 Define Alpha Range (Log Scale)
alpha_range = np.logspace(-7, 7, 20)  # Alpha values from 0.001 to 10

# 🔷 Perform Grid Search to Find Best Alpha
ridge = Ridge()
ridge_cv = GridSearchCV(ridge, {'alpha': alpha_range}, scoring='neg_m
ridge_cv.fit(X_train, y_train)

# 🔷 Get Best Alpha & MSE
best_alpha = ridge_cv.best_params_['alpha']
best_mse = -ridge_cv.best_score_

print(f"✅ Best Ridge Alpha: {best_alpha:.4f}")
print(f"📉 Best Cross-Validated MSE: {best_mse:.4f}")

# 🔷 Train Final Model with Best Alpha
best_ridge = Ridge(alpha=best_alpha)
best_ridge.fit(X_train, y_train)
y_predrid = best_ridge.predict(X_test)
final_mse = mean_squared_error(y_test, y_predrid)

print(f"🏆 Final Ridge MSE on Test Data: {final_mse:.4f}")
```

```
✅ Best Ridge Alpha: 61584.8211
📉 Best Cross-Validated MSE: 0.8625
🏆 Final Ridge MSE on Test Data: 0.7713
```

```
import matplotlib.pyplot as plt

alphas = ridge_cv.cv_results_['param_alpha'].data
mse_scores = -ridge_cv.cv_results_['mean_test_score']

plt.figure(figsize=(8, 5))
plt.plot(alphas, mse_scores, marker='o', linestyle='dashed')
plt.xscale('log')
plt.xlabel("Alpha (log scale)")
plt.ylabel("Cross-Validated MSE")
plt.title("Alpha vs. MSE in Ridge Regression")
plt.show()
```




```
xfinal = dfinal.drop(columns=['returnval'])
yfinal = dfinal[['returnval']]
```

```
X_train, X_test, y_train, y_test = train_test_split(x, y, train_size=
modelri = Ridge(alpha=best_alpha)
modelri.fit(X_train, y_train)
y_predri = modelri.predict(X_test)
mserid = mean_squared_error(y_test, y_pred)

print('mse:',mserid)
#print(y_pred)

ypredlas = modelri.predict(xfinal)
print('xfinal prediction:',ypredlas)
#print(y_test)
print('mse:',mserid)
rmse = np.sqrt(mserid)
print('rmse:',rmse)
```

```
mse: 0.7703089856942993
xfinal prediction: [0.08501012 0.09142292]
mse: 0.7703089856942993
rmse: 0.877672482019517
```

```
# Last observed closing NIFTY value
last_close = nif  # <-- replace with your actual last NIFTY close

predictions = []

for step, pred in enumerate(ypredlas, start=1):
    # Convert percentage return to decimal
    r = pred / 100.0

    # Forecasted close
    forecast_close = last_close * (1 + r)

    # Confidence intervals
    lower_68 = last_close * (1 + r - rmse/100)
    upper_68 = last_close * (1 + r + rmse/100)

    lower_95 = last_close * (1 + r - 1.96*rmse/100)
    upper_95 = last_close * (1 + r + 1.96*rmse/100)

    predictions.append([step, forecast_close, lower_68, upper_68, low

# Put results in a DataFrame
dflas = pd.DataFrame(predictions,
                       columns=["Horizon", "Forecasted Close", "68% L
                                "95% Lower", "95% Upper"])

dflas = dflas.iloc[1:].reset_index(drop=True)
print(dflas)
```

```
   Horizon  Forecasted Close     68% Lower    68% Upper     95% Lower
0        2      25136.959953  24916.541286  25357.37862  24704.939366

      95% Upper
0  25568.980541
```

DECISION TREE REGRESSOR

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, RandomizedSearc
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error, r2_score

# -----------------------------
# Example: dataset (replace with your data)
# -----------------------------
```

```
# Suppose X and y are your features and target
# X = ...
# y = ...

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(x, y, train_size=

# ------------------------------
# Step 1: Initialize model
# ------------------------------
tree = DecisionTreeRegressor(random_state=42)

# ------------------------------
# Step 2: Define parameter search space
# ------------------------------
param_dist = {
    "max_depth": [int(x) for x in np.linspace(3, 20, 10)] + [None],
    "min_samples_split": [2, 5, 10, 20, 50],
    "min_samples_leaf": [1, 2, 4, 10, 20],
    "max_features": [None, "sqrt", "log2"],
    "criterion": ["squared_error", "friedman_mse", "absolute_error"]
}

# ------------------------------
# Step 3: RandomizedSearchCV
# ------------------------------
random_search = RandomizedSearchCV(
    estimator=tree,
    param_distributions=param_dist,
    n_iter=50,             # number of random combinations to try
    cv=5,                  # 5-fold cross validation
    scoring="neg_mean_squared_error",
    n_jobs=-1,
    random_state=42,
    verbose=2)

# Fit search
random_search.fit(X_train, y_train)

# ------------------------------
# Step 4: Best model
# ------------------------------
print("Best Parameters:", random_search.best_params_)
print("Best CV Score (MSE):", -random_search.best_score_)

# Best fitted model
best_tree = random_search.best_estimator_
```

```
# -----------------------------
# Step 5: Predictions & Evaluation
# -----------------------------
y_pred_tree = best_tree.predict(X_test)

mse = mean_squared_error(y_test, y_pred_tree)
r2 = r2_score(y_test, y_pred_tree)

print("Test MSE:", mse)
print("Test R²:", r2)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best Parameters: {'min_samples_split': 20, 'min_samples_leaf': 20, 'ma
Best CV Score (MSE): 0.8749736570258912
Test MSE: 0.7705058371197857
Test R²: 0.003295617139308038
```

```
X_train, X_test, y_train, y_test = train_test_split(x, y, train_size=
best_tree = Ridge(alpha=best_alpha)
best_tree.fit(X_train, y_train)
y_preddt = best_tree.predict(X_test)
msedt = mean_squared_error(y_test, y_preddt)

print('mse:',msedt)
#print(y_pred)

ypreddt = best_tree.predict(xfinal)
print('xfinal prediction:',ypreddt)
#print(y_test)
print('mse:',msedt)
rmse = np.sqrt(msedt)
print('rmse:',rmse)
```

```
mse: 0.7712731485936027
xfinal prediction: [0.08501012 0.09142292]
mse: 0.7712731485936027
rmse: 0.8782215828557179
```

```
# Last observed closing NIFTY value
last_close = nif  # <-- replace with your actual last NIFTY close

predictions = []

for step, pred in enumerate(ypreddt, start=1):
    # Convert percentage return to decimal
    r = pred / 100.0

    # Forecasted close
    forecast_close = last_close * (1 + r)

    # Confidence intervals
    lower_68 = last_close * (1 + r - rmse/100)
    upper_68 = last_close * (1 + r + rmse/100)

    lower_95 = last_close * (1 + r - 1.96*rmse/100)
    upper_95 = last_close * (1 + r + 1.96*rmse/100)

    predictions.append([step, forecast_close, lower_68, upper_68, low

# Put results in a DataFrame
dfdt = pd.DataFrame(predictions,
                       columns=["Horizon", "Forecasted Close", "68% L
                                "95% Lower", "95% Upper"])

dfdt = dfdt.iloc[1:].reset_index(drop=True)
print(dfdt)
```

```
   Horizon  Forecasted Close     68% Lower     68% Upper     95% Lower
0        2      25136.959953  24916.403385  25357.516522  24704.669079

      95% Upper
0  25569.250827
```

RANDOM FOREST REGRESSOR

```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
```

```
# Split dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(x, y, train_size=

print("Train size:", X_train.shape, "Test size:", X_test.shape)

# Initialize and train the Random Forest model
rf = RandomForestRegressor(n_estimators=100, random_state=42)  # 100
rf.fit(X_train, y_train)
```

```
Train size: (3066, 9) Test size: (1314, 9)
/usr/local/lib/python3.12/dist-packages/sklearn/base.py:1389: DataConve
  return fit_method(estimator, *args, **kwargs)
▾ RandomForestRegressor
RandomForestRegressor(random_state=42)
```

```
# Predict on the test set
y_pred = rf.predict(X_test)

# Evaluate with common metrics
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"R² Score: {r2:.4f}")
```

```
Mean Squared Error (MSE): 0.8167
R² Score: -0.0565
```

```
yfinal = rf.predict(xfinal)
print('xfinal prediction:',yfinal)
#print(y_test)
print('mse:',mse)
rmse = np.sqrt(mse)
print('rmse:',rmse)
```

```
xfinal prediction: [-0.01353536  0.11082443]
mse: 0.8167329938079065
rmse: 0.9037328110718934
```

```
# Last observed closing NIFTY value
last_close = nif  # <-- replace with your actual last NIFTY close

predictions = []

for step, pred in enumerate(yfinal, start=1):
    # Convert percentage return to decimal
    r = pred / 100.0

    # Forecasted close
    forecast_close = last_close * (1 + r)

    # Confidence intervals
    lower_68 = last_close * (1 + r - rmse/100)
    upper_68 = last_close * (1 + r + rmse/100)

    lower_95 = last_close * (1 + r - 1.96*rmse/100)
    upper_95 = last_close * (1 + r + 1.96*rmse/100)

    predictions.append([step, forecast_close, lower_68, upper_68, low

# Put results in a DataFrame
dfrf = pd.DataFrame(predictions,
                       columns=["Horizon", "Forecasted Close", "68% L
                                "95% Lower", "95% Upper"])

dfrf = dfrf.iloc[1:].reset_index(drop=True)
print(dfrf)
```

```
   Horizon  Forecasted Close    68% Lower     68% Upper    95% Lower  \
0        2      25141.832448  24914.86899  25368.795906  24696.98407

     95% Upper
0  25586.680826
```

BAGGING REGRESSOR

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, RandomizedSearcl
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import BaggingRegressor
from sklearn.metrics import mean_squared_error, r2_score

# -----------------------------
# Example dataset (replace with your X, y)
```

```
# -----------------------------
# X = ...
# y = ...

X_train, X_test, y_train, y_test = train_test_split(x, y, train_size=

# -----------------------------
# Step 1: Define base estimator
# -----------------------------
base_tree = DecisionTreeRegressor(random_state=42)

# -----------------------------
# Step 2: Bagging Regressor
# -----------------------------
bagging = BaggingRegressor(
    estimator=base_tree,
    random_state=42,
    n_jobs=-1
)

# -----------------------------
# Step 3: Define parameter distributions
# -----------------------------
param_dist = {
    "n_estimators": [50, 100, 200, 300],
    "max_samples": [0.5, 0.7, 0.8, 1.0],
    "max_features": [0.5, 0.7, 1.0],
    "bootstrap": [True, False],
    "estimator__max_depth": [3, 5, 7, 10, None],
    "estimator__min_samples_split": [2, 5, 10, 20],
    "estimator__min_samples_leaf": [1, 2, 4, 10]
}

# -----------------------------
# Step 4: RandomizedSearchCV
# -----------------------------
random_search = RandomizedSearchCV(
    estimator=bagging,
    param_distributions=param_dist,
    n_iter=50,                  # number of random combinations
    cv=5,                       # 5-fold cross validation
    scoring="neg_mean_squared_error",
    n_jobs=-1,
    random_state=42,
    verbose=2
)

# Fit search
```

```
random_search.fit(X_train, y_train)

# -----------------------------
# Step 5: Best model
# -----------------------------
print("Best Parameters:", random_search.best_params_)
print("Best CV Score (MSE):", -random_search.best_score_)

best_bagging = random_search.best_estimator_

# -----------------------------
# Step 6: Predictions & Evaluation
# -----------------------------
y_pred_bagging = best_bagging.predict(X_test)

mse = mean_squared_error(y_test, y_pred_bagging)
r2 = r2_score(y_test, y_pred_bagging)

print("Test MSE:", mse)
print("Test R²:", r2)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
/usr/local/lib/python3.12/dist-packages/sklearn/ensemble/_bagging.py:5(
  return column_or_1d(y, warn=True)
Best Parameters: {'n_estimators': 200, 'max_samples': 0.5, 'max_featur(
Best CV Score (MSE): 0.8575539475240536
Test MSE: 0.774225255405518
Test R²: -0.0015157163100778526
```

```
ybag = best_bagging.predict(xfinal)
print('xfinal prediction:',ybag)
#print(y_test)
print('mse:',mse)
rmse = np.sqrt(mse)
print('rmse:',rmse)
```

```
xfinal prediction: [0.06434036 0.1492062 ]
mse: 0.774225255405518
rmse: 0.8799007076969071
```

```
# Last observed closing NIFTY value
last_close = nif  # <-- replace with your actual last NIFTY close

predictions = []

for step, pred in enumerate(ybag, start=1):
    # Convert percentage return to decimal
    r = pred / 100.0

    # Forecasted close
    forecast_close = last_close * (1 + r)

    # Confidence intervals
    lower_68 = last_close * (1 + r - rmse/100)
    upper_68 = last_close * (1 + r + rmse/100)

    lower_95 = last_close * (1 + r - 1.96*rmse/100)
    upper_95 = last_close * (1 + r + 1.96*rmse/100)

    predictions.append([step, forecast_close, lower_68, upper_68, low

# Put results in a DataFrame
dfbag = pd.DataFrame(predictions,
                     columns=["Horizon", "Forecasted Close", "68% L
                              "95% Lower", "95% Upper"])

dfbag = dfbag.iloc[1:].reset_index(drop=True)
print(dfbag)
```

```
   Horizon  Forecasted Close     68% Lower    68% Upper     95% Lower
0        2      25151.471646  24930.493382  25372.44991  24718.354249

     95% Upper
0  25584.589043
```

TIME SERIES FORECASTING

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from statsmodels.tsa.arima.model import ARIMA
from prophet import Prophet
from sklearn.model_selection import train_test_split
```

```
df = dw['Close']

# Visualize the data
plt.figure(figsize=(10,5))
plt.plot(df, label="Time Series Data")
plt.legend()
plt.show()
```

```
import pandas as pd
import numpy as np
import yfinance as yf
import matplotlib.pyplot as plt
#!pip install arch
from arch import arch_model
```

```
Collecting arch
  Downloading arch-7.2.0-cp312-cp312-manylinux_2_17_x86_64.manylinux201
Requirement already satisfied: numpy>=1.22.3 in /usr/local/lib/python3
Requirement already satisfied: scipy>=1.8 in /usr/local/lib/python3.12/
Requirement already satisfied: pandas>=1.4 in /usr/local/lib/python3.1
Requirement already satisfied: statsmodels>=0.12 in /usr/local/lib/pyt
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/li
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python
Requirement already satisfied: patsy>=0.5.6 in /usr/local/lib/python3.
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/pytho
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/d
Downloading arch-7.2.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_
                                        978.3/978.3 kB 12.9 MB/s e
Installing collected packages: arch
Successfully installed arch-7.2.0
```

```
returns = dw['returnval']

model = arch_model(returns, vol="GARCH", p=1, q=1, mean="AR", lags=1)
res = model.fit(disp="off")

# ---------------------------
# 3. Forecast next 8 steps
# ---------------------------
forecast = res.forecast(horizon=8, reindex=False)

# Extract mean returns and volatility forecasts
mean_forecast = forecast.mean.values[-1]      # shape (1,8)
var_forecast = forecast.variance.values[-1]   # shape (1,8)
vol_forecast = np.sqrt(var_forecast)

# ---------------------------
# 4. Convert forecasted returns → closing prices
# ---------------------------
last_close = 24868.6
predictions = []

for h in range(8):
    exp_return = mean_forecast[h] / 100  # convert % back to decimal
    exp_price = last_close * np.exp(exp_return)  # point forecast

    # 95% confidence interval (±1.96 sigma)
    lower = last_close * np.exp(exp_return - 1.96 * vol_forecast[h]/100
    upper = last_close * np.exp(exp_return + 1.96 * vol_forecast[h]/100

    predictions.append([h+1, exp_price, lower, upper])

# ---------------------------
# 5. Put into DataFrame
# ---------------------------
pred_df = pd.DataFrame(predictions,
                       columns=["Horizon", "Forecasted Close", "95% Low
print(pred_df)
```

```
   Horizon  Forecasted Close     95% Lower     95% Upper
0        1      24892.003128  24607.041763  25180.264484
1        2      24887.198241  24596.529667  25181.301780
2        3      24886.925326  24591.074456  25186.335527
3        4      24886.909823  24586.006325  25191.496023
4        5      24886.908943  24581.077042  25196.545931
5        6      24886.908893  24576.266684  25201.477596
6        7      24886.908890  24571.568714  25206.295996
7        8      24886.908890  24566.977789  25211.006393
```

```
print(dflog)
```

```
   Horizon  Forecasted Close     68% Lower    68% Upper     95% Lower
0        2      25154.559263  24934.140596  25374.97793  24722.538675

     95% Upper
0  25586.57985
```

```
print(dflas)
```

```
   Horizon  Forecasted Close     68% Lower    68% Upper     95% Lower
0        2      25136.959953  24916.541286  25357.37862  24704.939366

      95% Upper
0  25568.980541
```

```
print(dfdt)
```

```
   Horizon  Forecasted Close     68% Lower     68% Upper     95% Lower
0        2      25136.959953  24916.403385  25357.516522  24704.669079

      95% Upper
0  25569.250827
```

```
print(dfrf)
```

```
   Horizon  Forecasted Close    68% Lower     68% Upper    95% Lower  \
0        2      25141.832448  24914.86899  25368.795906  24696.98407

      95% Upper
0  25586.680826
```

```
print(dfbag)
```

```
   Horizon  Forecasted Close     68% Lower    68% Upper     95% Lower
0        2      25151.471646  24930.493382  25372.44991  24718.354249

      95% Upper
0  25584.589043
```

NEURAL NETWORK

```
import numpy as np
```

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import mean_squared_error, mean_absolute_error

# -------------------------------
# 1. Load your dataset
# -------------------------------
# Assuming df is your DataFrame from the earlier steps in your notebo
# and it has a "Close" column and engineered features.

x = dwf.drop(columns=['returnval'])
y = dwf[['returnval']]                  # target closing price

# Scale features
#scaler = MinMaxScaler()
#X_scaled = scaler.fit_transform(X)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0

# -------------------------------
# 2. Build Neural Network Model
# -------------------------------
model = Sequential()
model.add(Dense(128, input_dim=X_train.shape[1], activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='linear'))  # regression output

model.compile(optimizer='adam', loss='mse', metrics=['mae'])

# -------------------------------
# 3. Train Model
# -------------------------------
es = EarlyStopping(monitor='val_loss', patience=30, restore_best_weig

history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=100,
    batch_size=32,
    callbacks=[es],
```

```
    verbose=2
)

# ------------------------------
# 4. Evaluate Model
# ------------------------------
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mse)

print(f"MSE: {mse:.4f}, RMSE: {rmse:.4f}, MAE: {mae:.4f}")

# ------------------------------
# 5. Forecast Next NIFTY Closing
# ------------------------------
# Take the last row (latest features) and predict next close
latest_features = x.iloc[-1].values.reshape(1, -1)
next_close = model.predict(latest_features)[0][0]

print(f"Forecasted next NIFTY close: {next_close:.2f}")
```

```
110/110 - 1s - 6ms/step - loss: 0.9542 - mae: 0.7820 - val_loss: 0.57
Epoch 43/100
110/110 - 1s - 6ms/step - loss: 0.9508 - mae: 0.7803 - val_loss: 0.65
Epoch 44/100
110/110 - 0s - 3ms/step - loss: 0.9297 - mae: 0.7729 - val_loss: 0.57
Epoch 45/100
110/110 - 0s - 3ms/step - loss: 0.9430 - mae: 0.7818 - val_loss: 0.58
Epoch 46/100
110/110 - 1s - 6ms/step - loss: 0.9435 - mae: 0.7789 - val_loss: 0.58
Epoch 47/100
110/110 - 0s - 3ms/step - loss: 0.9423 - mae: 0.7770 - val_loss: 0.67
Epoch 48/100
110/110 - 1s - 6ms/step - loss: 0.9556 - mae: 0.7797 - val_loss: 0.65
Epoch 49/100
110/110 - 0s - 3ms/step - loss: 0.9748 - mae: 0.7821 - val_loss: 0.57
Epoch 50/100
110/110 - 0s - 4ms/step - loss: 0.9275 - mae: 0.7739 - val_loss: 0.57
Epoch 51/100
110/110 - 0s - 3ms/step - loss: 0.9302 - mae: 0.7734 - val_loss: 0.57
Epoch 52/100
110/110 - 1s - 6ms/step - loss: 0.9363 - mae: 0.7749 - val_loss: 0.58
Epoch 53/100
110/110 - 1s - 6ms/step - loss: 0.9093 - mae: 0.7669 - val_loss: 0.59
Epoch 54/100
110/110 - 1s - 6ms/step - loss: 0.9280 - mae: 0.7717 - val_loss: 0.57
Epoch 55/100
110/110 - 1s - 5ms/step - loss: 0.9175 - mae: 0.7702 - val_loss: 0.58
Epoch 56/100
110/110 - 0s - 3ms/step - loss: 0.9242 - mae: 0.7716 - val_loss: 0.58
```

```
Epoch 57/100
110/110 - 0s - 3ms/step - loss: 0.9630 - mae: 0.7739 - val_loss: 0.65
Epoch 58/100
110/110 - 1s - 6ms/step - loss: 0.9285 - mae: 0.7716 - val_loss: 0.59
Epoch 59/100
110/110 - 0s - 3ms/step - loss: 0.9217 - mae: 0.7702 - val_loss: 0.57
Epoch 60/100
110/110 - 0s - 3ms/step - loss: 0.9090 - mae: 0.7674 - val_loss: 0.57
Epoch 61/100
110/110 - 1s - 7ms/step - loss: 0.9133 - mae: 0.7683 - val_loss: 0.57
Epoch 62/100
110/110 - 1s - 6ms/step - loss: 0.9196 - mae: 0.7685 - val_loss: 0.57
Epoch 63/100
110/110 - 1s - 6ms/step - loss: 0.9070 - mae: 0.7651 - val_loss: 0.57
Epoch 64/100
110/110 - 1s - 6ms/step - loss: 0.9275 - mae: 0.7677 - val_loss: 0.57
Epoch 65/100
110/110 - 1s - 6ms/step - loss: 0.9094 - mae: 0.7665 - val_loss: 0.57
Epoch 66/100
110/110 - 1s - 5ms/step - loss: 0.9093 - mae: 0.7660 - val_loss: 0.57
Epoch 67/100
110/110 - 0s - 4ms/step - loss: 0.9079 - mae: 0.7661 - val_loss: 0.57
Epoch 68/100
110/110 - 1s - 6ms/step - loss: 0.9038 - mae: 0.7645 - val_loss: 0.57
Epoch 69/100
110/110 - 0s - 3ms/step - loss: 0.9038 - mae: 0.7650 - val_loss: 0.57
Epoch 70/100
110/110 - 0s - 3ms/step - loss: 0.9019 - mae: 0.7639 - val_loss: 0.57
28/28 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
```

Double-click (or enter) to edit

```
import numpy as np

# Inputs
last_close = nif      # replace with your actual last observed NIFTY (
pred_return = next_close       # model output in percent
rmse = rmse            # RMSE in percent units

# Expected close
expected_close = last_close * (1 + pred_return/100)

# Confidence intervals
lower_68 = last_close * (1 + (pred_return - rmse)/100)
upper_68 = last_close * (1 + (pred_return + rmse)/100)

lower_95 = last_close * (1 + (pred_return - 1.96*rmse)/100)
upper_95 = last_close * (1 + (pred_return + 1.96*rmse)/100)

print(f"Expected NIFTY close: {expected_close:.2f}")
print(f"68% CI: {lower_68:.2f} – {upper_68:.2f}")
print(f"95% CI: {lower_95:.2f} – {upper_95:.2f}")
```

```
Expected NIFTY close: 25123.06
68% CI: 24933.11 – 25313.00
95% CI: 24750.77 – 25495.34
```

LSTM NEURAL NETWORK

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping

# -------------------------------
# 1. Prepare Data (time series)
# -------------------------------
# Assuming df has a "Close" column (from your earlier preprocessing)

values = dw['returnval'].values.reshape(-1, 1)

# Scale data between 0 and 1
scaler = MinMaxScaler(feature_range=(0, 1))
scaled = scaler.fit_transform(values)
```

```
# Function to create sequences
def create_sequences(data, lookback=60):
    X, y = [], []
    for i in range(len(data) - lookback):
        X.append(data[i:i+lookback, 0])
        y.append(data[i+lookback, 0])
    return np.array(X), np.array(y)

LOOKBACK = 60
X, y = create_sequences(scaled, LOOKBACK)

# Reshape for LSTM: (samples, timesteps, features)
X = X.reshape((X.shape[0], X.shape[1], 1))

# Train-test split
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# -------------------------------
# 2. Build LSTM Model
# -------------------------------
model = Sequential()
model.add(LSTM(64, return_sequences=True, input_shape=(LOOKBACK, 1)))
model.add(Dropout(0.3))
model.add(LSTM(32, return_sequences=False))
model.add(Dropout(0.2))
model.add(Dense(16, activation='relu'))
model.add(Dense(1))  # output: next close value (scaled)

model.compile(optimizer='adam', loss='mse')

# -------------------------------
# 3. Train Model
# -------------------------------
es = EarlyStopping(monitor='val_loss', patience=20, restore_best_weigl

history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=50,
    batch_size=32,
    callbacks=[es],
    verbose=2
)

# -------------------------------
# 4. Evaluate Model
```

```
# -------------------------------
y_pred = model.predict(X_test)

# Inverse scale predictions & actual values
y_test_inv = scaler.inverse_transform(y_test.reshape(-1, 1))
y_pred_inv = scaler.inverse_transform(y_pred)

mse = mean_squared_error(y_test_inv, y_pred_inv)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test_inv, y_pred_inv)

print(f"MSE: {mse:.4f}, RMSE: {rmse:.4f}, MAE: {mae:.4f}")

# -------------------------------
# 5. Forecast Next NIFTY Close
# -------------------------------
last_seq = scaled[-LOOKBACK:].reshape(1, LOOKBACK, 1)
next_pred_scaled = model.predict(last_seq)
next_pred = scaler.inverse_transform(next_pred_scaled)[0][0]

print(f"Forecasted next NIFTY close: {next_pred:.2f}")
```

```
Epoch 23/50
109/109 - 11s - 103ms/step - loss: 0.0021 - val_loss: 8.1201e-04
Epoch 24/50
109/109 - 7s - 68ms/step - loss: 0.0021 - val_loss: 8.0017e-04
Epoch 25/50
109/109 - 9s - 83ms/step - loss: 0.0021 - val_loss: 7.4944e-04
Epoch 26/50
109/109 - 10s - 87ms/step - loss: 0.0021 - val_loss: 7.8630e-04
Epoch 27/50
109/109 - 12s - 113ms/step - loss: 0.0021 - val_loss: 7.5808e-04
Epoch 28/50
109/109 - 10s - 87ms/step - loss: 0.0021 - val_loss: 7.5095e-04
Epoch 29/50
109/109 - 6s - 51ms/step - loss: 0.0021 - val_loss: 7.6129e-04
Epoch 30/50
109/109 - 7s - 67ms/step - loss: 0.0021 - val_loss: 7.4988e-04
Epoch 31/50
109/109 - 6s - 51ms/step - loss: 0.0021 - val_loss: 7.7325e-04
Epoch 32/50
109/109 - 8s - 69ms/step - loss: 0.0021 - val_loss: 7.9506e-04
Epoch 33/50
109/109 - 8s - 75ms/step - loss: 0.0021 - val_loss: 7.6074e-04
Epoch 34/50
109/109 - 10s - 94ms/step - loss: 0.0021 - val_loss: 7.5945e-04
Epoch 35/50
109/109 - 11s - 98ms/step - loss: 0.0021 - val_loss: 8.6529e-04
Epoch 36/50
109/109 - 11s - 105ms/step - loss: 0.0021 - val_loss: 7.4955e-04
```

```
Epoch 37/50
109/109 - 6s - 53ms/step - loss: 0.0021 - val_loss: 7.7969e-04
Epoch 38/50
109/109 - 7s - 68ms/step - loss: 0.0021 - val_loss: 7.5540e-04
Epoch 39/50
109/109 - 6s - 50ms/step - loss: 0.0021 - val_loss: 7.5468e-04
Epoch 40/50
109/109 - 7s - 61ms/step - loss: 0.0021 - val_loss: 7.6876e-04
Epoch 41/50
109/109 - 9s - 85ms/step - loss: 0.0021 - val_loss: 7.5111e-04
Epoch 42/50
109/109 - 8s - 71ms/step - loss: 0.0021 - val_loss: 7.9870e-04
Epoch 43/50
109/109 - 7s - 63ms/step - loss: 0.0021 - val_loss: 7.4942e-04
Epoch 44/50
109/109 - 7s - 68ms/step - loss: 0.0021 - val_loss: 7.4934e-04
Epoch 45/50
109/109 - 8s - 76ms/step - loss: 0.0021 - val_loss: 7.6690e-04
Epoch 46/50
109/109 - 7s - 65ms/step - loss: 0.0021 - val_loss: 7.6598e-04
Epoch 47/50
109/109 - 10s - 88ms/step - loss: 0.0021 - val_loss: 7.5038e-04
Epoch 48/50
109/109 - 9s - 86ms/step - loss: 0.0021 - val_loss: 7.5548e-04
Epoch 49/50
109/109 - 7s - 68ms/step - loss: 0.0021 - val_loss: 7.4979e-04
Epoch 50/50
109/109 - 6s - 52ms/step - loss: 0.0021 - val_loss: 7.5944e-04
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step
MSE: 0.7074, RMSE: 0.8411, MAE: 0.6161
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
```

```
import numpy as np

# Inputs
last_close = nif      # replace with your actual last observed NIFTY cl
pred_return = next_pred       # model output in percent
rmse = rmse            # RMSE in percent units

# Expected close
expected_close = last_close * (1 + pred_return/100)

# Confidence intervals
lower_68 = last_close * (1 + (pred_return - rmse)/100)
upper_68 = last_close * (1 + (pred_return + rmse)/100)

lower_95 = last_close * (1 + (pred_return - 1.96*rmse)/100)
upper_95 = last_close * (1 + (pred_return + 1.96*rmse)/100)

print(f"Expected NIFTY close: {expected_close:.2f}")
print(f"68% CI: {lower_68:.2f} – {upper_68:.2f}")
print(f"95% CI: {lower_95:.2f} – {upper_95:.2f}")
```

```
Expected NIFTY close: 25124.45
68% CI: 24913.22 – 25335.67
95% CI: 24710.45 – 25538.44
```