GWP 2

Linear Discriminant Analysis

In [1]:
```
#import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import yfinance as yf

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
```

In [2]:
```
#import data
data_df = yf.download(["AAPL","AMZN","CSCO","GOOGL","IBM","MSFT","NVDA"],
                      start="2010-01-01", end="2021-12-31")["Adj Close"]

data_df.head()
```

```
[*********************100%***********************]  7 of 7 completed
```

Out[2]:

Ticker	AAPL	AMZN	CSCO	GOOGL	IBM	MSFT	NVDA
Date							
2010-01-04 00:00:00+00:00	6.454505	6.6950	16.601284	15.645692	75.353798	23.347315	0.423952
2010-01-05 00:00:00+00:00	6.465665	6.7345	16.527323	15.576794	74.443512	23.354868	0.430143
2010-01-06 00:00:00+00:00	6.362821	6.6125	16.419739	15.184124	73.959923	23.211533	0.432894
2010-01-07 00:00:00+00:00	6.351058	6.5000	16.493696	14.830644	73.703903	22.970142	0.424410
2010-01-08 00:00:00+00:00	6.393283	6.6760	16.581108	15.028353	74.443512	23.128548	0.425328

In [3]:
```
# Adjust date
data_df["Date"] = pd.to_datetime(data_df.index)
data_df["Date"] = data_df["Date"].dt.date
data_df.set_index("Date",inplace=True)
data_df.head(3)
```

Out[3]:

Ticker Date	AAPL	AMZN	CSCO	GOOGL	IBM	MSFT	NVDA
2010-01-04	6.454505	6.6950	16.601284	15.645692	75.353798	23.347315	0.423952
2010-01-05	6.465665	6.7345	16.527323	15.576794	74.443512	23.354868	0.430143
2010-01-06	6.362821	6.6125	16.419739	15.184124	73.959923	23.211533	0.432894

In [4]:
```
# calculate returns
returns_df = data_df.pct_change().dropna()
returns_df.head()
```

Out[4]:

Ticker Date	AAPL	AMZN	CSCO	GOOGL	IBM	MSFT	NVDA
2010-01-05	0.001729	0.005900	-0.004455	-0.004404	-0.012080	0.000324	0.014603
2010-01-06	-0.015906	-0.018116	-0.006509	-0.025209	-0.006496	-0.006137	0.006396
2010-01-07	-0.001849	-0.017013	0.004504	-0.023280	-0.003462	-0.010400	-0.019597
2010-01-08	0.006648	0.027077	0.005300	0.013331	0.010035	0.006896	0.002161
2010-01-11	-0.008822	-0.024041	-0.002838	-0.001512	-0.010470	-0.012720	-0.014016

In [5]:
```
# covariance matrix
cov = np.cov(returns_df, rowvar=False)
cov = pd.DataFrame(cov, index=returns_df.columns, columns=returns_df.columns)
cov
```

Out[5]:

Ticker Ticker	AAPL	AMZN	CSCO	GOOGL	IBM	MSFT	NVDA
AAPL	0.000312	0.000158	0.000136	0.000152	0.000102	0.000157	0.000222
AMZN	0.000158	0.000386	0.000125	0.000184	0.000092	0.000165	0.000223
CSCO	0.000136	0.000125	0.000279	0.000124	0.000123	0.000147	0.000197
GOOGL	0.000152	0.000184	0.000124	0.000265	0.000100	0.000158	0.000207
IBM	0.000102	0.000092	0.000123	0.000100	0.000199	0.000110	0.000137
MSFT	0.000157	0.000165	0.000147	0.000158	0.000110	0.000249	0.000227
NVDA	0.000222	0.000223	0.000197	0.000207	0.000137	0.000227	0.000722

In [6]:
```
# Create a new column called Target that defines strategy to take long position for
# Let 1 depict returns of NVDA exceeding 2.0% and 0 otherwise
returns_df["Target"] = np.where(
    (returns_df["NVDA"].abs() > 0.02), 1, 0)
```

In [7]:
```
# Checking target proportion
round(returns_df["Target"].sum() / len(returns_df), 4)
```

Out[7]:
```
0.3292
```

In [8]:
```
# define target variable (NVDA) as y and features (independent variables) as X
X, y = returns_df.iloc[:, 0:-2], returns_df.iloc[:, -1]
```

```
print(X.shape, y.shape)
```

```
(3019, 6) (3019,)
```

In [9]:
```
# split data into train and test sets using 80/20
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=int(len(y) * 0.2), shuffle=False
)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(2416, 6) (603, 6) (2416,) (603,)
```

In [10]:
```
# conduct LDA
lda = LinearDiscriminantAnalysis()
lda.fit(X_train, y_train)
```

Out[10]:
```
▾ LinearDiscriminantAnalysis
LinearDiscriminantAnalysis()
```

In [11]:
```
# make predictions on test data
y_pred = lda.predict(X_test)
print(y_pred)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0]
```

In [12]:
```
#check model accuracy
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)
```

In [13]:
```
# print the accuracy and confusion matrix
print(f"Accuracy : {accuracy:.4f}")
print(conf_matrix)
print(class_report)
```

```
Accuracy : 0.5837
[[351   0]
 [251   1]]
              precision    recall  f1-score   support

           0       0.58      1.00      0.74       351
           1       1.00      0.00      0.01       252

    accuracy                           0.58       603
   macro avg       0.79      0.50      0.37       603
weighted avg       0.76      0.58      0.43       603
```

SVM (Support Vector machine)

For our implementation we are going to do a simple BTC Price Movement Classification using SVM to predict whether a cryptocurrency's price (e.g., Bitcoin) will move up(1) or down(0) based on historical price data and technical indicators.

Installing necessary data sources and computations api's

In [14]:
```
!pip install yfinance
!pip install pandas_ta
```

```
Requirement already satisfied: yfinance in /usr/local/lib/python3.10/dist-packages
(0.2.43)
Requirement already satisfied: pandas>=1.3.0 in /usr/local/lib/python3.10/dist-pac
kages (from yfinance) (2.2.2)
Requirement already satisfied: numpy>=1.16.5 in /usr/local/lib/python3.10/dist-pac
kages (from yfinance) (1.26.4)
Requirement already satisfied: requests>=2.31 in /usr/local/lib/python3.10/dist-pa
ckages (from yfinance) (2.32.3)
Requirement already satisfied: multitasking>=0.0.7 in /usr/local/lib/python3.10/di
st-packages (from yfinance) (0.0.11)
Requirement already satisfied: lxml>=4.9.1 in /usr/local/lib/python3.10/dist-packa
ges (from yfinance) (4.9.4)
Requirement already satisfied: platformdirs>=2.0.0 in /usr/local/lib/python3.10/di
st-packages (from yfinance) (4.3.6)
Requirement already satisfied: pytz>=2022.5 in /usr/local/lib/python3.10/dist-pack
ages (from yfinance) (2024.2)
Requirement already satisfied: frozendict>=2.3.4 in /usr/local/lib/python3.10/dist
-packages (from yfinance) (2.4.4)
Requirement already satisfied: peewee>=3.16.2 in /usr/local/lib/python3.10/dist-pa
ckages (from yfinance) (3.17.6)
Requirement already satisfied: beautifulsoup4>=4.11.1 in /usr/local/lib/python3.1
0/dist-packages (from yfinance) (4.12.3)
Requirement already satisfied: html5lib>=1.1 in /usr/local/lib/python3.10/dist-pac
kages (from yfinance) (1.1)
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.10/dist-pac
kages (from beautifulsoup4>=4.11.1->yfinance) (2.6)
Requirement already satisfied: six>=1.9 in /usr/local/lib/python3.10/dist-packages
(from html5lib>=1.1->yfinance) (1.16.0)
Requirement already satisfied: webencodings in /usr/local/lib/python3.10/dist-pack
ages (from html5lib>=1.1->yfinance) (0.5.1)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.1
0/dist-packages (from pandas>=1.3.0->yfinance) (2.8.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-pa
ckages (from pandas>=1.3.0->yfinance) (2024.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.
10/dist-packages (from requests>=2.31->yfinance) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-pack
ages (from requests>=2.31->yfinance) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dis
t-packages (from requests>=2.31->yfinance) (2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dis
t-packages (from requests>=2.31->yfinance) (2024.8.30)
Collecting pandas_ta
  Downloading pandas_ta-0.3.14b.tar.gz (115 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 115.1/115.1 kB 2.7 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages
(from pandas_ta) (2.2.2)
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-pac
kages (from pandas->pandas_ta) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.1
0/dist-packages (from pandas->pandas_ta) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-pack
ages (from pandas->pandas_ta) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-pa
ckages (from pandas->pandas_ta) (2024.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages
(from python-dateutil>=2.8.2->pandas->pandas_ta) (1.16.0)
Building wheels for collected packages: pandas_ta
  Building wheel for pandas_ta (setup.py) ... done
  Created wheel for pandas_ta: filename=pandas_ta-0.3.14b0-py3-none-any.whl size=2
18909 sha256=9dcd3df85877dd3a68d160df51fa8aa1246b5f881978f084563a3e3ece18e2e4
  Stored in directory: /root/.cache/pip/wheels/69/00/ac/f7fa862c34b0e2ef320175100c
233377b4c558944f12474cf0
```

```
Successfully built pandas_ta
Installing collected packages: pandas_ta
Successfully installed pandas_ta-0.3.14b0
```

importing necessary libraries for computations

In [15]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
import random
import pandas_ta as ta
import seaborn as sns


from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn import svm
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV
from sklearn.decomposition import PCA
```

Downloading 5 years of daily OHLCV data og BTC-USD from yahoo finance Api

In [16]:
```
# Gathering BTC data

Start = '2019-01-01'
End = '2024-01-01'
df = yf.download('BTC-USD', start=Start, end=End).dropna()
```

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

In [17]:
```
df
```

Out[17]:

Date	Open	High	Low	Close	Adj Close	Volume
2019-01-01	3746.713379	3850.913818	3707.231201	3843.520020	3843.520020	4324200990
2019-01-02	3849.216309	3947.981201	3817.409424	3943.409424	3943.409424	5244856836
2019-01-03	3931.048584	3935.685059	3826.222900	3836.741211	3836.741211	4530215219
2019-01-04	3832.040039	3865.934570	3783.853760	3857.717529	3857.717529	4847965467
2019-01-05	3851.973877	3904.903076	3836.900146	3845.194580	3845.194580	5137609824
...	...	...	...	...	...	...
2023-12-27	42518.468750	43683.160156	42167.582031	43442.855469	43442.855469	25260941032
2023-12-28	43468.199219	43804.781250	42318.550781	42627.855469	42627.855469	22992093014
2023-12-29	42614.644531	43124.324219	41424.062500	42099.402344	42099.402344	26000021055
2023-12-30	42091.753906	42584.125000	41556.226562	42156.902344	42156.902344	16013925945
2023-12-31	42152.097656	42860.937500	41998.253906	42265.187500	42265.187500	16397498810

1826 rows × 6 columns

calculating all the important tecnical indicators to our data frame like Returns, SMA, RSI, MACD as part of our Feature engineering.

In [18]:
```
df['Returns'] = df['Adj Close'].pct_change()
df['10_SMA'] = df['Close'].rolling(window=10).mean()
df['50_SMA'] = df['Close'].rolling(window=50).mean()
df['RSI'] = ta.rsi(df['Close'])
macd_df = df.ta.macd(close='Close', fast=12, slow=26, signal=9, append=True)
df['MACD'] = macd_df['MACD_12_26_9']
df['Signal_Line'] = df['MACD'].ewm(span=9, adjust=False).mean()
```

Making a data of complete dataframe for further analysis

In [19]:
```
df_main = df.copy()
```

In [20]:
```
df_main
```

Out[20]:

	Open	High	Low	Close	Adj Close	Volume	Retu
Date							
2019-01-01	3746.713379	3850.913818	3707.231201	3843.520020	3843.520020	4324200990	N
2019-01-02	3849.216309	3947.981201	3817.409424	3943.409424	3943.409424	5244856836	0.025
2019-01-03	3931.048584	3935.685059	3826.222900	3836.741211	3836.741211	4530215219	-0.027
2019-01-04	3832.040039	3865.934570	3783.853760	3857.717529	3857.717529	4847965467	0.005
2019-01-05	3851.973877	3904.903076	3836.900146	3845.194580	3845.194580	5137609824	-0.003
...	...	...	...	...	...	...	
2023-12-27	42518.468750	43683.160156	42167.582031	43442.855469	43442.855469	25260941032	0.021
2023-12-28	43468.199219	43804.781250	42318.550781	42627.855469	42627.855469	22992093014	-0.018
2023-12-29	42614.644531	43124.324219	41424.062500	42099.402344	42099.402344	26000021055	-0.012
2023-12-30	42091.753906	42584.125000	41556.226562	42156.902344	42156.902344	16013925945	0.001
2023-12-31	42152.097656	42860.937500	41998.253906	42265.187500	42265.187500	16397498810	0.002

1826 rows × 15 columns

Building a strategy to calculate up and down of BTC using the main dataframe, it will help use to create our traget variable.

- if bulish condition is met then target value will be 1(up)
- and if not then the target value will be 0(down)

In [21]:

```
df_main['Next_Day_Close'] = df_main['Close'].shift(-1)
df_main['Bullish_Condition'] = ((df_main['10_SMA'] > df_main['50_SMA']) &
                               (df_main['RSI'] < 70) &
                               (df_main['MACD'] > df_main['Signal_Line'])).astype(int


df_main['Target'] = np.where((df_main['Next_Day_Close'] > df_main['Close']) & (df_m
```

Checking the total value counts of ups and downs using target variable.

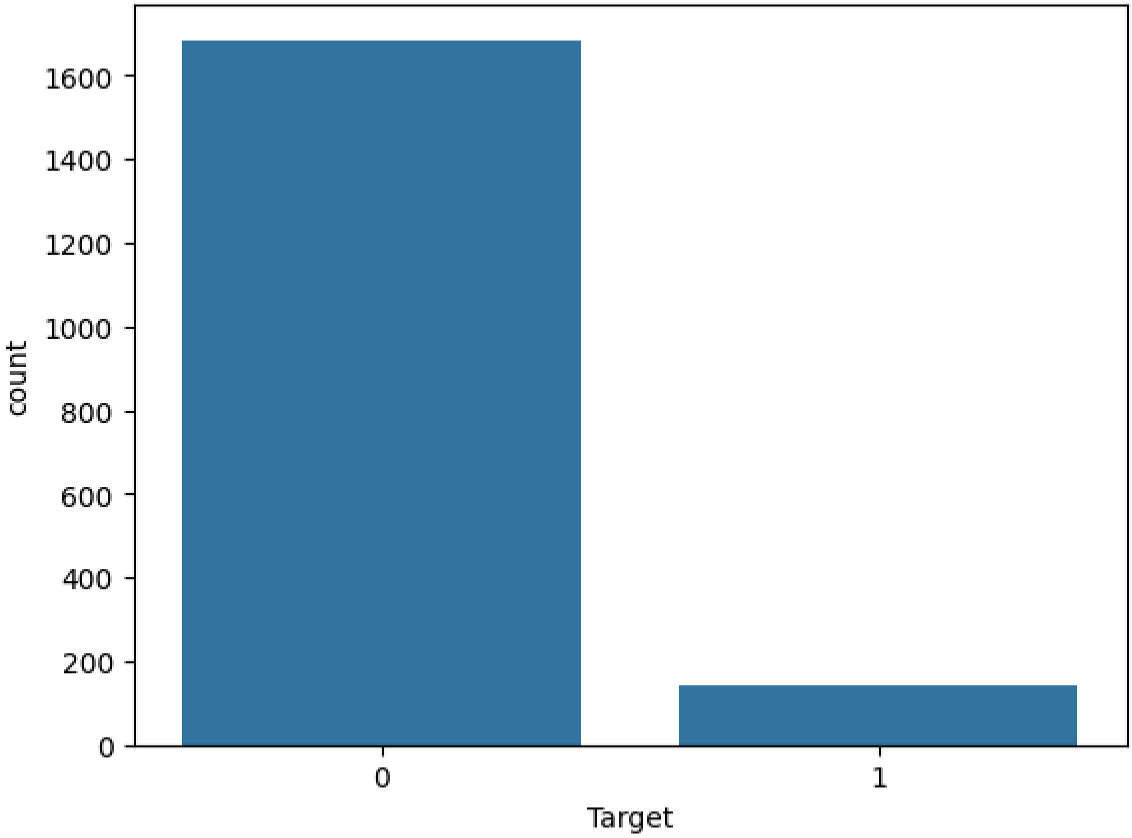
In [22]:

```
sns.countplot(x = "Target", data = df_main)
df_main.loc[:,"Target"].value_counts()
```

Out[22]:

Target	count
0	1684
1	142

dtype: int64



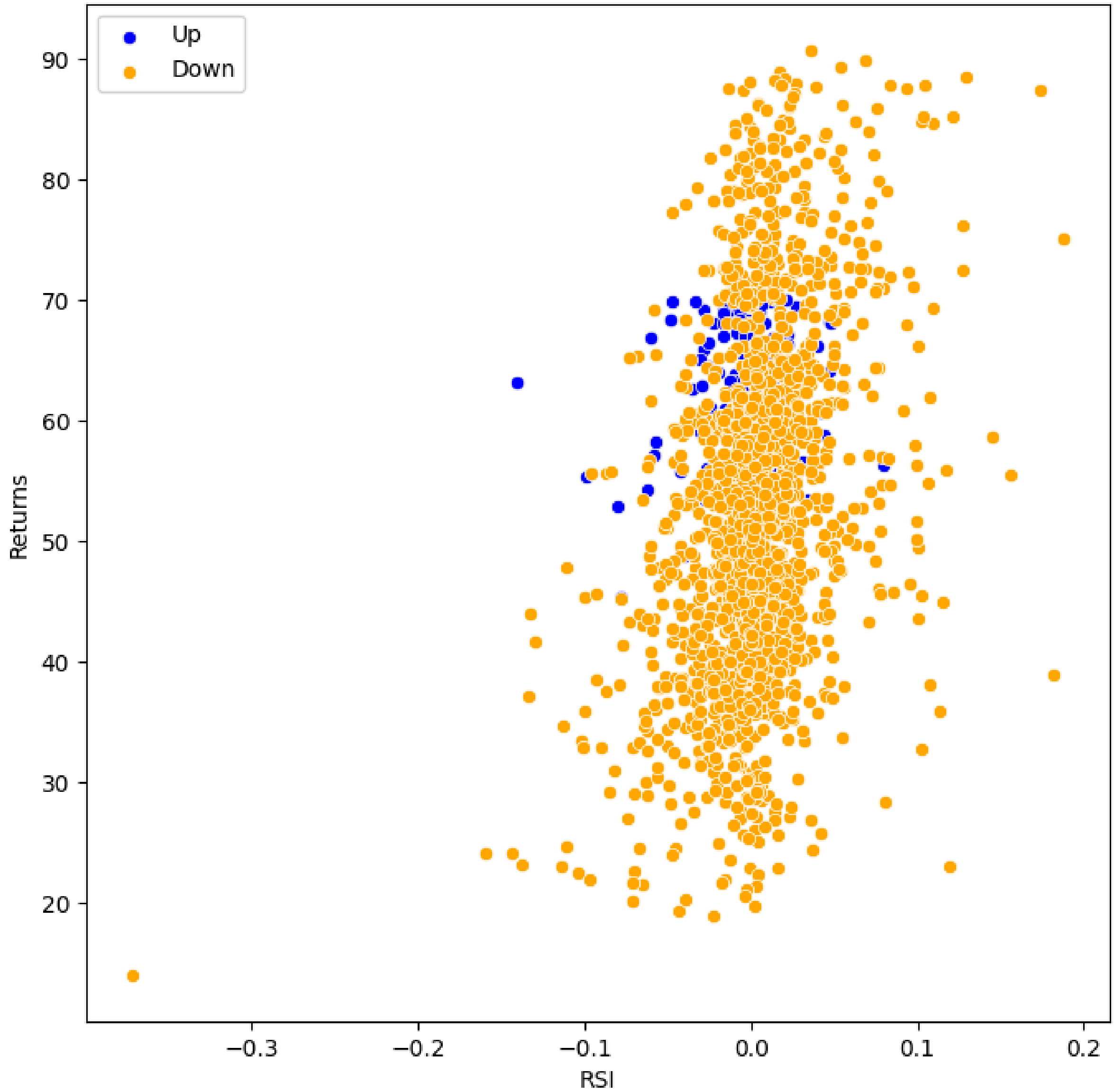
In [23]:
```
df_main.columns
```

Out[23]:
```
Index(['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume', 'Returns',
       '10_SMA', '50_SMA', 'RSI', 'MACD_12_26_9', 'MACDh_12_26_9',
       'MACDs_12_26_9', 'MACD', 'Signal_Line', 'Next_Day_Close',
       'Bullish_Condition', 'Target'],
      dtype='object')
```

we have also visualized the target varible of ups and down and we can see the how scatterd are they our goal is to classify them using SVM.

In [24]:
```
# Second Visual
Up = df_main[df_main.Target == 1]
Down = df_main[df_main.Target == 0]

plt.figure(figsize = (8,8))
plt.scatter(Up.Returns, Up.RSI, color = "blue", label = "Up", linewidths=0.5 ,edgec
plt.scatter(Down.Returns, Down.RSI, color = "orange", label = "Down", linewidths=0.
plt.xlabel("RSI")
plt.ylabel("Returns")
plt.legend()
plt.show()
```



here we have made the final copy of our preprocessed data for running the learning algorithm.

In [25]:
```
df_svm = df_main.copy()
```

In [26]:
```
df_svm
```

Out[26]:

Date	Open	High	Low	Close	Adj Close	Volume	Retu
2019-01-01	3746.713379	3850.913818	3707.231201	3843.520020	3843.520020	4324200990	N
2019-01-02	3849.216309	3947.981201	3817.409424	3943.409424	3943.409424	5244856836	0.025
2019-01-03	3931.048584	3935.685059	3826.222900	3836.741211	3836.741211	4530215219	-0.027
2019-01-04	3832.040039	3865.934570	3783.853760	3857.717529	3857.717529	4847965467	0.005
2019-01-05	3851.973877	3904.903076	3836.900146	3845.194580	3845.194580	5137609824	-0.003
...	...	...	...	...	...	...	
2023-12-27	42518.468750	43683.160156	42167.582031	43442.855469	43442.855469	25260941032	0.021
2023-12-28	43468.199219	43804.781250	42318.550781	42627.855469	42627.855469	22992093014	-0.018
2023-12-29	42614.644531	43124.324219	41424.062500	42099.402344	42099.402344	26000021055	-0.012
2023-12-30	42091.753906	42584.125000	41556.226562	42156.902344	42156.902344	16013925945	0.001
2023-12-31	42152.097656	42860.937500	41998.253906	42265.187500	42265.187500	16397498810	0.002

1826 rows × 18 columns

Here we have divided our data in two forms where in x_data we drop our target variable and in y_data we will only include Target values i.e. 0's and 1's.

In [27]:
```
# x_data
x_data = df_svm.drop(["Target"], axis = 1)

#y_data
y_data = df_svm.Target.values
```

In [28]:
```
x_data
```

Out[28]:

	Open	High	Low	Close	Adj Close	Volume	Retu
Date							
2019-01-01	3746.713379	3850.913818	3707.231201	3843.520020	3843.520020	4324200990	N
2019-01-02	3849.216309	3947.981201	3817.409424	3943.409424	3943.409424	5244856836	0.025
2019-01-03	3931.048584	3935.685059	3826.222900	3836.741211	3836.741211	4530215219	-0.027
2019-01-04	3832.040039	3865.934570	3783.853760	3857.717529	3857.717529	4847965467	0.005
2019-01-05	3851.973877	3904.903076	3836.900146	3845.194580	3845.194580	5137609824	-0.003
...	...	...	...	...	...	...	
2023-12-27	42518.468750	43683.160156	42167.582031	43442.855469	43442.855469	25260941032	0.021
2023-12-28	43468.199219	43804.781250	42318.550781	42627.855469	42627.855469	22992093014	-0.018
2023-12-29	42614.644531	43124.324219	41424.062500	42099.402344	42099.402344	26000021055	-0.012
2023-12-30	42091.753906	42584.125000	41556.226562	42156.902344	42156.902344	16013925945	0.001
2023-12-31	42152.097656	42860.937500	41998.253906	42265.187500	42265.187500	16397498810	0.002

1826 rows × 17 columns

In [29]:

```
y_data
```

Out[29]:

```
array([0, 0, 0, ..., 0, 0, 0])
```

before furthur analysis we have to normalize the data for that we are using MinMax Scaler to tranform our data, but we can also see there were some missing data due to our feature engineering we are going to take care of that here using interpolation technique we have used a linear interpolation method to fill the missing data.

In [30]:

```
scaler = MinMaxScaler()


x_data = scaler.fit_transform(x_data)

original_columns = df_svm.drop(["Target"], axis=1).columns


x_data = pd.DataFrame(x_data, columns=original_columns).interpolate(method='linear'

x_data
```

Out[30]:

	Open	High	Low	Close	Adj Close	Volume	Returns	10_SMA	50_SMA	
0	0.005383	0.006471	0.005020	0.006920	0.006920	0.000000	0.711217	0.007462	0.000049	0.3
1	0.006981	0.007956	0.006769	0.008477	0.008477	0.002656	0.711217	0.007462	0.000049	0.3
2	0.008257	0.007768	0.006909	0.006815	0.006815	0.000594	0.616363	0.007462	0.000049	0.3
3	0.006714	0.006701	0.006236	0.007141	0.007141	0.001511	0.674516	0.007462	0.000049	0.3
4	0.007024	0.007297	0.007078	0.006946	0.006946	0.002347	0.658933	0.007462	0.000049	0.3
...	...	...	...	...	...	...	...	...	...	
1821	0.609791	0.615884	0.615588	0.624046	0.624046	0.060398	0.703537	0.650150	0.636627	0.5
1822	0.624596	0.617745	0.617985	0.611345	0.611345	0.053853	0.631188	0.650157	0.639068	0.5
1823	0.611290	0.607334	0.603785	0.603109	0.603109	0.062531	0.642568	0.649877	0.640960	0.4
1824	0.603139	0.599069	0.605883	0.604005	0.604005	0.033723	0.667181	0.647435	0.642656	0.4
1825	0.604080	0.603304	0.612900	0.605693	0.605693	0.034829	0.669332	0.644816	0.644450	0.4

1826 rows × 17 columns

Now. we have splitted our data in 80:20 ratio for training and testing

In [31]:
```
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.2,
```

we are using sckit learn SVM library to run our learning algorithm, first we will run a random SVM model to check it's accuracy.

Our random model resulted with an accuracy of 0.9371584699453552, but we want to improve it so will further explore techniques.

In [32]:
```
Model_1 = svm.SVC(random_state = 1)

Model_1.fit(x_train, y_train)

print("accuracy of Model_1 algo:", Model_1.score(x_test, y_test))
```

```
accuracy of Model_1 algo: 0.9371584699453552
```

For selecting a better model we are going to run a grid Search algorithm will will give do all the hyperparameter tuning on our behalf for the provided techniques i.e. Kernel method (Linear, Polynomial, Sigmoid) and provide us the best estimator for our model.

In [33]:
```
grid={"C":[1, 100, 1000],
      "kernel":["linear","poly","sigmoid"],
      'gamma': ['scale', 'auto'],
      'degree': [2, 3]
}

model_selection = svm.SVC(random_state = 1)

model_selection.fit(x_train, y_train)

print("test accuracy {}".format(model_selection.score(x_test,y_test)))
print("Train accuracy {}".format(model_selection.score(x_train, y_train)))
```

```
model_selection_gscv=GridSearchCV(model_selection,grid,cv=10)
model_selection_gscv.fit(x_test,y_test)
```

```
test accuracy 0.9371584699453552
Train accuracy 0.9363013698630137
```

Out[33]:

```
▸ GridSearchCV
▸ best_estimator_: SVC
    ▸ SVC
```

After running the grid search CV we can the best estimator which should be used to improve the accuracy of the model.

In [34]:

```
print("best hyperparameters: ", model_selection_gscv.best_params_)
print("accuracy: ", model_selection_gscv.best_score_)
```

```
best hyperparameters:  {'C': 1000, 'degree': 2, 'gamma': 'scale', 'kernel': 'pol
y'}
accuracy:  0.9617117117117117
```

we can see {'C': 1000, 'kernel': 'poly'} is the best hyperparameter that the grid search cv have found so we are going to use these hyperparameter in our model and check if the accuracy improves or not.

In [35]:

```
main_model = svm.SVC(C = 1000, kernel="poly", degree= 2)

print("test accuracy: {} ".format(main_model.fit(x_test, y_test).score(x_test, y_te
print("train accuracy: {} ".format(main_model.fit(x_train, y_train).score(x_train,
```

```
test accuracy: 0.9918032786885246
train accuracy: 0.9910958904109589
```

here we can see in the results above test accuracy: 0.9972677595628415, train accuracy: 0.9965753424657534, how model accuray have improved a lot but also our training accuracy is also impressive.

Now we have to vislaize our data with final results but we have problem as our model arounf 17 features which will be very tricky to visualize so for that will have to improvise and call our good old friend PCA (we have already discussed about it briefly in GWP1), we will use PCA to reduce the dimesion of our model for plotting purposes. Then, we can plot the decision boundary and visualize how the SVM classifier separates the data. Although our accuracy reduced but still this is just for visuaization.

In [36]:

```
pca = PCA(n_components=2)
X_train_reduced = pca.fit_transform(x_train)
X_test_reduced = pca.transform(x_test)

# Train the SVM model on reduced data
main_model.fit(X_train_reduced, y_train)
print("test accuracy: {} ".format(main_model.fit(X_test_reduced, y_test).score(X_te
print("train accuracy: {} ".format(main_model.fit(X_train_reduced, y_train).score(X
```

```
test accuracy: 0.953551912568306
train accuracy: 0.9273972602739726
```

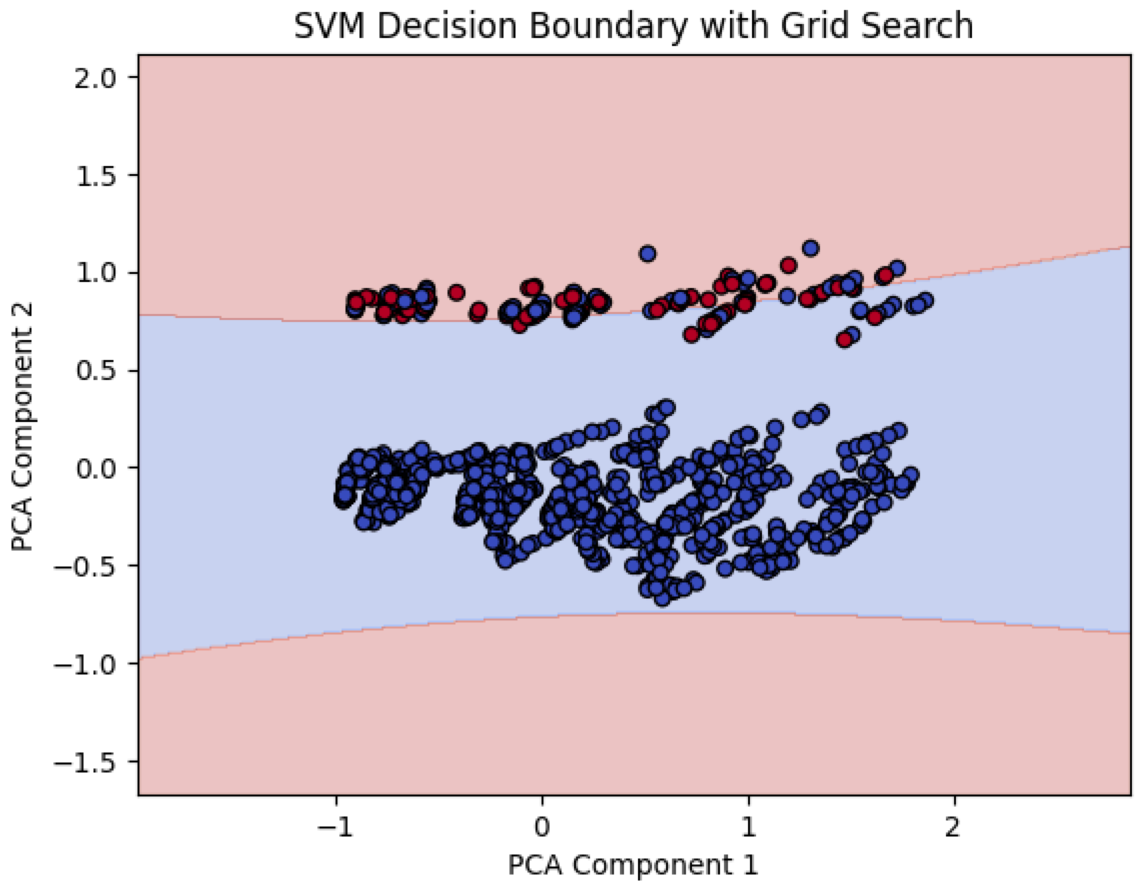
Here we have again trained the same SVM model with same hyperparameters but with reduced data and plotted it to visulaize our decision boundary.

In [37]:
```
# Function to plot decision boundary
def plot_svm_decision_boundary(model, X, y):
    # Create a mesh grid
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                         np.arange(y_min, y_max, 0.01))

    # Predict for each point in the mesh
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot the decision boundary and the margins
    plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
    plt.scatter(X[:, 0], X[:, 1], c=y, s=30, edgecolor='k', cmap='coolwarm')
    plt.title("SVM Decision Boundary with Grid Search")
    plt.xlabel("PCA Component 1")
    plt.ylabel("PCA Component 2")
    plt.show()

# Plot the decision boundary using training data
plot_svm_decision_boundary(main_model, X_train_reduced, y_train)
```



In [37]:

Neural Networks

For our implementation we are going to do a simple BTC Price Movement modeling using Neural Networks to predict whether a cryptocurrency's price (e.g., Bitcoin) will move up(1) or down(0) based on historical price data and technical indicators.

Installing necessary data sources and computations api's

In [38]:
```
!pip install yfinance
!pip install pandas_ta
!pip install scikeras
```

```
Requirement already satisfied: yfinance in /usr/local/lib/python3.10/dist-packages
(0.2.43)
Requirement already satisfied: pandas>=1.3.0 in /usr/local/lib/python3.10/dist-pac
kages (from yfinance) (2.2.2)
Requirement already satisfied: numpy>=1.16.5 in /usr/local/lib/python3.10/dist-pac
kages (from yfinance) (1.26.4)
Requirement already satisfied: requests>=2.31 in /usr/local/lib/python3.10/dist-pa
ckages (from yfinance) (2.32.3)
Requirement already satisfied: multitasking>=0.0.7 in /usr/local/lib/python3.10/di
st-packages (from yfinance) (0.0.11)
Requirement already satisfied: lxml>=4.9.1 in /usr/local/lib/python3.10/dist-packa
ges (from yfinance) (4.9.4)
Requirement already satisfied: platformdirs>=2.0.0 in /usr/local/lib/python3.10/di
st-packages (from yfinance) (4.3.6)
Requirement already satisfied: pytz>=2022.5 in /usr/local/lib/python3.10/dist-pack
ages (from yfinance) (2024.2)
Requirement already satisfied: frozendict>=2.3.4 in /usr/local/lib/python3.10/dist
-packages (from yfinance) (2.4.4)
Requirement already satisfied: peewee>=3.16.2 in /usr/local/lib/python3.10/dist-pa
ckages (from yfinance) (3.17.6)
Requirement already satisfied: beautifulsoup4>=4.11.1 in /usr/local/lib/python3.1
0/dist-packages (from yfinance) (4.12.3)
Requirement already satisfied: html5lib>=1.1 in /usr/local/lib/python3.10/dist-pac
kages (from yfinance) (1.1)
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.10/dist-pac
kages (from beautifulsoup4>=4.11.1->yfinance) (2.6)
Requirement already satisfied: six>=1.9 in /usr/local/lib/python3.10/dist-packages
(from html5lib>=1.1->yfinance) (1.16.0)
Requirement already satisfied: webencodings in /usr/local/lib/python3.10/dist-pack
ages (from html5lib>=1.1->yfinance) (0.5.1)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.1
0/dist-packages (from pandas>=1.3.0->yfinance) (2.8.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-pa
ckages (from pandas>=1.3.0->yfinance) (2024.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.
10/dist-packages (from requests>=2.31->yfinance) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-pack
ages (from requests>=2.31->yfinance) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dis
t-packages (from requests>=2.31->yfinance) (2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dis
t-packages (from requests>=2.31->yfinance) (2024.8.30)
Requirement already satisfied: pandas_ta in /usr/local/lib/python3.10/dist-package
s (0.3.14b0)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages
(from pandas_ta) (2.2.2)
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-pac
kages (from pandas->pandas_ta) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.1
0/dist-packages (from pandas->pandas_ta) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-pack
ages (from pandas->pandas_ta) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-pa
ckages (from pandas->pandas_ta) (2024.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages
(from python-dateutil>=2.8.2->pandas->pandas_ta) (1.16.0)
Collecting scikeras
  Downloading scikeras-0.13.0-py3-none-any.whl.metadata (3.1 kB)
Requirement already satisfied: keras>=3.2.0 in /usr/local/lib/python3.10/dist-pack
ages (from scikeras) (3.4.1)
Requirement already satisfied: scikit-learn>=1.4.2 in /usr/local/lib/python3.10/di
st-packages (from scikeras) (1.5.2)
Requirement already satisfied: absl-py in /usr/local/lib/python3.10/dist-packages
(from keras>=3.2.0->scikeras) (1.4.0)
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (f
rom keras>=3.2.0->scikeras) (1.26.4)
Requirement already satisfied: rich in /usr/local/lib/python3.10/dist-packages (fr
om keras>=3.2.0->scikeras) (13.8.1)
Requirement already satisfied: namex in /usr/local/lib/python3.10/dist-packages (f
rom keras>=3.2.0->scikeras) (0.0.8)
Requirement already satisfied: h5py in /usr/local/lib/python3.10/dist-packages (fr
om keras>=3.2.0->scikeras) (3.11.0)
Requirement already satisfied: optree in /usr/local/lib/python3.10/dist-packages
(from keras>=3.2.0->scikeras) (0.12.1)
Requirement already satisfied: ml-dtypes in /usr/local/lib/python3.10/dist-package
s (from keras>=3.2.0->scikeras) (0.4.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-package
s (from keras>=3.2.0->scikeras) (24.1)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.10/dist-pack
ages (from scikit-learn>=1.4.2->scikeras) (1.13.1)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.10/dist-pac
kages (from scikit-learn>=1.4.2->scikeras) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.10/d
ist-packages (from scikit-learn>=1.4.2->scikeras) (3.5.0)
Requirement already satisfied: typing-extensions>=4.5.0 in /usr/local/lib/python3.
10/dist-packages (from optree->keras>=3.2.0->scikeras) (4.12.2)
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.10/
dist-packages (from rich->keras>=3.2.0->scikeras) (3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.1
0/dist-packages (from rich->keras>=3.2.0->scikeras) (2.18.0)
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.10/dist-packag
es (from markdown-it-py>=2.2.0->rich->keras>=3.2.0->scikeras) (0.1.2)
Downloading scikeras-0.13.0-py3-none-any.whl (26 kB)
Installing collected packages: scikeras
Successfully installed scikeras-0.13.0
```

importing necessary libraries for computations

In [39]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
import random
import pandas_ta as ta
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

Downloading 5 years of daily OHLCV data og BTC-USD from yahoo finance Api

In [40]:
```
# Gathering BTC data

Start = '2019-01-01'
End = '2024-01-01'
df = yf.download('BTC-USD', start=Start, end=End).dropna()
```

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

In [41]:
```
df
```

Out[41]:

Date	Open	High	Low	Close	Adj Close	Volume
2019-01-01	3746.713379	3850.913818	3707.231201	3843.520020	3843.520020	4324200990
2019-01-02	3849.216309	3947.981201	3817.409424	3943.409424	3943.409424	5244856836
2019-01-03	3931.048584	3935.685059	3826.222900	3836.741211	3836.741211	4530215219
2019-01-04	3832.040039	3865.934570	3783.853760	3857.717529	3857.717529	4847965467
2019-01-05	3851.973877	3904.903076	3836.900146	3845.194580	3845.194580	5137609824
...	...	...	...	...	...	...
2023-12-27	42518.468750	43683.160156	42167.582031	43442.855469	43442.855469	25260941032
2023-12-28	43468.199219	43804.781250	42318.550781	42627.855469	42627.855469	22992093014
2023-12-29	42614.644531	43124.324219	41424.062500	42099.402344	42099.402344	26000021055
2023-12-30	42091.753906	42584.125000	41556.226562	42156.902344	42156.902344	16013925945
2023-12-31	42152.097656	42860.937500	41998.253906	42265.187500	42265.187500	16397498810

1826 rows × 6 columns

calculating all the important tecnical indicators to our data frame like Returns, SMA, RSI, MACD as part of our Feature engineering.

In [42]:
```
df['Returns'] = df['Adj Close'].pct_change()
df['10_SMA'] = df['Close'].rolling(window=10).mean()
df['50_SMA'] = df['Close'].rolling(window=50).mean()
df['RSI'] = ta.rsi(df['Close'])
macd_df = df.ta.macd(close='Close', fast=12, slow=26, signal=9, append=True)
df['MACD'] = macd_df['MACD_12_26_9']
df['Signal_Line'] = df['MACD'].ewm(span=9, adjust=False).mean()
```

Making a data of complete dataframe for further analysis

In [43]:
```
df_main = df.copy()
```

In [44]:
```
df_main
```

Out[44]:

Date	Open	High	Low	Close	Adj Close	Volume	Retu
2019-01-01	3746.713379	3850.913818	3707.231201	3843.520020	3843.520020	4324200990	N
2019-01-02	3849.216309	3947.981201	3817.409424	3943.409424	3943.409424	5244856836	0.025
2019-01-03	3931.048584	3935.685059	3826.222900	3836.741211	3836.741211	4530215219	-0.027
2019-01-04	3832.040039	3865.934570	3783.853760	3857.717529	3857.717529	4847965467	0.005
2019-01-05	3851.973877	3904.903076	3836.900146	3845.194580	3845.194580	5137609824	-0.003
...	...	...	...	...	...	...	
2023-12-27	42518.468750	43683.160156	42167.582031	43442.855469	43442.855469	25260941032	0.021
2023-12-28	43468.199219	43804.781250	42318.550781	42627.855469	42627.855469	22992093014	-0.018
2023-12-29	42614.644531	43124.324219	41424.062500	42099.402344	42099.402344	26000021055	-0.012
2023-12-30	42091.753906	42584.125000	41556.226562	42156.902344	42156.902344	16013925945	0.001
2023-12-31	42152.097656	42860.937500	41998.253906	42265.187500	42265.187500	16397498810	0.002

1826 rows × 15 columns

Building a strategy to calculate up and down of BTC using the main dataframe, it will help use to create our traget variable.

- if bulish condition is met then target value will be 1(up)
- and if not then the target value will be 0(down)

In [45]:
```
df_main['Next_Day_Close'] = df_main['Close'].shift(-1)
df_main['Bullish_Condition'] = ((df_main['10_SMA'] > df_main['50_SMA']) &
                                (df_main['RSI'] < 70) &
                                (df_main['MACD'] > df_main['Signal_Line'])).astype(int


df_main['Target'] = np.where((df_main['Next_Day_Close'] > df_main['Close']) & (df_m
```

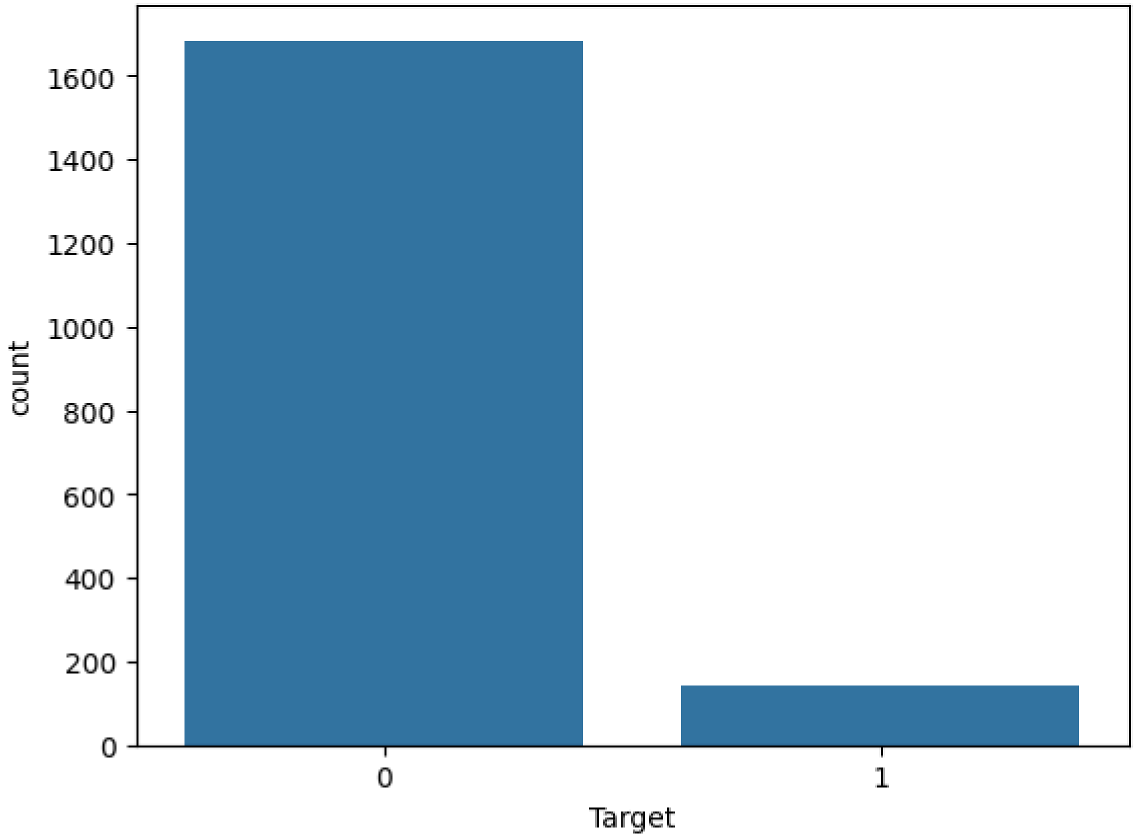
Checking the total value counts of ups and downs using target variable.

In [46]:
```
sns.countplot(x = "Target", data = df_main)
df_main.loc[:,"Target"].value_counts()
```

Out[46]:

Target	count
0	1684
1	142

dtype: int64



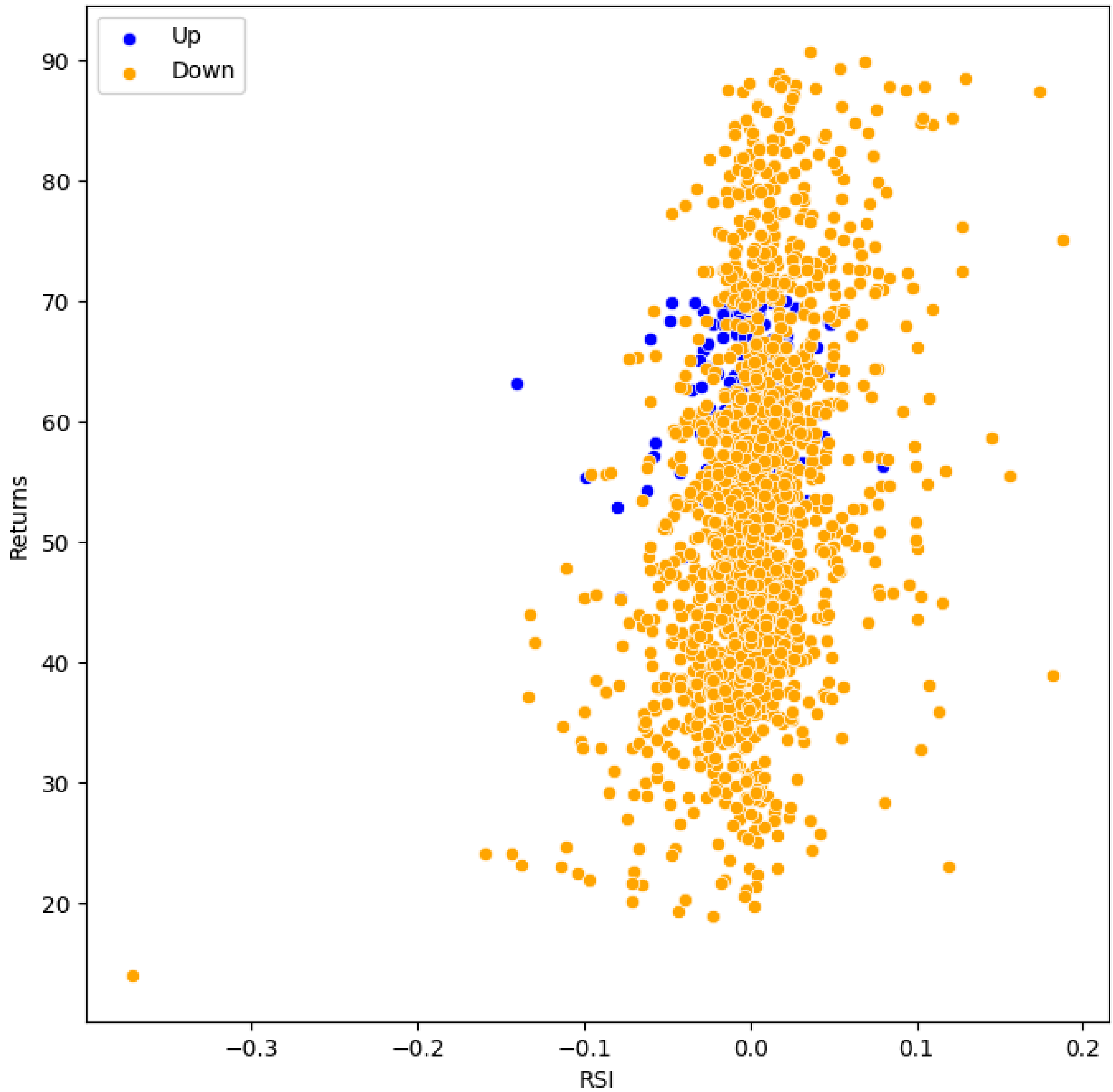
In [47]:
```
df_main.columns
```

Out[47]:
```
Index(['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume', 'Returns',
       '10_SMA', '50_SMA', 'RSI', 'MACD_12_26_9', 'MACDh_12_26_9',
       'MACDs_12_26_9', 'MACD', 'Signal_Line', 'Next_Day_Close',
       'Bullish_Condition', 'Target'],
      dtype='object')
```

we have also visualized the target varible of ups and down and we can see the how scatterd are they our goal is to model them using NN.

In [48]:
```
# Second Visual
Up = df_main[df_main.Target == 1]
Down = df_main[df_main.Target == 0]

plt.figure(figsize = (8,8))
plt.scatter(Up.Returns, Up.RSI, color = "blue", label = "Up", linewidths=0.5 ,edgec
plt.scatter(Down.Returns, Down.RSI, color = "orange", label = "Down", linewidths=0.
plt.xlabel("RSI")
plt.ylabel("Returns")
plt.legend()
plt.show()
```



here we have made the final copy of our preprocessed data for running the learning algorithm.

In [49]:
```
df_nn = df_main.copy()
```

In [50]:
```
df_nn
```

Out[50]:

	Open	High	Low	Close	Adj Close	Volume	Retu
Date							
2019-01-01	3746.713379	3850.913818	3707.231201	3843.520020	3843.520020	4324200990	N
2019-01-02	3849.216309	3947.981201	3817.409424	3943.409424	3943.409424	5244856836	0.025
2019-01-03	3931.048584	3935.685059	3826.222900	3836.741211	3836.741211	4530215219	-0.027
2019-01-04	3832.040039	3865.934570	3783.853760	3857.717529	3857.717529	4847965467	0.005
2019-01-05	3851.973877	3904.903076	3836.900146	3845.194580	3845.194580	5137609824	-0.003
...	...	...	...	...	...	...	
2023-12-27	42518.468750	43683.160156	42167.582031	43442.855469	43442.855469	25260941032	0.021
2023-12-28	43468.199219	43804.781250	42318.550781	42627.855469	42627.855469	22992093014	-0.018
2023-12-29	42614.644531	43124.324219	41424.062500	42099.402344	42099.402344	26000021055	-0.012
2023-12-30	42091.753906	42584.125000	41556.226562	42156.902344	42156.902344	16013925945	0.001
2023-12-31	42152.097656	42860.937500	41998.253906	42265.187500	42265.187500	16397498810	0.002

1826 rows × 18 columns

Here we have divided our data in two forms where in x_data we drop our target variable and in y_data we will only include Target values i.e. 0's and 1's.

In [51]:
```
# x_data
x_data = df_nn.drop(["Target"], axis = 1)

#y_data
y_data = df_nn.Target.values
```

In [52]:
```
x_data
```

Out[52]:

	Open	High	Low	Close	Adj Close	Volume	Retu
Date							
2019-01-01	3746.713379	3850.913818	3707.231201	3843.520020	3843.520020	4324200990	N
2019-01-02	3849.216309	3947.981201	3817.409424	3943.409424	3943.409424	5244856836	0.025
2019-01-03	3931.048584	3935.685059	3826.222900	3836.741211	3836.741211	4530215219	-0.027
2019-01-04	3832.040039	3865.934570	3783.853760	3857.717529	3857.717529	4847965467	0.005
2019-01-05	3851.973877	3904.903076	3836.900146	3845.194580	3845.194580	5137609824	-0.003
...	...	...	...	...	...	...	
2023-12-27	42518.468750	43683.160156	42167.582031	43442.855469	43442.855469	25260941032	0.021
2023-12-28	43468.199219	43804.781250	42318.550781	42627.855469	42627.855469	22992093014	-0.018
2023-12-29	42614.644531	43124.324219	41424.062500	42099.402344	42099.402344	26000021055	-0.012
2023-12-30	42091.753906	42584.125000	41556.226562	42156.902344	42156.902344	16013925945	0.001
2023-12-31	42152.097656	42860.937500	41998.253906	42265.187500	42265.187500	16397498810	0.002

1826 rows × 17 columns

In [53]:

```
y_data
```

Out[53]:

```
array([0, 0, 0, ..., 0, 0, 0])
```

before furthur analysis we have to normalize the data for that we are using MinMax Scaler to tranform our data, but we can also see there were some missing data due to our feature engineering we are going to take care of that here using interpolation technique we have used a linear interpolation method to fill the missing data.

In [54]:

```
scaler = MinMaxScaler()


x_data = scaler.fit_transform(x_data)

original_columns = df_nn.drop(["Target"], axis=1).columns


x_data = pd.DataFrame(x_data, columns=original_columns).interpolate(method='linear'

x_data
```

Out[54]:

	Open	High	Low	Close	Adj Close	Volume	Returns	10_SMA	50_SMA	
0	0.005383	0.006471	0.005020	0.006920	0.006920	0.000000	0.711217	0.007462	0.000049	0.3
1	0.006981	0.007956	0.006769	0.008477	0.008477	0.002656	0.711217	0.007462	0.000049	0.3
2	0.008257	0.007768	0.006909	0.006815	0.006815	0.000594	0.616363	0.007462	0.000049	0.3
3	0.006714	0.006701	0.006236	0.007141	0.007141	0.001511	0.674516	0.007462	0.000049	0.3
4	0.007024	0.007297	0.007078	0.006946	0.006946	0.002347	0.658933	0.007462	0.000049	0.3
...	...	...	...	...	...	...	...	...	...	
1821	0.609791	0.615884	0.615588	0.624046	0.624046	0.060398	0.703537	0.650150	0.636627	0.5
1822	0.624596	0.617745	0.617985	0.611345	0.611345	0.053853	0.631188	0.650157	0.639068	0.5
1823	0.611290	0.607334	0.603785	0.603109	0.603109	0.062531	0.642568	0.649877	0.640960	0.4
1824	0.603139	0.599069	0.605883	0.604005	0.604005	0.033723	0.667181	0.647435	0.642656	0.4
1825	0.604080	0.603304	0.612900	0.605693	0.605693	0.034829	0.669332	0.644816	0.644450	0.4

1826 rows × 17 columns

Now. we have splitted our data in 80:20 ratio for training and testing

In [55]:
```
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.2,
```

we are using Keras library to run our learning algorithm

In [56]:
```
# Define the model
model = keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=(x_train.shape[1],)),  # Input
    layers.Dense(32, activation='relu'),                                  # Hidden
    layers.Dense(1, activation='sigmoid')                                 # Output
])

# Compile the model
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Fit the model
history = model.fit(x_train, y_train, epochs=50, batch_size=32, validation_split=0.

test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test accuracy using Random Model: {test_accuracy}')
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWar
ning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Seq
uential models, prefer using an `Input(shape)` object as the first layer in the mo
del instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Epoch 1/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.4514 - loss: 0.6983 - val_acc
uracy: 0.9315 - val_loss: 0.3881
Epoch 2/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9157 - loss: 0.3624 - val_acc
uracy: 0.9315 - val_loss: 0.2339
Epoch 3/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9167 - loss: 0.2642 - val_acc
uracy: 0.9315 - val_loss: 0.1853
Epoch 4/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9089 - loss: 0.2197 - val_acc
uracy: 0.9315 - val_loss: 0.1528
Epoch 5/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9044 - loss: 0.1674 - val_acc
uracy: 0.9281 - val_loss: 0.1208
Epoch 6/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9199 - loss: 0.1391 - val_acc
uracy: 0.9281 - val_loss: 0.1074
Epoch 7/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9086 - loss: 0.1345 - val_acc
uracy: 0.9281 - val_loss: 0.1007
Epoch 8/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9178 - loss: 0.1207 - val_acc
uracy: 0.9315 - val_loss: 0.0979
Epoch 9/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9362 - loss: 0.1079 - val_acc
uracy: 0.9384 - val_loss: 0.0962
Epoch 10/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9407 - loss: 0.1114 - val_acc
uracy: 0.9315 - val_loss: 0.0949
Epoch 11/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9257 - loss: 0.1212 - val_acc
uracy: 0.9281 - val_loss: 0.0937
Epoch 12/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9307 - loss: 0.1052 - val_acc
uracy: 0.9418 - val_loss: 0.0937
Epoch 13/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9071 - loss: 0.1185 - val_acc
uracy: 0.9384 - val_loss: 0.0931
Epoch 14/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9375 - loss: 0.1078 - val_acc
uracy: 0.9212 - val_loss: 0.0928
Epoch 15/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9354 - loss: 0.1094 - val_acc
uracy: 0.9349 - val_loss: 0.0925
Epoch 16/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9496 - loss: 0.0937 - val_acc
uracy: 0.9349 - val_loss: 0.0922
Epoch 17/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9225 - loss: 0.1130 - val_acc
uracy: 0.9452 - val_loss: 0.0951
Epoch 18/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9261 - loss: 0.1123 - val_acc
uracy: 0.9452 - val_loss: 0.0933
Epoch 19/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9411 - loss: 0.1074 - val_acc
uracy: 0.9212 - val_loss: 0.0953
Epoch 20/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9322 - loss: 0.1206 - val_acc
uracy: 0.9075 - val_loss: 0.0938
Epoch 21/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9329 - loss: 0.1098 - val_acc
uracy: 0.9418 - val_loss: 0.0917
Epoch 22/50
```

```
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9349 - loss: 0.1054 - val_accuracy: 0.9349 - val_loss: 0.0915
Epoch 23/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9345 - loss: 0.1069 - val_accuracy: 0.9418 - val_loss: 0.0918
Epoch 24/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9394 - loss: 0.1054 - val_accuracy: 0.9452 - val_loss: 0.0931
Epoch 25/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9398 - loss: 0.1048 - val_accuracy: 0.9075 - val_loss: 0.0930
Epoch 26/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9241 - loss: 0.1243 - val_accuracy: 0.9281 - val_loss: 0.0972
Epoch 27/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9194 - loss: 0.1023 - val_accuracy: 0.9452 - val_loss: 0.0922
Epoch 28/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9286 - loss: 0.1110 - val_accuracy: 0.9384 - val_loss: 0.0918
Epoch 29/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9275 - loss: 0.1164 - val_accuracy: 0.9075 - val_loss: 0.0933
Epoch 30/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9353 - loss: 0.1054 - val_accuracy: 0.9384 - val_loss: 0.0911
Epoch 31/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9375 - loss: 0.1063 - val_accuracy: 0.9384 - val_loss: 0.0912
Epoch 32/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9306 - loss: 0.1090 - val_accuracy: 0.9110 - val_loss: 0.0933
Epoch 33/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9422 - loss: 0.0982 - val_accuracy: 0.9384 - val_loss: 0.0912
Epoch 34/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.9377 - loss: 0.1034 - val_accuracy: 0.9384 - val_loss: 0.0916
Epoch 35/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9394 - loss: 0.1045 - val_accuracy: 0.9349 - val_loss: 0.0913
Epoch 36/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9242 - loss: 0.1171 - val_accuracy: 0.9452 - val_loss: 0.0913
Epoch 37/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.9221 - loss: 0.1154 - val_accuracy: 0.9452 - val_loss: 0.0954
Epoch 38/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.9339 - loss: 0.1092 - val_accuracy: 0.9418 - val_loss: 0.0915
Epoch 39/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.9261 - loss: 0.1126 - val_accuracy: 0.9315 - val_loss: 0.0916
Epoch 40/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.9402 - loss: 0.1040 - val_accuracy: 0.9212 - val_loss: 0.0922
Epoch 41/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9216 - loss: 0.1074 - val_accuracy: 0.9384 - val_loss: 0.0909
Epoch 42/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9310 - loss: 0.1080 - val_accuracy: 0.9418 - val_loss: 0.0910
Epoch 43/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9336 - loss: 0.1148 - val_acc
```

```
uracy: 0.9452 - val_loss: 0.0913
Epoch 44/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9409 - loss: 0.0948 - val_acc
uracy: 0.9452 - val_loss: 0.0934
Epoch 45/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9288 - loss: 0.1170 - val_acc
uracy: 0.9349 - val_loss: 0.0921
Epoch 46/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9232 - loss: 0.1158 - val_acc
uracy: 0.9384 - val_loss: 0.0909
Epoch 47/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9330 - loss: 0.1113 - val_acc
uracy: 0.9247 - val_loss: 0.0920
Epoch 48/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9474 - loss: 0.0942 - val_acc
uracy: 0.9384 - val_loss: 0.0904
Epoch 49/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9437 - loss: 0.1028 - val_acc
uracy: 0.9452 - val_loss: 0.0928
Epoch 50/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9349 - loss: 0.1073 - val_acc
uracy: 0.9144 - val_loss: 0.0926
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9323 - loss: 0.0951
Test accuracy using Random Model: 0.9398906826972961
```

For selecting a better model we are going to run a grid Search algorithm and will give do all the hyperparameter tuning on our behalf for the provided techniques

In [57]:

```python
def create_and_train_model(neurons, optimizer, batch_size, epochs):
    # Define the model
    model = keras.Sequential([
        layers.Dense(neurons, activation='relu', input_shape=(x_train.shape[1],)),
        layers.Dense(32, activation='relu'),
        layers.Dense(1, activation='sigmoid')
    ])

    # Compile the model
    model.compile(optimizer=optimizer,
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

    # Fit the model
    history = model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size, val

    return model, history
```

In [58]:

```python
# Define hyperparameters to tune
neurons_list = [16, 32, 64]
optimizers = ['adam', 'sgd']
batch_sizes = [16, 32]
epochs = 50

results = []

# Loop through hyperparameter combinations
for neurons in neurons_list:
    for optimizer in optimizers:
        for batch_size in batch_sizes:
            # Train the model with current hyperparameters
            model, history = create_and_train_model(neurons, optimizer, batch_size,

            # Evaluate the model
            test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=0)
```

```
            # Store the results
            results.append({
                'neurons': neurons,
                'optimizer': optimizer,
                'batch_size': batch_size,
                'test_accuracy': test_accuracy
            })

            print(f'Neurons: {neurons}, Optimizer: {optimizer}, Batch Size: {batch_
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Neurons: 16, Optimizer: adam, Batch Size: 16, Test Accuracy: 0.9344262480735779
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Neurons: 16, Optimizer: adam, Batch Size: 32, Test Accuracy: 0.931693971157074
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Neurons: 16, Optimizer: sgd, Batch Size: 16, Test Accuracy: 0.9344262480735779
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Neurons: 16, Optimizer: sgd, Batch Size: 32, Test Accuracy: 0.9289617538452148
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Neurons: 32, Optimizer: adam, Batch Size: 16, Test Accuracy: 0.937158465385437
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Neurons: 32, Optimizer: adam, Batch Size: 32, Test Accuracy: 0.9398906826972961
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Neurons: 32, Optimizer: sgd, Batch Size: 16, Test Accuracy: 0.931693971157074
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Neurons: 32, Optimizer: sgd, Batch Size: 32, Test Accuracy: 0.9344262480735779
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Neurons: 64, Optimizer: adam, Batch Size: 16, Test Accuracy: 0.9426229596138
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Neurons: 64, Optimizer: adam, Batch Size: 32, Test Accuracy: 0.9344262480735779
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Neurons: 64, Optimizer: sgd, Batch Size: 16, Test Accuracy: 0.937158465385437
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Neurons: 64, Optimizer: sgd, Batch Size: 32, Test Accuracy: 0.937158465385437
```

In [59]:

```python
# Convert results to a DataFrame for easier analysis
results_df = pd.DataFrame(results)

# Find the best configuration
best_result = results_df.loc[results_df['test_accuracy'].idxmax()]

print("\nBest Hyperparameters:")
print(best_result)
```

```
Best Hyperparameters:
neurons                64
optimizer            adam
batch_size             16
test_accuracy    0.942623
Name: 8, dtype: object
```

we can see {'neurons': 32, 'optimizer': 'adam', 'batch_size':16} is the best hyperparameter that the grid search cv have found so we are going to use these hyperparameter in our model and check if the accuracy improves or not.

In [60]:

```python
model, history = create_and_train_model(32, 'adam', 16, 50)
test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=0)

print("test accuracy: {} ".format(test_accuracy))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
test accuracy: 0.9398906826972961
```

here we can see in the results above test accuracy: 0.94, model accuracy have improved a lot

In [ ]: