# Training Pipeline Runner and Analysis

This notebook serves as the interactive interface for a modular Deep Learning project.

**Objectives:**

1. Load the project configurations from the `config.yaml` file.
2. Execute the full training pipeline, which is encapsulated in the `TrainingPipeline` class.
3. Capture the results (trained model, history, test data) into a single variable.
4. Perform interactive analysis on the results to understand the model's performance.

In [1]:
```
import sys
import os

# Get the absolute path of the project's root directory
# os.getcwd() gets the current folder ('/notebooks')
# os.path.join(..., '..') goes one level up to the project root
project_root = os.path.abspath(os.path.join(os.getcwd(), '..'))

# Add the project root to the Python path
if project_root not in sys.path:
    sys.path.append(project_root)

# --- Imports and Setup ---
import yaml
import pandas as pd
import matplotlib.pyplot as plt

# Import CUSTOM training pipeline class
# AQUI É FEITA A PONTE COM O ARQUIVO training_pipeline.py
from src.training_pipeline import TrainingPipeline

# --- Settings ---
# Set pandas to display all columns in a dataframe for better inspection
pd.set_option('display.max_columns', None)
```

```
# --- Load Configuration ---
# Construct the absolute path to the config file using project_root
config_path = os.path.join(project_root, 'configs', 'config.yaml')
# PASSO 1: CHAMA O ARQUIVO DE CONFIG
print(f"Loading configuration from: {config_path}")
with open(config_path, 'r') as file:
    config = yaml.safe_load(file)

print("Configuration loaded successfully.")
```

```
Loading configuration from: C:\Projetos_Python\gld_lstm_strategy\configs\config.yaml
Configuration loaded successfully.
```

In [2]:
```
# Create an instance of the training pipeline, passing the project_root
# PASSO 2: CHAMA O ARQUIVO training_pipeline.py, QUE TEM A CLASSE TrainingPipeline
# ESTA MESMA CLASSE CONTÉM TAMBÉM A FUNÇÃO _load_and_prepare_data, QUE CHAMA O ARQUIVO data_loader.py
# NO ARQUIVO data_loader.py, A CLASSE DataLoader CONTÉM AS FUNÇÕES load_main_data() E load_all_macro_data()
pipeline = TrainingPipeline(config=config, project_root=project_root)

# PASSO 3: CHAMA A FUNÇÃO run_static_test() CONTIDA NA CLASSE TrainingPipeline
# Run the STATIC test pipeline and capture the results
# This method encapsulates the original single train/test split logic
static_results = pipeline.run_static_test()

print("\n\n✅ --- Static Test Pipeline execution finished! --- ✅")
print("All results are now available in the 'static_results' variable.")
```

```
Random seeds set to 2025 for reproducibility.


===== Starting STATIC Test Run for model_type='lstm' =====
===== Step 1: Loading and Preparing Full Dataset =====
--- Loading Main Asset Data ---
Loading GLD data from local cache: C:\Projetos_Python\gld_lstm_strategy\data\gld_data.csv

--- Loading Macroeconomic Data ---
Loading DX-Y.NYB data from local cache: C:\Projetos_Python\gld_lstm_strategy\data\dx-y.nyb_data.csv
Loading ^TNX data from local cache: C:\Projetos_Python\gld_lstm_strategy\data\^tnx_data.csv
Loading ^VIX data from local cache: C:\Projetos_Python\gld_lstm_strategy\data\^vix_data.csv
Loading CL=F data from local cache: C:\Projetos_Python\gld_lstm_strategy\data\cl=f_data.csv
Loading SI=F data from local cache: C:\Projetos_Python\gld_lstm_strategy\data\si=f_data.csv
Loading TIP data from local cache: C:\Projetos_Python\gld_lstm_strategy\data\tip_data.csv
Loading HG=F data from local cache: C:\Projetos_Python\gld_lstm_strategy\data\hg=f_data.csv

===== Starting Feature Engineering Pipeline =====
Step 1: Creating custom OHLCV features...
Step 2: Applying 'All' technical indicator strategy from pandas_ta...
```

```
130it [00:10, 12.70it/s]
```

```
 -> Dropped 2 redundant TA columns.
Step 3: Creating custom interaction and ratio features...
Step 4: Creating lagged and momentum features...
Step 5: Merging macroeconomic features...
 -> Macro features merged and forward-filled.
Step 6: Defining target variable...

Pipeline complete. Dropped 77 rows with NaN values.
Final dataset shape: (2438, 247)
===============================================



--- Splitting data chronologically ---
Train set size: 1761, Validation set size: 311, Test set size: 366

--- Running Feature Selection ---
Selected 37 features via BorutaPy.

--- Scaling data ---

--- Preparing data and training LSTM Model ---
Epoch 1/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 17s 280ms/step - accuracy: 0.5030 - loss: 0.7019 - val_accuracy: 0.4180 - val_loss: 0.7305
Epoch 2/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 267ms/step - accuracy: 0.5585 - loss: 0.6879 - val_accuracy: 0.4180 - val_loss: 0.7121
Epoch 3/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 250ms/step - accuracy: 0.5489 - loss: 0.6861 - val_accuracy: 0.6557 - val_loss: 0.6872
Epoch 4/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 252ms/step - accuracy: 0.5691 - loss: 0.6838 - val_accuracy: 0.4180 - val_loss: 0.6974
Epoch 5/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 252ms/step - accuracy: 0.5838 - loss: 0.6729 - val_accuracy: 0.4180 - val_loss: 0.7160
Epoch 6/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 251ms/step - accuracy: 0.6016 - loss: 0.6685 - val_accuracy: 0.4180 - val_loss: 0.7056
Epoch 7/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 257ms/step - accuracy: 0.6108 - loss: 0.6620 - val_accuracy: 0.4590 - val_loss: 0.7094
Epoch 8/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 14s 286ms/step - accuracy: 0.6115 - loss: 0.6596 - val_accuracy: 0.4426 - val_loss: 0.7162
Epoch 9/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 15s 306ms/step - accuracy: 0.6270 - loss: 0.6525 - val_accuracy: 0.4918 - val_loss: 0.7050
Epoch 10/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 259ms/step - accuracy: 0.6184 - loss: 0.6473 - val_accuracy: 0.5082 - val_loss: 0.7017
```

```
Epoch 11/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 14s 274ms/step - accuracy: 0.6291 - loss: 0.6392 - val_accuracy: 0.5164 - val_loss: 0.7020
Epoch 12/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 254ms/step - accuracy: 0.6591 - loss: 0.6368 - val_accuracy: 0.5902 - val_loss: 0.6677
Epoch 13/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 250ms/step - accuracy: 0.6671 - loss: 0.6273 - val_accuracy: 0.6311 - val_loss: 0.6537
Epoch 14/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 256ms/step - accuracy: 0.6855 - loss: 0.6150 - val_accuracy: 0.5984 - val_loss: 0.6434
Epoch 15/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 250ms/step - accuracy: 0.6800 - loss: 0.6257 - val_accuracy: 0.5410 - val_loss: 0.6906
Epoch 16/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 254ms/step - accuracy: 0.6736 - loss: 0.6041 - val_accuracy: 0.5738 - val_loss: 0.6412
Epoch 17/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 257ms/step - accuracy: 0.7016 - loss: 0.5977 - val_accuracy: 0.6721 - val_loss: 0.6043
Epoch 18/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 251ms/step - accuracy: 0.7139 - loss: 0.5822 - val_accuracy: 0.6230 - val_loss: 0.6172
Epoch 19/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 255ms/step - accuracy: 0.7208 - loss: 0.5667 - val_accuracy: 0.6721 - val_loss: 0.6169
Epoch 20/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 263ms/step - accuracy: 0.7158 - loss: 0.5691 - val_accuracy: 0.6803 - val_loss: 0.6158
Epoch 21/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 268ms/step - accuracy: 0.7198 - loss: 0.5646 - val_accuracy: 0.6885 - val_loss: 0.5929
Epoch 22/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 12s 249ms/step - accuracy: 0.7396 - loss: 0.5445 - val_accuracy: 0.6148 - val_loss: 0.6275
Epoch 23/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 256ms/step - accuracy: 0.7159 - loss: 0.5560 - val_accuracy: 0.6803 - val_loss: 0.6010
Epoch 24/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 251ms/step - accuracy: 0.7295 - loss: 0.5453 - val_accuracy: 0.7049 - val_loss: 0.5656
Epoch 25/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 255ms/step - accuracy: 0.7227 - loss: 0.5308 - val_accuracy: 0.6803 - val_loss: 0.6030
Epoch 26/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 252ms/step - accuracy: 0.7489 - loss: 0.5219 - val_accuracy: 0.7213 - val_loss: 0.5546
Epoch 27/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 259ms/step - accuracy: 0.7563 - loss: 0.5015 - val_accuracy: 0.6885 - val_loss: 0.5792
Epoch 28/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 250ms/step - accuracy: 0.7274 - loss: 0.5211 - val_accuracy: 0.7131 - val_loss: 0.5323
Epoch 29/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 14s 273ms/step - accuracy: 0.7863 - loss: 0.4833 - val_accuracy: 0.6721 - val_loss: 0.5674
Epoch 30/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 252ms/step - accuracy: 0.7763 - loss: 0.4730 - val_accuracy: 0.7049 - val_loss: 0.5492
Epoch 31/100
```

```
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 251ms/step - accuracy: 0.7885 - loss: 0.4418 - val_accuracy: 0.7377 - val_loss: 0.5541
Epoch 32/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 14s 271ms/step - accuracy: 0.7878 - loss: 0.4426 - val_accuracy: 0.7213 - val_loss: 0.5174
Epoch 33/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 14s 284ms/step - accuracy: 0.7949 - loss: 0.4380 - val_accuracy: 0.7787 - val_loss: 0.4662
Epoch 34/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 19s 253ms/step - accuracy: 0.8123 - loss: 0.4249 - val_accuracy: 0.7295 - val_loss: 0.5130
Epoch 35/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 253ms/step - accuracy: 0.8041 - loss: 0.4283 - val_accuracy: 0.7131 - val_loss: 0.5576
Epoch 36/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 252ms/step - accuracy: 0.8196 - loss: 0.4213 - val_accuracy: 0.7459 - val_loss: 0.5098
Epoch 37/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 255ms/step - accuracy: 0.8138 - loss: 0.4153 - val_accuracy: 0.6967 - val_loss: 0.5521
Epoch 38/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 12s 245ms/step - accuracy: 0.8091 - loss: 0.4115 - val_accuracy: 0.7131 - val_loss: 0.5675
Epoch 39/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 252ms/step - accuracy: 0.8183 - loss: 0.4031 - val_accuracy: 0.7459 - val_loss: 0.5175
Epoch 40/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 12s 248ms/step - accuracy: 0.8211 - loss: 0.4185 - val_accuracy: 0.7049 - val_loss: 0.5543
Epoch 41/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 255ms/step - accuracy: 0.8385 - loss: 0.3914 - val_accuracy: 0.7295 - val_loss: 0.5626
Epoch 42/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 253ms/step - accuracy: 0.8278 - loss: 0.3902 - val_accuracy: 0.7295 - val_loss: 0.5882
Epoch 43/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 259ms/step - accuracy: 0.8073 - loss: 0.3991 - val_accuracy: 0.7295 - val_loss: 0.6361
Epoch 44/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 253ms/step - accuracy: 0.8337 - loss: 0.3714 - val_accuracy: 0.7459 - val_loss: 0.5831
Epoch 45/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 251ms/step - accuracy: 0.8271 - loss: 0.3825 - val_accuracy: 0.7459 - val_loss: 0.5808
Epoch 46/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 262ms/step - accuracy: 0.8444 - loss: 0.3550 - val_accuracy: 0.7705 - val_loss: 0.6124
Epoch 47/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 13s 252ms/step - accuracy: 0.8418 - loss: 0.3579 - val_accuracy: 0.7705 - val_loss: 0.6309
Epoch 48/100
50/50 ━━━━━━━━━━━━━━━━━━━━ 12s 248ms/step - accuracy: 0.8472 - loss: 0.3670 - val_accuracy: 0.6967 - val_loss: 0.6306
6/6 ━━━━━━━━━━━━━━━━━━━━ 1s 137ms/step

--- Evaluating Model on Unseen Test Data ---
```

```
✅ --- Static Test Pipeline execution finished! --- ✅
All results are now available in the 'static_results' variable.
```

In [3]:
```python
# Now, let's start analyzing the results.
# First, let's see what keys are in our results dictionary to know what we can inspect.
print("Objects returned by the pipeline:")
print(list(static_results.keys()))
```

```
Objects returned by the pipeline:
['model', 'history', 'selected_features', 'scaler', 'true_labels', 'pred_probas', 'processed_data_for_backtest']
```

In [4]:
```python
# --- 1. Trained Model Analysis ---
print("Extracting the trained model...")
trained_model = static_results['model']

# Print the model's summary to review its architecture
print("\nModel Summary:")
trained_model.summary()
```

```
Extracting the trained model...

Model Summary:
```

**Model: "sequential"**

| Layer (type) | Output Shape | Param # |
|---|---|---|
| lstm (LSTM) | (None, 189, 150) | 112,800 |
| dropout (Dropout) | (None, 189, 150) | 0 |
| lstm_1 (LSTM) | (None, 75) | 67,800 |
| dropout_1 (Dropout) | (None, 75) | 0 |
| dense (Dense) | (None, 1) | 76 |

**Total params:** 542,030 (2.07 MB)

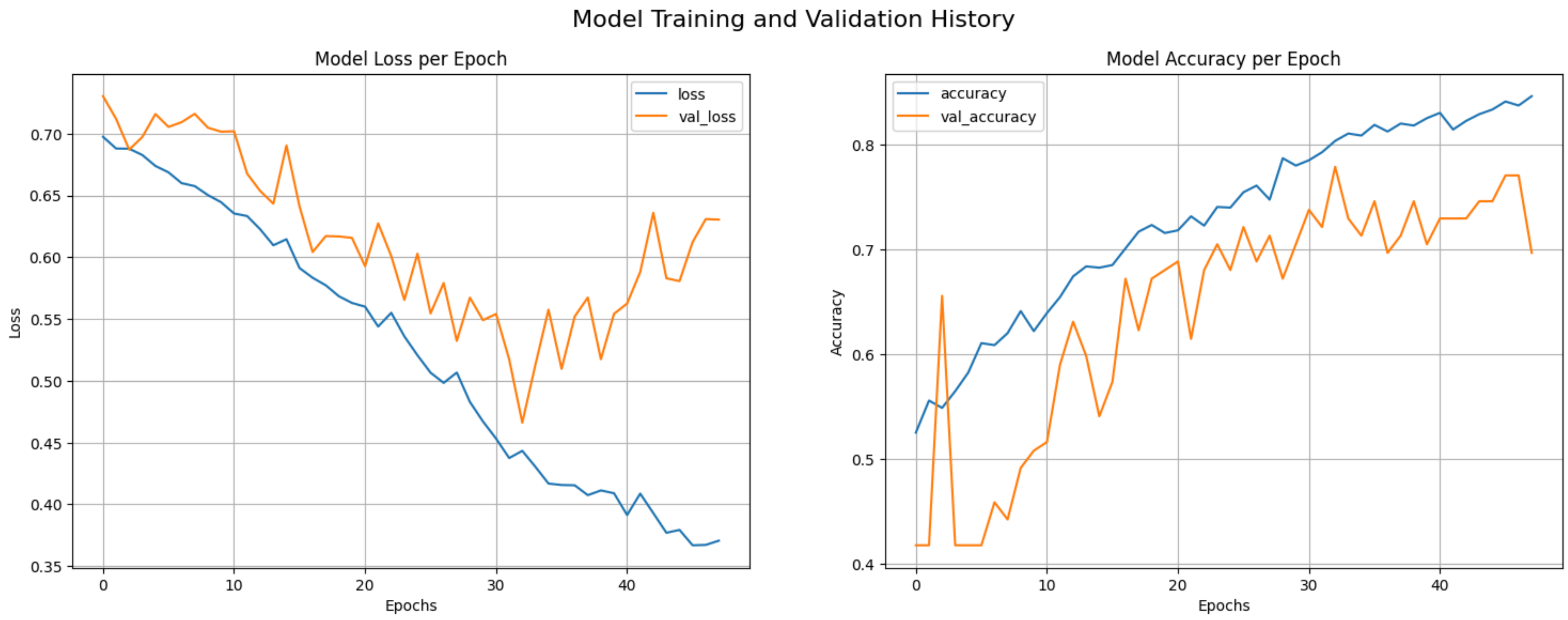
**Trainable params:** 180,676 (705.77 KB)

**Non-trainable params:** 0 (0.00 B)

**Optimizer params:** 361,354 (1.38 MB)

In [5]:
```
# --- 2. Training History Analysis ---
print("Analyzing the training history to check for overfitting...")

# Convert the history object to a pandas DataFrame for easy plotting
history_df = pd.DataFrame(static_results['history'].history)

# Create a figure with two subplots
fig, ax = plt.subplots(1, 2, figsize=(18, 6))

# Plot training & validation loss
history_df[['loss', 'val_loss']].plot(ax=ax[0], title='Model Loss per Epoch', grid=True)
ax[0].set_xlabel("Epochs")
ax[0].set_ylabel("Loss")

# Plot training & validation accuracy
history_df[['accuracy', 'val_accuracy']].plot(ax=ax[1], title='Model Accuracy per Epoch', grid=True)
ax[1].set_xlabel("Epochs")
ax[1].set_ylabel("Accuracy")

plt.suptitle("Model Training and Validation History", fontsize=16)
plt.show()
```

```
Analyzing the training history to check for overfitting...
```




In [6]:
```python
# --- 3. Selected Features Review ---
print("These are the features that BorutaPy selected during the pipeline run:")

selected_features = static_results['selected_features']
for i, feature in enumerate(selected_features):
    print(f"{i+1:02d}: {feature}")
```

```
These are the features that BorutaPy selected during the pipeline run:
01: close
02: AD
03: ADX_14
04: BIAS_SMA_26
05: AR_26
06: BR_26
07: CCI_14_0.015
08: CMF_20
09: CMO_14
10: DPO_20
11: BULLP_13
12: BEARP_13
13: J_9_3
14: KURT_30
15: KVO_34_55_13
16: KVOs_34_55_13
17: MACDh_12_26_9
18: MASSI_9_25
19: NVI_1
20: PGO_14
21: PVI_1
22: PVO_12_26_9
23: PVOs_12_26_9
24: PVT
25: RVGI_14_4
26: SKEW_30
27: SMIo_5_20_5
28: TOS_STDEVALL_LR
29: TRIX_30_9
30: WILLR_14
31: BBB_20_2.0
32: rsi_14_momentum_5d
33: dxy_close
34: tnx_close
35: vix_close
36: oil_close
37: silver_close
```

In [7]:
```
# --- 4. Professional Backtesting with vectorbt ---
from src.backtester import VectorizedBacktester
```

```
import pandas as pd

print("\n\n--- Preparing data for professional backtest ---")

# The 'static_results' dictionary contains the data we need
processed_data = static_results['processed_data_for_backtest']

# Recreate X_test to get the prices and the correct date index
X = processed_data.drop(columns=[config['TARGET_NAME']])
train_val_size = int(len(X) * (1 - config['TEST_SIZE']))
X_test = X.iloc[train_val_size:]

# Access the prediction probabilities directly from the main results dictionary
y_pred_proba = static_results['pred_probas']

# Convert probabilities to binary signals (0 or 1) using a 0.5 threshold
test_predictions = (y_pred_proba > 0.5).astype(int)

# Create a pandas Series for the signals with the correct date index
signal_dates = X_test.index[config['TIME_STEPS']:]
signals_series = pd.Series(test_predictions, index=signal_dates, name="signal")

# Get the price data for the same period
price_data_for_backtest = X_test.loc[signal_dates]

print(f"Backtest will run on {len(signals_series)} signals from {signals_series.index.min().date()} to {signals_series.index.m

# Instantiate and run the backtester
backtester = VectorizedBacktester(
    price_data=price_data_for_backtest,
    signals=signals_series,
    config=config
)

# Run the backtest with exit signals enabled
portfolio = backtester.run(commission=0.001, slippage=0.001)
```

```
--- Preparing data for professional backtest ---
Backtest will run on 177 signals from 2024-04-18 to 2024-12-30

===== Starting Vectorized Backtest with vectorbt =====

--- Backtest Performance Stats ---
Start                          2024-04-18 00:00:00
End                            2024-12-30 00:00:00
Period                           177 days 00:00:00
Start Value                                  100.0
End Value                                114.64216
Total Return [%]                          14.64216
Benchmark Return [%]                        9.2085
Max Gross Exposure [%]                       100.0
Total Fees Paid                           2.349421
Max Drawdown [%]                          3.349057
Max Drawdown Duration             36 days 00:00:00
Total Trades                                    11
Total Closed Trades                             11
Total Open Trades                                0
Open Trade PnL                                 0.0
Win Rate [%]                             72.727273
Best Trade [%]                            4.984358
Worst Trade [%]                          -0.796262
Avg Winning Trade [%]                     1.972687
Avg Losing Trade [%]                     -0.622551
Avg Winning Trade Duration         5 days 06:00:00
Avg Losing Trade Duration          3 days 16:00:00
Profit Factor                             8.454068
Expectancy                                1.331105
Sharpe Ratio                              2.994728
Calmar Ratio                              9.718887
Omega Ratio                               2.018344
Sortino Ratio                             6.045033
dtype: object

--- Plotting Equity Curve and Drawdowns ---
```

```
===== Backtest Finished =====
```