GWP 2

Step 1

In this GWP we will build a Deep Learning model that tries to predict short-term market trends across different asset classes.

Importing necessary libraries

In [1]:
```
import yfinance as yf
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from scipy.stats import skew, kurtosis, norm, probplot
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.stats.diagnostic import acorr_ljungbox
import tensorflow as tf
from keras.layers import LSTM, Dense, Dropout, Bidirectional
from sklearn.preprocessing import MinMaxScaler
from keras import Sequential
from keras.callbacks import EarlyStopping
```

First we will download Data of 5 different ETf's SPY, TLT, SHY, GLD, DBO from January 1st, 2018 to December 30th, 2022.

In [2]:
```
## Download Data for ETFs
etfs = ["SPY", "TLT", "SHY", "GLD", "DBO"]
start_date = "2018-01-01"
end_date = "2022-12-30"

# Download data
data = {etf: yf.download(etf, start=start_date, end=end_date)['Close'] for etf i

# Align all valid data to the same index
aligned_data = pd.concat(data.values(), axis=1)

# Drop rows with missing data (if any)
close_prices = aligned_data.dropna()

# Convert the Date index to datetime format without time
close_prices.index = pd.to_datetime(close_prices.index).date

# Display the first few rows of the DataFrame
print("Successfully created DataFrame for valid ETFs with aligned dates.")
print(close_prices.head())
```

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

```
Successfully created DataFrame for valid ETFs with aligned dates.
Ticker             SPY         TLT        SHY         GLD    DBO
2018-01-02  268.769989  125.489998  83.820000  125.150002  10.19
2018-01-03  270.470001  126.089996  83.820000  124.820000  10.41
2018-01-04  271.609985  126.070000  83.779999  125.459999  10.43
2018-01-05  273.420013  125.709999  83.779999  125.330002  10.38
2018-01-08  273.920013  125.629997  83.779999  125.309998  10.44
```

In [3]:
```
close_prices.info()
```

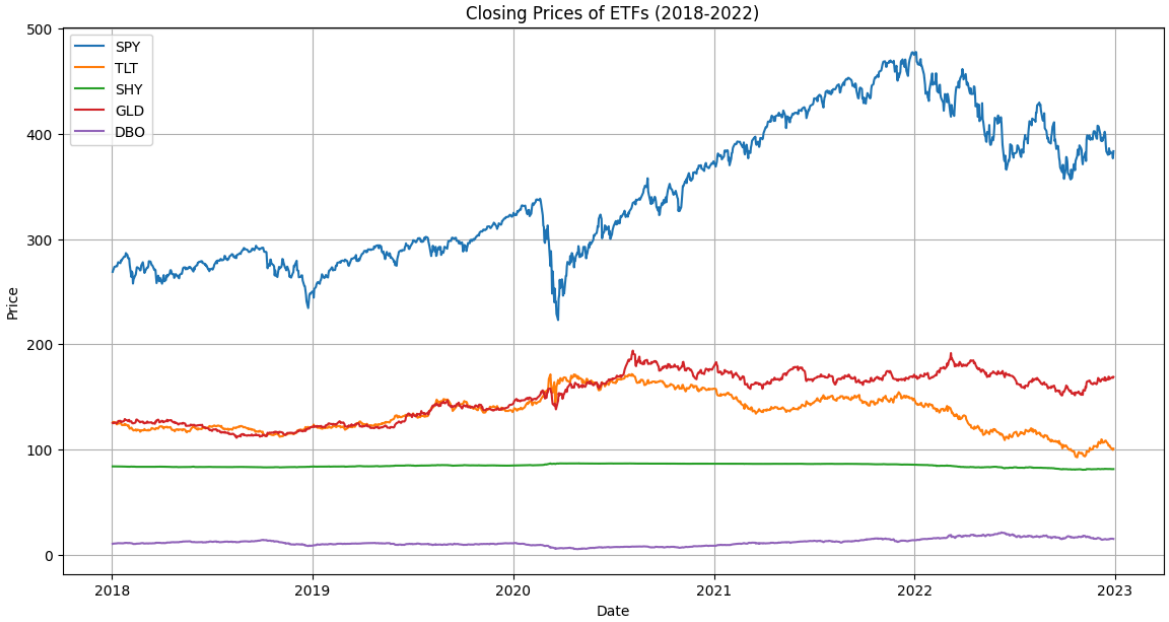
```
<class 'pandas.core.frame.DataFrame'>
Index: 1258 entries, 2018-01-02 to 2022-12-29
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   SPY     1258 non-null   float64
 1   TLT     1258 non-null   float64
 2   SHY     1258 non-null   float64
 3   GLD     1258 non-null   float64
 4   DBO     1258 non-null   float64
dtypes: float64(5)
memory usage: 59.0+ KB
```

Now we will start our EDA, we will see some descriptive statistics of the dataset then we will find the return and plot it.

In [4]:
```
# Save data for reproducibility
close_prices.to_csv("etf_close_prices.csv")

# Step 1b: Exploratory Data Analysis
# 1. Summary Statistics
summary_stats = close_prices.describe()
print("Summary Statistics:\n", summary_stats)
```

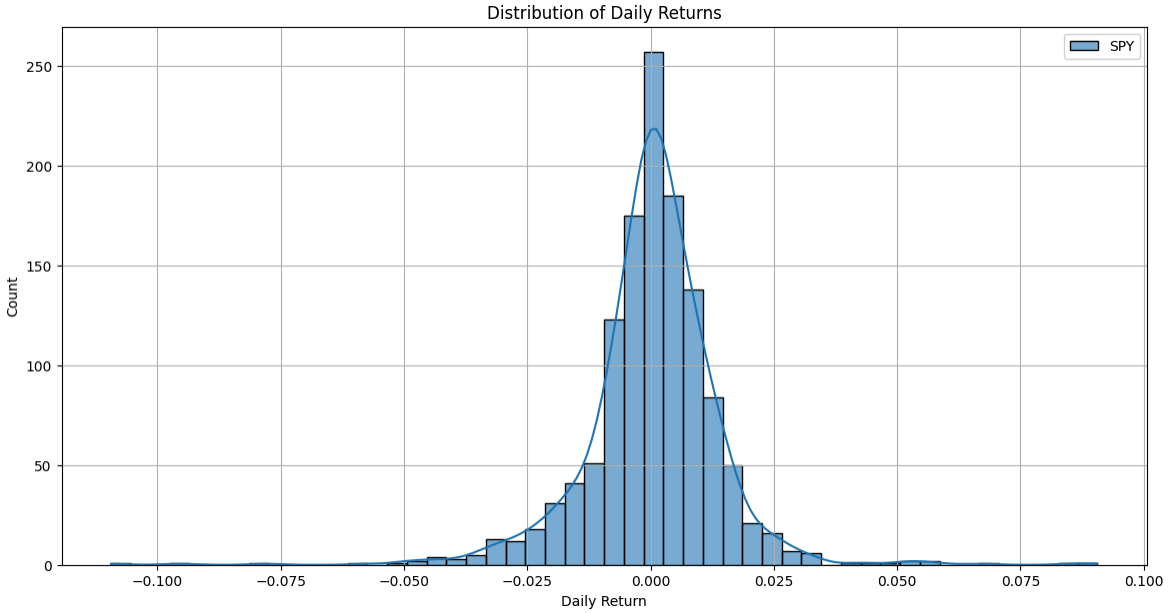
```
Summary Statistics:
 Ticker          SPY          TLT          SHY          GLD          DBO
count   1258.000000  1258.000000  1258.000000  1258.000000  1258.000000
mean     344.308625   134.630079    84.553426   150.921431    11.588172
std       66.613497    18.284340     1.595424    22.501465     3.341802
min      222.949997    92.400002    80.589996   111.099998     5.200000
25%      283.965012   119.635002    83.269997   125.865000     9.530000
50%      326.759995   136.504997    84.620003   159.434998    10.915000
75%      402.577507   148.054996    86.250000   169.345005    13.252500
max      477.709991   171.570007    86.800003   193.889999    21.080000
```

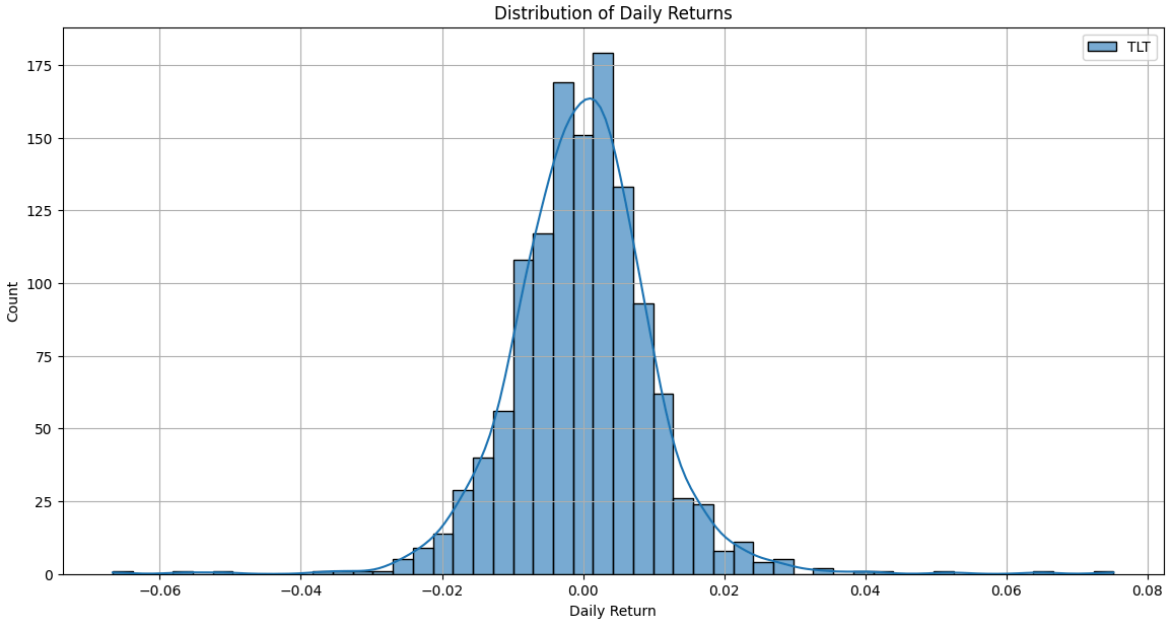
In [5]:
```
# 2. Daily Returns
daily_returns = close_prices.pct_change().dropna()

# Visualize closing prices
plt.figure(figsize=(14, 7))
for etf in etfs:
    plt.plot(close_prices[etf], label=etf)
plt.title("Closing Prices of ETFs (2018-2022)")
plt.xlabel("Date")
plt.ylabel("Price")
plt.legend()
plt.grid()
plt.show()
```

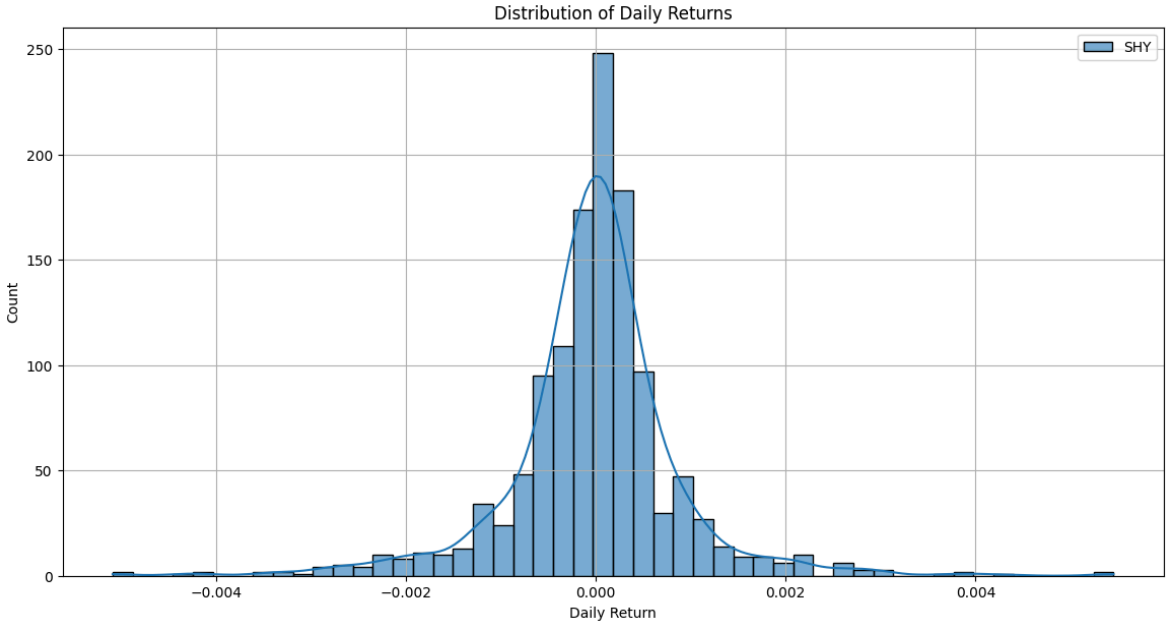


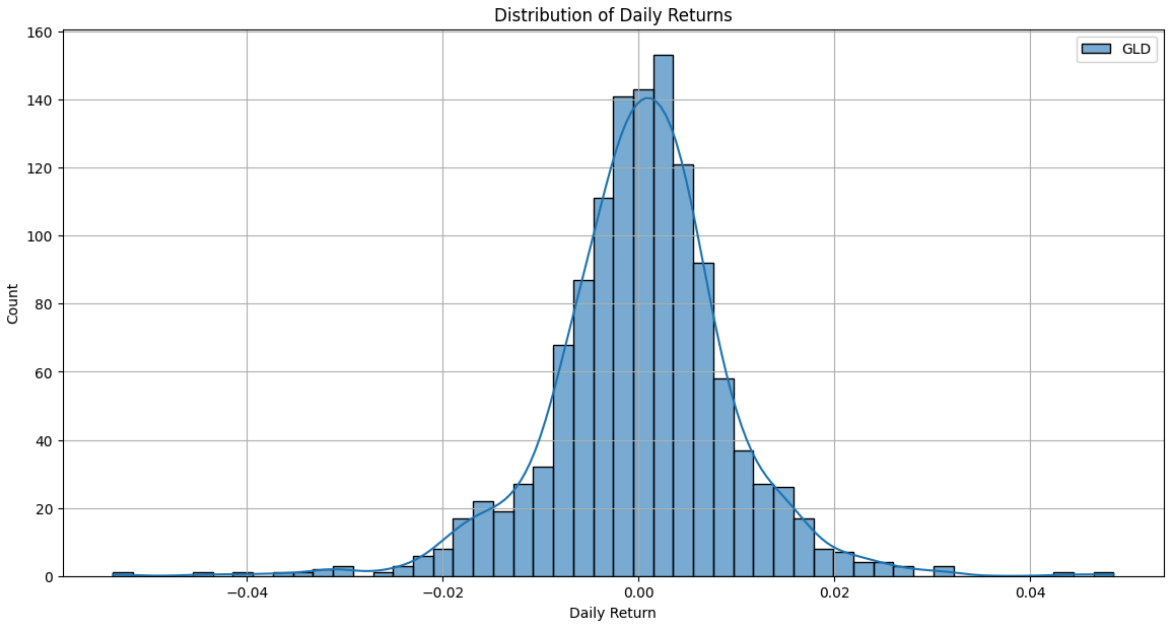
Here will visulaize daily return histogram and we will see distribution of our returns.

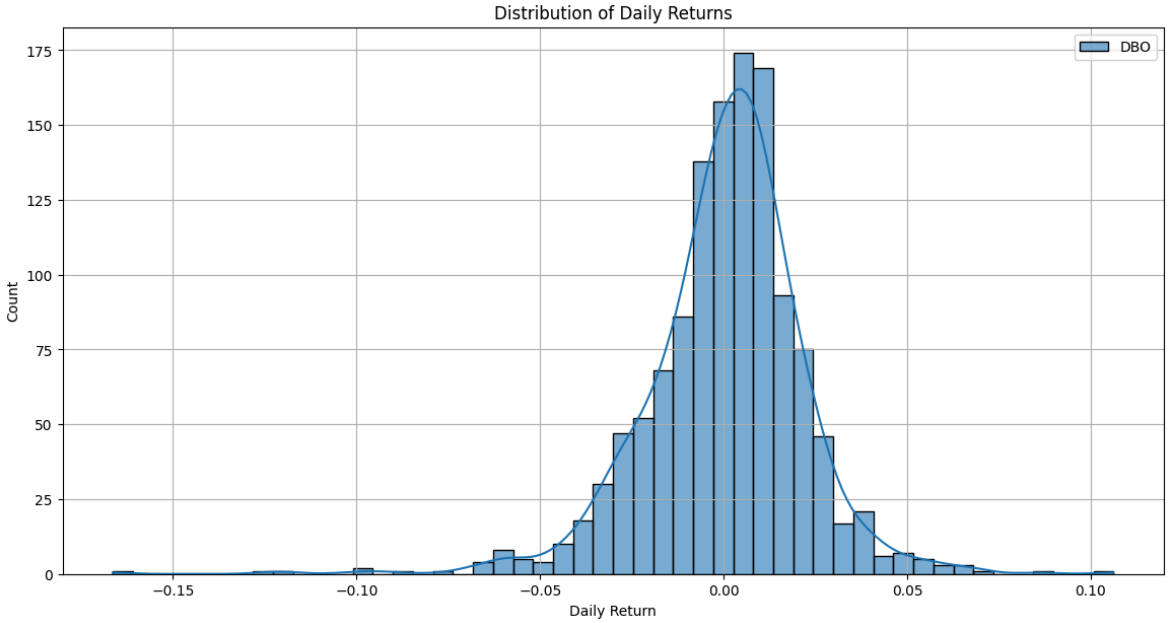
In [6]:
```
# Visualize daily returns distributions

for etf in etfs:
  plt.figure(figsize=(14, 7))
  sns.histplot(daily_returns[etf], kde=True, label=etf, bins=50, alpha=0.6)
  plt.title("Distribution of Daily Returns")
  plt.xlabel("Daily Return")
  plt.legend()
  plt.grid()
  plt.show()
```




Distribution of Daily Returns
TLT
Count
Daily Return

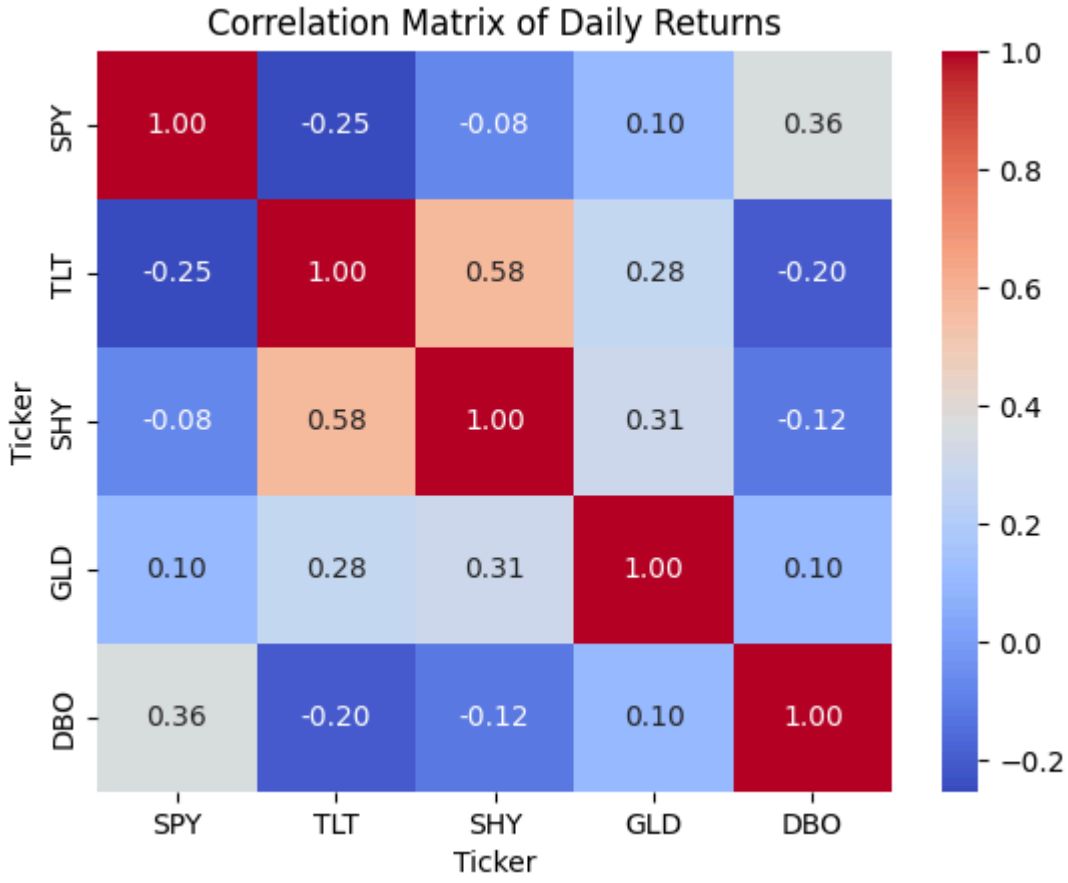

Distribution of Daily Returns
SHY
Count
Daily Return


Distribution of Daily Returns
GLD
Count
Daily Return



Here is the correlations matrix of the retruns

In [7]:
```
# Correlation Matrix of Returns
correlation_matrix = daily_returns.corr()
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Correlation Matrix of Daily Returns")
plt.show()
```



Now we will do ADF to check for stationarity for our time series dataset.

In [8]:
```
# Stationarity Test (Augmented Dickey-Fuller)
from statsmodels.tsa.stattools import adfuller
```

```
def test_stationarity(series, name):
    result = adfuller(series)
    print(f"ADF Statistic for {name}: {result[0]:.4f}")
    print(f"p-value for {name}: {result[1]:.4f}")

for etf in etfs:
    test_stationarity(close_prices[etf].dropna(), etf)
```

```
ADF Statistic for SPY: -1.2506
p-value for SPY: 0.6515
ADF Statistic for TLT: -0.4058
p-value for TLT: 0.9091
ADF Statistic for SHY: -0.0643
p-value for SHY: 0.9529
ADF Statistic for GLD: -1.1950
p-value for GLD: 0.6757
ADF Statistic for DBO: -1.0006
p-value for DBO: 0.7531
```

Here we can see returns of our etf's are not stationary, so will will move forward to build a stationary dataset.

In [9]:
```
close_prices
```

Out[9]:

Ticker	SPY	TLT	SHY	GLD	DBO
2018-01-02	268.769989	125.489998	83.820000	125.150002	10.19
2018-01-03	270.470001	126.089996	83.820000	124.820000	10.41
2018-01-04	271.609985	126.070000	83.779999	125.459999	10.43
2018-01-05	273.420013	125.709999	83.779999	125.330002	10.38
2018-01-08	273.920013	125.629997	83.779999	125.309998	10.44
...	...	...	...	...	...
2022-12-22	380.720001	103.680000	81.360001	166.759995	14.64
2022-12-23	382.910004	102.160004	81.320000	167.259995	15.00
2022-12-27	381.399994	100.139999	81.209999	168.669998	15.11
2022-12-28	376.660004	99.550003	81.209999	167.910004	14.95
2022-12-29	383.440002	100.680000	81.269997	168.850006	14.88

1258 rows × 5 columns

To build a stationary Dataset we will first fins the Log returns of the etf's, you can also see it in plot below.

In [10]:
```
# Log Returns Transformation for Stationarity
log_returns = np.log(close_prices / close_prices.shift(1)).dropna()
```

In [11]:
```
# Plot Transformed Series
plt.figure(figsize=(12, 6))
```

```
plt.plot(log_returns, label='Log Returns')
plt.title('Stationary Log Returns')
plt.legend()

plt.tight_layout()
plt.show()
```

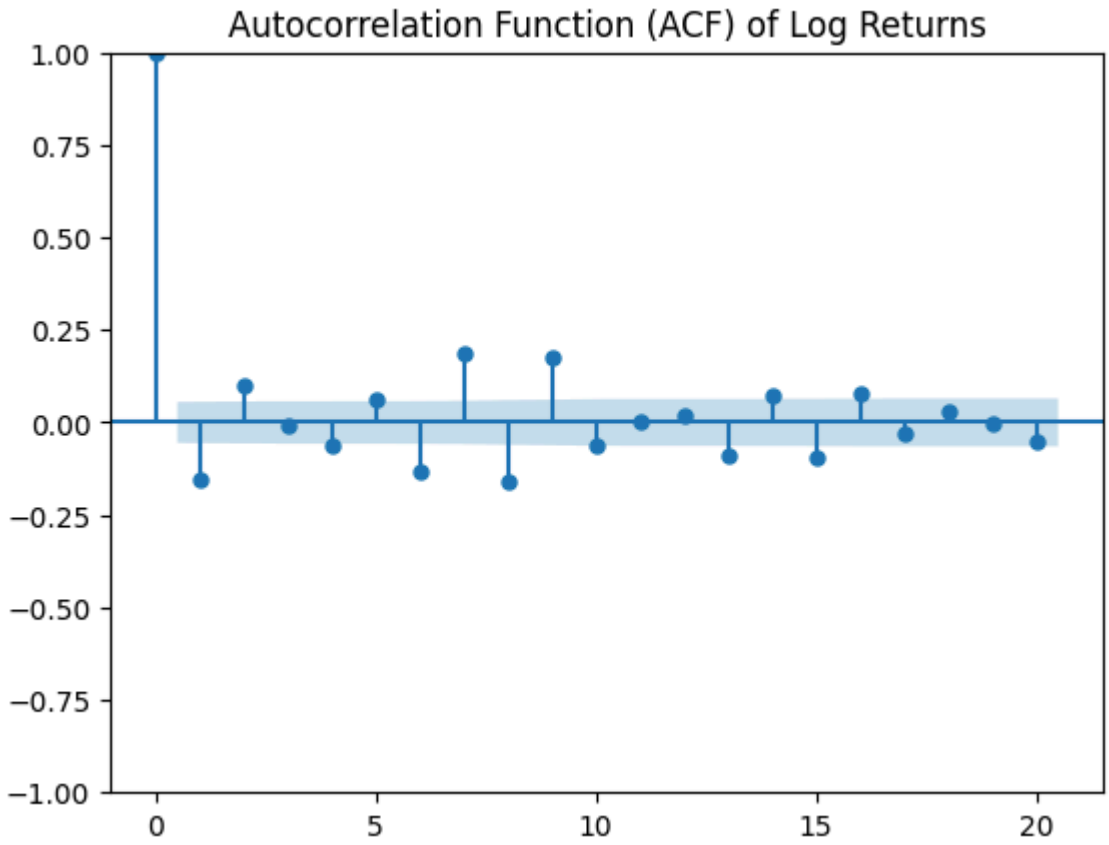
Here is the descriptive statistics summary for all the etf's log return.

In [12]:
```
# Summary statistics
for etf in etfs:
    print(f"Summary statistics for {etf}:")
    print(log_returns[etf].describe())
    print("\n")
```

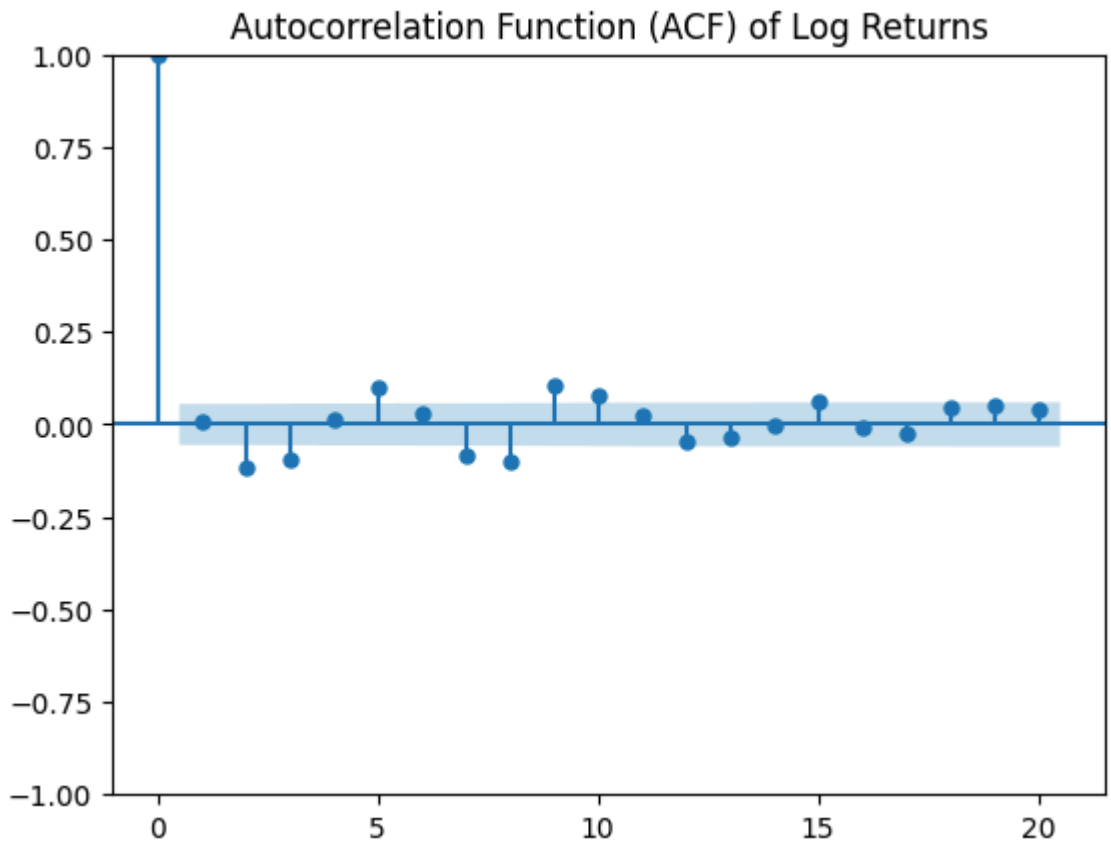
```
Summary statistics for SPY:
count    1257.000000
mean        0.000283
std         0.013687
min        -0.115887
25%        -0.005266
50%         0.000727
75%         0.007088
max         0.086731
Name: SPY, dtype: float64


Summary statistics for TLT:
count    1257.000000
mean       -0.000175
std         0.010159
min        -0.069010
25%        -0.005884
50%         0.000000
75%         0.005495
max         0.072503
Name: TLT, dtype: float64


Summary statistics for SHY:
count    1257.000000
mean       -0.000025
std         0.000918
min        -0.005101
25%        -0.000354
50%         0.000000
75%         0.000351
max         0.005438
Name: SHY, dtype: float64


Summary statistics for GLD:
count    1257.000000
mean        0.000238
std         0.009057
min        -0.055190
25%        -0.004336
50%         0.000518
75%         0.005080
max         0.047390
Name: GLD, dtype: float64


Summary statistics for DBO:
count    1257.000000
mean        0.000301
std         0.021876
min        -0.182065
25%        -0.009655
50%         0.002411
75%         0.012613
max         0.100955
Name: DBO, dtype: float64
```

Now we will do ACF plots of our return to check for stationarity, and below you guys can see the ACF plots suggest that the data is stationary.

In [13]:
```
for etf in etfs:
  # Autocorrelation plot to check for persistence
  plot_acf(log_returns[etf], lags=20)
  plt.title("Autocorrelation Function (ACF) of Log Returns")
  plt.show()

  # Ljung-Box test for autocorrelation
  ljung_box_result = acorr_ljungbox(log_returns[etf], lags=[10], return_df=True)
  print("Ljung-Box Test for Autocorrelation (lags=10):")
  print(ljung_box_result)
```

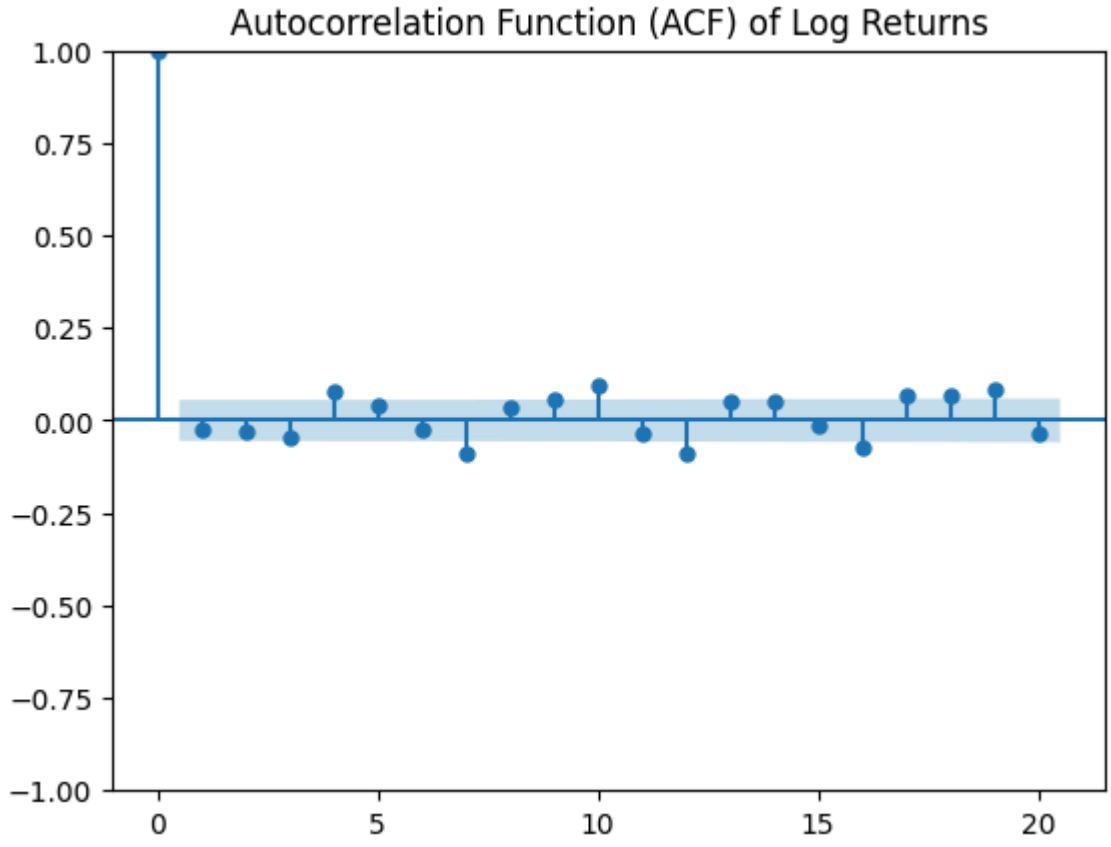



```
Ljung-Box Test for Autocorrelation (lags=10):
       lb_stat     lb_pvalue
10  195.354275  1.500667e-36
```



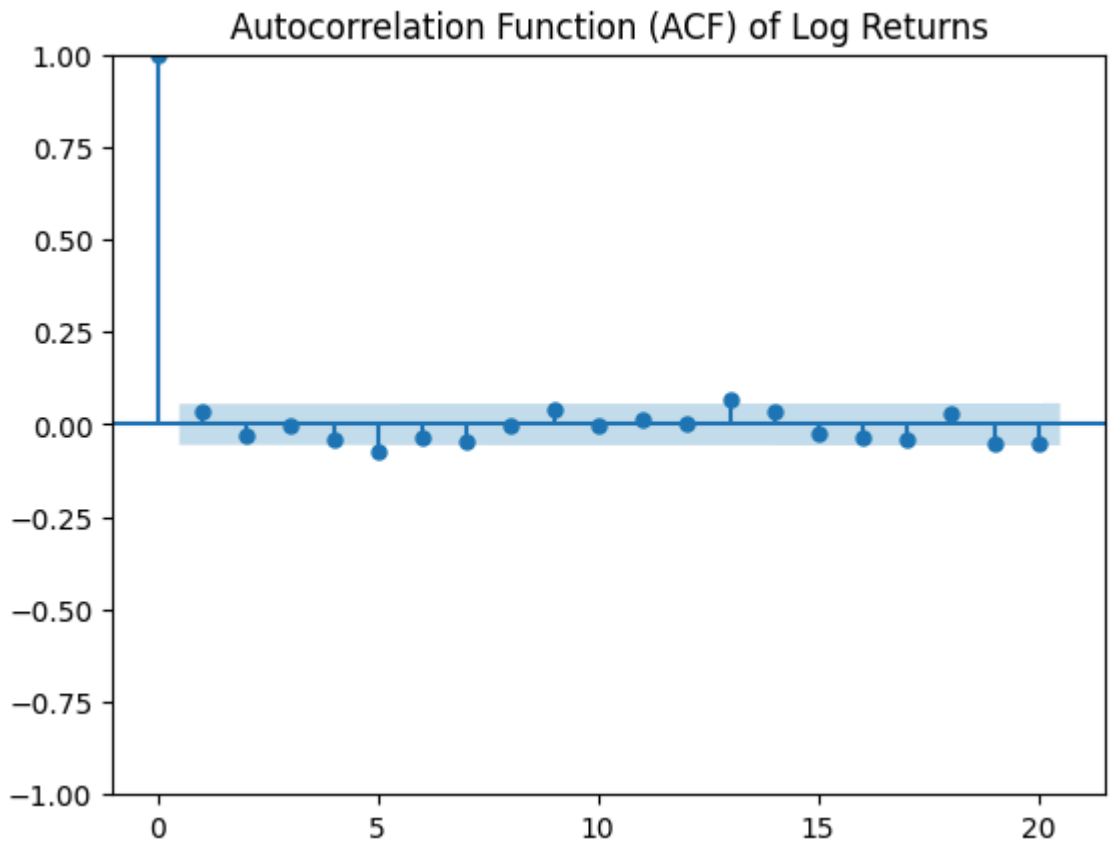
Ljung-Box Test for Autocorrelation (lags=10):

	lb_stat	lb_pvalue
10	86.175949	3.059002e-14

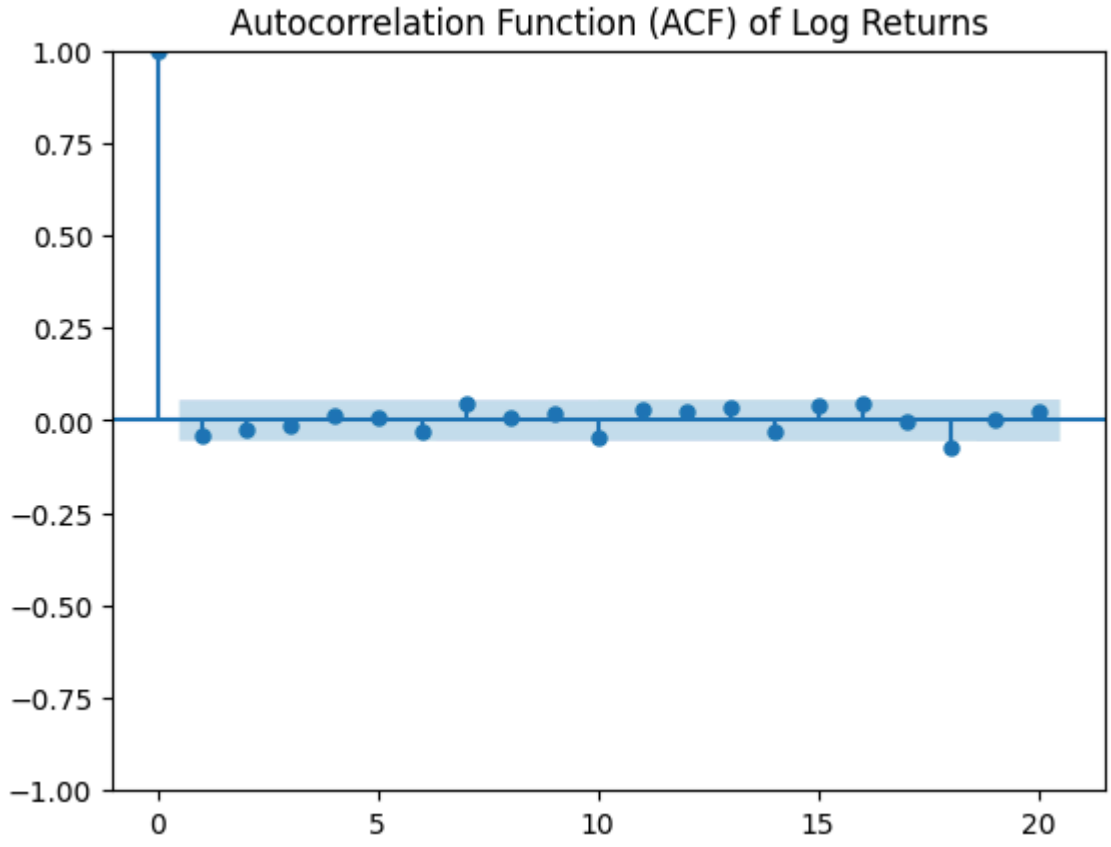


Ljung-Box Test for Autocorrelation (lags=10):

	lb_stat	lb_pvalue
10	41.980712	0.000008




```
Ljung-Box Test for Autocorrelation (lags=10):
      lb_stat  lb_pvalue
10  17.283599    0.06832
```




```
Ljung-Box Test for Autocorrelation (lags=10):
      lb_stat  lb_pvalue
10  10.509729   0.396966
```

And finally we will do the ADF test to confirm for our stationarity in our dataset and the ADFtest suggest that our data is indeed stationary.

In [14]:
```
def test_stationarity(series, name):
    result = adfuller(series)
    print(f"ADF Statistic for {name}: {result[0]:.4f}")
    print(f"p-value for {name}: {result[1]:.4f}")

for etf in etfs:
    test_stationarity(log_returns[etf].dropna(), etf)
```

```
ADF Statistic for SPY: -10.8025
p-value for SPY: 0.0000
ADF Statistic for TLT: -10.6434
p-value for TLT: 0.0000
ADF Statistic for SHY: -6.6507
p-value for SHY: 0.0000
ADF Statistic for GLD: -15.4176
p-value for GLD: 0.0000
ADF Statistic for DBO: -36.9405
p-value for DBO: 0.0000
```

Step 2

Now In this step we will build DL models using RNN, here how we are going to do :

- We will first build a DL model that uses LSTM network architecture for each of the 5 different asset classes, to predict the 25-day ahead return of each ETF.
- After that we will Train each 5 models and do an in sample predective performance and also Test each of the model out of sample and compare the results of both across the assets.
- Then we will a trading strategy that uses the out-of-sample predection of all the different models and the backtest it check our predective performance.

Data prep

We will first start with our data prepration, we will first copy the log returns in a new data set to work further, and prepare it accordingly so that the data can be used as input for each model.

In [15]:
```
df = log_returns.copy()
```

In [16]:
```
df
```

Out[16]:

Ticker	SPY	TLT	SHY	GLD	DBO
2018-01-03	0.006305	0.004770	0.000000	-0.002640	0.021360
2018-01-04	0.004206	-0.000159	-0.000477	0.005114	0.001919
2018-01-05	0.006642	-0.002860	0.000000	-0.001037	-0.004805
2018-01-08	0.001827	-0.000637	0.000000	-0.000160	0.005764
2018-01-09	0.002261	-0.013463	-0.000358	-0.004639	0.017094
...	...	...	...	...	...
2022-12-22	-0.014369	-0.000193	-0.000614	-0.012159	-0.007486
2022-12-23	0.005736	-0.014769	-0.000492	0.002994	0.024293
2022-12-27	-0.003951	-0.019971	-0.001354	0.008395	0.007307
2022-12-28	-0.012506	-0.005909	0.000000	-0.004516	-0.010645
2022-12-29	0.017840	0.011287	0.000739	0.005583	-0.004693

1257 rows × 5 columns

In [17]:
```
df.reset_index(inplace=True)

# Rename 'Ticker' to 'Date'
df.rename(columns={"index": "Date"}, inplace=True)

df
```

Out[17]:

Ticker	Date	SPY	TLT	SHY	GLD	DBO
0	2018-01-03	0.006305	0.004770	0.000000	-0.002640	0.021360
1	2018-01-04	0.004206	-0.000159	-0.000477	0.005114	0.001919
2	2018-01-05	0.006642	-0.002860	0.000000	-0.001037	-0.004805
3	2018-01-08	0.001827	-0.000637	0.000000	-0.000160	0.005764
4	2018-01-09	0.002261	-0.013463	-0.000358	-0.004639	0.017094
...	...	...	...	...	...	...
1252	2022-12-22	-0.014369	-0.000193	-0.000614	-0.012159	-0.007486
1253	2022-12-23	0.005736	-0.014769	-0.000492	0.002994	0.024293
1254	2022-12-27	-0.003951	-0.019971	-0.001354	0.008395	0.007307
1255	2022-12-28	-0.012506	-0.005909	0.000000	-0.004516	-0.010645
1256	2022-12-29	0.017840	0.011287	0.000739	0.005583	-0.004693

1257 rows × 6 columns

In [18]:
```
df = df.reindex(
    columns=[
```

```
        "Date",
        "SPY",
        "TLT",
        "SHY",
        "GLD",
        "DBO",
    ]
)

df.head()
```

Out[18]:

Ticker	Date	SPY	TLT	SHY	GLD	DBO
0	2018-01-03	0.006305	0.004770	0.000000	-0.002640	0.021360
1	2018-01-04	0.004206	-0.000159	-0.000477	0.005114	0.001919
2	2018-01-05	0.006642	-0.002860	0.000000	-0.001037	-0.004805
3	2018-01-08	0.001827	-0.000637	0.000000	-0.000160	0.005764
4	2018-01-09	0.002261	-0.013463	-0.000358	-0.004639	0.017094

All model description :

Here is the model description for all the models, As we are using same data structure, LSTM architecture we are going to define our models workflow at one go here:

- First we will prepare the data for each model as we have to build individual model for each ETF, in this we will copy only the required ETF that is needed for that model
- Then we will prepare the data for input by doing some feature engineering in which we will calculate 1-day, 10-days, and 50-days cumulative returns of repective ETF to predict the 25-day ahead return.
- After that we will define our data into train, test and validation set, also we will also have an window i.e sequence length of 30 as we are using LSTM architecture.
- In the next step we will we will scale the different variables under the min max scalar to ease our computation then finally we will split the data in train and test split.
- Now Finally we will define our LSTM model architecture, which will be same for all the respective ETF models :

> our model architecture consists of a sequential LSTM network which we have designed for regression. It begins with stacking three LSTM layers, each containing 50 units and utilizing the tanh activation function, with L2 regularization (L2=0.01) applied to the kernel weights to prevent overfitting. The first two LSTM layers return sequences, allowing the propagation of sequential information through the network, while the third LSTM layer does not return sequences, outputting only the final hidden state. Each LSTM layer is followed by a dropout layer with a dropout rate of 0.2, ensuring robustness and generalization. After the

> LSTM layers, a dense layer with 20 units and ReLU activation is added, followed by another dropout layer. The network ends with an output layer comprising a single neuron with a linear activation function, suitable for regression predictions. The model is compiled with the Adam optimizer, using a learning rate of 1e-4, and the loss function is mean absolute error (MAE). Early stopping is also included to monitor the validation loss, with a patience of 5 epochs and the ability to restore the best-performing weights.

- After that we will final features (learning rate, loss function, early stopping criteria, batch size, ...) and train the model, and after training we need to do the job of testing and to find how our model is performing we will do and final features (learning rate, loss function, early stopping criteria, batch size, ...) and train the model, and to check the perfomance of our model we will do an out of sample predection using the standard out-of-sample R-squared measured and check the results.

LSTM Model 1 : SPY

In [19]:
```
df_spy = df[["Date", "SPY"]].copy()
df_spy
```

Out[19]:

Ticker	Date	SPY
0	2018-01-03	0.006305
1	2018-01-04	0.004206
2	2018-01-05	0.006642
3	2018-01-08	0.001827
4	2018-01-09	0.002261
...	...	...
1252	2022-12-22	-0.014369
1253	2022-12-23	0.005736
1254	2022-12-27	-0.003951
1255	2022-12-28	-0.012506
1256	2022-12-29	0.017840

1257 rows × 2 columns

In [20]:
```
df_spy["Ret_10"] = df_spy["SPY"].rolling(10).apply(lambda x: np.exp(np.sum(x)) -
df_spy["Ret_50"] = df_spy["SPY"].rolling(50).apply(lambda x: np.exp(np.sum(x)) -
df_spy["Ret_25"] = df_spy["SPY"].rolling(25).apply(lambda x: np.exp(np.sum(x)) -

df_spy["Ret25"] = df_spy["Ret_25"].shift(-25)
del df_spy["Ret_25"]
```

```
df_spy = df_spy.dropna()
df_spy.head()
```

Out[20]:

Ticker	Date	SPY	Ret_10	Ret_50	Ret25
49	2018-03-15	-0.001090	0.027269	0.023180	-0.030509
50	2018-03-16	-0.002913	0.019028	0.013791	-0.027826
51	2018-03-19	-0.013623	-0.006246	-0.004124	-0.027764
52	2018-03-20	0.001699	-0.007073	-0.009034	-0.027016
53	2018-03-21	-0.001921	-0.008615	-0.012741	-0.015235

In [21]:
```
Xdf, ydf = df_spy.iloc[:, 1:-1], df_spy.iloc[:, -1]
X = Xdf.astype("float32")
y = ydf.astype("float32")
```

In [22]:
```
val_split = 0.2
train_split = 0.625
train_size = int(len(df_spy) * train_split)
val_size = int(train_size * val_split)
test_size = int(len(df_spy) - train_size)

window_size = 30

ts = test_size
split_time = len(df_spy) - ts
test_time = df_spy.iloc[split_time + window_size :, 0:1].values


y_train_set = y[:split_time]
y_test_set = y[split_time:]

X_train_set = X[:split_time]
X_test_set = X[split_time:]

n_features = X_train_set.shape[1]
```

In [23]:
```
scaler_input = MinMaxScaler(feature_range=(-1, 1))
scaler_input.fit(X_train_set)
X_train_set_scaled = scaler_input.transform(X_train_set)
X_test_set_scaled = scaler_input.transform(X_test_set)

mean_ret = np.mean(y_train_set)

scaler_output = MinMaxScaler(feature_range=(-1, 1))
y_train_set = y_train_set.values.reshape(len(y_train_set), 1)
y_test_set = y_test_set.values.reshape(len(y_test_set), 1)
scaler_output.fit(y_train_set)
y_train_set_scaled = scaler_output.transform(y_train_set)
```

In [24]:
```
training_time = df_spy.iloc[:split_time, 0:1].values

X_train = []
y_train = []

for i in range(window_size, y_train_set_scaled.shape[0]):
```

```
    X_train.append(X_train_set_scaled[i - window_size : i, :])
    y_train.append(y_train_set_scaled[i])

X_train, y_train = np.array(X_train), np.array(y_train)

print("Shape of training data", X_train.shape, y_train.shape)

X_test = []
y_test = y_test_set

for i in range(window_size, y_test_set.shape[0]):
    X_test.append(X_test_set_scaled[i - window_size : i, :])

X_test, y_test = np.array(X_test), np.array(y_test)

print("Shape of test data", X_test.shape, y_test.shape)
```

```
Shape of training data (709, 30, 3) (709, 1)
Shape of test data (414, 30, 3) (444, 1)
```

In [25]:
```
from keras.regularizers import l2
from keras.optimizers import Adam
```

In [26]:
```
SEED = 321
# Define hyperparameters
units_lstm = 50
n_dropout = 0.2
act_fun = "relu"

# Define the model
model = Sequential()

# First LSTM layer with Tanh activation and L2 regularization
model.add(
    LSTM(
        units=units_lstm,
        return_sequences=True,
        activation="tanh",
        input_shape=(X_train.shape[1], n_features),
        kernel_regularizer=l2(0.01)
    )
)
model.add(Dropout(n_dropout, seed=SEED))

# Second LSTM layer with Tanh activation and L2 regularization
model.add(LSTM(units=units_lstm, return_sequences=True, activation="tanh", kerne
model.add(Dropout(n_dropout, seed=SEED))

# Third LSTM layer with Tanh activation and L2 regularization
model.add(LSTM(units=units_lstm, return_sequences=False, activation="tanh", kern
model.add(Dropout(n_dropout, seed=SEED))

# Dense layer with ReLU activation
model.add(Dense(units=20, activation=act_fun))
model.add(Dropout(n_dropout, seed=SEED))

# Output layer with Linear activation (for regression)
model.add(Dense(units=1, activation='linear'))

# Compile the model with Adam optimizer and MAE loss
```

```
hp_lr = 1e-4
model.compile(optimizer=Adam(learning_rate=hp_lr), loss='mean_absolute_error')

es = EarlyStopping(
    monitor="val_loss", mode="min", verbose=0, patience=5, restore_best_weights=
)

# Summary of the model
model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/rnn.py:204: UserWarn
ing: 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 m
odel instead.
  super().__init__(**kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	
lstm (LSTM)	(None, 30, 50)	
dropout (Dropout)	(None, 30, 50)	
lstm_1 (LSTM)	(None, 30, 50)	
dropout_1 (Dropout)	(None, 30, 50)	
lstm_2 (LSTM)	(None, 50)	
dropout_2 (Dropout)	(None, 50)	
dense (Dense)	(None, 20)	
dropout_3 (Dropout)	(None, 20)	
dense_1 (Dense)	(None, 1)	

Total params: 52,241 (204.07 KB)

Trainable params: 52,241 (204.07 KB)

Non-trainable params: 0 (0.00 B)

In [27]:
```
# fit the models
model.fit(
    X_train,
    y_train,
    validation_split=val_split,
    epochs=100,
    batch_size=64,
    verbose=2,
    callbacks=[es],
)
```

```
Epoch 1/100
9/9 - 6s - 635ms/step - loss: 1.8691 - val_loss: 1.8151
Epoch 2/100
9/9 - 2s - 223ms/step - loss: 1.8311 - val_loss: 1.7757
Epoch 3/100
9/9 - 0s - 32ms/step - loss: 1.7961 - val_loss: 1.7382
Epoch 4/100
9/9 - 0s - 16ms/step - loss: 1.7606 - val_loss: 1.7012
Epoch 5/100
9/9 - 0s - 16ms/step - loss: 1.7251 - val_loss: 1.6626
Epoch 6/100
9/9 - 0s - 17ms/step - loss: 1.6906 - val_loss: 1.6211
Epoch 7/100
9/9 - 0s - 14ms/step - loss: 1.6544 - val_loss: 1.5800
Epoch 8/100
9/9 - 0s - 17ms/step - loss: 1.6255 - val_loss: 1.5423
Epoch 9/100
9/9 - 0s - 37ms/step - loss: 1.5989 - val_loss: 1.5150
Epoch 10/100
9/9 - 0s - 34ms/step - loss: 1.5734 - val_loss: 1.4945
Epoch 11/100
9/9 - 0s - 33ms/step - loss: 1.5477 - val_loss: 1.4730
Epoch 12/100
9/9 - 0s - 21ms/step - loss: 1.5250 - val_loss: 1.4521
Epoch 13/100
9/9 - 0s - 34ms/step - loss: 1.5027 - val_loss: 1.4292
Epoch 14/100
9/9 - 0s - 21ms/step - loss: 1.4784 - val_loss: 1.4054
Epoch 15/100
9/9 - 0s - 23ms/step - loss: 1.4541 - val_loss: 1.3809
Epoch 16/100
9/9 - 0s - 25ms/step - loss: 1.4300 - val_loss: 1.3583
Epoch 17/100
9/9 - 0s - 27ms/step - loss: 1.4090 - val_loss: 1.3356
Epoch 18/100
9/9 - 0s - 32ms/step - loss: 1.3886 - val_loss: 1.3141
Epoch 19/100
9/9 - 0s - 15ms/step - loss: 1.3630 - val_loss: 1.2927
Epoch 20/100
9/9 - 0s - 33ms/step - loss: 1.3410 - val_loss: 1.2719
Epoch 21/100
9/9 - 0s - 15ms/step - loss: 1.3225 - val_loss: 1.2532
Epoch 22/100
9/9 - 0s - 15ms/step - loss: 1.2990 - val_loss: 1.2328
Epoch 23/100
9/9 - 0s - 15ms/step - loss: 1.2805 - val_loss: 1.2116
Epoch 24/100
9/9 - 0s - 14ms/step - loss: 1.2611 - val_loss: 1.1951
Epoch 25/100
9/9 - 0s - 16ms/step - loss: 1.2408 - val_loss: 1.1743
Epoch 26/100
9/9 - 0s - 15ms/step - loss: 1.2233 - val_loss: 1.1544
Epoch 27/100
9/9 - 0s - 13ms/step - loss: 1.2040 - val_loss: 1.1334
Epoch 28/100
9/9 - 0s - 16ms/step - loss: 1.1823 - val_loss: 1.1174
Epoch 29/100
9/9 - 0s - 13ms/step - loss: 1.1660 - val_loss: 1.0988
Epoch 30/100
9/9 - 0s - 15ms/step - loss: 1.1465 - val_loss: 1.0794
```

```
Epoch 31/100
9/9 - 0s - 14ms/step - loss: 1.1306 - val_loss: 1.0596
Epoch 32/100
9/9 - 0s - 15ms/step - loss: 1.1142 - val_loss: 1.0447
Epoch 33/100
9/9 - 0s - 16ms/step - loss: 1.0955 - val_loss: 1.0295
Epoch 34/100
9/9 - 0s - 15ms/step - loss: 1.0755 - val_loss: 1.0122
Epoch 35/100
9/9 - 0s - 15ms/step - loss: 1.0608 - val_loss: 0.9940
Epoch 36/100
9/9 - 0s - 13ms/step - loss: 1.0440 - val_loss: 0.9758
Epoch 37/100
9/9 - 0s - 17ms/step - loss: 1.0279 - val_loss: 0.9609
Epoch 38/100
9/9 - 0s - 13ms/step - loss: 1.0106 - val_loss: 0.9467
Epoch 39/100
9/9 - 0s - 14ms/step - loss: 0.9976 - val_loss: 0.9302
Epoch 40/100
9/9 - 0s - 13ms/step - loss: 0.9822 - val_loss: 0.9175
Epoch 41/100
9/9 - 0s - 14ms/step - loss: 0.9649 - val_loss: 0.8991
Epoch 42/100
9/9 - 0s - 14ms/step - loss: 0.9512 - val_loss: 0.8822
Epoch 43/100
9/9 - 0s - 16ms/step - loss: 0.9373 - val_loss: 0.8694
Epoch 44/100
9/9 - 0s - 15ms/step - loss: 0.9225 - val_loss: 0.8555
Epoch 45/100
9/9 - 0s - 15ms/step - loss: 0.9101 - val_loss: 0.8448
Epoch 46/100
9/9 - 0s - 14ms/step - loss: 0.8951 - val_loss: 0.8328
Epoch 47/100
9/9 - 0s - 17ms/step - loss: 0.8810 - val_loss: 0.8173
Epoch 48/100
9/9 - 0s - 15ms/step - loss: 0.8673 - val_loss: 0.8023
Epoch 49/100
9/9 - 0s - 16ms/step - loss: 0.8547 - val_loss: 0.7869
Epoch 50/100
9/9 - 0s - 31ms/step - loss: 0.8426 - val_loss: 0.7741
Epoch 51/100
9/9 - 0s - 14ms/step - loss: 0.8273 - val_loss: 0.7608
Epoch 52/100
9/9 - 0s - 14ms/step - loss: 0.8175 - val_loss: 0.7507
Epoch 53/100
9/9 - 0s - 15ms/step - loss: 0.8026 - val_loss: 0.7391
Epoch 54/100
9/9 - 0s - 14ms/step - loss: 0.7919 - val_loss: 0.7256
Epoch 55/100
9/9 - 0s - 15ms/step - loss: 0.7763 - val_loss: 0.7149
Epoch 56/100
9/9 - 0s - 16ms/step - loss: 0.7658 - val_loss: 0.7020
Epoch 57/100
9/9 - 0s - 13ms/step - loss: 0.7573 - val_loss: 0.6935
Epoch 58/100
9/9 - 0s - 14ms/step - loss: 0.7440 - val_loss: 0.6814
Epoch 59/100
9/9 - 0s - 13ms/step - loss: 0.7324 - val_loss: 0.6671
Epoch 60/100
9/9 - 0s - 16ms/step - loss: 0.7225 - val_loss: 0.6559
```

```
Epoch 61/100
9/9 - 0s - 13ms/step - loss: 0.7100 - val_loss: 0.6488
Epoch 62/100
9/9 - 0s - 15ms/step - loss: 0.6964 - val_loss: 0.6368
Epoch 63/100
9/9 - 0s - 16ms/step - loss: 0.6887 - val_loss: 0.6266
Epoch 64/100
9/9 - 0s - 16ms/step - loss: 0.6803 - val_loss: 0.6164
Epoch 65/100
9/9 - 0s - 14ms/step - loss: 0.6661 - val_loss: 0.6042
Epoch 66/100
9/9 - 0s - 14ms/step - loss: 0.6564 - val_loss: 0.5931
Epoch 67/100
9/9 - 0s - 16ms/step - loss: 0.6511 - val_loss: 0.5856
Epoch 68/100
9/9 - 0s - 15ms/step - loss: 0.6408 - val_loss: 0.5787
Epoch 69/100
9/9 - 0s - 14ms/step - loss: 0.6292 - val_loss: 0.5688
Epoch 70/100
9/9 - 0s - 15ms/step - loss: 0.6222 - val_loss: 0.5580
Epoch 71/100
9/9 - 0s - 16ms/step - loss: 0.6125 - val_loss: 0.5483
Epoch 72/100
9/9 - 0s - 14ms/step - loss: 0.6005 - val_loss: 0.5407
Epoch 73/100
9/9 - 0s - 15ms/step - loss: 0.5924 - val_loss: 0.5322
Epoch 74/100
9/9 - 0s - 14ms/step - loss: 0.5865 - val_loss: 0.5238
Epoch 75/100
9/9 - 0s - 14ms/step - loss: 0.5728 - val_loss: 0.5125
Epoch 76/100
9/9 - 0s - 15ms/step - loss: 0.5639 - val_loss: 0.5027
Epoch 77/100
9/9 - 0s - 13ms/step - loss: 0.5602 - val_loss: 0.4959
Epoch 78/100
9/9 - 0s - 16ms/step - loss: 0.5520 - val_loss: 0.4901
Epoch 79/100
9/9 - 0s - 14ms/step - loss: 0.5422 - val_loss: 0.4824
Epoch 80/100
9/9 - 0s - 13ms/step - loss: 0.5322 - val_loss: 0.4732
Epoch 81/100
9/9 - 0s - 14ms/step - loss: 0.5252 - val_loss: 0.4661
Epoch 82/100
9/9 - 0s - 15ms/step - loss: 0.5179 - val_loss: 0.4571
Epoch 83/100
9/9 - 0s - 15ms/step - loss: 0.5115 - val_loss: 0.4511
Epoch 84/100
9/9 - 0s - 15ms/step - loss: 0.5037 - val_loss: 0.4463
Epoch 85/100
9/9 - 0s - 16ms/step - loss: 0.4970 - val_loss: 0.4408
Epoch 86/100
9/9 - 0s - 14ms/step - loss: 0.4893 - val_loss: 0.4328
Epoch 87/100
9/9 - 0s - 21ms/step - loss: 0.4821 - val_loss: 0.4257
Epoch 88/100
9/9 - 0s - 34ms/step - loss: 0.4759 - val_loss: 0.4200
Epoch 89/100
9/9 - 0s - 34ms/step - loss: 0.4693 - val_loss: 0.4146
Epoch 90/100
9/9 - 0s - 23ms/step - loss: 0.4621 - val_loss: 0.4062
```

```
Epoch 91/100
9/9 - 0s - 20ms/step - loss: 0.4559 - val_loss: 0.4004
Epoch 92/100
9/9 - 0s - 22ms/step - loss: 0.4504 - val_loss: 0.3938
Epoch 93/100
9/9 - 0s - 22ms/step - loss: 0.4421 - val_loss: 0.3858
Epoch 94/100
9/9 - 0s - 22ms/step - loss: 0.4374 - val_loss: 0.3810
Epoch 95/100
9/9 - 0s - 37ms/step - loss: 0.4298 - val_loss: 0.3755
Epoch 96/100
9/9 - 0s - 24ms/step - loss: 0.4246 - val_loss: 0.3734
Epoch 97/100
9/9 - 0s - 15ms/step - loss: 0.4186 - val_loss: 0.3633
Epoch 98/100
9/9 - 0s - 14ms/step - loss: 0.4090 - val_loss: 0.3579
Epoch 99/100
9/9 - 0s - 15ms/step - loss: 0.4071 - val_loss: 0.3548
Epoch 100/100
9/9 - 0s - 18ms/step - loss: 0.3996 - val_loss: 0.3473
```

Out[27]: <keras.src.callbacks.history.History at 0x7d360bc17430>

In [28]:
```python
#Train and test

prediction = model.predict(X_test)
prediction = scaler_output.inverse_transform(prediction)
prediction = prediction.flatten()

values = np.array(y_test[window_size:])
values = values.flatten()


def R2_campbell(y_true, y_predicted, mean_ret):
    y_predicted = y_predicted.reshape((-1,))
    sse = sum((y_true - y_predicted) ** 2)
    tse = sum((y_true - mean_ret) ** 2)
    r2_score = 1 - (sse / tse)

    return r2_score


R2_Campbell = R2_campbell(values, prediction, mean_ret)
print("Out-of-sample R-squared:", R2_Campbell)

# %%
df_predictions = pd.DataFrame(
    {"Date": test_time.flatten(), "Pred LSTM": prediction, "values": values}
)
df_predictions.head()
```

```
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
Out-of-sample R-squared: -0.07894918546054952
```

Out[28]:

	Date	Pred LSTM	values
0	2021-04-06	0.012331	0.019920
1	2021-04-07	0.012389	-0.002902
2	2021-04-08	0.012460	0.004308
3	2021-04-09	0.012442	0.012370
4	2021-04-12	0.012460	0.009426

convert to numpy to evaluate

In [29]:
```
df1_actual=df_predictions[['values']].to_numpy()
df1_pred=df_predictions[['Pred LSTM']].to_numpy()
```

LSTM Model 2 : TLT

In [30]:
```
df_tlt = df[["Date", "TLT"]].copy()
df_tlt
```

Out[30]:

Ticker	Date	TLT
0	2018-01-03	0.004770
1	2018-01-04	-0.000159
2	2018-01-05	-0.002860
3	2018-01-08	-0.000637
4	2018-01-09	-0.013463
...	...	...
1252	2022-12-22	-0.000193
1253	2022-12-23	-0.014769
1254	2022-12-27	-0.019971
1255	2022-12-28	-0.005909
1256	2022-12-29	0.011287

1257 rows × 2 columns

In [31]:
```
df_tlt["Ret_10"] = df_tlt["TLT"].rolling(10).apply(lambda x: np.exp(np.sum(x)) -
df_tlt["Ret_50"] = df_tlt["TLT"].rolling(50).apply(lambda x: np.exp(np.sum(x)) -
df_tlt["Ret_25"] = df_tlt["TLT"].rolling(25).apply(lambda x: np.exp(np.sum(x)) -

df_tlt["Ret25"] = df_tlt["Ret_25"].shift(-25)
del df_tlt["Ret_25"]
df_tlt = df_tlt.dropna()
df_tlt.head()
```

Out[31]:

	Ticker	Date	TLT	Ret_10	Ret_50	Ret25
49		2018-03-15	0.000000	0.007710	-0.041836	-0.014804
50		2018-03-16	-0.003583	0.012336	-0.049806	-0.011017
51		2018-03-19	-0.003177	0.011861	-0.052669	-0.012392
52		2018-03-20	-0.004027	0.006856	-0.053775	-0.015048
53		2018-03-21	0.001092	0.009067	-0.052137	-0.009237

In [32]:
```
Xdf, ydf = df_tlt.iloc[:, 1:-1], df_tlt.iloc[:, -1]
X = Xdf.astype("float32")
y = ydf.astype("float32")
```

In [33]:
```
val_split = 0.2
train_split = 0.625
train_size = int(len(df_tlt) * train_split)
val_size = int(train_size * val_split)
test_size = int(len(df_tlt) - train_size)

window_size = 30

ts = test_size
split_time = len(df_tlt) - ts
test_time = df_tlt.iloc[split_time + window_size :, 0:1].values


y_train_set = y[:split_time]
y_test_set = y[split_time:]

X_train_set = X[:split_time]
X_test_set = X[split_time:]

n_features = X_train_set.shape[1]
```

In [34]:
```
scaler_input = MinMaxScaler(feature_range=(-1, 1))
scaler_input.fit(X_train_set)
X_train_set_scaled = scaler_input.transform(X_train_set)
X_test_set_scaled = scaler_input.transform(X_test_set)

mean_ret = np.mean(y_train_set)

scaler_output = MinMaxScaler(feature_range=(-1, 1))
y_train_set = y_train_set.values.reshape(len(y_train_set), 1)
y_test_set = y_test_set.values.reshape(len(y_test_set), 1)
scaler_output.fit(y_train_set)
y_train_set_scaled = scaler_output.transform(y_train_set)
```

In [35]:
```
training_time = df_tlt.iloc[:split_time, 0:1].values

X_train = []
y_train = []

for i in range(window_size, y_train_set_scaled.shape[0]):
    X_train.append(X_train_set_scaled[i - window_size : i, :])
    y_train.append(y_train_set_scaled[i])
```

```
X_train, y_train = np.array(X_train), np.array(y_train)

print("Shape of training data", X_train.shape, y_train.shape)

X_test = []
y_test = y_test_set

for i in range(window_size, y_test_set.shape[0]):
    X_test.append(X_test_set_scaled[i - window_size : i, :])

X_test, y_test = np.array(X_test), np.array(y_test)

print("Shape of test data", X_test.shape, y_test.shape)
```

```
Shape of training data (709, 30, 3) (709, 1)
Shape of test data (414, 30, 3) (444, 1)
```

In [36]:
```
from keras.regularizers import l2
from keras.optimizers import Adam
```

In [37]:
```
SEED = 321
# Define hyperparameters
units_lstm = 50
n_dropout = 0.2
act_fun = "relu"

# Define the model
model = Sequential()

# First LSTM layer with Tanh activation and L2 regularization
model.add(
    LSTM(
        units=units_lstm,
        return_sequences=True,
        activation="tanh",
        input_shape=(X_train.shape[1], n_features),
        kernel_regularizer=l2(0.01)
    )
)
model.add(Dropout(n_dropout, seed=SEED))

# Second LSTM layer with Tanh activation and L2 regularization
model.add(LSTM(units=units_lstm, return_sequences=True, activation="tanh", kerne
model.add(Dropout(n_dropout, seed=SEED))

# Third LSTM layer with Tanh activation and L2 regularization
model.add(LSTM(units=units_lstm, return_sequences=False, activation="tanh", kern
model.add(Dropout(n_dropout, seed=SEED))

# Dense layer with ReLU activation
model.add(Dense(units=20, activation=act_fun))
model.add(Dropout(n_dropout, seed=SEED))

# Output layer with Linear activation (for regression)
model.add(Dense(units=1, activation='linear'))

# Compile the model with Adam optimizer and MAE loss
hp_lr = 1e-4
model.compile(optimizer=Adam(learning_rate=hp_lr), loss='mean_absolute_error')
```

```
es = EarlyStopping(
    monitor="val_loss", mode="min", verbose=0, patience=5, restore_best_weights=
)

# Summary of the model
model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/rnn.py:204: UserWarn
ing: 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 m
odel instead.
  super().__init__(**kwargs)
```

Model: "sequential_1"

Layer (type)	Output Shape	
lstm_3 (LSTM)	(None, 30, 50)	
dropout_4 (Dropout)	(None, 30, 50)	
lstm_4 (LSTM)	(None, 30, 50)	
dropout_5 (Dropout)	(None, 30, 50)	
lstm_5 (LSTM)	(None, 50)	
dropout_6 (Dropout)	(None, 50)	
dense_2 (Dense)	(None, 20)	
dropout_7 (Dropout)	(None, 20)	
dense_3 (Dense)	(None, 1)	

Total params: 52,241 (204.07 KB)

Trainable params: 52,241 (204.07 KB)

Non-trainable params: 0 (0.00 B)

In [38]:
```
# fit the models
model.fit(
    X_train,
    y_train,
    validation_split=val_split,
    epochs=100,
    batch_size=64,
    verbose=2,
    callbacks=[es],
)
```

```
Epoch 1/100
9/9 - 3s - 315ms/step - loss: 1.9529 - val_loss: 2.1209
Epoch 2/100
9/9 - 0s - 16ms/step - loss: 1.9110 - val_loss: 2.0576
Epoch 3/100
9/9 - 0s - 18ms/step - loss: 1.8793 - val_loss: 2.0017
Epoch 4/100
9/9 - 0s - 32ms/step - loss: 1.8439 - val_loss: 1.9505
Epoch 5/100
9/9 - 0s - 16ms/step - loss: 1.8129 - val_loss: 1.8951
Epoch 6/100
9/9 - 0s - 15ms/step - loss: 1.7825 - val_loss: 1.8407
Epoch 7/100
9/9 - 0s - 16ms/step - loss: 1.7498 - val_loss: 1.7965
Epoch 8/100
9/9 - 0s - 16ms/step - loss: 1.7191 - val_loss: 1.7614
Epoch 9/100
9/9 - 0s - 14ms/step - loss: 1.6971 - val_loss: 1.7394
Epoch 10/100
9/9 - 0s - 15ms/step - loss: 1.6696 - val_loss: 1.7168
Epoch 11/100
9/9 - 0s - 15ms/step - loss: 1.6447 - val_loss: 1.7092
Epoch 12/100
9/9 - 0s - 14ms/step - loss: 1.6196 - val_loss: 1.6813
Epoch 13/100
9/9 - 0s - 15ms/step - loss: 1.5922 - val_loss: 1.6642
Epoch 14/100
9/9 - 0s - 13ms/step - loss: 1.5702 - val_loss: 1.6447
Epoch 15/100
9/9 - 0s - 14ms/step - loss: 1.5465 - val_loss: 1.6188
Epoch 16/100
9/9 - 0s - 14ms/step - loss: 1.5262 - val_loss: 1.6045
Epoch 17/100
9/9 - 0s - 16ms/step - loss: 1.5016 - val_loss: 1.5857
Epoch 18/100
9/9 - 0s - 14ms/step - loss: 1.4799 - val_loss: 1.5680
Epoch 19/100
9/9 - 0s - 16ms/step - loss: 1.4572 - val_loss: 1.5474
Epoch 20/100
9/9 - 0s - 15ms/step - loss: 1.4399 - val_loss: 1.5364
Epoch 21/100
9/9 - 0s - 15ms/step - loss: 1.4182 - val_loss: 1.5234
Epoch 22/100
9/9 - 0s - 13ms/step - loss: 1.3980 - val_loss: 1.5090
Epoch 23/100
9/9 - 0s - 14ms/step - loss: 1.3788 - val_loss: 1.4793
Epoch 24/100
9/9 - 0s - 15ms/step - loss: 1.3566 - val_loss: 1.4424
Epoch 25/100
9/9 - 0s - 15ms/step - loss: 1.3359 - val_loss: 1.4189
Epoch 26/100
9/9 - 0s - 15ms/step - loss: 1.3236 - val_loss: 1.4185
Epoch 27/100
9/9 - 0s - 14ms/step - loss: 1.3008 - val_loss: 1.4117
Epoch 28/100
9/9 - 0s - 13ms/step - loss: 1.2796 - val_loss: 1.3875
Epoch 29/100
9/9 - 0s - 15ms/step - loss: 1.2639 - val_loss: 1.3643
Epoch 30/100
9/9 - 0s - 17ms/step - loss: 1.2466 - val_loss: 1.3337
```

```
Epoch 31/100
9/9 - 0s - 15ms/step - loss: 1.2270 - val_loss: 1.3154
Epoch 32/100
9/9 - 0s - 14ms/step - loss: 1.2107 - val_loss: 1.2981
Epoch 33/100
9/9 - 0s - 15ms/step - loss: 1.1970 - val_loss: 1.2956
Epoch 34/100
9/9 - 0s - 16ms/step - loss: 1.1797 - val_loss: 1.2856
Epoch 35/100
9/9 - 0s - 13ms/step - loss: 1.1605 - val_loss: 1.2588
Epoch 36/100
9/9 - 0s - 15ms/step - loss: 1.1474 - val_loss: 1.2483
Epoch 37/100
9/9 - 0s - 16ms/step - loss: 1.1322 - val_loss: 1.2382
Epoch 38/100
9/9 - 0s - 15ms/step - loss: 1.1148 - val_loss: 1.2140
Epoch 39/100
9/9 - 0s - 16ms/step - loss: 1.0990 - val_loss: 1.2006
Epoch 40/100
9/9 - 0s - 18ms/step - loss: 1.0795 - val_loss: 1.1759
Epoch 41/100
9/9 - 0s - 21ms/step - loss: 1.0684 - val_loss: 1.1612
Epoch 42/100
9/9 - 0s - 32ms/step - loss: 1.0518 - val_loss: 1.1461
Epoch 43/100
9/9 - 0s - 21ms/step - loss: 1.0418 - val_loss: 1.1478
Epoch 44/100
9/9 - 0s - 21ms/step - loss: 1.0264 - val_loss: 1.1329
Epoch 45/100
9/9 - 0s - 34ms/step - loss: 1.0113 - val_loss: 1.1204
Epoch 46/100
9/9 - 0s - 35ms/step - loss: 0.9986 - val_loss: 1.1053
Epoch 47/100
9/9 - 0s - 23ms/step - loss: 0.9823 - val_loss: 1.0931
Epoch 48/100
9/9 - 0s - 26ms/step - loss: 0.9727 - val_loss: 1.0729
Epoch 49/100
9/9 - 0s - 30ms/step - loss: 0.9540 - val_loss: 1.0518
Epoch 50/100
9/9 - 0s - 28ms/step - loss: 0.9443 - val_loss: 1.0439
Epoch 51/100
9/9 - 0s - 16ms/step - loss: 0.9338 - val_loss: 1.0368
Epoch 52/100
9/9 - 0s - 16ms/step - loss: 0.9174 - val_loss: 1.0125
Epoch 53/100
9/9 - 0s - 16ms/step - loss: 0.9035 - val_loss: 0.9967
Epoch 54/100
9/9 - 0s - 32ms/step - loss: 0.8938 - val_loss: 0.9864
Epoch 55/100
9/9 - 0s - 14ms/step - loss: 0.8834 - val_loss: 0.9810
Epoch 56/100
9/9 - 0s - 16ms/step - loss: 0.8726 - val_loss: 0.9686
Epoch 57/100
9/9 - 0s - 16ms/step - loss: 0.8580 - val_loss: 0.9673
Epoch 58/100
9/9 - 0s - 13ms/step - loss: 0.8500 - val_loss: 0.9526
Epoch 59/100
9/9 - 0s - 17ms/step - loss: 0.8346 - val_loss: 0.9327
Epoch 60/100
9/9 - 0s - 14ms/step - loss: 0.8264 - val_loss: 0.9239
```

```
Epoch 61/100
9/9 - 0s - 14ms/step - loss: 0.8146 - val_loss: 0.9138
Epoch 62/100
9/9 - 0s - 16ms/step - loss: 0.8058 - val_loss: 0.9156
Epoch 63/100
9/9 - 0s - 15ms/step - loss: 0.7919 - val_loss: 0.8913
Epoch 64/100
9/9 - 0s - 15ms/step - loss: 0.7816 - val_loss: 0.8771
Epoch 65/100
9/9 - 0s - 13ms/step - loss: 0.7732 - val_loss: 0.8677
Epoch 66/100
9/9 - 0s - 17ms/step - loss: 0.7655 - val_loss: 0.8625
Epoch 67/100
9/9 - 0s - 16ms/step - loss: 0.7548 - val_loss: 0.8682
Epoch 68/100
9/9 - 0s - 15ms/step - loss: 0.7439 - val_loss: 0.8467
Epoch 69/100
9/9 - 0s - 14ms/step - loss: 0.7327 - val_loss: 0.8289
Epoch 70/100
9/9 - 0s - 13ms/step - loss: 0.7228 - val_loss: 0.8209
Epoch 71/100
9/9 - 0s - 14ms/step - loss: 0.7128 - val_loss: 0.8220
Epoch 72/100
9/9 - 0s - 15ms/step - loss: 0.7052 - val_loss: 0.8105
Epoch 73/100
9/9 - 0s - 15ms/step - loss: 0.6953 - val_loss: 0.7997
Epoch 74/100
9/9 - 0s - 15ms/step - loss: 0.6882 - val_loss: 0.7852
Epoch 75/100
9/9 - 0s - 16ms/step - loss: 0.6782 - val_loss: 0.7704
Epoch 76/100
9/9 - 0s - 15ms/step - loss: 0.6712 - val_loss: 0.7676
Epoch 77/100
9/9 - 0s - 13ms/step - loss: 0.6610 - val_loss: 0.7618
Epoch 78/100
9/9 - 0s - 17ms/step - loss: 0.6577 - val_loss: 0.7525
Epoch 79/100
9/9 - 0s - 14ms/step - loss: 0.6505 - val_loss: 0.7454
Epoch 80/100
9/9 - 0s - 14ms/step - loss: 0.6363 - val_loss: 0.7354
Epoch 81/100
9/9 - 0s - 14ms/step - loss: 0.6317 - val_loss: 0.7329
Epoch 82/100
9/9 - 0s - 16ms/step - loss: 0.6232 - val_loss: 0.7282
Epoch 83/100
9/9 - 0s - 14ms/step - loss: 0.6175 - val_loss: 0.7225
Epoch 84/100
9/9 - 0s - 16ms/step - loss: 0.6065 - val_loss: 0.7112
Epoch 85/100
9/9 - 0s - 16ms/step - loss: 0.5993 - val_loss: 0.6975
Epoch 86/100
9/9 - 0s - 15ms/step - loss: 0.5933 - val_loss: 0.6927
Epoch 87/100
9/9 - 0s - 14ms/step - loss: 0.5866 - val_loss: 0.6906
Epoch 88/100
9/9 - 0s - 15ms/step - loss: 0.5816 - val_loss: 0.6788
Epoch 89/100
9/9 - 0s - 14ms/step - loss: 0.5722 - val_loss: 0.6695
Epoch 90/100
9/9 - 0s - 14ms/step - loss: 0.5633 - val_loss: 0.6655
```

```
Epoch 91/100
9/9 - 0s - 15ms/step - loss: 0.5596 - val_loss: 0.6642
Epoch 92/100
9/9 - 0s - 13ms/step - loss: 0.5528 - val_loss: 0.6571
Epoch 93/100
9/9 - 0s - 16ms/step - loss: 0.5465 - val_loss: 0.6457
Epoch 94/100
9/9 - 0s - 13ms/step - loss: 0.5419 - val_loss: 0.6353
Epoch 95/100
9/9 - 0s - 15ms/step - loss: 0.5355 - val_loss: 0.6288
Epoch 96/100
9/9 - 0s - 14ms/step - loss: 0.5273 - val_loss: 0.6245
Epoch 97/100
9/9 - 0s - 14ms/step - loss: 0.5213 - val_loss: 0.6247
Epoch 98/100
9/9 - 0s - 15ms/step - loss: 0.5148 - val_loss: 0.6216
Epoch 99/100
9/9 - 0s - 14ms/step - loss: 0.5089 - val_loss: 0.6153
Epoch 100/100
9/9 - 0s - 13ms/step - loss: 0.5034 - val_loss: 0.6033
```

Out[38]: <keras.src.callbacks.history.History at 0x7d35dc387e20>

In [39]:
```python
#Train and test

prediction = model.predict(X_test)
prediction = scaler_output.inverse_transform(prediction)
prediction = prediction.flatten()

values = np.array(y_test[window_size:])
values = values.flatten()


def R2_campbell(y_true, y_predicted, mean_ret):
    y_predicted = y_predicted.reshape((-1,))
    sse = sum((y_true - y_predicted) ** 2)
    tse = sum((y_true - mean_ret) ** 2)
    r2_score = 1 - (sse / tse)

    return r2_score


R2_Campbell = R2_campbell(values, prediction, mean_ret)
print("Out-of-sample R-squared:", R2_Campbell)

# %%
df_predictions = pd.DataFrame(
    {"Date": test_time.flatten(), "Pred LSTM": prediction, "values": values}
)
df_predictions.head()
```

```
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
Out-of-sample R-squared: -0.14217422275184166
```

Out[39]:

	Date	Pred LSTM	values
0	2021-04-06	0.019597	-0.005804
1	2021-04-07	0.019484	-0.009351
2	2021-04-08	0.019378	-0.015941
3	2021-04-09	0.019264	-0.003127
4	2021-04-12	0.019132	-0.004802

In [40]:
```
df2_actual=df_predictions[['values']].to_numpy()
df2_pred=df_predictions[['Pred LSTM']].to_numpy()
```

LSTM Model 3 : SHY

In [41]:
```
df_shy = df[["Date", "SHY"]].copy()
df_shy
```

Out[41]:

Ticker	Date	SHY
0	2018-01-03	0.000000
1	2018-01-04	-0.000477
2	2018-01-05	0.000000
3	2018-01-08	0.000000
4	2018-01-09	-0.000358
...	...	...
1252	2022-12-22	-0.000614
1253	2022-12-23	-0.000492
1254	2022-12-27	-0.001354
1255	2022-12-28	0.000000
1256	2022-12-29	0.000739

1257 rows × 2 columns

In [42]:
```
df_shy["Ret_10"] = df_shy["SHY"].rolling(10).apply(lambda x: np.exp(np.sum(x)) -
df_shy["Ret_50"] = df_shy["SHY"].rolling(50).apply(lambda x: np.exp(np.sum(x)) -
df_shy["Ret_25"] = df_shy["SHY"].rolling(25).apply(lambda x: np.exp(np.sum(x)) -

df_shy["Ret25"] = df_shy["Ret_25"].shift(-25)
del df_shy["Ret_25"]
df_shy = df_shy.dropna()
df_shy.head()
```

Out[42]:

	Ticker	Date	SHY	Ret_10	Ret_50	Ret25
49		2018-03-15	0.000120	-0.00024	-0.004414	-0.002277
50		2018-03-16	-0.000479	-0.00024	-0.004891	-0.001918
51		2018-03-19	0.000000	-0.00024	-0.004416	-0.001679
52		2018-03-20	-0.000360	-0.00036	-0.004774	-0.001439
53		2018-03-21	0.000719	0.00036	-0.004058	-0.001918

In [43]:
```
Xdf, ydf = df_shy.iloc[:, 1:-1], df_shy.iloc[:, -1]
X = Xdf.astype("float32")
y = ydf.astype("float32")
```

In [44]:
```
val_split = 0.2
train_split = 0.625
train_size = int(len(df_shy) * train_split)
val_size = int(train_size * val_split)
test_size = int(len(df_shy) - train_size)

window_size = 30

ts = test_size
split_time = len(df_shy) - ts
test_time = df_shy.iloc[split_time + window_size :, 0:1].values


y_train_set = y[:split_time]
y_test_set = y[split_time:]

X_train_set = X[:split_time]
X_test_set = X[split_time:]

n_features = X_train_set.shape[1]
```

In [45]:
```
scaler_input = MinMaxScaler(feature_range=(-1, 1))
scaler_input.fit(X_train_set)
X_train_set_scaled = scaler_input.transform(X_train_set)
X_test_set_scaled = scaler_input.transform(X_test_set)

mean_ret = np.mean(y_train_set)

scaler_output = MinMaxScaler(feature_range=(-1, 1))
y_train_set = y_train_set.values.reshape(len(y_train_set), 1)
y_test_set = y_test_set.values.reshape(len(y_test_set), 1)
scaler_output.fit(y_train_set)
y_train_set_scaled = scaler_output.transform(y_train_set)
```

In [46]:
```
training_time = df_shy.iloc[:split_time, 0:1].values

X_train = []
y_train = []

for i in range(window_size, y_train_set_scaled.shape[0]):
    X_train.append(X_train_set_scaled[i - window_size : i, :])
    y_train.append(y_train_set_scaled[i])
```

```
X_train, y_train = np.array(X_train), np.array(y_train)

print("Shape of training data", X_train.shape, y_train.shape)

X_test = []
y_test = y_test_set

for i in range(window_size, y_test_set.shape[0]):
    X_test.append(X_test_set_scaled[i - window_size : i, :])

X_test, y_test = np.array(X_test), np.array(y_test)

print("Shape of test data", X_test.shape, y_test.shape)
```

```
Shape of training data (709, 30, 3) (709, 1)
Shape of test data (414, 30, 3) (444, 1)
```

In [47]:
```
from keras.regularizers import l2
from keras.optimizers import Adam
```

In [48]:
```
SEED = 321
# Define hyperparameters
units_lstm = 50
n_dropout = 0.2
act_fun = "relu"

# Define the model
model = Sequential()

# First LSTM layer with Tanh activation and L2 regularization
model.add(
    LSTM(
        units=units_lstm,
        return_sequences=True,
        activation="tanh",
        input_shape=(X_train.shape[1], n_features),
        kernel_regularizer=l2(0.01)
    )
)
model.add(Dropout(n_dropout, seed=SEED))

# Second LSTM layer with Tanh activation and L2 regularization
model.add(LSTM(units=units_lstm, return_sequences=True, activation="tanh", kerne
model.add(Dropout(n_dropout, seed=SEED))

# Third LSTM layer with Tanh activation and L2 regularization
model.add(LSTM(units=units_lstm, return_sequences=False, activation="tanh", kern
model.add(Dropout(n_dropout, seed=SEED))

# Dense layer with ReLU activation
model.add(Dense(units=20, activation=act_fun))
model.add(Dropout(n_dropout, seed=SEED))

# Output layer with Linear activation (for regression)
model.add(Dense(units=1, activation='linear'))

# Compile the model with Adam optimizer and MAE loss
hp_lr = 1e-4
model.compile(optimizer=Adam(learning_rate=hp_lr), loss='mean_absolute_error')
```

```
es = EarlyStopping(
    monitor="val_loss", mode="min", verbose=0, patience=5, restore_best_weights=
)

# Summary of the model
model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/rnn.py:204: UserWarn
ing: 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 m
odel instead.
  super().__init__(**kwargs)
```

Model: "sequential_2"

Layer (type)	Output Shape	
lstm_6 (LSTM)	(None, 30, 50)	
dropout_8 (Dropout)	(None, 30, 50)	
lstm_7 (LSTM)	(None, 30, 50)	
dropout_9 (Dropout)	(None, 30, 50)	
lstm_8 (LSTM)	(None, 50)	
dropout_10 (Dropout)	(None, 50)	
dense_4 (Dense)	(None, 20)	
dropout_11 (Dropout)	(None, 20)	
dense_5 (Dense)	(None, 1)	

Total params: 52,241 (204.07 KB)

Trainable params: 52,241 (204.07 KB)

Non-trainable params: 0 (0.00 B)

In [49]:
```
# fit the models
model.fit(
    X_train,
    y_train,
    validation_split=val_split,
    epochs=100,
    batch_size=64,
    verbose=2,
    callbacks=[es],
)
```

```
Epoch 1/100
9/9 - 4s - 430ms/step - loss: 2.1008 - val_loss: 2.1303
Epoch 2/100
9/9 - 0s - 18ms/step - loss: 2.0270 - val_loss: 2.0254
Epoch 3/100
9/9 - 0s - 17ms/step - loss: 1.9537 - val_loss: 1.8989
Epoch 4/100
9/9 - 0s - 15ms/step - loss: 1.8778 - val_loss: 1.7493
Epoch 5/100
9/9 - 0s - 15ms/step - loss: 1.8403 - val_loss: 1.6237
Epoch 6/100
9/9 - 0s - 15ms/step - loss: 1.8068 - val_loss: 1.5883
Epoch 7/100
9/9 - 0s - 14ms/step - loss: 1.7750 - val_loss: 1.5899
Epoch 8/100
9/9 - 0s - 16ms/step - loss: 1.7410 - val_loss: 1.5906
Epoch 9/100
9/9 - 0s - 14ms/step - loss: 1.7184 - val_loss: 1.5748
Epoch 10/100
9/9 - 0s - 15ms/step - loss: 1.6934 - val_loss: 1.5540
Epoch 11/100
9/9 - 0s - 13ms/step - loss: 1.6637 - val_loss: 1.5279
Epoch 12/100
9/9 - 0s - 16ms/step - loss: 1.6400 - val_loss: 1.5181
Epoch 13/100
9/9 - 0s - 16ms/step - loss: 1.6151 - val_loss: 1.5054
Epoch 14/100
9/9 - 0s - 14ms/step - loss: 1.5869 - val_loss: 1.5087
Epoch 15/100
9/9 - 0s - 16ms/step - loss: 1.5662 - val_loss: 1.4664
Epoch 16/100
9/9 - 0s - 14ms/step - loss: 1.5465 - val_loss: 1.4434
Epoch 17/100
9/9 - 0s - 15ms/step - loss: 1.5184 - val_loss: 1.4352
Epoch 18/100
9/9 - 0s - 13ms/step - loss: 1.4998 - val_loss: 1.4026
Epoch 19/100
9/9 - 0s - 14ms/step - loss: 1.4851 - val_loss: 1.3688
Epoch 20/100
9/9 - 0s - 16ms/step - loss: 1.4611 - val_loss: 1.3901
Epoch 21/100
9/9 - 0s - 16ms/step - loss: 1.4469 - val_loss: 1.3545
Epoch 22/100
9/9 - 0s - 15ms/step - loss: 1.4231 - val_loss: 1.3306
Epoch 23/100
9/9 - 0s - 16ms/step - loss: 1.4005 - val_loss: 1.3017
Epoch 24/100
9/9 - 0s - 15ms/step - loss: 1.3819 - val_loss: 1.2994
Epoch 25/100
9/9 - 0s - 14ms/step - loss: 1.3640 - val_loss: 1.3054
Epoch 26/100
9/9 - 0s - 14ms/step - loss: 1.3491 - val_loss: 1.2742
Epoch 27/100
9/9 - 0s - 14ms/step - loss: 1.3284 - val_loss: 1.2138
Epoch 28/100
9/9 - 0s - 16ms/step - loss: 1.3067 - val_loss: 1.2242
Epoch 29/100
9/9 - 0s - 16ms/step - loss: 1.2937 - val_loss: 1.2070
Epoch 30/100
9/9 - 0s - 15ms/step - loss: 1.2719 - val_loss: 1.1906
```

```
Epoch 31/100
9/9 - 0s - 15ms/step - loss: 1.2578 - val_loss: 1.1739
Epoch 32/100
9/9 - 0s - 13ms/step - loss: 1.2490 - val_loss: 1.1764
Epoch 33/100
9/9 - 0s - 13ms/step - loss: 1.2280 - val_loss: 1.1255
Epoch 34/100
9/9 - 0s - 15ms/step - loss: 1.2071 - val_loss: 1.1317
Epoch 35/100
9/9 - 0s - 15ms/step - loss: 1.1936 - val_loss: 1.1171
Epoch 36/100
9/9 - 0s - 15ms/step - loss: 1.1764 - val_loss: 1.0701
Epoch 37/100
9/9 - 0s - 16ms/step - loss: 1.1646 - val_loss: 1.0593
Epoch 38/100
9/9 - 0s - 14ms/step - loss: 1.1520 - val_loss: 1.0960
Epoch 39/100
9/9 - 0s - 14ms/step - loss: 1.1332 - val_loss: 1.0343
Epoch 40/100
9/9 - 0s - 15ms/step - loss: 1.1203 - val_loss: 1.0094
Epoch 41/100
9/9 - 0s - 13ms/step - loss: 1.1026 - val_loss: 1.0357
Epoch 42/100
9/9 - 0s - 15ms/step - loss: 1.0944 - val_loss: 1.0459
Epoch 43/100
9/9 - 0s - 14ms/step - loss: 1.0767 - val_loss: 0.9947
Epoch 44/100
9/9 - 0s - 16ms/step - loss: 1.0617 - val_loss: 0.9530
Epoch 45/100
9/9 - 0s - 17ms/step - loss: 1.0467 - val_loss: 0.9934
Epoch 46/100
9/9 - 0s - 14ms/step - loss: 1.0405 - val_loss: 0.9842
Epoch 47/100
9/9 - 0s - 16ms/step - loss: 1.0185 - val_loss: 0.9281
Epoch 48/100
9/9 - 0s - 14ms/step - loss: 1.0110 - val_loss: 0.9208
Epoch 49/100
9/9 - 0s - 17ms/step - loss: 0.9946 - val_loss: 0.9022
Epoch 50/100
9/9 - 0s - 14ms/step - loss: 0.9784 - val_loss: 0.9007
Epoch 51/100
9/9 - 0s - 14ms/step - loss: 0.9735 - val_loss: 0.8865
Epoch 52/100
9/9 - 0s - 16ms/step - loss: 0.9595 - val_loss: 0.8819
Epoch 53/100
9/9 - 0s - 14ms/step - loss: 0.9460 - val_loss: 0.8680
Epoch 54/100
9/9 - 0s - 13ms/step - loss: 0.9339 - val_loss: 0.8554
Epoch 55/100
9/9 - 0s - 13ms/step - loss: 0.9214 - val_loss: 0.8211
Epoch 56/100
9/9 - 0s - 16ms/step - loss: 0.9131 - val_loss: 0.8228
Epoch 57/100
9/9 - 0s - 15ms/step - loss: 0.9018 - val_loss: 0.8275
Epoch 58/100
9/9 - 0s - 14ms/step - loss: 0.8963 - val_loss: 0.7783
Epoch 59/100
9/9 - 0s - 15ms/step - loss: 0.8732 - val_loss: 0.7913
Epoch 60/100
9/9 - 0s - 14ms/step - loss: 0.8630 - val_loss: 0.7827
```

```
Epoch 61/100
9/9 - 0s - 13ms/step - loss: 0.8561 - val_loss: 0.7722
Epoch 62/100
9/9 - 0s - 16ms/step - loss: 0.8450 - val_loss: 0.7488
Epoch 63/100
9/9 - 0s - 15ms/step - loss: 0.8304 - val_loss: 0.7396
Epoch 64/100
9/9 - 0s - 15ms/step - loss: 0.8196 - val_loss: 0.7496
Epoch 65/100
9/9 - 0s - 13ms/step - loss: 0.8155 - val_loss: 0.7535
Epoch 66/100
9/9 - 0s - 18ms/step - loss: 0.8054 - val_loss: 0.7233
Epoch 67/100
9/9 - 0s - 14ms/step - loss: 0.7995 - val_loss: 0.7017
Epoch 68/100
9/9 - 0s - 13ms/step - loss: 0.7830 - val_loss: 0.7311
Epoch 69/100
9/9 - 0s - 15ms/step - loss: 0.7820 - val_loss: 0.7146
Epoch 70/100
9/9 - 0s - 14ms/step - loss: 0.7658 - val_loss: 0.6793
Epoch 71/100
9/9 - 0s - 14ms/step - loss: 0.7600 - val_loss: 0.6498
Epoch 72/100
9/9 - 0s - 17ms/step - loss: 0.7526 - val_loss: 0.6924
Epoch 73/100
9/9 - 0s - 13ms/step - loss: 0.7427 - val_loss: 0.6598
Epoch 74/100
9/9 - 0s - 18ms/step - loss: 0.7258 - val_loss: 0.6162
Epoch 75/100
9/9 - 0s - 16ms/step - loss: 0.7240 - val_loss: 0.6490
Epoch 76/100
9/9 - 0s - 36ms/step - loss: 0.7113 - val_loss: 0.6196
Epoch 77/100
9/9 - 0s - 33ms/step - loss: 0.7038 - val_loss: 0.6234
Epoch 78/100
9/9 - 0s - 22ms/step - loss: 0.6982 - val_loss: 0.6268
Epoch 79/100
9/9 - 0s - 19ms/step - loss: 0.6954 - val_loss: 0.6087
Epoch 80/100
9/9 - 0s - 36ms/step - loss: 0.6844 - val_loss: 0.6160
Epoch 81/100
9/9 - 0s - 35ms/step - loss: 0.6775 - val_loss: 0.5859
Epoch 82/100
9/9 - 0s - 22ms/step - loss: 0.6673 - val_loss: 0.5978
Epoch 83/100
9/9 - 0s - 23ms/step - loss: 0.6546 - val_loss: 0.5660
Epoch 84/100
9/9 - 0s - 17ms/step - loss: 0.6596 - val_loss: 0.5693
Epoch 85/100
9/9 - 0s - 14ms/step - loss: 0.6449 - val_loss: 0.5893
Epoch 86/100
9/9 - 0s - 15ms/step - loss: 0.6380 - val_loss: 0.5385
Epoch 87/100
9/9 - 0s - 13ms/step - loss: 0.6371 - val_loss: 0.5626
Epoch 88/100
9/9 - 0s - 13ms/step - loss: 0.6321 - val_loss: 0.5683
Epoch 89/100
9/9 - 0s - 17ms/step - loss: 0.6161 - val_loss: 0.5206
Epoch 90/100
9/9 - 0s - 14ms/step - loss: 0.6042 - val_loss: 0.5350
```

```
Epoch 91/100
9/9 - 0s - 14ms/step - loss: 0.5992 - val_loss: 0.5383
Epoch 92/100
9/9 - 0s - 14ms/step - loss: 0.5978 - val_loss: 0.5171
Epoch 93/100
9/9 - 0s - 15ms/step - loss: 0.5922 - val_loss: 0.5008
Epoch 94/100
9/9 - 0s - 13ms/step - loss: 0.5870 - val_loss: 0.5110
Epoch 95/100
9/9 - 0s - 17ms/step - loss: 0.5794 - val_loss: 0.5005
Epoch 96/100
9/9 - 0s - 18ms/step - loss: 0.5787 - val_loss: 0.4775
Epoch 97/100
9/9 - 0s - 30ms/step - loss: 0.5676 - val_loss: 0.4738
Epoch 98/100
9/9 - 0s - 15ms/step - loss: 0.5591 - val_loss: 0.4696
Epoch 99/100
9/9 - 0s - 16ms/step - loss: 0.5534 - val_loss: 0.4843
Epoch 100/100
9/9 - 0s - 14ms/step - loss: 0.5580 - val_loss: 0.4732
```

Out[49]: <keras.src.callbacks.history.History at 0x7d357e7138e0>

In [50]:
```python
#Train and test

prediction = model.predict(X_test)
prediction = scaler_output.inverse_transform(prediction)
prediction = prediction.flatten()

values = np.array(y_test[window_size:])
values = values.flatten()


def R2_campbell(y_true, y_predicted, mean_ret):
    y_predicted = y_predicted.reshape((-1,))
    sse = sum((y_true - y_predicted) ** 2)
    tse = sum((y_true - mean_ret) ** 2)
    r2_score = 1 - (sse / tse)

    return r2_score


R2_Campbell = R2_campbell(values, prediction, mean_ret)
print("Out-of-sample R-squared:", R2_Campbell)

# %%
df_predictions = pd.DataFrame(
    {"Date": test_time.flatten(), "Pred LSTM": prediction, "values": values}
)
df_predictions.head()
```

```
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
Out-of-sample R-squared: 0.04833190355893968
```

Out[50]:

	Date	Pred LSTM	values
0	2021-04-06	0.001095	0.000464
1	2021-04-07	0.001111	0.000232
2	2021-04-08	0.001126	0.000116
3	2021-04-09	0.001138	0.000464
4	2021-04-12	0.001163	0.000696

In [51]:
```
df3_actual=df_predictions[['values']].to_numpy()
df3_pred=df_predictions[['Pred LSTM']].to_numpy()
```

LSTM Model 4 : GLD

In [52]:
```
df_gld = df[["Date", "GLD"]].copy()
df_gld
```

Out[52]:

Ticker	Date	GLD
0	2018-01-03	-0.002640
1	2018-01-04	0.005114
2	2018-01-05	-0.001037
3	2018-01-08	-0.000160
4	2018-01-09	-0.004639
...	...	...
1252	2022-12-22	-0.012159
1253	2022-12-23	0.002994
1254	2022-12-27	0.008395
1255	2022-12-28	-0.004516
1256	2022-12-29	0.005583

1257 rows × 2 columns

In [53]:
```
df_gld["Ret_10"] = df_gld["GLD"].rolling(10).apply(lambda x: np.exp(np.sum(x)) -
df_gld["Ret_50"] = df_gld["GLD"].rolling(50).apply(lambda x: np.exp(np.sum(x)) -
df_gld["Ret_25"] = df_gld["GLD"].rolling(25).apply(lambda x: np.exp(np.sum(x)) -

df_gld["Ret25"] = df_gld["Ret_25"].shift(-25)
del df_gld["Ret_25"]
df_gld = df_gld.dropna()
df_gld.head()
```

Out[53]:

	Ticker	Date	GLD	Ret_10	Ret_50	Ret25
49		2018-03-15	-0.006385	0.001443	-0.001998	0.013851
50		2018-03-16	-0.002405	-0.006300	-0.001763	0.008186
51		2018-03-19	0.002165	-0.002476	-0.004703	0.010891
52		2018-03-20	-0.004495	-0.017545	-0.008139	0.008849
53		2018-03-21	0.017306	0.006045	0.009337	-0.011939

In [54]:
```
Xdf, ydf = df_gld.iloc[:, 1:-1], df_gld.iloc[:, -1]
X = Xdf.astype("float32")
y = ydf.astype("float32")
```

In [55]:
```
val_split = 0.2
train_split = 0.625
train_size = int(len(df_gld) * train_split)
val_size = int(train_size * val_split)
test_size = int(len(df_gld) - train_size)

window_size = 30

ts = test_size
split_time = len(df_gld) - ts
test_time = df_gld.iloc[split_time + window_size :, 0:1].values


y_train_set = y[:split_time]
y_test_set = y[split_time:]

X_train_set = X[:split_time]
X_test_set = X[split_time:]

n_features = X_train_set.shape[1]
```

In [56]:
```
scaler_input = MinMaxScaler(feature_range=(-1, 1))
scaler_input.fit(X_train_set)
X_train_set_scaled = scaler_input.transform(X_train_set)
X_test_set_scaled = scaler_input.transform(X_test_set)

mean_ret = np.mean(y_train_set)

scaler_output = MinMaxScaler(feature_range=(-1, 1))
y_train_set = y_train_set.values.reshape(len(y_train_set), 1)
y_test_set = y_test_set.values.reshape(len(y_test_set), 1)
scaler_output.fit(y_train_set)
y_train_set_scaled = scaler_output.transform(y_train_set)
```

In [57]:
```
training_time = df_gld.iloc[:split_time, 0:1].values

X_train = []
y_train = []

for i in range(window_size, y_train_set_scaled.shape[0]):
    X_train.append(X_train_set_scaled[i - window_size : i, :])
    y_train.append(y_train_set_scaled[i])
```

```
X_train, y_train = np.array(X_train), np.array(y_train)

print("Shape of training data", X_train.shape, y_train.shape)

X_test = []
y_test = y_test_set

for i in range(window_size, y_test_set.shape[0]):
    X_test.append(X_test_set_scaled[i - window_size : i, :])

X_test, y_test = np.array(X_test), np.array(y_test)

print("Shape of test data", X_test.shape, y_test.shape)
```

```
Shape of training data (709, 30, 3) (709, 1)
Shape of test data (414, 30, 3) (444, 1)
```

In [58]:
```
from keras.regularizers import l2
from keras.optimizers import Adam
```

In [59]:
```
SEED = 321
# Define hyperparameters
units_lstm = 50
n_dropout = 0.2
act_fun = "relu"

# Define the model
model = Sequential()

# First LSTM layer with Tanh activation and L2 regularization
model.add(
    LSTM(
        units=units_lstm,
        return_sequences=True,
        activation="tanh",
        input_shape=(X_train.shape[1], n_features),
        kernel_regularizer=l2(0.01)
    )
)
model.add(Dropout(n_dropout, seed=SEED))

# Second LSTM layer with Tanh activation and L2 regularization
model.add(LSTM(units=units_lstm, return_sequences=True, activation="tanh", kerne
model.add(Dropout(n_dropout, seed=SEED))

# Third LSTM layer with Tanh activation and L2 regularization
model.add(LSTM(units=units_lstm, return_sequences=False, activation="tanh", kern
model.add(Dropout(n_dropout, seed=SEED))

# Dense layer with ReLU activation
model.add(Dense(units=20, activation=act_fun))
model.add(Dropout(n_dropout, seed=SEED))

# Output layer with Linear activation (for regression)
model.add(Dense(units=1, activation='linear'))

# Compile the model with Adam optimizer and MAE loss
hp_lr = 1e-4
model.compile(optimizer=Adam(learning_rate=hp_lr), loss='mean_absolute_error')
```

```
es = EarlyStopping(
    monitor="val_loss", mode="min", verbose=0, patience=5, restore_best_weights=
)

# Summary of the model
model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/rnn.py:204: UserWarn
ing: 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 m
odel instead.
  super().__init__(**kwargs)
```

Model: "sequential_3"

Layer (type)	Output Shape	
lstm_9 (LSTM)	(None, 30, 50)	
dropout_12 (Dropout)	(None, 30, 50)	
lstm_10 (LSTM)	(None, 30, 50)	
dropout_13 (Dropout)	(None, 30, 50)	
lstm_11 (LSTM)	(None, 50)	
dropout_14 (Dropout)	(None, 50)	
dense_6 (Dense)	(None, 20)	
dropout_15 (Dropout)	(None, 20)	
dense_7 (Dense)	(None, 1)	

Total params: 52,241 (204.07 KB)

Trainable params: 52,241 (204.07 KB)

Non-trainable params: 0 (0.00 B)

In [60]:
```
# fit the models
model.fit(
    X_train,
    y_train,
    validation_split=val_split,
    epochs=100,
    batch_size=64,
    verbose=2,
    callbacks=[es],
)
```

```
Epoch 1/100
9/9 - 3s - 326ms/step - loss: 1.9456 - val_loss: 2.1192
Epoch 2/100
9/9 - 0s - 16ms/step - loss: 1.9096 - val_loss: 2.0731
Epoch 3/100
9/9 - 0s - 16ms/step - loss: 1.8734 - val_loss: 2.0276
Epoch 4/100
9/9 - 0s - 15ms/step - loss: 1.8396 - val_loss: 1.9808
Epoch 5/100
9/9 - 0s - 15ms/step - loss: 1.8092 - val_loss: 1.9343
Epoch 6/100
9/9 - 0s - 15ms/step - loss: 1.7766 - val_loss: 1.8893
Epoch 7/100
9/9 - 0s - 15ms/step - loss: 1.7493 - val_loss: 1.8512
Epoch 8/100
9/9 - 0s - 14ms/step - loss: 1.7220 - val_loss: 1.8173
Epoch 9/100
9/9 - 0s - 17ms/step - loss: 1.6974 - val_loss: 1.7868
Epoch 10/100
9/9 - 0s - 15ms/step - loss: 1.6691 - val_loss: 1.7571
Epoch 11/100
9/9 - 0s - 14ms/step - loss: 1.6466 - val_loss: 1.7301
Epoch 12/100
9/9 - 0s - 14ms/step - loss: 1.6216 - val_loss: 1.7021
Epoch 13/100
9/9 - 0s - 14ms/step - loss: 1.5992 - val_loss: 1.6787
Epoch 14/100
9/9 - 0s - 15ms/step - loss: 1.5765 - val_loss: 1.6553
Epoch 15/100
9/9 - 0s - 13ms/step - loss: 1.5505 - val_loss: 1.6312
Epoch 16/100
9/9 - 0s - 13ms/step - loss: 1.5261 - val_loss: 1.6072
Epoch 17/100
9/9 - 0s - 15ms/step - loss: 1.5074 - val_loss: 1.5813
Epoch 18/100
9/9 - 0s - 15ms/step - loss: 1.4818 - val_loss: 1.5527
Epoch 19/100
9/9 - 0s - 13ms/step - loss: 1.4613 - val_loss: 1.5241
Epoch 20/100
9/9 - 0s - 16ms/step - loss: 1.4361 - val_loss: 1.4994
Epoch 21/100
9/9 - 0s - 15ms/step - loss: 1.4154 - val_loss: 1.4786
Epoch 22/100
9/9 - 0s - 14ms/step - loss: 1.4002 - val_loss: 1.4564
Epoch 23/100
9/9 - 0s - 14ms/step - loss: 1.3804 - val_loss: 1.4329
Epoch 24/100
9/9 - 0s - 16ms/step - loss: 1.3542 - val_loss: 1.4093
Epoch 25/100
9/9 - 0s - 15ms/step - loss: 1.3328 - val_loss: 1.3917
Epoch 26/100
9/9 - 0s - 14ms/step - loss: 1.3207 - val_loss: 1.3684
Epoch 27/100
9/9 - 0s - 16ms/step - loss: 1.2998 - val_loss: 1.3537
Epoch 28/100
9/9 - 0s - 20ms/step - loss: 1.2846 - val_loss: 1.3440
Epoch 29/100
9/9 - 0s - 32ms/step - loss: 1.2639 - val_loss: 1.3197
Epoch 30/100
9/9 - 0s - 35ms/step - loss: 1.2384 - val_loss: 1.2927
```

```
Epoch 31/100
9/9 - 0s - 32ms/step - loss: 1.2258 - val_loss: 1.2681
Epoch 32/100
9/9 - 0s - 34ms/step - loss: 1.2116 - val_loss: 1.2597
Epoch 33/100
9/9 - 0s - 35ms/step - loss: 1.1892 - val_loss: 1.2442
Epoch 34/100
9/9 - 0s - 21ms/step - loss: 1.1736 - val_loss: 1.2200
Epoch 35/100
9/9 - 0s - 22ms/step - loss: 1.1583 - val_loss: 1.1969
Epoch 36/100
9/9 - 0s - 31ms/step - loss: 1.1464 - val_loss: 1.1940
Epoch 37/100
9/9 - 0s - 16ms/step - loss: 1.1269 - val_loss: 1.1814
Epoch 38/100
9/9 - 0s - 17ms/step - loss: 1.1152 - val_loss: 1.1583
Epoch 39/100
9/9 - 0s - 15ms/step - loss: 1.0947 - val_loss: 1.1417
Epoch 40/100
9/9 - 0s - 14ms/step - loss: 1.0773 - val_loss: 1.1322
Epoch 41/100
9/9 - 0s - 15ms/step - loss: 1.0688 - val_loss: 1.1109
Epoch 42/100
9/9 - 0s - 15ms/step - loss: 1.0489 - val_loss: 1.0939
Epoch 43/100
9/9 - 0s - 16ms/step - loss: 1.0400 - val_loss: 1.0838
Epoch 44/100
9/9 - 0s - 14ms/step - loss: 1.0234 - val_loss: 1.0720
Epoch 45/100
9/9 - 0s - 17ms/step - loss: 1.0121 - val_loss: 1.0521
Epoch 46/100
9/9 - 0s - 15ms/step - loss: 0.9973 - val_loss: 1.0453
Epoch 47/100
9/9 - 0s - 14ms/step - loss: 0.9783 - val_loss: 1.0306
Epoch 48/100
9/9 - 0s - 15ms/step - loss: 0.9657 - val_loss: 1.0142
Epoch 49/100
9/9 - 0s - 14ms/step - loss: 0.9577 - val_loss: 0.9954
Epoch 50/100
9/9 - 0s - 13ms/step - loss: 0.9363 - val_loss: 0.9823
Epoch 51/100
9/9 - 0s - 16ms/step - loss: 0.9308 - val_loss: 0.9719
Epoch 52/100
9/9 - 0s - 17ms/step - loss: 0.9172 - val_loss: 0.9548
Epoch 53/100
9/9 - 0s - 13ms/step - loss: 0.9040 - val_loss: 0.9411
Epoch 54/100
9/9 - 0s - 16ms/step - loss: 0.8888 - val_loss: 0.9384
Epoch 55/100
9/9 - 0s - 17ms/step - loss: 0.8801 - val_loss: 0.9263
Epoch 56/100
9/9 - 0s - 15ms/step - loss: 0.8671 - val_loss: 0.9098
Epoch 57/100
9/9 - 0s - 15ms/step - loss: 0.8538 - val_loss: 0.9077
Epoch 58/100
9/9 - 0s - 15ms/step - loss: 0.8450 - val_loss: 0.8907
Epoch 59/100
9/9 - 0s - 17ms/step - loss: 0.8358 - val_loss: 0.8832
Epoch 60/100
9/9 - 0s - 14ms/step - loss: 0.8267 - val_loss: 0.8633
```

```
Epoch 61/100
9/9 - 0s - 15ms/step - loss: 0.8155 - val_loss: 0.8656
Epoch 62/100
9/9 - 0s - 13ms/step - loss: 0.8021 - val_loss: 0.8477
Epoch 63/100
9/9 - 0s - 15ms/step - loss: 0.7917 - val_loss: 0.8397
Epoch 64/100
9/9 - 0s - 31ms/step - loss: 0.7774 - val_loss: 0.8271
Epoch 65/100
9/9 - 0s - 15ms/step - loss: 0.7706 - val_loss: 0.8160
Epoch 66/100
9/9 - 0s - 15ms/step - loss: 0.7609 - val_loss: 0.8117
Epoch 67/100
9/9 - 0s - 14ms/step - loss: 0.7561 - val_loss: 0.7955
Epoch 68/100
9/9 - 0s - 15ms/step - loss: 0.7425 - val_loss: 0.8134
Epoch 69/100
9/9 - 0s - 16ms/step - loss: 0.7329 - val_loss: 0.7912
Epoch 70/100
9/9 - 0s - 15ms/step - loss: 0.7235 - val_loss: 0.7742
Epoch 71/100
9/9 - 0s - 17ms/step - loss: 0.7139 - val_loss: 0.7724
Epoch 72/100
9/9 - 0s - 14ms/step - loss: 0.7071 - val_loss: 0.7632
Epoch 73/100
9/9 - 0s - 15ms/step - loss: 0.6987 - val_loss: 0.7428
Epoch 74/100
9/9 - 0s - 14ms/step - loss: 0.6913 - val_loss: 0.7318
Epoch 75/100
9/9 - 0s - 13ms/step - loss: 0.6848 - val_loss: 0.7522
Epoch 76/100
9/9 - 0s - 16ms/step - loss: 0.6728 - val_loss: 0.7295
Epoch 77/100
9/9 - 0s - 17ms/step - loss: 0.6611 - val_loss: 0.6975
Epoch 78/100
9/9 - 0s - 13ms/step - loss: 0.6547 - val_loss: 0.7032
Epoch 79/100
9/9 - 0s - 16ms/step - loss: 0.6465 - val_loss: 0.6906
Epoch 80/100
9/9 - 0s - 31ms/step - loss: 0.6417 - val_loss: 0.7035
Epoch 81/100
9/9 - 0s - 35ms/step - loss: 0.6343 - val_loss: 0.6936
Epoch 82/100
9/9 - 0s - 52ms/step - loss: 0.6257 - val_loss: 0.6792
Epoch 83/100
9/9 - 0s - 14ms/step - loss: 0.6173 - val_loss: 0.6805
Epoch 84/100
9/9 - 0s - 16ms/step - loss: 0.6101 - val_loss: 0.6699
Epoch 85/100
9/9 - 0s - 15ms/step - loss: 0.6048 - val_loss: 0.6524
Epoch 86/100
9/9 - 0s - 13ms/step - loss: 0.5964 - val_loss: 0.6475
Epoch 87/100
9/9 - 0s - 18ms/step - loss: 0.5886 - val_loss: 0.6446
Epoch 88/100
9/9 - 0s - 31ms/step - loss: 0.5879 - val_loss: 0.6447
Epoch 89/100
9/9 - 0s - 14ms/step - loss: 0.5746 - val_loss: 0.6427
Epoch 90/100
9/9 - 0s - 15ms/step - loss: 0.5695 - val_loss: 0.6086
```

```
Epoch 91/100
9/9 - 0s - 18ms/step - loss: 0.5596 - val_loss: 0.6236
Epoch 92/100
9/9 - 0s - 13ms/step - loss: 0.5597 - val_loss: 0.6209
Epoch 93/100
9/9 - 0s - 17ms/step - loss: 0.5520 - val_loss: 0.6169
Epoch 94/100
9/9 - 0s - 14ms/step - loss: 0.5436 - val_loss: 0.6002
Epoch 95/100
9/9 - 0s - 14ms/step - loss: 0.5394 - val_loss: 0.5911
Epoch 96/100
9/9 - 0s - 16ms/step - loss: 0.5341 - val_loss: 0.5915
Epoch 97/100
9/9 - 0s - 14ms/step - loss: 0.5312 - val_loss: 0.5828
Epoch 98/100
9/9 - 0s - 13ms/step - loss: 0.5275 - val_loss: 0.5762
Epoch 99/100
9/9 - 0s - 13ms/step - loss: 0.5170 - val_loss: 0.5763
Epoch 100/100
9/9 - 0s - 16ms/step - loss: 0.5125 - val_loss: 0.5712
```

Out[60]: <keras.src.callbacks.history.History at 0x7d36050ba710>

In [61]:
```python
#Train and test

prediction = model.predict(X_test)
prediction = scaler_output.inverse_transform(prediction)
prediction = prediction.flatten()

values = np.array(y_test[window_size:])
values = values.flatten()


def R2_campbell(y_true, y_predicted, mean_ret):
    y_predicted = y_predicted.reshape((-1,))
    sse = sum((y_true - y_predicted) ** 2)
    tse = sum((y_true - mean_ret) ** 2)
    r2_score = 1 - (sse / tse)

    return r2_score


R2_Campbell = R2_campbell(values, prediction, mean_ret)
print("Out-of-sample R-squared:", R2_Campbell)

# %%
df_predictions = pd.DataFrame(
    {"Date": test_time.flatten(), "Pred LSTM": prediction, "values": values}
)
df_predictions.head()
```

```
13/13 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step
Out-of-sample R-squared: -0.0749841115727965
```

Out[61]:

	Date	Pred LSTM	values
0	2021-04-06	-0.001686	0.054650
1	2021-04-07	-0.001281	0.047125
2	2021-04-08	0.000052	0.040241
3	2021-04-09	0.001573	0.057696
4	2021-04-12	0.004156	0.076534

In [62]:
```
df4_actual=df_predictions[['values']].to_numpy()
df4_pred=df_predictions[['Pred LSTM']].to_numpy()
```

LSTM Model 5 : DBO

In [63]:
```
df_dbo = df[["Date", "DBO"]].copy()
df_dbo
```

Out[63]:

Ticker	Date	DBO
0	2018-01-03	0.021360
1	2018-01-04	0.001919
2	2018-01-05	-0.004805
3	2018-01-08	0.005764
4	2018-01-09	0.017094
...	...	...
1252	2022-12-22	-0.007486
1253	2022-12-23	0.024293
1254	2022-12-27	0.007307
1255	2022-12-28	-0.010645
1256	2022-12-29	-0.004693

1257 rows × 2 columns

In [64]:
```
df_dbo["Ret_10"] = df_dbo["DBO"].rolling(10).apply(lambda x: np.exp(np.sum(x)) -
df_dbo["Ret_50"] = df_dbo["DBO"].rolling(50).apply(lambda x: np.exp(np.sum(x)) -
df_dbo["Ret_25"] = df_dbo["DBO"].rolling(25).apply(lambda x: np.exp(np.sum(x)) -

df_dbo["Ret25"] = df_dbo["Ret_25"].shift(-25)
del df_dbo["Ret_25"]
df_dbo = df_dbo.dropna()
df_dbo.head()
```

Out[64]:

	Ticker	Date	DBO	Ret_10	Ret_50	Ret25
49		2018-03-15	0.002818	0.016206	0.046124	0.098499
50		2018-03-16	0.012121	0.023719	0.036503	0.096386
51		2018-03-19	-0.005576	0.006567	0.028763	0.086673
52		2018-03-20	0.014801	0.020619	0.049133	0.076217
53		2018-03-21	0.026282	0.062738	0.070881	0.054562

In [65]:
```
Xdf, ydf = df_dbo.iloc[:, 1:-1], df_dbo.iloc[:, -1]
X = Xdf.astype("float32")
y = ydf.astype("float32")
```

In [66]:
```
val_split = 0.2
train_split = 0.625
train_size = int(len(df_dbo) * train_split)
val_size = int(train_size * val_split)
test_size = int(len(df_dbo) - train_size)

window_size = 30

ts = test_size
split_time = len(df_dbo) - ts
test_time = df_dbo.iloc[split_time + window_size :, 0:1].values


y_train_set = y[:split_time]
y_test_set = y[split_time:]

X_train_set = X[:split_time]
X_test_set = X[split_time:]

n_features = X_train_set.shape[1]
```

In [67]:
```
scaler_input = MinMaxScaler(feature_range=(-1, 1))
scaler_input.fit(X_train_set)
X_train_set_scaled = scaler_input.transform(X_train_set)
X_test_set_scaled = scaler_input.transform(X_test_set)

mean_ret = np.mean(y_train_set)

scaler_output = MinMaxScaler(feature_range=(-1, 1))
y_train_set = y_train_set.values.reshape(len(y_train_set), 1)
y_test_set = y_test_set.values.reshape(len(y_test_set), 1)
scaler_output.fit(y_train_set)
y_train_set_scaled = scaler_output.transform(y_train_set)
```

In [68]:
```
training_time = df_dbo.iloc[:split_time, 0:1].values

X_train = []
y_train = []

for i in range(window_size, y_train_set_scaled.shape[0]):
    X_train.append(X_train_set_scaled[i - window_size : i, :])
    y_train.append(y_train_set_scaled[i])
```

```
X_train, y_train = np.array(X_train), np.array(y_train)

print("Shape of training data", X_train.shape, y_train.shape)

X_test = []
y_test = y_test_set

for i in range(window_size, y_test_set.shape[0]):
    X_test.append(X_test_set_scaled[i - window_size : i, :])

X_test, y_test = np.array(X_test), np.array(y_test)

print("Shape of test data", X_test.shape, y_test.shape)
```

```
Shape of training data (709, 30, 3) (709, 1)
Shape of test data (414, 30, 3) (444, 1)
```

In [69]:
```
from keras.regularizers import l2
from keras.optimizers import Adam
```

In [70]:
```
SEED = 321
# Define hyperparameters
units_lstm = 50
n_dropout = 0.2
act_fun = "relu"

# Define the model
model = Sequential()

# First LSTM layer with Tanh activation and L2 regularization
model.add(
    LSTM(
        units=units_lstm,
        return_sequences=True,
        activation="tanh",
        input_shape=(X_train.shape[1], n_features),
        kernel_regularizer=l2(0.01)
    )
)
model.add(Dropout(n_dropout, seed=SEED))

# Second LSTM layer with Tanh activation and L2 regularization
model.add(LSTM(units=units_lstm, return_sequences=True, activation="tanh", kerne
model.add(Dropout(n_dropout, seed=SEED))

# Third LSTM layer with Tanh activation and L2 regularization
model.add(LSTM(units=units_lstm, return_sequences=False, activation="tanh", kern
model.add(Dropout(n_dropout, seed=SEED))

# Dense layer with ReLU activation
model.add(Dense(units=20, activation=act_fun))
model.add(Dropout(n_dropout, seed=SEED))

# Output layer with Linear activation (for regression)
model.add(Dense(units=1, activation='linear'))

# Compile the model with Adam optimizer and MAE loss
hp_lr = 1e-4
model.compile(optimizer=Adam(learning_rate=hp_lr), loss='mean_absolute_error')
```

```
es = EarlyStopping(
    monitor="val_loss", mode="min", verbose=0, patience=5, restore_best_weights=
)

# Summary of the model
model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/rnn.py:204: UserWarn
ing: 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 m
odel instead.
  super().__init__(**kwargs)
```

Model: "sequential_4"

Layer (type)	Output Shape	
lstm_12 (LSTM)	(None, 30, 50)	
dropout_16 (Dropout)	(None, 30, 50)	
lstm_13 (LSTM)	(None, 30, 50)	
dropout_17 (Dropout)	(None, 30, 50)	
lstm_14 (LSTM)	(None, 50)	
dropout_18 (Dropout)	(None, 50)	
dense_8 (Dense)	(None, 20)	
dropout_19 (Dropout)	(None, 20)	
dense_9 (Dense)	(None, 1)	

Total params: 52,241 (204.07 KB)

Trainable params: 52,241 (204.07 KB)

Non-trainable params: 0 (0.00 B)

In [71]:
```
# fit the models
model.fit(
    X_train,
    y_train,
    validation_split=val_split,
    epochs=100,
    batch_size=64,
    verbose=2,
    callbacks=[es],
)
```

```
Epoch 1/100
9/9 - 3s - 344ms/step - loss: 1.9276 - val_loss: 1.9802
Epoch 2/100
9/9 - 0s - 16ms/step - loss: 1.8887 - val_loss: 1.9347
Epoch 3/100
9/9 - 0s - 16ms/step - loss: 1.8525 - val_loss: 1.8897
Epoch 4/100
9/9 - 0s - 14ms/step - loss: 1.8139 - val_loss: 1.8460
Epoch 5/100
9/9 - 0s - 15ms/step - loss: 1.7832 - val_loss: 1.8063
Epoch 6/100
9/9 - 0s - 15ms/step - loss: 1.7570 - val_loss: 1.7733
Epoch 7/100
9/9 - 0s - 14ms/step - loss: 1.7295 - val_loss: 1.7493
Epoch 8/100
9/9 - 0s - 15ms/step - loss: 1.7025 - val_loss: 1.7257
Epoch 9/100
9/9 - 0s - 15ms/step - loss: 1.6802 - val_loss: 1.7075
Epoch 10/100
9/9 - 0s - 14ms/step - loss: 1.6514 - val_loss: 1.6863
Epoch 11/100
9/9 - 0s - 16ms/step - loss: 1.6244 - val_loss: 1.6629
Epoch 12/100
9/9 - 0s - 15ms/step - loss: 1.6031 - val_loss: 1.6377
Epoch 13/100
9/9 - 0s - 14ms/step - loss: 1.5810 - val_loss: 1.6124
Epoch 14/100
9/9 - 0s - 16ms/step - loss: 1.5544 - val_loss: 1.5883
Epoch 15/100
9/9 - 0s - 15ms/step - loss: 1.5335 - val_loss: 1.5635
Epoch 16/100
9/9 - 0s - 17ms/step - loss: 1.5119 - val_loss: 1.5477
Epoch 17/100
9/9 - 0s - 34ms/step - loss: 1.4904 - val_loss: 1.5275
Epoch 18/100
9/9 - 0s - 13ms/step - loss: 1.4667 - val_loss: 1.5064
Epoch 19/100
9/9 - 0s - 14ms/step - loss: 1.4450 - val_loss: 1.4834
Epoch 20/100
9/9 - 0s - 15ms/step - loss: 1.4223 - val_loss: 1.4617
Epoch 21/100
9/9 - 0s - 16ms/step - loss: 1.4040 - val_loss: 1.4393
Epoch 22/100
9/9 - 0s - 18ms/step - loss: 1.3837 - val_loss: 1.4183
Epoch 23/100
9/9 - 0s - 15ms/step - loss: 1.3667 - val_loss: 1.4049
Epoch 24/100
9/9 - 0s - 15ms/step - loss: 1.3437 - val_loss: 1.3877
Epoch 25/100
9/9 - 0s - 14ms/step - loss: 1.3261 - val_loss: 1.3680
Epoch 26/100
9/9 - 0s - 13ms/step - loss: 1.3048 - val_loss: 1.3477
Epoch 27/100
9/9 - 0s - 14ms/step - loss: 1.2886 - val_loss: 1.3319
Epoch 28/100
9/9 - 0s - 14ms/step - loss: 1.2694 - val_loss: 1.3148
Epoch 29/100
9/9 - 0s - 14ms/step - loss: 1.2519 - val_loss: 1.2932
Epoch 30/100
9/9 - 0s - 17ms/step - loss: 1.2329 - val_loss: 1.2685
```

```
Epoch 31/100
9/9 - 0s - 13ms/step - loss: 1.2184 - val_loss: 1.2603
Epoch 32/100
9/9 - 0s - 15ms/step - loss: 1.1997 - val_loss: 1.2464
Epoch 33/100
9/9 - 0s - 13ms/step - loss: 1.1816 - val_loss: 1.2284
Epoch 34/100
9/9 - 0s - 13ms/step - loss: 1.1634 - val_loss: 1.2024
Epoch 35/100
9/9 - 0s - 17ms/step - loss: 1.1528 - val_loss: 1.1903
Epoch 36/100
9/9 - 0s - 14ms/step - loss: 1.1315 - val_loss: 1.1833
Epoch 37/100
9/9 - 0s - 18ms/step - loss: 1.1169 - val_loss: 1.1614
Epoch 38/100
9/9 - 0s - 14ms/step - loss: 1.1041 - val_loss: 1.1449
Epoch 39/100
9/9 - 0s - 14ms/step - loss: 1.0857 - val_loss: 1.1360
Epoch 40/100
9/9 - 0s - 14ms/step - loss: 1.0701 - val_loss: 1.1218
Epoch 41/100
9/9 - 0s - 15ms/step - loss: 1.0600 - val_loss: 1.1092
Epoch 42/100
9/9 - 0s - 16ms/step - loss: 1.0433 - val_loss: 1.0896
Epoch 43/100
9/9 - 0s - 14ms/step - loss: 1.0276 - val_loss: 1.0772
Epoch 44/100
9/9 - 0s - 14ms/step - loss: 1.0137 - val_loss: 1.0670
Epoch 45/100
9/9 - 0s - 16ms/step - loss: 1.0004 - val_loss: 1.0525
Epoch 46/100
9/9 - 0s - 14ms/step - loss: 0.9862 - val_loss: 1.0351
Epoch 47/100
9/9 - 0s - 15ms/step - loss: 0.9739 - val_loss: 1.0274
Epoch 48/100
9/9 - 0s - 16ms/step - loss: 0.9612 - val_loss: 1.0116
Epoch 49/100
9/9 - 0s - 14ms/step - loss: 0.9461 - val_loss: 0.9949
Epoch 50/100
9/9 - 0s - 14ms/step - loss: 0.9339 - val_loss: 0.9805
Epoch 51/100
9/9 - 0s - 16ms/step - loss: 0.9240 - val_loss: 0.9733
Epoch 52/100
9/9 - 0s - 16ms/step - loss: 0.9070 - val_loss: 0.9677
Epoch 53/100
9/9 - 0s - 21ms/step - loss: 0.8979 - val_loss: 0.9512
Epoch 54/100
9/9 - 0s - 21ms/step - loss: 0.8826 - val_loss: 0.9398
Epoch 55/100
9/9 - 0s - 34ms/step - loss: 0.8727 - val_loss: 0.9243
Epoch 56/100
9/9 - 0s - 35ms/step - loss: 0.8615 - val_loss: 0.9120
Epoch 57/100
9/9 - 0s - 20ms/step - loss: 0.8485 - val_loss: 0.9027
Epoch 58/100
9/9 - 0s - 21ms/step - loss: 0.8409 - val_loss: 0.8901
Epoch 59/100
9/9 - 0s - 22ms/step - loss: 0.8287 - val_loss: 0.8804
Epoch 60/100
9/9 - 0s - 24ms/step - loss: 0.8167 - val_loss: 0.8725
```

```
Epoch 61/100
9/9 - 0s - 24ms/step - loss: 0.8070 - val_loss: 0.8604
Epoch 62/100
9/9 - 0s - 22ms/step - loss: 0.7945 - val_loss: 0.8546
Epoch 63/100
9/9 - 0s - 23ms/step - loss: 0.7856 - val_loss: 0.8398
Epoch 64/100
9/9 - 0s - 28ms/step - loss: 0.7750 - val_loss: 0.8277
Epoch 65/100
9/9 - 0s - 15ms/step - loss: 0.7652 - val_loss: 0.8221
Epoch 66/100
9/9 - 0s - 34ms/step - loss: 0.7531 - val_loss: 0.8054
Epoch 67/100
9/9 - 0s - 15ms/step - loss: 0.7454 - val_loss: 0.7969
Epoch 68/100
9/9 - 0s - 14ms/step - loss: 0.7387 - val_loss: 0.7964
Epoch 69/100
9/9 - 0s - 15ms/step - loss: 0.7279 - val_loss: 0.7787
Epoch 70/100
9/9 - 0s - 16ms/step - loss: 0.7194 - val_loss: 0.7715
Epoch 71/100
9/9 - 0s - 16ms/step - loss: 0.7089 - val_loss: 0.7672
Epoch 72/100
9/9 - 0s - 17ms/step - loss: 0.7005 - val_loss: 0.7555
Epoch 73/100
9/9 - 0s - 31ms/step - loss: 0.6922 - val_loss: 0.7439
Epoch 74/100
9/9 - 0s - 14ms/step - loss: 0.6835 - val_loss: 0.7428
Epoch 75/100
9/9 - 0s - 14ms/step - loss: 0.6739 - val_loss: 0.7328
Epoch 76/100
9/9 - 0s - 15ms/step - loss: 0.6640 - val_loss: 0.7222
Epoch 77/100
9/9 - 0s - 14ms/step - loss: 0.6536 - val_loss: 0.7167
Epoch 78/100
9/9 - 0s - 15ms/step - loss: 0.6495 - val_loss: 0.7055
Epoch 79/100
9/9 - 0s - 16ms/step - loss: 0.6441 - val_loss: 0.7018
Epoch 80/100
9/9 - 0s - 14ms/step - loss: 0.6341 - val_loss: 0.6969
Epoch 81/100
9/9 - 0s - 15ms/step - loss: 0.6275 - val_loss: 0.6820
Epoch 82/100
9/9 - 0s - 14ms/step - loss: 0.6217 - val_loss: 0.6802
Epoch 83/100
9/9 - 0s - 14ms/step - loss: 0.6128 - val_loss: 0.6766
Epoch 84/100
9/9 - 0s - 15ms/step - loss: 0.6003 - val_loss: 0.6640
Epoch 85/100
9/9 - 0s - 14ms/step - loss: 0.5967 - val_loss: 0.6540
Epoch 86/100
9/9 - 0s - 16ms/step - loss: 0.5895 - val_loss: 0.6516
Epoch 87/100
9/9 - 0s - 15ms/step - loss: 0.5814 - val_loss: 0.6479
Epoch 88/100
9/9 - 0s - 15ms/step - loss: 0.5797 - val_loss: 0.6391
Epoch 89/100
9/9 - 0s - 14ms/step - loss: 0.5704 - val_loss: 0.6256
Epoch 90/100
9/9 - 0s - 14ms/step - loss: 0.5630 - val_loss: 0.6229
```

```
Epoch 91/100
9/9 - 0s - 15ms/step - loss: 0.5592 - val_loss: 0.6225
Epoch 92/100
9/9 - 0s - 14ms/step - loss: 0.5510 - val_loss: 0.6124
Epoch 93/100
9/9 - 0s - 16ms/step - loss: 0.5430 - val_loss: 0.6053
Epoch 94/100
9/9 - 0s - 14ms/step - loss: 0.5382 - val_loss: 0.6012
Epoch 95/100
9/9 - 0s - 15ms/step - loss: 0.5327 - val_loss: 0.5906
Epoch 96/100
9/9 - 0s - 14ms/step - loss: 0.5275 - val_loss: 0.5889
Epoch 97/100
9/9 - 0s - 13ms/step - loss: 0.5194 - val_loss: 0.5808
Epoch 98/100
9/9 - 0s - 14ms/step - loss: 0.5116 - val_loss: 0.5746
Epoch 99/100
9/9 - 0s - 15ms/step - loss: 0.5079 - val_loss: 0.5701
Epoch 100/100
9/9 - 0s - 17ms/step - loss: 0.5030 - val_loss: 0.5654
```

Out[71]: <keras.src.callbacks.history.History at 0x7d357c7db790>

In [72]:
```python
#Train and test

prediction = model.predict(X_test)
prediction = scaler_output.inverse_transform(prediction)
prediction = prediction.flatten()

values = np.array(y_test[window_size:])
values = values.flatten()


def R2_campbell(y_true, y_predicted, mean_ret):
    y_predicted = y_predicted.reshape((-1,))
    sse = sum((y_true - y_predicted) ** 2)
    tse = sum((y_true - mean_ret) ** 2)
    r2_score = 1 - (sse / tse)

    return r2_score


R2_Campbell = R2_campbell(values, prediction, mean_ret)
print("Out-of-sample R-squared:", R2_Campbell)

# %%
df_predictions = pd.DataFrame(
    {"Date": test_time.flatten(), "Pred LSTM": prediction, "values": values}
)
df_predictions.head()
```

```
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
Out-of-sample R-squared: -0.21064766830888093
```

Out[72]:

	Date	Pred LSTM	values
0	2021-04-06	-0.005765	0.097106
1	2021-04-07	-0.009215	0.104478
2	2021-04-08	-0.012329	0.077934
3	2021-04-09	-0.015165	0.097744
4	2021-04-12	-0.017567	0.103545

In [73]:
```
df5_actual=df_predictions[['values']].to_numpy()
df5_pred=df_predictions[['Pred LSTM']].to_numpy()
```

In [73]:
```
```

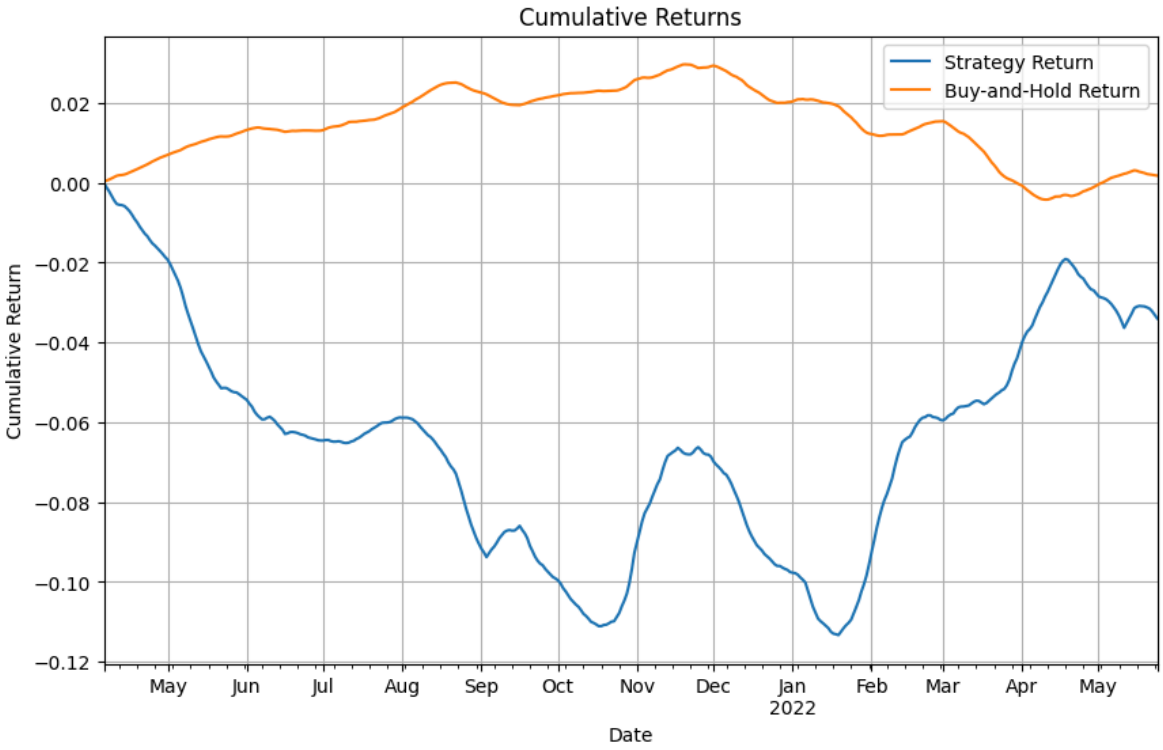
Trading Strategy & backtest

In this part of the code, we are going to implement a trading strategy based on the performance of the models that we built. Consequently, we are going to carry out a backtest to see how well our model performs in the past.

In [74]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Generate example dataset with 5 models predicting 5 asset classes
np.random.seed(42)  # For reproducibility
dates = pd.date_range(start="2021-04-06", periods=414)
dates = pd.date_range(start="2021-04-06", periods=414)
data = {
    "Model1_Asset1": df1_pred.flatten(),
    "Model1_Asset2": df2_pred.flatten(),
    "Model1_Asset3": df3_pred.flatten(),
    "Model1_Asset4": df4_pred.flatten(),
    "Model1_Asset5": df5_pred.flatten(),

    # Simulated actual returns
    "Asset1_Actual": df1_actual.flatten(),
    "Asset2_Actual": df2_actual.flatten(),
    "Asset3_Actual": df3_actual.flatten(),
    "Asset4_Actual": df4_actual.flatten(),
    "Asset5_Actual": df5_actual.flatten(),
}


df_prediction1 = pd.DataFrame(data, index=dates)

# Step 1: Normalize actual returns
scaling_factor = 100  # Ensure actual returns are realistic percentages
for asset in ["Asset1", "Asset2", "Asset3", "Asset4", "Asset5"]:
    df_prediction1[f"{asset}_Actual"] /= scaling_factor

# Step 2: Aggregate predictions from models for each asset class
asset_classes = ["Asset1", "Asset2", "Asset3", "Asset4", "Asset5"]
for asset in asset_classes:
```

```
    model_columns = [col for col in df_prediction1.columns if asset in col]
    df_prediction1[f"{asset}_AggPred"] = df_prediction1[model_columns].mean(axis

# Normalize aggregated predictions
for asset in asset_classes:
    df_prediction1[f"{asset}_AggPred"] = (df_prediction1[f"{asset}_AggPred"] - d

# Step 3: Define a 25-day rebalancing strategy
rebalance_period = 25
positions = []

for i in range(0, len(df_prediction1), rebalance_period):
    # Get predictions for the rebalancing window
    window = df_prediction1.iloc[i:i + rebalance_period]
    # Rank asset classes by aggregated predictions
    aggregated_predictions = {asset: window[f"{asset}_AggPred"].mean() for asset
    ranked_assets = sorted(aggregated_predictions.items(), key=lambda x: x[1], r

    # Determine long and short positions
    long_assets = [ranked_assets[j][0] for j in range(2)]  # Top 2 assets
    short_assets = [ranked_assets[j][0] for j in range(-2, 0)]  # Bottom 2 asset

    # Assign positions for the window
    for day in range(len(window)):
        position = {asset: 1 if asset in long_assets else (-1 if asset in short_
        positions.append(position)

# Step 4: Add positions to the DataFrame
positions_df = pd.DataFrame(positions, index=df_prediction1.index[:len(positions
for asset in asset_classes:
    df_prediction1[f"{asset}_Position"] = positions_df[asset]

# Step 5: Calculate strategy returns
df_prediction1["Strat_ret"] = 0
for asset in asset_classes:
    df_prediction1["Strat_ret"] += (
        df_prediction1[f"{asset}_Position"].shift(1).fillna(0) * df_prediction1[
    )

# Step 6: Calculate cumulative returns
df_prediction1["CumRet"] = (1 + df_prediction1["Strat_ret"]).cumprod() - 1
df_prediction1["bhRet"] = (1 + df_prediction1[["Asset1_Actual", "Asset2_Actual",

# Step 7: Plot cumulative returns
plt.figure(figsize=(10, 6))
ax = plt.gca()
df_prediction1.plot(y="CumRet", label="Strategy Return", ax=ax)
df_prediction1.plot(y="bhRet", label="Buy-and-Hold Return", ax=ax)
plt.title("Cumulative Returns")
plt.xlabel("Date")
plt.ylabel("Cumulative Return")
plt.legend()
plt.grid(True)
plt.show()

# Step 8: Debugging Outputs
print("\n===== Debugging Outputs =====")
print("Strategy Returns Summary:")
print(df_prediction1["Strat_ret"].describe())
```

```
print("\nActual Returns Summary:")
for asset in asset_classes:
    print(f"{asset}_Actual Summary:")
    print(df_prediction1[f"{asset}_Actual"].describe())

print("\nAggregated Predictions Summary:")
for asset in asset_classes:
    print(f"{asset}_AggPred Summary:")
    print(df_prediction1[f"{asset}_AggPred"].describe())
```



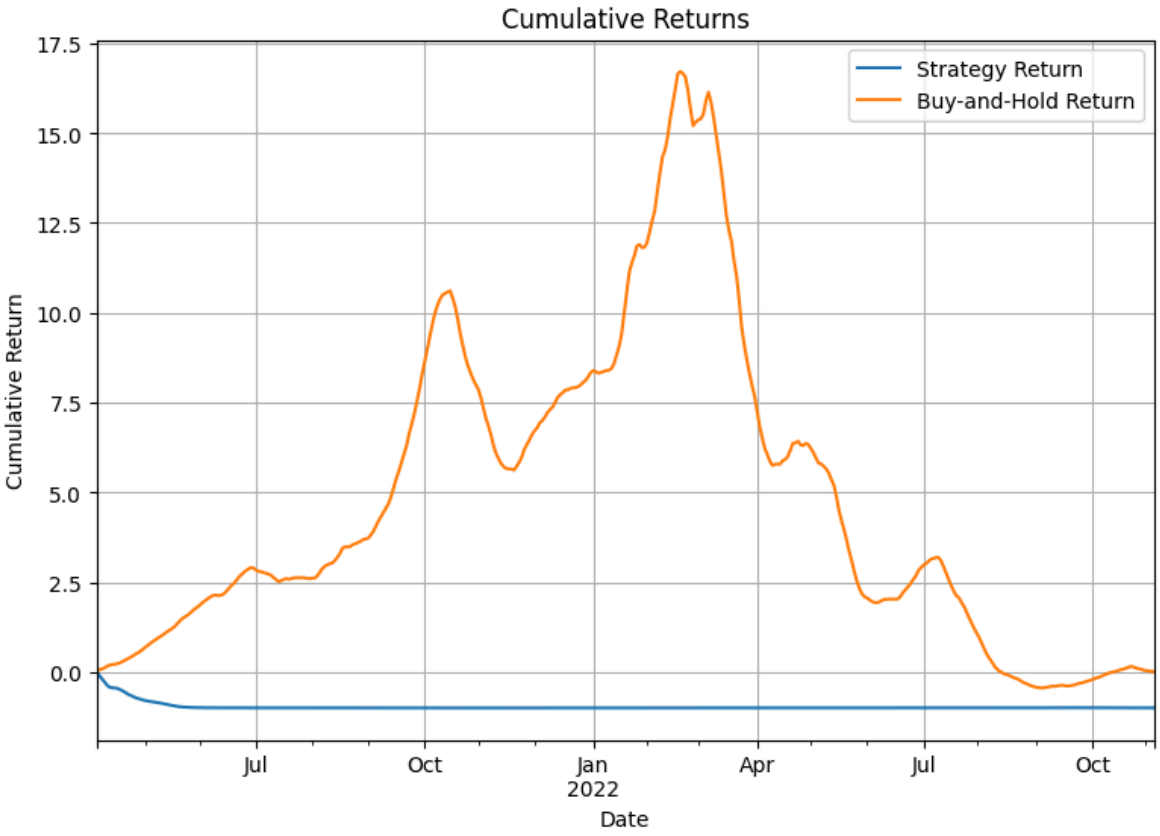
```
===== Debugging Outputs =====
Strategy Returns Summary:
count    414.000000
mean      -0.000083
std        0.001196
min       -0.002576
25%       -0.000871
50%       -0.000249
75%        0.000516
max        0.004570
Name: Strat_ret, dtype: float64

Actual Returns Summary:
Asset1_Actual Summary:
count    414.000000
mean      -0.000017
std        0.000558
min       -0.001486
25%       -0.000422
50%        0.000078
75%        0.000345
max        0.001323
Name: Asset1_Actual, dtype: float64
Asset2_Actual Summary:
count    414.000000
mean      -0.000154
std        0.000533
min       -0.001394
25%       -0.000521
50%       -0.000182
75%        0.000200
max        0.001546
Name: Asset2_Actual, dtype: float64
Asset3_Actual Summary:
count    414.000000
mean      -0.000035
std        0.000054
min       -0.000199
25%       -0.000071
50%       -0.000024
75%        0.000001
max        0.000112
Name: Asset3_Actual, dtype: float64
Asset4_Actual Summary:
count    414.000000
mean       0.000009
std        0.000413
min       -0.000859
25%       -0.000289
50%       -0.000033
75%        0.000279
max        0.001393
Name: Asset4_Actual, dtype: float64
Asset5_Actual Summary:
count    414.000000
mean       0.000220
std        0.000962
min       -0.002185
25%       -0.000439
50%        0.000288
```

```
75%        0.001014
max        0.002354
Name: Asset5_Actual, dtype: float64

Aggregated Predictions Summary:
Asset1_AggPred Summary:
count    4.140000e+02
mean    -1.474279e-07
std      1.000000e+00
min     -1.054328e+00
25%     -7.270955e-01
50%     -4.445593e-01
75%      5.692315e-01
max      3.310513e+00
Name: Asset1_AggPred, dtype: float64
Asset2_AggPred Summary:
count    4.140000e+02
mean    -3.685698e-08
std      9.999995e-01
min     -2.430957e+00
25%     -5.712716e-01
50%      6.565616e-02
75%      7.557077e-01
max      1.857373e+00
Name: Asset2_AggPred, dtype: float64
Asset3_AggPred Summary:
count    4.140000e+02
mean     1.842849e-08
std      9.999995e-01
min     -2.492889e+00
25%     -6.746811e-01
50%      2.800760e-01
75%      7.348438e-01
max      2.111022e+00
Name: Asset3_AggPred, dtype: float64
Asset4_AggPred Summary:
count    4.140000e+02
mean    -9.214244e-08
std      1.000000e+00
min     -2.293982e+00
25%     -7.207858e-01
50%      8.108197e-02
75%      7.641779e-01
max      1.889821e+00
Name: Asset4_AggPred, dtype: float64
Asset5_AggPred Summary:
count    4.140000e+02
mean     1.842849e-08
std      1.000000e+00
min     -2.521746e+00
25%     -4.213583e-01
50%     -1.178829e-01
75%      5.178796e-01
max      2.097086e+00
Name: Asset5_AggPred, dtype: float64
```

In [75]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
# Generate example dataset with single model predictions for 5 asset classes
np.random.seed(42)  # For reproducibility
dates = pd.bdate_range(start="2021-04-06", periods=414)
data = {
    "Model1_Asset1": df1_pred.flatten(),
    "Model1_Asset2": df2_pred.flatten(),
    "Model1_Asset3": df3_pred.flatten(),
    "Model1_Asset4": df4_pred.flatten(),
    "Model1_Asset5": df5_pred.flatten(),

    # Simulated actual returns
    "Asset1_Actual": df1_actual.flatten(),
    "Asset2_Actual": df2_actual.flatten(),
    "Asset3_Actual": df3_actual.flatten(),
    "Asset4_Actual": df4_actual.flatten(),
    "Asset5_Actual": df5_actual.flatten(),
}

df_prediction1 = pd.DataFrame(data, index=dates)

# Step 1: Normalize actual returns (if required)
# Since the actual returns are already realistic (as decimals), no scaling is ap
# Remove unnecessary division by scaling factor.

# Step 2: Normalize single model predictions for each asset
asset_classes = ["Asset1", "Asset2", "Asset3", "Asset4", "Asset5"]
for asset in asset_classes:
    # Normalize each model's predictions
    df_prediction1[f"{asset}_NormPred"] = (
        df_prediction1[f"Model1_{asset}"] - df_prediction1[f"Model1_{asset}"].me
    ) / df_prediction1[f"Model1_{asset}"].std()

# Step 3: Define a 25-day rebalancing strategy
rebalance_period = 25
positions = []

for i in range(0, len(df_prediction1), rebalance_period):
    # Get predictions for the rebalancing window
    window = df_prediction1.iloc[i:i + rebalance_period]
    # Rank asset classes by normalized predictions
    aggregated_predictions = {asset: window[f"{asset}_NormPred"].mean() for asse
    ranked_assets = sorted(aggregated_predictions.items(), key=lambda x: x[1], r

    # Determine long and short positions
    long_assets = [ranked_assets[j][0] for j in range(2)]  # Top 2 assets
    short_assets = [ranked_assets[j][0] for j in range(-2, 0)]  # Bottom 2 asset

    # Assign positions for the window
    for day in range(len(window)):
        position = {asset: 1 if asset in long_assets else (-1 if asset in short_
        positions.append(position)

# Step 4: Add positions to the DataFrame
positions_df = pd.DataFrame(positions, index=df_prediction1.index[:len(positions
for asset in asset_classes:
    df_prediction1[f"{asset}_Position"] = positions_df[asset]

# Step 5: Calculate strategy returns
df_prediction1["Strat_ret"] = 0
for asset in asset_classes:
```

```
    df_prediction1["Strat_ret"] += (
        df_prediction1[f"{asset}_Position"].shift(1).fillna(0) * df_prediction1[
    )

# Step 6: Calculate cumulative returns
df_prediction1["CumRet"] = (1 + df_prediction1["Strat_ret"]).cumprod() - 1
df_prediction1["bhRet"] = (1 + df_prediction1[["Asset1_Actual", "Asset2_Actual",

# Step 7: Plot cumulative returns
plt.figure(figsize=(9, 6))
ax = plt.gca()
df_prediction1.plot(y="CumRet", label="Strategy Return", ax=ax)
df_prediction1.plot(y="bhRet", label="Buy-and-Hold Return", ax=ax)
plt.title("Cumulative Returns")
plt.xlabel("Date")
plt.ylabel("Cumulative Return")
plt.legend()
plt.grid(True)
plt.show()

# Step 8: Debugging Outputs
print("\n===== Debugging Outputs =====")
print("Strategy Returns Summary:")
print(df_prediction1["Strat_ret"].describe())

print("\nActual Returns Summary:")
for asset in asset_classes:
    print(f"{asset}_Actual Summary:")
    print(df_prediction1[f"{asset}_Actual"].describe())

print("\nNormalized Predictions Summary:")
for asset in asset_classes:
    print(f"{asset}_NormPred Summary:")
    print(df_prediction1[f"{asset}_NormPred"].describe())
```



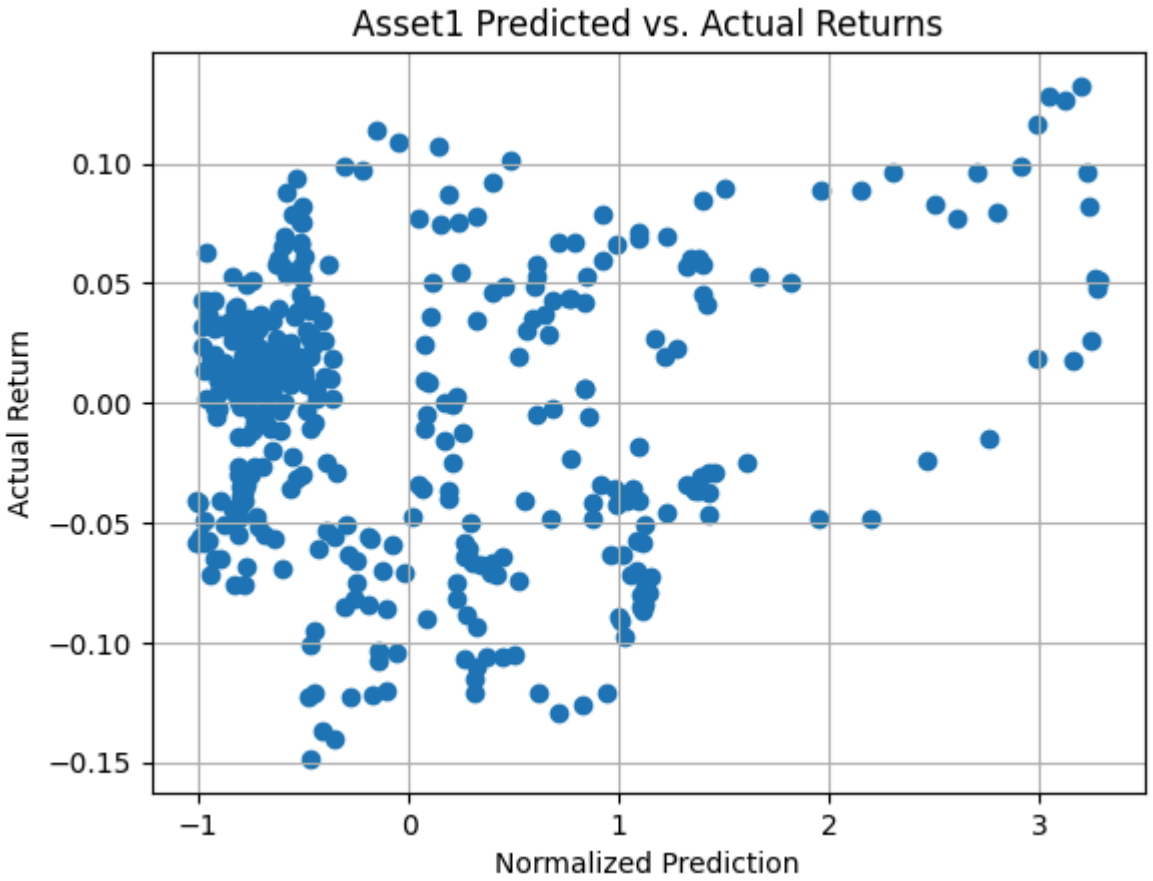
```
===== Debugging Outputs =====
Strategy Returns Summary:
count    414.000000
mean      -0.008281
std        0.119638
min       -0.257634
25%       -0.087080
50%       -0.024868
75%        0.051563
max        0.457011
Name: Strat_ret, dtype: float64

Actual Returns Summary:
Asset1_Actual Summary:
count    414.000000
mean      -0.001744
std        0.055836
min       -0.148578
25%       -0.042159
50%        0.007755
75%        0.034521
max        0.132254
Name: Asset1_Actual, dtype: float64
Asset2_Actual Summary:
count    414.000000
mean      -0.015412
std        0.053331
min       -0.139385
25%       -0.052149
50%       -0.018177
75%        0.019964
max        0.154588
Name: Asset2_Actual, dtype: float64
Asset3_Actual Summary:
count    414.000000
mean      -0.003538
std        0.005396
min       -0.019946
25%       -0.007060
50%       -0.002382
75%        0.000119
max        0.011165
Name: Asset3_Actual, dtype: float64
Asset4_Actual Summary:
count    414.000000
mean       0.000886
std        0.041299
min       -0.085940
25%       -0.028918
50%       -0.003343
75%        0.027858
max        0.139330
Name: Asset4_Actual, dtype: float64
Asset5_Actual Summary:
count    414.000000
mean       0.022036
std        0.096196
min       -0.218547
25%       -0.043875
50%        0.028758
```

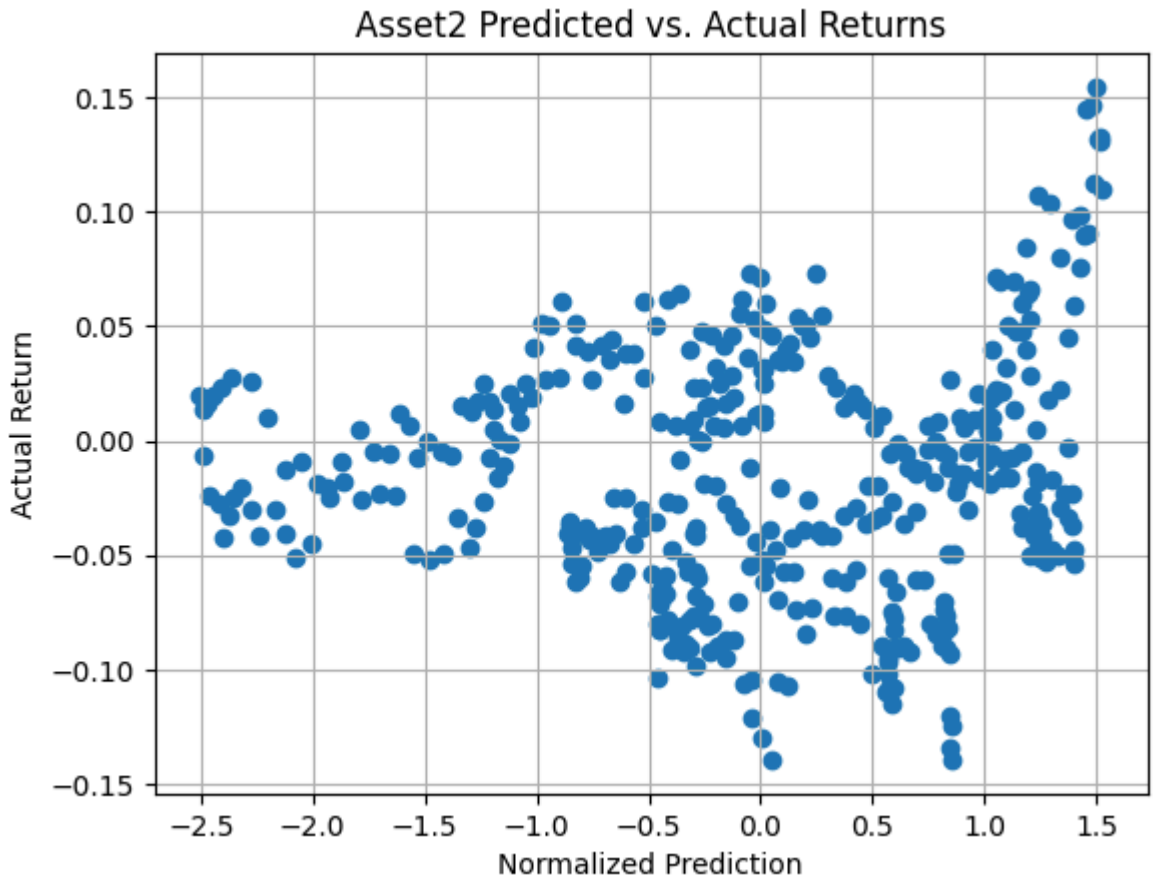
```
75%        0.101388
max        0.235450
Name: Asset5_Actual, dtype: float64

Normalized Predictions Summary:
Asset1_NormPred Summary:
count    414.000000
mean       0.000000
std        1.000000
min       -1.009284
25%       -0.740353
50%       -0.466650
75%        0.555653
max        3.285987
Name: Asset1_NormPred, dtype: float64
Asset2_NormPred Summary:
count    4.140000e+02
mean     7.371396e-08
std      1.000000e+00
min     -2.505626e+00
25%     -5.295504e-01
50%      1.636557e-02
75%      8.379345e-01
max      1.525519e+00
Name: Asset2_NormPred, dtype: float64
Asset3_NormPred Summary:
count    4.140000e+02
mean    -7.371396e-08
std      1.000000e+00
min     -2.551192e+00
25%     -6.651722e-01
50%      2.963652e-01
75%      7.130983e-01
max      1.974099e+00
Name: Asset3_NormPred, dtype: float64
Asset4_NormPred Summary:
count    4.140000e+02
mean     2.764273e-08
std      1.000000e+00
min     -2.283347e+00
25%     -7.257001e-01
50%      8.107599e-02
75%      7.264480e-01
max      1.953859e+00
Name: Asset4_NormPred, dtype: float64
Asset5_NormPred Summary:
count    4.140000e+02
mean    -3.685698e-08
std      9.999998e-01
min     -2.494257e+00
25%     -4.077196e-01
50%     -1.084992e-01
75%      5.336576e-01
max      2.022602e+00
Name: Asset5_NormPred, dtype: float64
```

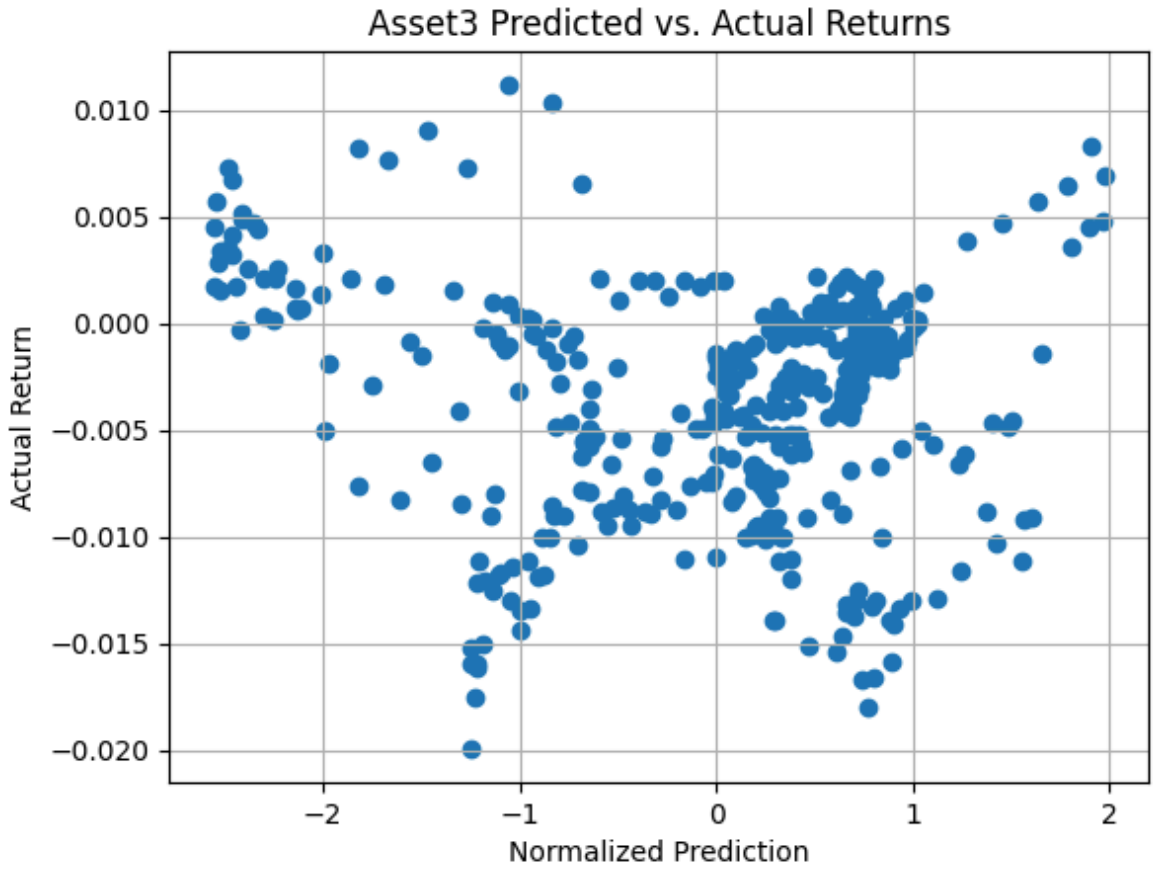
In [76]:
```
for asset in asset_classes:
    correlation = df_prediction1[f"{asset}_NormPred"].corr(df_prediction1[f"{ass
    print(f"Correlation between {asset}_NormPred and {asset}_Actual: {correlatio
```

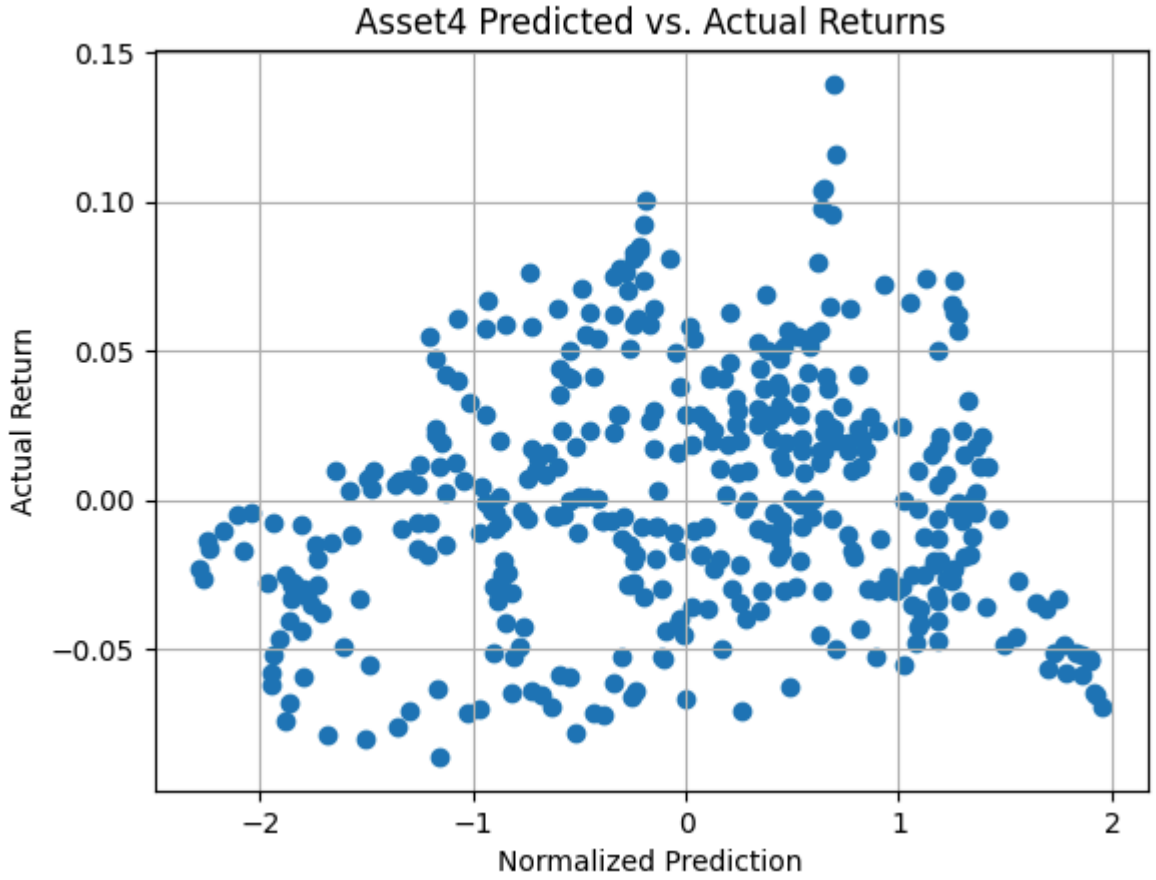
```
Correlation between Asset1_NormPred and Asset1_Actual: 0.12
Correlation between Asset2_NormPred and Asset2_Actual: 0.10
Correlation between Asset3_NormPred and Asset3_Actual: -0.09
Correlation between Asset4_NormPred and Asset4_Actual: 0.10
Correlation between Asset5_NormPred and Asset5_Actual: -0.13
```

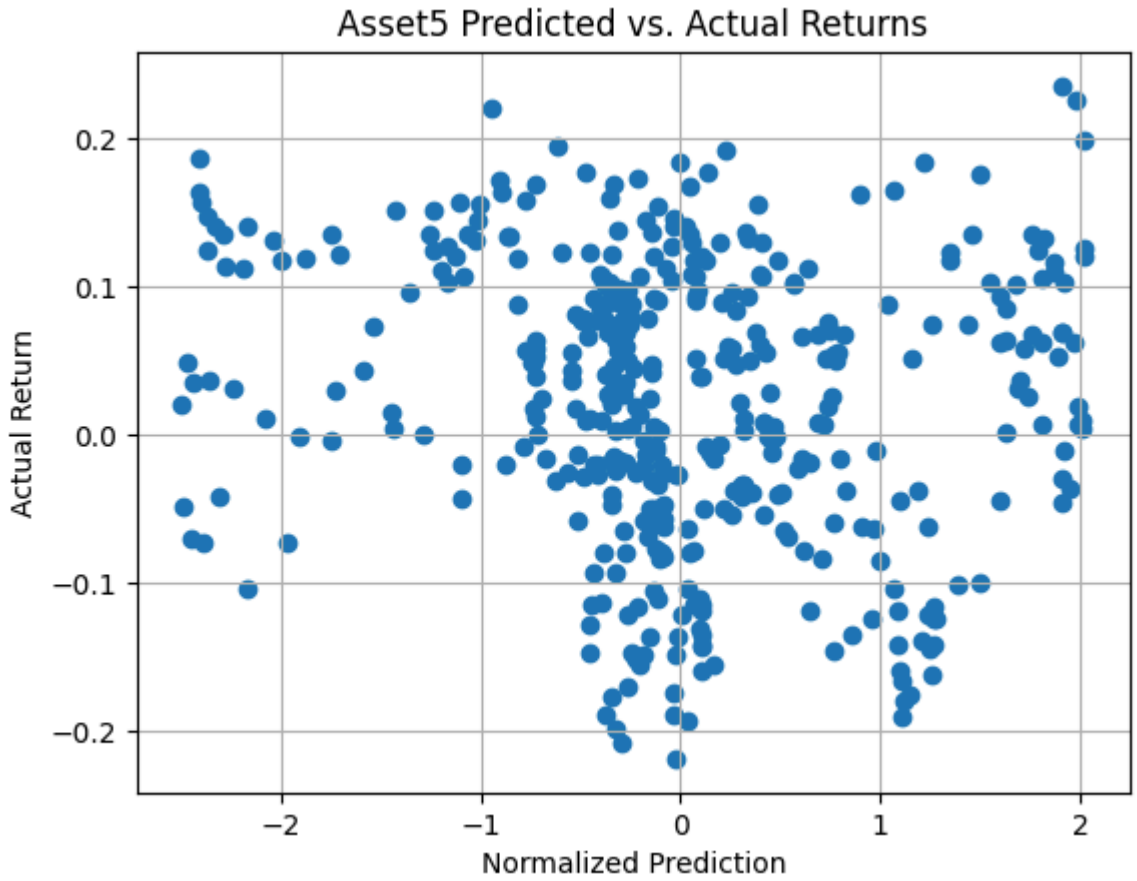
In [77]:
```
for asset in asset_classes:
    plt.scatter(df_prediction1[f"{asset}_NormPred"], df_prediction1[f"{asset}_Ac
    plt.title(f"{asset} Predicted vs. Actual Returns")
    plt.xlabel("Normalized Prediction")
    plt.ylabel("Actual Return")
    plt.grid(True)
    plt.show()
```




Asset2 Predicted vs. Actual Returns
Actual Return
Normalized Prediction
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−2.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Asset3 Predicted vs. Actual Returns
Actual Return
Normalized Prediction
0.010
0.005
0.000
−0.005
−0.010
−0.015
−0.020
−2
−1
0
1
2


Asset4 Predicted vs. Actual Returns
Actual Return
Normalized Prediction
0.15
0.10
0.05
0.00
−0.05
−2
−1
0
1
2


Asset5 Predicted vs. Actual Returns
Actual Return
Normalized Prediction
0.2
0.1
0.0
−0.1
−0.2
−2
−1
0
1
2

Step 3

So, In this step we will also build a DL models but this time of multi-output, here how we are going to do :

We will first build a DL model with multi-ouptut that uses LSTM network architecture that blends together all of the information from the 5 models of Step 2, to predict the 25-day ahead return of each ETF.

After that we will Train the single model with 5 ouptuts and do an in sample predective performance and also Test each of the model out of sample and compare the results of both across the assets.

Then we will a trading strategy that uses the out-of-sample predection of all the different models and the backtest it check our predective performance.

model description :

Here is the model description for multi-ouput model description, here we are going to use Multi-output Bidirectional LSTM architecture, here how the workflow goes :

- First we will prepare the data, we will concatenate all the ETF data together in single dataset.
- Then we will prepare the data for input by doing the same feature engineering in which we will calculate 1-day, 10-days, and 50-days cumulative returns of all the ETF at one go to predict the 25-day ahead return.
- After that we will define our data into feature, train, test and validation set, also we will also have an window i.e sequence length of 30 as we are using LSTM architecture.
- In the next step we will we will scale the different variables under the min max scalar to ease our computation then finally we will split the data in train and test split.
- Now Finally we will define our LSTM model architecture, This time we are going to do things little bit differently we are going to use bidirectional LSTM model for our predection, here's how our model work :

  > This time also our model architecture consists of a LSTM model designed for a regression task, predicting continuous values for 5 different ETFs. It begins with four Bidirectional LSTM layers, each containing 50 units and utilizing the same tanh activation function, with L2 regularization (L2=0.01) applied to the kernel weights to prevent overfitting. The first three Bidirectional LSTM layers return sequences, allowing the propagation of sequential information through the network, while the fourth Bidirectional LSTM layer does not return sequences, outputting only the final hidden state. Each Bidirectional LSTM layer is followed by a Dropout layer with a dropout rate of 0.2, ensuring robustness and generalization. After the LSTM layers, the model includes two Dense layers with 20 and 10 units respectively, both using ReLU activation, followed by Dropout layers. The

> network finishes with an output layer comprising 5 neurons with a linear activation function, suitable for regression predictions. The model is compiled with the Adam optimizer, using a learning rate of 1e-4, and the loss function is mean squared error (MSE). Early stopping is employed to monitor the validation loss, with a patience of 5 epochs and the ability to restore the best-performing weights.

- After that we will define the final features (learning rate, loss function, early stopping criteria, batch size, ...) and train the model, and after training we need to do the job of testing and to find how our model is performing we will do and final features (learning rate, loss function, early stopping criteria, batch size, ...) and train the model, and to check the perfomance of our model we will do an out of sample predection using the standard out-of-sample R-squared measured and check the results.

In [78]:
```
df
```

Out[78]:

Ticker	Date	SPY	TLT	SHY	GLD	DBO
0	2018-01-03	0.006305	0.004770	0.000000	-0.002640	0.021360
1	2018-01-04	0.004206	-0.000159	-0.000477	0.005114	0.001919
2	2018-01-05	0.006642	-0.002860	0.000000	-0.001037	-0.004805
3	2018-01-08	0.001827	-0.000637	0.000000	-0.000160	0.005764
4	2018-01-09	0.002261	-0.013463	-0.000358	-0.004639	0.017094
...	...	...	...	...	...	...
1252	2022-12-22	-0.014369	-0.000193	-0.000614	-0.012159	-0.007486
1253	2022-12-23	0.005736	-0.014769	-0.000492	0.002994	0.024293
1254	2022-12-27	-0.003951	-0.019971	-0.001354	0.008395	0.007307
1255	2022-12-28	-0.012506	-0.005909	0.000000	-0.004516	-0.010645
1256	2022-12-29	0.017840	0.011287	0.000739	0.005583	-0.004693

1257 rows × 6 columns

In [79]:
```
etfs = ['SPY', 'TLT', 'SHY', 'GLD', 'DBO']

# Create an empty list to store processed DataFrames
processed_etfs = []

for etf in etfs:
    df_etf = df[['Date', etf]].copy()
    df_etf[f"Ret_10_{etf}"] = df_etf[etf].rolling(10).apply(lambda x: np.exp(np.
    df_etf[f"Ret_50_{etf}"] = df_etf[etf].rolling(50).apply(lambda x: np.exp(np.
    df_etf[f"Ret25_{etf}"] = df_etf[etf].rolling(25).apply(lambda x: np.exp(np.s

    del df_etf[etf]  # Remove the original ETF column to keep only features
    processed_etfs.append(df_etf)
```

```
# Merge all processed DataFrames
df_combined = pd.concat(processed_etfs, axis=1)

# Drop rows with NaNs introduced by rolling and shifting
df_combined = df_combined.dropna()
```

In [80]:
```
df_combined.head()
```

Out[80]:

Ticker	Date	Ret_10_SPY	Ret_50_SPY	Ret25_SPY	Date	Ret_10_TLT	Ret_50_TLT	Ret2
49	2018-03-15	0.027269	0.023180	-0.030509	2018-03-15	0.007710	-0.041836	-0.0
50	2018-03-16	0.019028	0.013791	-0.027826	2018-03-16	0.012336	-0.049806	-0.0
51	2018-03-19	-0.006246	-0.004124	-0.027764	2018-03-19	0.011861	-0.052669	-0.0
52	2018-03-20	-0.007073	-0.009034	-0.027016	2018-03-20	0.006856	-0.053775	-0.0
53	2018-03-21	-0.008615	-0.012741	-0.015235	2018-03-21	0.009067	-0.052137	-0.0

In [81]:
```
# Separate features and targets
feature_columns = [col for col in df_combined.columns if "Ret25" not in col and
target_columns = [col for col in df_combined.columns if "Ret25" in col]

X = df_combined[feature_columns].values.astype("float32")
y = df_combined[target_columns].values.astype("float32")

# Print shapes to verify
print("Features shape:", X.shape)
print("Targets shape:", y.shape)
```

```
Features shape: (1183, 10)
Targets shape: (1183, 5)
```

In [82]:
```
val_split = 0.2
train_split = 0.70
train_size = int(len(df_combined) * train_split)
val_size = int(train_size * val_split)
test_size = int(len(df_combined) - train_size)

window_size = 30

ts = test_size
split_time = len(df_combined) - ts
test_time = df_combined.iloc[split_time + window_size :, 0:1].values


y_train_set = y[:split_time]
y_test_set = y[split_time:]

X_train_set = X[:split_time]
X_test_set = X[split_time:]
```

```
n_features = X_train_set.shape[1]
```

In [83]:
```
scaler_input = MinMaxScaler(feature_range=(-1, 1))
scaler_input.fit(X_train_set)
X_train_set_scaled = scaler_input.transform(X_train_set)
X_test_set_scaled = scaler_input.transform(X_test_set)

mean_ret = np.mean(y_train_set)

scaler_output = MinMaxScaler(feature_range=(-1, 1))
y_train_set_scaled = scaler_output.fit_transform(y_train_set)
y_test_set_scaled = scaler_output.transform(y_test_set)
```

In [84]:
```
# Prepare training data with sliding windows (for scaled targets)
X_train = []
y_train = []

for i in range(window_size, len(X_train_set_scaled)):
    # Features: Last 'window_size' observations
    X_train.append(X_train_set_scaled[i - window_size:i, :])
    # Targets: Scaled multi-output targets for the next time step
    y_train.append(y_train_set_scaled[i])

X_train, y_train = np.array(X_train), np.array(y_train)

# Prepare test data with sliding windows (for scaled targets)
X_test = []
y_test = []

for i in range(window_size, len(X_test_set_scaled)):
    # Features: Last 'window_size' observations
    X_test.append(X_test_set_scaled[i - window_size:i, :])
    # Targets: Scaled multi-output targets for the next time step
    y_test.append(y_test_set_scaled[i])

X_test, y_test = np.array(X_test), np.array(y_test)

# Verify shapes
print("Training data shapes -> X_train:", X_train.shape, ", y_train:", y_train.s
print("Test data shapes -> X_test:", X_test.shape, ", y_test:", y_test.shape)
```

```
Training data shapes -> X_train: (798, 30, 10) , y_train: (798, 5)
Test data shapes -> X_test: (325, 30, 10) , y_test: (325, 5)
```

In [85]:
```
# Set the random seed for reproducibility
SEED = 4321
units_lstm = 50
n_dropout = 0.2
act_fun = "relu"
n_outputs = 5  # Number of outputs (one for each ETF)

# Build the model
model = Sequential()

# Bidirectional LSTM layers with regularization
model.add(
    Bidirectional(
        LSTM(
            units=units_lstm,
```

```
            return_sequences=True,
            activation="tanh",
            input_shape=(X_train.shape[1], X_train.shape[2]),
            kernel_regularizer=l2(0.01),
        )
    )
)
model.add(
    Bidirectional(
        LSTM(
            units=units_lstm,
            return_sequences=True,
            activation="tanh",
            kernel_regularizer=l2(0.01),
        )
    )
)
model.add(Dropout(n_dropout, seed=SEED))

model.add(
    Bidirectional(
        LSTM(
            units=units_lstm,
            return_sequences=True,
            activation="tanh",
            kernel_regularizer=l2(0.01),
        )
    )
)
model.add(Dropout(n_dropout, seed=SEED))

model.add(
    Bidirectional(
        LSTM(
            units=units_lstm,
            return_sequences=True,
            activation="tanh",
            kernel_regularizer=l2(0.01),
        )
    )
)
model.add(Dropout(n_dropout, seed=SEED))

model.add(
    Bidirectional(
        LSTM(
            units=units_lstm,
            return_sequences=False,
            activation="tanh",
            kernel_regularizer=l2(0.01),
        )
    )
)
model.add(Dropout(n_dropout, seed=SEED))

# Dense layers to reduce dimensionality before output
model.add(Dense(units=20, activation=act_fun))
model.add(Dropout(n_dropout, seed=SEED))

model.add(Dense(units=10, activation=act_fun))
```

```
model.add(Dropout(n_dropout, seed=SEED))

# Output layer with 5 units (one for each ETF)
model.add(Dense(units=n_outputs, activation="linear"))

# Compile the model
hp_lr = 1e-4  # Learning rate
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=hp_lr), loss="mean_squared_
)

# Early stopping callback
es = EarlyStopping(
    monitor="val_loss", mode="min", verbose=1, patience=5, restore_best_weights=
)
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/rnn.py:204: UserWarn
ing: 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 m
odel instead.
  super().__init__(**kwargs)
```

In [86]:
```
# Train the model
history = model.fit(
    X_train,
    y_train,
    epochs=100,
    batch_size=64,
    validation_split=val_split,
    callbacks=[es],
    verbose=2
)

# Evaluate the model on the test set
test_loss = model.evaluate(X_test, y_test, verbose=1)
print(f"Test Loss: {test_loss}")
```

```
Epoch 1/100
10/10 - 11s - 1s/step - loss: 11.1294 - val_loss: 11.0456
Epoch 2/100
10/10 - 0s - 30ms/step - loss: 10.8948 - val_loss: 10.8087
Epoch 3/100
10/10 - 0s - 29ms/step - loss: 10.6643 - val_loss: 10.5715
Epoch 4/100
10/10 - 0s - 33ms/step - loss: 10.4388 - val_loss: 10.3406
Epoch 5/100
10/10 - 0s - 31ms/step - loss: 10.2165 - val_loss: 10.1165
Epoch 6/100
10/10 - 0s - 30ms/step - loss: 10.0010 - val_loss: 9.8995
Epoch 7/100
10/10 - 0s - 29ms/step - loss: 9.7902 - val_loss: 9.6891
Epoch 8/100
10/10 - 0s - 30ms/step - loss: 9.5817 - val_loss: 9.4815
Epoch 9/100
10/10 - 0s - 30ms/step - loss: 9.3768 - val_loss: 9.2794
Epoch 10/100
10/10 - 0s - 29ms/step - loss: 9.1788 - val_loss: 9.0819
Epoch 11/100
10/10 - 0s - 42ms/step - loss: 8.9861 - val_loss: 8.8923
Epoch 12/100
10/10 - 0s - 42ms/step - loss: 8.7964 - val_loss: 8.7053
Epoch 13/100
10/10 - 0s - 40ms/step - loss: 8.6113 - val_loss: 8.5228
Epoch 14/100
10/10 - 0s - 44ms/step - loss: 8.4311 - val_loss: 8.3443
Epoch 15/100
10/10 - 1s - 64ms/step - loss: 8.2498 - val_loss: 8.1647
Epoch 16/100
10/10 - 1s - 51ms/step - loss: 8.0734 - val_loss: 7.9877
Epoch 17/100
10/10 - 1s - 61ms/step - loss: 7.9040 - val_loss: 7.8239
Epoch 18/100
10/10 - 1s - 61ms/step - loss: 7.7353 - val_loss: 7.6603
Epoch 19/100
10/10 - 0s - 29ms/step - loss: 7.5710 - val_loss: 7.4940
Epoch 20/100
10/10 - 0s - 31ms/step - loss: 7.4069 - val_loss: 7.3281
Epoch 21/100
10/10 - 0s - 29ms/step - loss: 7.2476 - val_loss: 7.1816
Epoch 22/100
10/10 - 0s - 29ms/step - loss: 7.0938 - val_loss: 7.0245
Epoch 23/100
10/10 - 0s - 30ms/step - loss: 6.9404 - val_loss: 6.8759
Epoch 24/100
10/10 - 0s - 32ms/step - loss: 6.7914 - val_loss: 6.7289
Epoch 25/100
10/10 - 1s - 63ms/step - loss: 6.6450 - val_loss: 6.5884
Epoch 26/100
10/10 - 1s - 59ms/step - loss: 6.5058 - val_loss: 6.4490
Epoch 27/100
10/10 - 0s - 30ms/step - loss: 6.3608 - val_loss: 6.3077
Epoch 28/100
10/10 - 0s - 32ms/step - loss: 6.2239 - val_loss: 6.1760
Epoch 29/100
10/10 - 0s - 32ms/step - loss: 6.0901 - val_loss: 6.0380
Epoch 30/100
10/10 - 1s - 61ms/step - loss: 5.9565 - val_loss: 5.9122
```

```
Epoch 31/100
10/10 - 0s - 29ms/step - loss: 5.8278 - val_loss: 5.7865
Epoch 32/100
10/10 - 0s - 34ms/step - loss: 5.6975 - val_loss: 5.6562
Epoch 33/100
10/10 - 1s - 60ms/step - loss: 5.5764 - val_loss: 5.5375
Epoch 34/100
10/10 - 0s - 28ms/step - loss: 5.4527 - val_loss: 5.4228
Epoch 35/100
10/10 - 0s - 31ms/step - loss: 5.3337 - val_loss: 5.3010
Epoch 36/100
10/10 - 0s - 35ms/step - loss: 5.2155 - val_loss: 5.1885
Epoch 37/100
10/10 - 1s - 56ms/step - loss: 5.1000 - val_loss: 5.0755
Epoch 38/100
10/10 - 0s - 32ms/step - loss: 4.9885 - val_loss: 4.9581
Epoch 39/100
10/10 - 0s - 29ms/step - loss: 4.8761 - val_loss: 4.8562
Epoch 40/100
10/10 - 0s - 29ms/step - loss: 4.7690 - val_loss: 4.7584
Epoch 41/100
10/10 - 0s - 29ms/step - loss: 4.6644 - val_loss: 4.6355
Epoch 42/100
10/10 - 0s - 43ms/step - loss: 4.5586 - val_loss: 4.5484
Epoch 43/100
10/10 - 1s - 61ms/step - loss: 4.4563 - val_loss: 4.4422
Epoch 44/100
10/10 - 0s - 46ms/step - loss: 4.3569 - val_loss: 4.3390
Epoch 45/100
10/10 - 1s - 64ms/step - loss: 4.2567 - val_loss: 4.2518
Epoch 46/100
10/10 - 0s - 50ms/step - loss: 4.1651 - val_loss: 4.1516
Epoch 47/100
10/10 - 1s - 62ms/step - loss: 4.0704 - val_loss: 4.0607
Epoch 48/100
10/10 - 1s - 58ms/step - loss: 3.9776 - val_loss: 3.9729
Epoch 49/100
10/10 - 0s - 31ms/step - loss: 3.8862 - val_loss: 3.8821
Epoch 50/100
10/10 - 0s - 30ms/step - loss: 3.7994 - val_loss: 3.7957
Epoch 51/100
10/10 - 0s - 28ms/step - loss: 3.7124 - val_loss: 3.7026
Epoch 52/100
10/10 - 0s - 29ms/step - loss: 3.6277 - val_loss: 3.6337
Epoch 53/100
10/10 - 0s - 30ms/step - loss: 3.5455 - val_loss: 3.5482
Epoch 54/100
10/10 - 0s - 30ms/step - loss: 3.4652 - val_loss: 3.4797
Epoch 55/100
10/10 - 0s - 29ms/step - loss: 3.3846 - val_loss: 3.4027
Epoch 56/100
10/10 - 0s - 31ms/step - loss: 3.3040 - val_loss: 3.3216
Epoch 57/100
10/10 - 0s - 30ms/step - loss: 3.2300 - val_loss: 3.2471
Epoch 58/100
10/10 - 0s - 29ms/step - loss: 3.1542 - val_loss: 3.1784
Epoch 59/100
10/10 - 0s - 30ms/step - loss: 3.0805 - val_loss: 3.1151
Epoch 60/100
10/10 - 0s - 30ms/step - loss: 3.0095 - val_loss: 3.0349
```

```
Epoch 61/100
10/10 - 0s - 29ms/step - loss: 2.9395 - val_loss: 2.9908
Epoch 62/100
10/10 - 0s - 30ms/step - loss: 2.8714 - val_loss: 2.9138
Epoch 63/100
10/10 - 0s - 29ms/step - loss: 2.8023 - val_loss: 2.8529
Epoch 64/100
10/10 - 1s - 63ms/step - loss: 2.7395 - val_loss: 2.8087
Epoch 65/100
10/10 - 1s - 62ms/step - loss: 2.6733 - val_loss: 2.7247
Epoch 66/100
10/10 - 0s - 29ms/step - loss: 2.6131 - val_loss: 2.6635
Epoch 67/100
10/10 - 0s - 30ms/step - loss: 2.5517 - val_loss: 2.6012
Epoch 68/100
10/10 - 0s - 30ms/step - loss: 2.4876 - val_loss: 2.5391
Epoch 69/100
10/10 - 0s - 29ms/step - loss: 2.4297 - val_loss: 2.4690
Epoch 70/100
10/10 - 0s - 31ms/step - loss: 2.3731 - val_loss: 2.4348
Epoch 71/100
10/10 - 1s - 61ms/step - loss: 2.3158 - val_loss: 2.3658
Epoch 72/100
10/10 - 0s - 30ms/step - loss: 2.2620 - val_loss: 2.3248
Epoch 73/100
10/10 - 0s - 29ms/step - loss: 2.2078 - val_loss: 2.2446
Epoch 74/100
10/10 - 0s - 43ms/step - loss: 2.1532 - val_loss: 2.2073
Epoch 75/100
10/10 - 0s - 43ms/step - loss: 2.1012 - val_loss: 2.1665
Epoch 76/100
10/10 - 0s - 43ms/step - loss: 2.0495 - val_loss: 2.0989
Epoch 77/100
10/10 - 0s - 44ms/step - loss: 2.0017 - val_loss: 2.0713
Epoch 78/100
10/10 - 0s - 46ms/step - loss: 1.9551 - val_loss: 2.0158
Epoch 79/100
10/10 - 0s - 47ms/step - loss: 1.9068 - val_loss: 1.9832
Epoch 80/100
10/10 - 0s - 46ms/step - loss: 1.8606 - val_loss: 1.9298
Epoch 81/100
10/10 - 0s - 33ms/step - loss: 1.8152 - val_loss: 1.8908
Epoch 82/100
10/10 - 1s - 58ms/step - loss: 1.7709 - val_loss: 1.8652
Epoch 83/100
10/10 - 0s - 31ms/step - loss: 1.7263 - val_loss: 1.8042
Epoch 84/100
10/10 - 0s - 34ms/step - loss: 1.6851 - val_loss: 1.7298
Epoch 85/100
10/10 - 1s - 58ms/step - loss: 1.6436 - val_loss: 1.6968
Epoch 86/100
10/10 - 0s - 30ms/step - loss: 1.6039 - val_loss: 1.6610
Epoch 87/100
10/10 - 0s - 31ms/step - loss: 1.5630 - val_loss: 1.6404
Epoch 88/100
10/10 - 0s - 29ms/step - loss: 1.5273 - val_loss: 1.5951
Epoch 89/100
10/10 - 0s - 32ms/step - loss: 1.4862 - val_loss: 1.5585
Epoch 90/100
10/10 - 0s - 31ms/step - loss: 1.4508 - val_loss: 1.5110
```

```
Epoch 91/100
10/10 - 0s - 30ms/step - loss: 1.4150 - val_loss: 1.4760
Epoch 92/100
10/10 - 0s - 32ms/step - loss: 1.3802 - val_loss: 1.4470
Epoch 93/100
10/10 - 1s - 61ms/step - loss: 1.3470 - val_loss: 1.4256
Epoch 94/100
10/10 - 0s - 32ms/step - loss: 1.3127 - val_loss: 1.3665
Epoch 95/100
10/10 - 1s - 61ms/step - loss: 1.2793 - val_loss: 1.3417
Epoch 96/100
10/10 - 0s - 31ms/step - loss: 1.2498 - val_loss: 1.3197
Epoch 97/100
10/10 - 0s - 31ms/step - loss: 1.2169 - val_loss: 1.2795
Epoch 98/100
10/10 - 0s - 31ms/step - loss: 1.1860 - val_loss: 1.2280
Epoch 99/100
10/10 - 1s - 62ms/step - loss: 1.1589 - val_loss: 1.2118
Epoch 100/100
10/10 - 0s - 32ms/step - loss: 1.1292 - val_loss: 1.2182
Restoring model weights from the end of the best epoch: 99.
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 1.2311
Test Loss: 1.2597569227218628
```

In [87]:
```
# Model summary
model.summary()
```

Model: "sequential_5"

Layer (type)	Output Shape	
bidirectional (Bidirectional)	(None, 30, 100)	
bidirectional_1 (Bidirectional)	(None, 30, 100)	
dropout_20 (Dropout)	(None, 30, 100)	
bidirectional_2 (Bidirectional)	(None, 30, 100)	
dropout_21 (Dropout)	(None, 30, 100)	
bidirectional_3 (Bidirectional)	(None, 30, 100)	
dropout_22 (Dropout)	(None, 30, 100)	
bidirectional_4 (Bidirectional)	(None, 100)	
dropout_23 (Dropout)	(None, 100)	
dense_10 (Dense)	(None, 20)	
dropout_24 (Dropout)	(None, 20)	
dense_11 (Dense)	(None, 10)	
dropout_25 (Dropout)	(None, 10)	
dense_12 (Dense)	(None, 5)	

Total params: 804,857 (3.07 MB)

Trainable params: 268,285 (1.02 MB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 536,572 (2.05 MB)

In [88]:
```
# Predict on test data
predictions = model.predict(X_test)

# Inverse transform predictions and true values if they are scaled
predictions = scaler_output.inverse_transform(predictions)  # Shape: (num_sample
true_values = scaler_output.inverse_transform(y_test)       # Shape: (num_sample

# Compute R² for each output (ETF)
def R2_campbell_multi(y_true, y_pred, mean_ret):
    r2_scores = []
    for i in range(y_true.shape[1]):  # Iterate over each ETF
        y_test_i = y_true[:, i]
        y_pred_i = y_pred[:, i]
        sse = sum((y_test_i - y_pred_i) ** 2)  # Sum of squared errors
        tse = sum((y_test_i - mean_ret[i]) ** 2)  # Total sum of squares
        r2_score = 1 - (sse / tse)
        r2_scores.append(r2_score)
    return r2_scores

# Compute mean return for each ETF (for the R² calculation)
mean_ret_multi = np.mean(true_values, axis=0)

# Compute R² scores
r2_scores = R2_campbell_multi(true_values, predictions, mean_ret_multi)
for i, r2 in enumerate(r2_scores):
    print(f"Out-of-sample R-squared for ETF {i + 1}: {r2}")
```

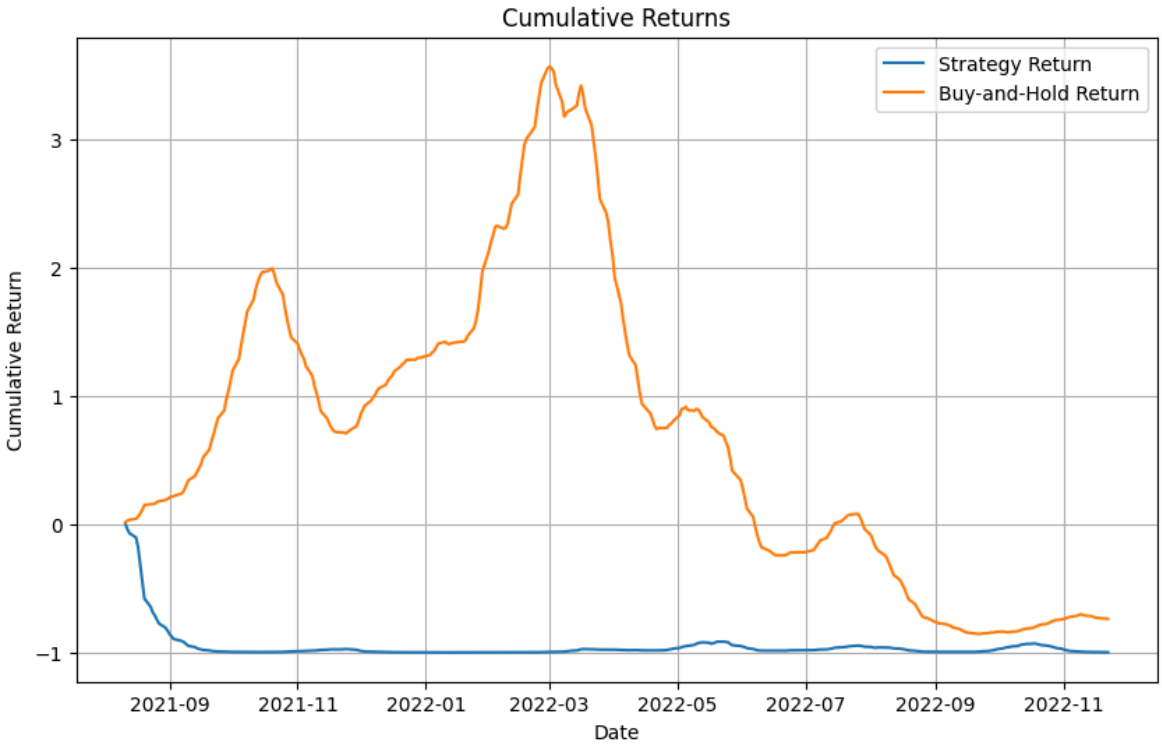
```
11/11 ━━━━━━━━━━━━━━━━━━━━ 1s 76ms/step
Out-of-sample R-squared for ETF 1: -0.264696308032323
Out-of-sample R-squared for ETF 2: -0.39265593732273096
Out-of-sample R-squared for ETF 3: -1.155273637298654
Out-of-sample R-squared for ETF 4: 0.037505848765581495
Out-of-sample R-squared for ETF 5: -0.6893508460948767
```

In [89]:
```
# Optional: Create a DataFrame for visualization
df_predictions2 = pd.DataFrame(
    {
        "Date": test_time.flatten(),  # Ensure this is aligned with your test se
        "ETF_1_Pred": predictions[:, 0],
        "ETF_1_True": true_values[:, 0],
        "ETF_2_Pred": predictions[:, 1],
        "ETF_2_True": true_values[:, 1],
        "ETF_3_Pred": predictions[:, 2],
        "ETF_3_True": true_values[:, 2],
        "ETF_4_Pred": predictions[:, 3],
        "ETF_4_True": true_values[:, 3],
        "ETF_5_Pred": predictions[:, 4],
        "ETF_5_True": true_values[:, 4],
    }
)
print(df_predictions2.head())
```

	Date	ETF_1_Pred	ETF_1_True	ETF_2_Pred	ETF_2_True	ETF_3_Pred	ETF_3_True	ETF_4_Pred	ETF_4_True	ETF_5_Pred	ETF_5_True
0	2021-08-11	0.017327	0.007639	0.001673	0.023280	0.001475	0.000232	0.009400	0.000183	0.020415	0.053344
1	2021-08-12	0.016810	-0.008335	0.002105	0.020036	0.001562	0.000000	0.010070	-0.001646	0.018952	0.048761
2	2021-08-13	0.016707	-0.026642	0.002184	0.016627	0.001579	0.000116	0.010216	-0.008714	0.018658	0.048622
3	2021-08-16	0.016719	-0.029845	0.002165	0.013297	0.001577	0.000116	0.010216	-0.007057	0.018688	0.057190
4	2021-08-17	0.017084	-0.013918	0.001847	0.019751	0.001515	-0.000232	0.009771	-0.009283	0.019713	0.087459

In [90]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Step 1: Create DataFrame from the given data
df_predictions2 = pd.DataFrame(
    {
        "Date": test_time.flatten(),  # Ensure this matches your test set timeli
        "ETF_1_Pred": predictions[:, 0],
        "ETF_1_True": true_values[:, 0],
        "ETF_2_Pred": predictions[:, 1],
        "ETF_2_True": true_values[:, 1],
        "ETF_3_Pred": predictions[:, 2],
        "ETF_3_True": true_values[:, 2],
        "ETF_4_Pred": predictions[:, 3],
        "ETF_4_True": true_values[:, 3],
        "ETF_5_Pred": predictions[:, 4],
        "ETF_5_True": true_values[:, 4],
    }
)

# Step 2: Normalize Predictions (if required)
for i in range(1, 6):  # ETFs 1 to 5
    df_predictions2[f"ETF_{i}_NormPred"] = (
        df_predictions2[f"ETF_{i}_Pred"] - df_predictions2[f"ETF_{i}_Pred"].mean
    ) / df_predictions2[f"ETF_{i}_Pred"].std()

# Step 3: Define a 25-day rebalancing strategy
rebalance_period = 25
positions = []

etfs = ["ETF_1", "ETF_2", "ETF_3", "ETF_4", "ETF_5"]
for i in range(0, len(df_predictions2), rebalance_period):
    # Select predictions for the rebalancing window
    window = df_predictions2.iloc[i:i + rebalance_period]

    # Calculate average normalized predictions per ETF over the window
    aggregated_predictions = {etf: window[f"{etf}_NormPred"].mean() for etf in e

    # Rank ETFs based on predictions
    ranked_etfs = sorted(aggregated_predictions.items(), key=lambda x: x[1], rev

    # Determine long and short positions
    long_etfs = [ranked_etfs[j][0] for j in range(2)]  # Top 2 ETFs
    short_etfs = [ranked_etfs[j][0] for j in range(-2, 0)]  # Bottom 2 ETFs
```

```
    # Assign positions for the window
    for day in range(len(window)):
        position = {etf: 1 if etf in long_etfs else (-1 if etf in short_etfs els
        positions.append(position)

# Step 4: Add positions to the DataFrame
positions_df = pd.DataFrame(positions, index=df_predictions2.index[:len(position
for etf in etfs:
    df_predictions2[f"{etf}_Position"] = positions_df[etf]

# Step 5: Calculate strategy returns
df_predictions2["Strat_ret"] = 0
for etf in etfs:
    df_predictions2["Strat_ret"] += (
        df_predictions2[f"{etf}_Position"].shift(1).fillna(0) * df_predictions2[
    )

# Step 6: Calculate cumulative returns
df_predictions2["CumRet"] = (1 + df_predictions2["Strat_ret"]).cumprod() - 1
df_predictions2["bhRet"] = (1 + df_predictions2[[f"{etf}_True" for etf in etfs]]

# Step 7: Plot cumulative returns
plt.figure(figsize=(10, 6))
ax = plt.gca()
df_predictions2.plot(x="Date", y="CumRet", label="Strategy Return", ax=ax)
df_predictions2.plot(x="Date", y="bhRet", label="Buy-and-Hold Return", ax=ax)
plt.title("Cumulative Returns")
plt.xlabel("Date")
plt.ylabel("Cumulative Return")
plt.legend()
plt.grid(True)
plt.show()

# Step 8: Debugging Outputs
print("\n===== Debugging Outputs =====")
print("Strategy Returns Summary:")
print(df_predictions2["Strat_ret"].describe())

print("\nBuy-and-Hold Returns Summary:")
print(df_predictions2["bhRet"].describe())

print("\nETF Predictions and True Values Correlation:")
for etf in etfs:
    correlation = df_predictions2[f"{etf}_Pred"].corr(df_predictions2[f"{etf}_Tr
    print(f"Correlation between {etf}_Pred and {etf}_True: {correlation:.2f}")
```




```
===== Debugging Outputs =====
Strategy Returns Summary:
count    325.000000
mean      -0.011963
std        0.129046
min       -0.271715
25%       -0.116313
50%       -0.021736
75%        0.076067
max        0.317077
Name: Strat_ret, dtype: float64

Buy-and-Hold Returns Summary:
count    325.000000
mean       0.689635
std        1.187499
min       -0.855609
25%       -0.224296
50%        0.719548
75%        1.404324
max        3.571723
Name: bhRet, dtype: float64

ETF Predictions and True Values Correlation:
Correlation between ETF_1_Pred and ETF_1_True: 0.00
Correlation between ETF_2_Pred and ETF_2_True: -0.19
Correlation between ETF_3_Pred and ETF_3_True: -0.20
Correlation between ETF_4_Pred and ETF_4_True: 0.22
Correlation between ETF_5_Pred and ETF_5_True: -0.51
```

In [91]:
```
df_predictions2.head()
```

Out[91]:

	Date	ETF_1_Pred	ETF_1_True	ETF_2_Pred	ETF_2_True	ETF_3_Pred	ETF_3_True	ETF
0	2021-08-11	0.017327	0.007639	0.001673	0.023280	0.001475	0.000232	0
1	2021-08-12	0.016810	-0.008335	0.002105	0.020036	0.001562	0.000000	0
2	2021-08-13	0.016707	-0.026642	0.002184	0.016627	0.001579	0.000116	0
3	2021-08-16	0.016719	-0.029845	0.002165	0.013297	0.001577	0.000116	0
4	2021-08-17	0.017084	-0.013918	0.001847	0.019751	0.001515	-0.000232	0

5 rows × 24 columns