MACHINE LEARNING & AI

A COLLECTION OF ARTICLES ON AI AND TRADING

Ali AZARY

PyQuantLab

Table of Contents

Preface

Welcome to **"Machine Learning & AI: A Collection of Articles on AI and Trading"**—your gateway to understanding how artificial intelligence and modern machine learning techniques are revolutionizing trading, investing, and financial analysis.

This curated collection brings together my most popular and insightful articles, each exploring a unique facet of AI and quantitative trading, from practical Python implementations to advanced strategies like neural networks, regime detection, and algorithmic Bitcoin trading. Whether you're a quant enthusiast, a professional trader, or simply curious about the intersection of technology and finance, you'll find actionable ideas and real-world code you can use today.

But why stop here? If you enjoy this collection, you'll find even more depth, structure, and ready-to-use strategies in my other books and guides, available now on our website:

https://www.pyquantlab.com/#books

- **Advanced Quantitative Trading**: Master powerful Python-based strategies and backtesting techniques for real-world edge.
- **Backtrader Essentials**: Your fast track to building, testing, and optimizing strategies with the Backtrader library.
- **Practical Financial Machine Learning**: A step-by-step guide for applying cutting-edge ML to finance and trading.
- **The Complete Technical Analysis Guide**: Proven, ready-to-use technical trading systems you can start using right away.
- **Desktop App Development with PyQt5**: Build professional financial and trading apps in Python.
- **Moving Average Convergence**: Discover ten crossover and ribbon strategies for consistent results.

All these resources are designed to help you take your trading and programming to the next level—whether you want to automate your trading, analyze data like a pro, or just get started with Python in finance.

Thank you for reading, and I hope this collection sparks new ideas for your journey into AI-powered trading!

Ali Azary

Algorithmic Bitcoin Trading Strategy using Machine Learning Classification

This tutorial provides a comprehensive guide to developing an algorithmic trading strategy for Bitcoin using machine learning classification techniques. We'll cover everything from fetching real-time Bitcoin data and engineering predictive features to building and evaluating classification models, and finally, backtesting the strategy. This guide is designed to be self-contained, with all necessary Python code and explanations.

1. Introduction: Classification for Trading Signals

Cryptocurrency markets, known for their volatility and 24/7 trading, present unique challenges and opportunities for algorithmic trading. Machine learning, particularly classification, can be employed to predict market movements and generate trading signals (e.g., buy, sell, or hold).

The core idea is to transform the problem of predicting price movements into a classification task. For instance, we can classify the next period's expected price movement into categories like "price will rise" (buy signal) or "price will fall" (sell signal). One powerful aspect of machine learning is **feature engineering**, where we create new, informative features from raw data (like price and volume) to improve model performance. Technical indicators are a common source for such features.

This tutorial will focus on:

- Building a trading strategy based on classifying buy/sell signals.
- Engineering features using common technical indicators.
- Developing a framework to backtest the trading strategy's performance.
- Choosing appropriate evaluation metrics for a trading strategy.

2. Problem Definition: Predicting Buy/Sell Signals

We aim to predict whether the current trading signal for Bitcoin is to **buy (1)** or **sell (0)**. This signal will be determined by comparing short-term and long-term price trends. For example, if a short-term moving average of the price is above a long-term moving average, it might indicate an uptrend (buy signal), and vice-versa.

- **Data:** We'll use historical Bitcoin price data. We will fetch up-to-date data using `yfinance`.
- **Features:** We will create various trend and momentum technical indicators from the price data to serve as input features for our classification model.
- **Target Variable:** A binary signal (1 for buy, 0 for sell) derived from the relationship between short-term and long-term moving averages.

3. Getting Started: Setting Up the Environment

3.1. Python Packages

We'll need several Python libraries:

- **`yfinance`**: For fetching financial data (Bitcoin prices).
- **`pandas`**: For data manipulation and analysis.
- **`numpy`**: For numerical operations.
- **`matplotlib.pyplot`** and **`seaborn`**: For data visualization.
- **`scikit-learn`**: For machine learning tasks, including:
    - `model_selection` (for `train_test_split`, `KFold`, `cross_val_score`, `GridSearchCV`)
    - Various classifiers (e.g., `LogisticRegression`, `DecisionTreeClassifier`, `RandomForestClassifier`)
    - `metrics` (for `accuracy_score`, `confusion_matrix`, `classification_report`)

```
import yfinance as yf
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, KFold, cross_val_score,
GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier, AdaBoostClassifier, ExtraTreesClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
import warnings
warnings.filterwarnings(action='ignore')

# Set a consistent style for plots
plt.style.use('seaborn-v0_8-whitegrid')
pd.set_option('display.width', 100)
```

3.2. Loading the Data

We will fetch Bitcoin (BTC-USD) data using `yfinance`. The original context uses minute-by-minute data; for simplicity and common practice with `yfinance` for daily strategies, we'll fetch daily data. The principles remain the same.

```
ticker = 'BTC-USD'
start_date = '2018-01-01'
end_date = pd.to_datetime('today').strftime('%Y-%m-%d')

try:
    raw_data = yf.download(ticker, start=start_date, end=end_date,
auto_adjust=False, progress=False)
    if raw_data.empty:
        raise ValueError("No data downloaded. Check ticker or date range.")

    dataset = raw_data[['Open', 'High', 'Low', 'Close', 'Volume']].copy()
    dataset.rename(columns={'Volume': 'Volume_(BTC)'}, inplace=True)
    print("Successfully downloaded Bitcoin data.")
except Exception as e:
    print(f"Error downloading data: {e}")
    print("Using a dummy dataset for demonstration purposes.")
    dates = pd.date_range(start='2020-01-01', periods=1000, freq='D')
    data_dummy = {
        'Open': np.random.rand(1000) * 10000 + 30000,
        'High': np.random.rand(1000) * 10000 + 35000,
        'Low': np.random.rand(1000) * 10000 + 25000,
        'Close': np.random.rand(1000) * 10000 + 30000,
        'Volume_(BTC)': np.random.rand(1000) * 100 + 10
    }
    dataset = pd.DataFrame(data_dummy, index=dates)

print("\nDataset shape:", dataset.shape)
dataset.dropna(axis=0, how='all', inplace=True) # Drop rows if all values are
NaN (can happen with yfinance for some dates)
print("Dataset shape after dropping all-NaN rows:", dataset.shape)


# 4. Exploratory Data Analysis (EDA)
print("\nDataset Info:")
dataset.info()
```

4. Exploratory Data Analysis (EDA)

A quick look at the data structure.

```
print("\nDataset Info:")
dataset.info()

print("\nSummary Statistics:")
print(dataset.describe())
```

Visualizing the closing price helps understand its trend and volatility.

```
plt.figure(figsize=(14, 7))
dataset['Close'].plot(grid=True)
plt.title(f'{ticker} Closing Price ({start_date} to {end_date})')
plt.ylabel('Price (USD)')
plt.savefig('bitcoin_closing_price.png')
print("\nSaved Bitcoin closing price plot to bitcoin_closing_price.png")
# plt.show()
plt.close()
```

Bitcoin's price chart typically shows significant volatility and distinct trend periods.

5. Data Preparation

5.1. Data Cleaning

Financial data can have missing values, especially for less liquid assets or specific exchanges. For daily `yfinance` data, NaNs are less common for major assets like BTC-USD but should still be checked. The PDF uses `ffill()` (forward fill) to handle NaNs.

```
print("\nMissing values before cleaning (after initial load):")
print(dataset.isnull().sum())
dataset.fillna(method='ffill', inplace=True)
dataset.fillna(method='bfill', inplace=True)
print("\nMissing values after initial ffill/bfill:")
print(dataset.isnull().sum())
dataset.dropna(inplace=True) # Drop any remaining rows with NaNs, if any
print("Dataset shape after full NaN drop:", dataset.shape)

if dataset.empty:
    print("Dataset is empty after initial cleaning. Exiting.")
    exit()
```

The `Timestamp` column in the original PDF's dataset (minute data) was not useful for modeling and was dropped. For our daily data, the DatetimeIndex is useful and kept.

5.2. Preparing the Target Variable (`signal`)

The trading signal (our target variable) is generated by comparing a short-term moving average (MAVG) with a long-term MAVG.

- If short-term MAVG > long-term MAVG: Buy signal (1)
- Otherwise: Sell signal (0)

We'll use a 10-period rolling mean for the short-term MAVG and a 60-period rolling mean for the long-term MAVG, applied to the 'Close' price.

```
short_window = 10
long_window = 60
dataset['short_mavg'] = dataset['Close'].rolling(window=short_window,
min_periods=1).mean()
```

```
dataset['long_mavg'] = dataset['Close'].rolling(window=long_window,
min_periods=1).mean()
dataset['signal'] = 0.0
valid_signal_idx_start = max(short_window, long_window) -1
if len(dataset) > valid_signal_idx_start :
    dataset.loc[dataset.index[valid_signal_idx_start:], 'signal'] = np.where(
        dataset['short_mavg'][valid_signal_idx_start:] >
dataset['long_mavg'][valid_signal_idx_start:], 1.0, 0.0
    )
```

5.3. Feature Engineering: Technical Indicators

Raw price/volume data might not be sufficient for a model to learn complex patterns. Technical indicators can extract underlying trend, momentum, volatility, and other characteristics from the market data. We will create several common indicators to use as features.

Technical Indicators to Implement:

1. **Exponential Moving Average (EMA):** Similar to SMA but gives more weight to recent prices.
   $$\text{EMA}_{\text{today}} = (\text{Value}_{\text{today}} \times \text{Multiplier}) + \text{EMA}_{\text{yesterday}} \times (1 - \text{Multiplier})$$
   where $\text{Multiplier} = \frac{2}{\text{Period}+1}$
2. **Rate of Change (ROC):** Measures the percentage change in price between the current price and the price n periods ago.
   $$\text{ROC} = \left(\frac{\text{Close}_{\text{today}} - \text{Close}_{\text{n periods ago}}}{\text{Close}_{\text{n periods ago}}}\right) \times 100$$
3. **Momentum (MOM):** Measures the absolute change in price over n periods.
   $$\text{MOM} = \text{Close}_{\text{today}} - \text{Close}_{\text{n periods ago}}$$
4. **Relative Strength Index (RSI):** A momentum oscillator that measures the speed and change of price movements. RSI oscillates between 0 and 100.
   - Typically, RSI > 70 indicates overbought conditions, and RSI < 30 indicates oversold conditions.
   - Calculation involves average gains and average losses over a period.
     $$\text{RS} = \frac{\text{Average Gain}}{\text{Average Loss}} \text{RSI} = 100 - \frac{100}{1+\text{RS}}$$
5. **Stochastic Oscillator (%K and %D):** Compares a particular closing price of an asset to a range of its prices over a certain period of time.
   - %K Line:
     $$\%K = \left(\frac{\text{Current Close} - \text{Lowest Low over period}}{\text{Highest High over period} - \text{Lowest Low over period}}\right) \times 100$$
   - %D Line: Typically a 3-period SMA of %K (slow stochastic).

6. **Moving Average (MA):** Simple moving average (already used for signal, but can be features too).

```
for n_ema in [10, 30, 200]:
    dataset[f'EMA{n_ema}'] = EMA(dataset['Close'], n_ema)
for n_roc in [10, 30]:
    dataset[f'ROC{n_roc}'] = ROC(dataset['Close'], n_roc)
for n_mom in [10, 30]:
    dataset[f'MOM{n_mom}'] = MOM(dataset['Close'], n_mom)
for n_rsi in [10, 30, 200]:
    dataset[f'RSI{n_rsi}'] = RSI(dataset['Close'], n_rsi)
stoch_periods = [10, 30, 200]
d_smooth_period = 3
for n_stoch in stoch_periods:
    dataset[f'%K_{n_stoch}'] = STOK(dataset['Close'], dataset['Low'],
dataset['High'], n_stoch)
    dataset[f'%D_{n_stoch}_{d_smooth_period}'] =
STOD(dataset[f'%K_{n_stoch}'], d_smooth_period)
for n_ma in [21, 63, 252]:
    dataset[f'MA{n_ma}'] = MA(dataset['Close'], n_ma)

initial_rows = len(dataset)
dataset.replace([np.inf, -np.inf], np.nan, inplace=True) # Replace infs
created by indicators like RSI if loss is 0
dataset.dropna(inplace=True)
print(f"\nDropped {initial_rows - len(dataset)} rows due to NaNs/infs from
feature engineering.")

if dataset.empty:
    print("Dataset is empty after feature engineering and NaN drop. Cannot
proceed.")
    exit()
```

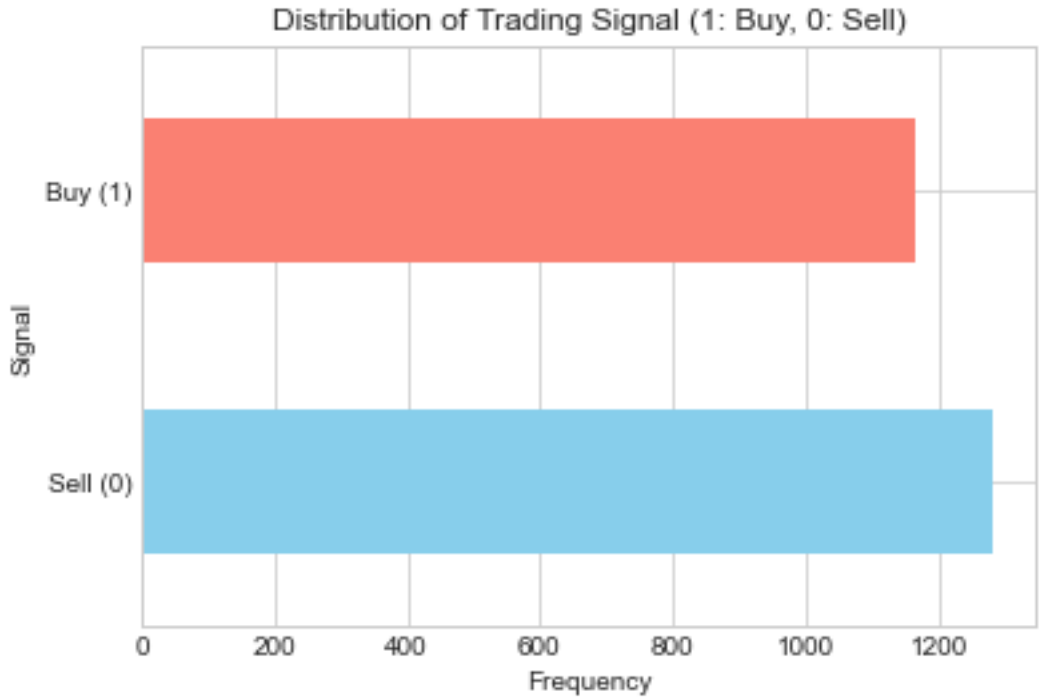
5.4. Data Visualization (Post Feature Engineering)

Let's check the distribution of our target variable `signal` after all data preparation.

```
plt.figure(figsize=(6, 4))
dataset['signal'].value_counts().plot(kind='barh', color=['skyblue',
'salmon'])
plt.title('Distribution of Trading Signal (1: Buy, 0: Sell)')
plt.xlabel('Frequency')
plt.ylabel('Signal')
plt.yticks(ticks=[0,1], labels=['Sell (0)', 'Buy (1)']) # Adjust based on
value_counts order
# plt.show()
plt.savefig('bitcoin_signal_distribution.png')
print("\nSaved trading signal distribution plot to
```

```
bitcoin_signal_distribution.png")
plt.close()
```

The distribution might be relatively balanced or slightly skewed depending on the market period and MAVG parameters. The PDF's example shows it as relatively balanced.



6. Evaluate Algorithms and Models

6.1. Prepare Data for Modeling

Separate features (X) and target (y). Drop columns used for target creation if they are not intended as features.

```
if 'signal' not in dataset.columns:
    print("Error: 'signal' column is missing from the dataset before
splitting.")
    exit()

features_to_drop_for_X = ['signal', 'short_mavg', 'long_mavg']
X = dataset.drop(columns=features_to_drop_for_X, errors='ignore')
y = dataset['signal']

X = X.apply(pd.to_numeric, errors='coerce').dropna(axis=1,
how='all').fillna(0)

if X.empty or len(X) != len(y) or X.shape[1] == 0:
    print("Feature set X is empty, mismatched with y, or has no columns after
```

```
final processing. Cannot proceed.")
    exit()
```

6.2. Train-Test Split

The PDF uses the last 100,000 observations for faster calculation. For daily data, this is a very long period. Let's use a standard chronological split for time series, e.g., 80% for training, 20% for testing.

```
split_index = int(len(X) * 0.8)
if split_index < 1 or split_index >= len(X) -1 :
    print(f"Cannot perform train-test split with current data size: {len(X)}.
Need more data after NaN drops.")
    exit()

X_train = X.iloc[:split_index]
X_test = X.iloc[split_index:]
y_train = y.iloc[:split_index]
y_test = y.iloc[split_index:]

if X_train.empty or X_test.empty or y_train.empty or y_test.empty:
    print("Training or testing set is empty. Cannot proceed with model
evaluation.")
    exit()
```

6.3. Test Options and Evaluation Metric

Given the signal distribution, **accuracy** can be a reasonable starting metric if the classes are somewhat balanced. We also need to look at precision, recall, and F1-score for buy/sell signals.

```
scoring_metric = 'accuracy'
num_folds = 5
kfold = KFold(n_splits=num_folds, shuffle=True, random_state=42)
```

6.4. Compare Models and Algorithms

Spot-check various classification algorithms.

```
models_btc = []
models_btc.append(('LR', LogisticRegression(solver='liblinear', max_iter=200,
random_state=42)))
models_btc.append(('LDA', LinearDiscriminantAnalysis()))
models_btc.append(('CART', DecisionTreeClassifier(random_state=42)))
models_btc.append(('RF', RandomForestClassifier(random_state=42, n_jobs=-1)))
models_btc.append(('GBM', GradientBoostingClassifier(random_state=42)))

results_btc = []
names_btc = []
print(f"\nSpot-checking models using {scoring_metric}:")
```

```
for name, model in models_btc:
    try:
        cv_results = cross_val_score(model, X_train, y_train, cv=kfold,
scoring=scoring_metric, n_jobs=-1)
        results_btc.append(cv_results)
        names_btc.append(name)
        print(f"{name}: {cv_results.mean():.4f} ({cv_results.std():.4f})")
    except Exception as e:
        print(f"Could not evaluate {name}: {e}")
```

The PDF identifies Random Forest as performing well among ensemble models. Let's assume it's a good candidate.

7. Model Tuning and Grid Search (Random Forest)

We'll tune hyperparameters for Random Forest using `GridSearchCV`.

```
best_model_btc = None
chosen_model_name_for_tuning = 'RF'
model_to_tune_proto = None
for name, model_proto_iter in models_btc:
    if name == chosen_model_name_for_tuning:
        model_to_tune_proto = model_proto_iter
        break

if model_to_tune_proto is not None:
    param_grid = {
        'n_estimators': [50, 100], 'max_depth': [5, 10, None], 'criterion':
['gini', 'entropy']
    } if isinstance(model_to_tune_proto, RandomForestClassifier) else {
        'n_estimators': [50, 100], 'learning_rate': [0.05, 0.1], 'max_depth':
[3,5]
    }
    grid = GridSearchCV(estimator=model_to_tune_proto, param_grid=param_grid,
scoring=scoring_metric, cv=kfold, n_jobs=-1)
    try:
        grid_result = grid.fit(X_train, y_train)
        print(f"\nBest {scoring_metric} for {chosen_model_name_for_tuning}:
{grid_result.best_score_:.4f} using {grid_result.best_params_}")
        best_model_btc = grid_result.best_estimator_
    except Exception as e:
        print(f"GridSearchCV failed for {chosen_model_name_for_tuning}: {e}")
        best_model_btc = model_to_tune_proto
        print(f"Using default (untuned) {chosen_model_name_for_tuning}
parameters due to GridSearchCV error.")
        best_model_btc.fit(X_train, y_train)
else:
    print(f"\nModel '{chosen_model_name_for_tuning}' not found or CV failed.
Using a default RF.")
```

```
    best_model_btc = RandomForestClassifier(random_state=42,
n_estimators=100, n_jobs=-1)
    if not X_train.empty and not y_train.empty:
         best_model_btc.fit(X_train, y_train)
    else:
        print("Cannot fit default model as training data is empty.")
        best_model_btc = None
```

8. Finalize the Model and Evaluate

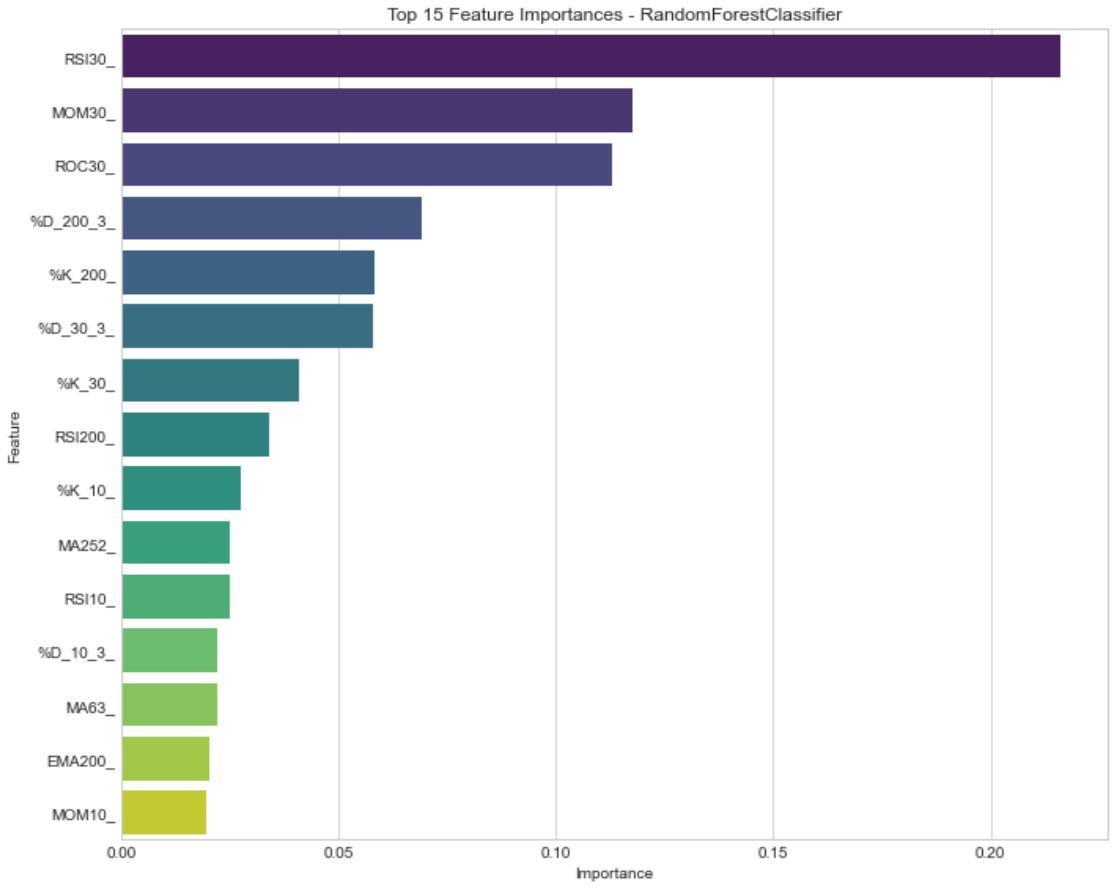
8.1. Results on the Test Dataset

Evaluate the tuned (or best chosen) model on the unseen test set.

```
if best_model_btc and not X_test.empty and not y_test.empty:
    y_pred_test = best_model_btc.predict(X_test)
    print(f"\nPerformance of Final Model
({best_model_btc.__class__.__name__}) on Test Set:")
    print(f"Accuracy: {accuracy_score(y_test, y_pred_test):.4f}")
    cm_test = confusion_matrix(y_test, y_pred_test)
    print("\nConfusion Matrix (Test Set):\n", cm_test)

    print("\nClassification Report (Test Set):")
    print(f"Unique values in y_test: {np.unique(y_test,
return_counts=True)}")
    print(f"Unique values in y_pred_test: {np.unique(y_pred_test,
return_counts=True)}")
    print(classification_report(y_test, y_pred_test, target_names=['Sell
(0)', 'Buy (1)'], labels=[0, 1], zero_division=0))

    if hasattr(best_model_btc, 'feature_importances_'):
        importances = best_model_btc.feature_importances_
        feature_names_original = X_train.columns

        str_feature_names = []
        for name in feature_names_original:
            if isinstance(name, tuple):
                str_feature_names.append('_'.join(map(str, name)))
            else:
                str_feature_names.append(str(name))

        feature_importance_df = pd.DataFrame({'feature': str_feature_names,
'importance': importances})
        feature_importance_df =
feature_importance_df.sort_values(by='importance', ascending=False)
        print("\nTop 15 Feature Importances (with stringified feature
names):")
        print(feature_importance_df.head(15))
        plt.figure(figsize=(10, 8))
        sns.barplot(x='importance', y='feature',
```

```
data=feature_importance_df.head(15), palette='viridis')
        plt.title(f'Top 15 Feature Importances -
{best_model_btc.__class__.__name__}')
        plt.xlabel('Importance')
        plt.ylabel('Feature')
        plt.tight_layout()
        # plt.savefig('bitcoin_feature_importance.png')
        print("\nSaved feature importance plot to
bitcoin_feature_importance.png")
        # plt.close()
else:
    print("\nNo model was finalized for evaluation or test set is empty.")
```

The model's accuracy and other metrics on the test set give an indication of its real-world performance. For tree-based models like Random Forest or GBM, we can examine feature importances.



This helps understand which technical indicators were most influential in the model's predictions. Momentum indicators like RSI and MOM often show high importance.

9. Backtesting the Trading Strategy (Simplified)

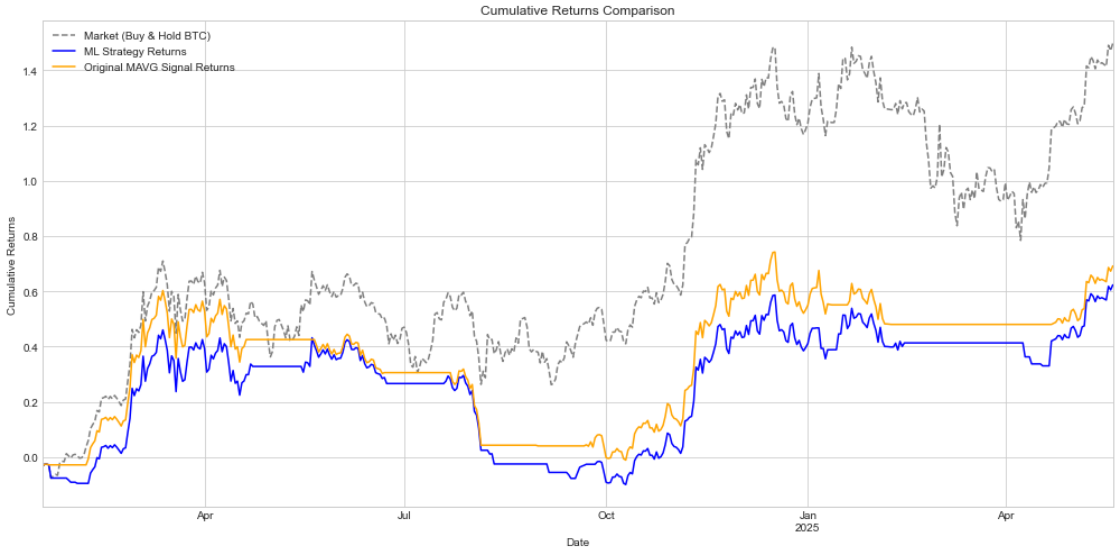
Backtesting simulates how the strategy would have performed on historical data. We'll create a simple backtest:

- Calculate daily market returns.
- Calculate strategy returns by multiplying market returns by the *predicted signal from the previous day* (since we trade on the next bar after a signal). A `1` means hold (or buy if not holding), a `0` means be out of the market (or sell if holding). This is a long-only interpretation for simplicity.

```
if best_model_btc and not X_test.empty and 'y_pred_test' in locals() and not
y_test.empty:
    backtest_df = pd.DataFrame(index=X_test.index)
    if 'Close' in dataset.columns and 'signal' in dataset.columns and
X_test.index.isin(dataset.index).all():
        backtest_df['Market_Returns'] = dataset.loc[X_test.index,
'Close'].pct_change()
        backtest_df['Predicted_Signal'] = y_pred_test
        backtest_df['Strategy_Returns'] = backtest_df['Market_Returns'] *
backtest_df['Predicted_Signal'].shift(1)
        backtest_df['Actual_MAVG_Signal_Returns'] =
backtest_df['Market_Returns'] * dataset.loc[X_test.index, 'signal'].shift(1)
        backtest_df.dropna(inplace=True)

        if not backtest_df.empty:
            backtest_df['Cumulative_Market_Returns'] = (1 +
backtest_df['Market_Returns']).cumprod() - 1
            backtest_df['Cumulative_Strategy_Returns'] = (1 +
backtest_df['Strategy_Returns']).cumprod() - 1
            backtest_df['Cumulative_Actual_MAVG_Signal_Returns'] = (1 +
backtest_df['Actual_MAVG_Signal_Returns']).cumprod() - 1
            print("\nBacktesting Results (Last 5 days):\n",
backtest_df.tail())
            plt.figure(figsize=(14, 7))
            backtest_df['Cumulative_Market_Returns'].plot(label='Market (Buy
& Hold BTC)', color='gray', linestyle='--')
            backtest_df['Cumulative_Strategy_Returns'].plot(label='ML
Strategy Returns', color='blue')

backtest_df['Cumulative_Actual_MAVG_Signal_Returns'].plot(label='Original
MAVG Signal Returns', color='orange')
            plt.title('Cumulative Returns Comparison')
            plt.ylabel('Cumulative Returns')
            plt.legend()
            plt.tight_layout()
            # plt.savefig('bitcoin_backtest_returns.png')
            print("\nSaved backtesting returns plot to
bitcoin_backtest_returns.png")
```

```
            # plt.close()
        else:
            print("\nBacktest DataFrame is empty after processing; cannot
plot returns.")
    else:
        print("\nCould not perform backtesting: 'Close' or 'signal' column
missing or index mismatch.")
else:
    print("\nSkipping backtesting as no model was finalized or
test/prediction data is unavailable.")

print("\n--- Tutorial: Algorithmic Bitcoin Trading Strategy Finished ---")
```

The plot comparing cumulative returns helps assess if the machine learning strategy added value over a simple buy-and-hold or the original MAVG crossover rule.



10. Conclusion and Next Steps

This tutorial demonstrated a complete workflow for building a Bitcoin trading strategy using machine learning classification. We covered:

- Defining the problem as a classification task.
- Fetching real market data using `yfinance`.
- Extensive feature engineering using technical indicators.
- Training, tuning, and evaluating various classification models.
- Assessing feature importance.
- Performing a simplified backtest.

The results of such a strategy can vary greatly depending on the chosen period, features, model, and market conditions. Key takeaways include the importance of robust feature engineering and careful model evaluation.

Further improvements and considerations could include:

- More sophisticated feature engineering (e.g., volatility measures, order book data if available).
- Different ways to define the target variable (e.g., predicting price change magnitude, multi-class signals like buy/sell/hold).
- Advanced backtesting with considerations for transaction costs, slippage, and risk management.
- Time series cross-validation techniques.
- Exploring more complex models like LSTMs or other deep learning architectures, though they require more data and computational resources.

This framework provides a solid foundation for developing and testing algorithmic trading strategies based on machine learning.

Build Your Own AI Coding Assistant From Plan to Execution with Python and Ollama

In today's fast-paced development world, Large Language Models (LLMs) are becoming invaluable assistants. But what if you could build an AI agent that not only writes code but also plans its approach, asks for your approval, and even debugs its own work until it's successful?

This tutorial will guide you through creating such an AI Coding Assistant using Python, the LangChain library for interacting with LLMs, and Ollama to run powerful open-source models locally. Our agent will take your request, propose a plan, get your green light, write the code, test it, debug it iteratively if needed, and finally, engage you with a thoughtful follow-up.

1. What You'll Build:

An AI agent that can:

1. **Understand Your Goal:** Take a natural language request for a Python script.
2. **Propose a Plan:** Ask an LLM to outline a high-level plan (in pseudocode) to achieve the goal.
3. **Seek Your Confirmation:** Present the plan to you for approval, allowing for one round of adjustments.
4. **Generate Code:** Instruct the LLM to write the full Python script based on the approved plan.

5. **Execute and Test:** Run the generated script.
6. **Iteratively Debug:** If the script fails, the agent feeds the error and the faulty code back to the LLM to get a corrected version, repeating this process until the script works or a maximum number of attempts is reached.
7. **Engage with Follow-up:** After a successful execution, the agent uses the LLM to ask you a relevant follow-up question, demonstrating contextual awareness.

2. Prerequisites:

- **Python 3.7+:** Ensure Python is installed on your system.
- **Ollama:** You need Ollama installed and running. Ollama allows you to run open-source LLMs like Llama 3, Mistral, Gemma, etc., locally.
    - Download Ollama: https://ollama.ai/
    - Pull a model: After installing Ollama, pull a model you want to use. For example, in your terminal:

      ```
      ollama pull gemma3:12b
      ```
- **LangChain Libraries:** Install the necessary Python packages:

  ```
  pip install langchain langchain-community
  ```

3. Code Deep Dive

Let's break down the script's components.

3.1. Configuration

```
import os, re, subprocess
from langchain_community.llms import Ollama
import warnings

warnings.filterwarnings(action="ignore")

# --- Configuration ---
MODEL_NAME   = "gemma3:12b"     # Your Ollama model tag
MAX_ATTEMPTS = 5                # How many retry loops before giving up
PROMPT_FILE  = "prompt.txt"     # Optional text file for your request
TEMP_SCRIPT  = "temp_script.py" # Where generated scripts get saved

# Patterns to catch errors even when exit code == 0
ERROR_PATTERNS = [
    r"Traceback \(most recent call last\):",
    r"Exception:", r"Error occurred", r"Error:",
    r"SyntaxError:", r"NameError:", r"TypeError:", r"AttributeError:",
    r"ImportError:", r"IndexError:", r"KeyError:", r"ValueError:",
```

```
r"FileNotFoundError:"
]
```

- `MODEL_NAME`: Specifies the Ollama model tag. **Crucially, change this to a model you have downloaded.**
- `MAX_ATTEMPTS`: The maximum number of times the agent will try to generate and debug code for a single request after the plan is approved.
- `PROMPT_FILE`: An optional text file (e.g., `prompt.txt`) where you can write your detailed script request. If this file isn't found, the agent will ask for input directly.
- `TEMP_SCRIPT`: The filename used to save and execute the LLM-generated Python code.
- `ERROR_PATTERNS`: A list of regular expressions used to scan the output of the generated script for common error indicators.

3.2. Helper Functions

These functions perform essential tasks:

- **`extract_code_block(text: str) -> str | None`**:

  ```
  def extract_code_block(text: str) -> str | None:
      if not text:
          return None
      m = re.search(r"```(?:python)?\s*(.*?)\s*```", text, re.DOTALL)
      return m.group(1).strip() if m else None
  ```

  Uses regular expressions to find and extract Python code enclosed in Markdown-style triple backticks (e.g., ```python ... ``` or ```...```). The `re.DOTALL` flag is important for code blocks that span multiple lines.

- **`run_script(path: str, timeout: int = 180) -> tuple[int, str]`**:

  ```
  def run_script(path: str, timeout: int = 180) -> tuple[int, str]:
      try:
          p = subprocess.run(
              ["python", path], capture_output=True, text=True,
              timeout=timeout, check=False
          )
          return p.returncode, (p.stdout or "") + (p.stderr or "")
      except subprocess.TimeoutExpired:
          return -1, f"⏰ Timeout after {timeout}s"
      except FileNotFoundError:
          return -1, f"❗ Script '{path}' not found."
      except Exception as e:
          return -1, f"❗ Error running script: {e}"
  ```

Executes the Python script saved at `path` using `subprocess.run`. It captures `stdout` and `stderr`, returns the script's exit code, and handles potential timeouts or other execution errors.

- **`invoke_llm(llm_instance: Ollama, prompt: str, extract_code: bool = True) -> tuple[str|None, str]`**:

```
def invoke_llm(llm_instance: Ollama, prompt: str, extract_code: bool =
True) -> tuple[str|None, str]:
    print("🧠 Thinking…")
    full = llm_instance.invoke(prompt)
    if extract_code:
        return extract_code_block(full), full
    return full, full
```

This is the gateway to your LLM. It sends a `prompt`, gets the `full_response`, and optionally tries to extract a code block. It prints a "Thinking..." message to let you know the LLM is working, keeping the actual prompt hidden for a cleaner interface.

- **`save_code(code: str, path: str)`**: A straightforward function to write the LLM-generated code to `TEMP_SCRIPT`.

- **`output_has_errors(output: str) -> bool`**: Checks if the script's captured output string contains any of the patterns listed in `ERROR_PATTERNS`. This helps detect failures even if the script exits with a return code of 0.

3.3. The `main()` Function: Orchestrating the Agent

This is where the magic happens, following a clear, phased approach:

Phase 1 & 2: LLM Initialization and Loading User Request

```
def main_interactive_loop():
    print("\n🔮 AI Agent: Plan ▶ Confirm ▶ Generate ▶ Debug ▶ Follow-up
🔮 \n")

    llm = None # Initialize llm to None
    try:
        llm = Ollama(model=MODEL_NAME)
        print(f"🤖 LLM '{MODEL_NAME}' initialized.")
    except Exception as e:
        print(f"❌ Cannot start LLM '{MODEL_NAME}': {e}")
        print("   Ensure Ollama is running and the model name is correct
(e.g., 'ollama list' to check).")
        return

    user_req_original = "" # This will be updated in each iteration of the
outer loop
```

```
    # Outer loop for continuous interaction
    while True:
        # 2) Load User Request (or get follow-up as new request)
        if not user_req_original: # First time or after an explicit 'new'
            if os.path.isfile(PROMPT_FILE) and os.path.getsize(PROMPT_FILE) >
0: # Check if prompt file exists and is not empty
                try:
                    with open(PROMPT_FILE, 'r+', encoding="utf-8") as f: #
Open in r+ to read and then truncate
                        user_req_original = f.read().strip()
                        f.seek(0) # Go to the beginning of the file
                        f.truncate() # Empty the file
                    if user_req_original:
                        print(f"📄 Loaded request from '{PROMPT_FILE}' (file
will be cleared after use).")
                    else: # File was empty
                        user_req_original = input("Enter your Python-script
request (or type 'exit' to quit): ").strip()
                except Exception as e:
                    print(f"Error reading or clearing {PROMPT_FILE}: {e}")
                    user_req_original = input("Enter your Python-script
request (or type 'exit' to quit): ").strip()
            else:
                user_req_original = input("Enter your Python-script request
(or type 'exit' to quit): ").strip()

        if user_req_original.lower() == 'exit':
            print("👋 Exiting agent.")
            break
        if not user_req_original:
            print("❌ No request provided. Please enter a request or type
'exit'.")
            user_req_original = "" # Reset to ensure it asks again
            continue

        current_contextual_request = user_req_original # Initialize for the
current task cycle
```

The LLM is initialized. Note the absence of `StreamingStdOutCallbackHandler` to prevent token-by-token printing of the LLM's raw response. The user's initial request for the script is loaded either from `prompt.txt` or by asking for input.

Phase 3: Planning and User Confirmation

```
# 3) PLAN PHASE
        plan_approved = False
        plan_code = ""
```

```
        for plan_attempt in range(2): # Allow one initial plan + one
adjustment attempt
            print(f"\n🧠 Phase: Proposing Plan (Attempt {plan_attempt + 1}/2
for current request)")
            plan_prompt = (
                "You are an expert Python developer and system architect.\n"
                "Your task is to create a super short super high-level plan
just in 3 to 5 sentences "
                "(in Python-style pseudocode with numbered comments) "
                "to implement the following user request. Do NOT write the
full Python script yet, only the plan.\n\n"
                f"User Request:\n'''{current_contextual_request}'''\n\n"
                "Instructions for your plan:\n"
                "- Use numbered comments (e.g., # 1. Initialize
variables).\n"
                "- Keep it high-level but clear enough to guide
implementation.\n"
                "- Wrap ONLY the pseudocode plan in a ```python ... ```
block."
            )
            extracted_plan, plan_resp_full = invoke_llm(llm, plan_prompt)

            if not extracted_plan:
                print(f"❌ LLM did not return a plan in the expected format
(attempt {plan_attempt + 1}).")
                if plan_attempt == 0:
                     retry_plan = input("Try generating plan again? (Y/n):
").strip().lower()
                     if retry_plan not in ("", "y", "yes"):
                        print("Aborting plan phase for current request.")
                        # Go to end of inner task cycle, which will then loop
outer for new request
                        plan_code = None # Signal plan failure
                        break
                else: # Second attempt also failed
                    print("Aborting plan phase after adjustment attempt
failed.")
                    plan_code = None # Signal plan failure
                    break
                continue # To next plan attempt

            plan_code = extracted_plan
            print("\n📝 Here's the proposed plan:\n")
            print(plan_code)

            ok = input("\nIs this plan OK? (Y/n/edit) ").strip().lower()
            if ok in ("", "y", "yes"):
                plan_approved = True
```

```
                print("✅ Plan approved by user.")
                break
            elif ok == "edit":
                adjustment_notes = input("What should be adjusted in the plan
or original request? (Your notes will be added to the request context):
").strip()
                if adjustment_notes:
                    current_contextual_request =
f"{user_req_original}\n\nUser's Plan Adjustment
Notes:\n'''{adjustment_notes}'''"
                    print("✅ Plan adjustment notes added. Regenerating
plan...")
                else:
                    print("No adjustment notes provided. Assuming current
plan is OK.")
                    plan_approved = True
                    break
            else:
                print("Plan not approved. This task will be skipped.")
                plan_code = None # Signal plan rejection
                break # Exit plan loop for this task

        if not plan_approved or not plan_code:
            print("❌ Plan not finalized or approved for the current
request.")
            user_req_original = "" # Reset to ask for a new request in the
next outer loop iteration
            print("-" * 30)
            continue # Go to next iteration of the outer while loop
```

This is a crucial interactive step.

- A carefully crafted `plan_prompt` asks the LLM for a *short, high-level pseudocode plan* (3-5 sentences as per your latest script's prompt addition), not the full code.
- The extracted plan is shown to you.
- You can type `Y` (or just Enter) to approve, `n` to reject (which exits), or `edit`.
- If you choose `edit`, you can provide adjustment notes. These notes are appended to the original request to form `current_contextual_request`, and the agent tries to generate an updated plan (one retry).

Phase 4: Code Generation and Iterative Debugging

```
# 4) GENERATE & DEBUG PHASE
        print("\n🧠 Phase: Generating and Debugging Code...")
        last_script_output = ""
        final_working_code = ""
        script_succeeded_this_cycle = False
```

```
        for attempt in range(1, MAX_ATTEMPTS + 1):
            print(f"🔄 Code Generation/Debug Attempt
{attempt}/{MAX_ATTEMPTS}")
            gen_prompt = ""
            # ... (gen_prompt logic for attempt 1 and debug attempts -
remains the same) ...
            if attempt == 1:
                gen_prompt = (
                    "You are an expert Python programmer.\n"
                    "Based on the following **approved plan**:\n"
                    f"```python\n{plan_code}\n```\n\n"
                    "And the original user request (with any adjustment
notes):\n"
                    f"'''{current_contextual_request}'''\n\n"
                    "Write a Python script as short and simple as possible.
Ensure all necessary imports are included. "
                    "Focus on fulfilling the plan and request accurately.\n"
                    "Wrap your answer ONLY in a ```python ... ``` code block.
No explanations outside the block."
                )
            else: # Debugging
                gen_prompt = (
                    "You are an expert Python debugger.\n"
                    "The goal was to implement this plan:\n"
                    f"```python\n{plan_code}\n```\n"
                    "And this overall request:\n"
                    f"'''{current_contextual_request}'''\n\n"
                    "The previous attempt at the script was:\n"
                    f"```python\n{final_working_code}\n```\n"
                    "Which produced this output (indicating errors):\n"
                    f"```text\n{last_script_output}\n```\n\n"
                    "Please meticulously analyze the errors, the code's
deviation from the plan, and the original request. "
                    "Provide a **fully corrected, complete Python script**
that fixes the issues and aligns with the plan and request. "
                    "Wrap your answer ONLY in a ```python ... ``` code
block."
                )

            code_block, code_resp_full = invoke_llm(llm, gen_prompt)
            if not code_block:
                print(f"❌ LLM did not return a code block in attempt
{attempt}.")
                if attempt == MAX_ATTEMPTS: break
                last_script_output = f"LLM failed to provide a code block.
Response: {code_resp_full}"
                continue

            final_working_code = code_block
```

```
            save_code(final_working_code, TEMP_SCRIPT)
            print(f"💾 The followig script generated and saved to
'{TEMP_SCRIPT}':\n\n f{final_working_code}.\n\n Running…")

            rc, out = run_script(TEMP_SCRIPT)
            print(f"  ▶ Script Return code: {rc}")
            if len(out or "") < 600: print(f"  📋 Script Output:\n{out}")
            else: print(f"  📋 Script Output (last 500 chars):\n{(out or
'')[-500:]}")
            last_script_output = out
```

Once the plan is approved:

- **Initial Generation:** For `attempt == 1`, `gen_prompt` instructs the LLM to write the full Python script based on `plan_code` and `current_contextual_request`. Your script now includes "Write a Python script as short and simple as possible."
- **Debugging:** If the script fails (non-zero `rc` or error patterns in `out`), for subsequent attempts, `gen_prompt` provides the LLM with:
  - The original plan and request.
  - The `final_working_code` (which was the code that just failed).
  - The `last_script_output` (the error messages from the failed run). It explicitly asks the LLM to analyze and correct the script.
- This loop continues for `MAX_ATTEMPTS`.

Phase 5: Follow-up Question (After Success)

```
    if rc == 0 and not output_has_errors(out):
                print("\n✅ 🎉 Success! Script ran cleanly for the current
request.")
                script_succeeded_this_cycle = True
                break # Exit debug loop on success
            else:
                print("⚠️ Errors detected or non-zero return code; will
attempt to debug...")

        if not script_succeeded_this_cycle:
            print(f"\n❌ All {MAX_ATTEMPTS} debug attempts exhausted for the
current request. Last script is in '{TEMP_SCRIPT}'.")
            user_req_original = "" # Reset to ask for new request
            print("-" * 30)
            continue # Go to next iteration of the outer while loop

        # 5) FOLLOW-UP QUESTION PHASE (Only if script_succeeded_this_cycle is
True)
        print("\n🧠 Phase: Follow-up")
        follow_up_context_prompt = (
```

```
            "You are a helpful AI assistant.\n"
            "The user had an initial request:\n"
            f"'''{user_req_original}'''\n" # Use the original request for
this specific cycle for context
            "An execution plan was approved:\n"
            f"```python\n{plan_code}\n```\n"
            "The following Python script was successfully generated and
executed to fulfill this:\n"
            f"```python\n{final_working_code}\n```\n"
            "The script's output (last 500 chars) was:\n"
            f"```text\n{last_script_output[-500:]}\n```\n\n"
            "Now, explain the code first very shortly and then ask the user a
concise and relevant follow-up question based on this success. "
            "For example, ask if they want to modify the script, save its
output differently, "
            "run it with new parameters, or tackle a related task. Do not
wrap your question in any special tags."
        )
        follow_up_question_text, _ = invoke_llm(llm,
follow_up_context_prompt, extract_code=False)
        print(f"\n🤖 Assistant: {follow_up_question_text.strip()}")

        user_response_to_follow_up = input("Your response (or type 'new' for
a new unrelated task, 'exit' to quit): ").strip()

        if user_response_to_follow_up.lower() == 'exit':
            print("👋 Exiting agent.")
            break # Exit outer while loop
        elif user_response_to_follow_up.lower() == 'new':
            user_req_original = "" # Clear it so it asks for a fresh prompt
        else:
            # Treat the response as a new request, potentially related to the
last one.
            # The LLM doesn't have explicit memory of this Q&A for the *next*
planning phase
            # unless we build that into the prompt. For now, it's a new
user_req_original.
            user_req_original = "The following Python script was successfully
generated and executed to fulfill this:\n"
            f"```python\n{final_working_code}\n```\n" + \
                "user had the following follow-up request:" + \
                user_response_to_follow_up


        print("-" * 30) # Separator for the next cycle
```

If the script runs successfully:

- A detailed `follow_up_context_prompt` is constructed, giving the LLM the full story: the initial request, the plan, the successful code, and a snippet of its output.
- The LLM is then tasked to ask you a relevant follow-up question. This demonstrates a simple form of memory and contextual awareness.
- The agent prints the LLM's question and then exits. (For a continuous conversation, you'd add an input loop here).

4. How to Use the AI Coding Assistant

1. **Save the Code:** Copy the entire Python script above and save it as a file, for example, `ai_agent.py`.
2. **Set `MODEL_NAME`:** Open `ai_agent.py` and change the `MODEL_NAME` variable to the exact tag of an LLM you have downloaded in Ollama (e.g., `"llama3:8b"`, `"mistral:latest"`, `"gemma2:9b"`).
3. **Run Ollama:** Ensure your Ollama application is running and the chosen model is available.
4. **Run the Agent:** Open your terminal or command prompt, navigate to the directory where you saved `ai_agent.py`, and run:

   ```
   python ai_agent.py
   ```

5. **Interact:**
   - The agent will ask for your request.
   - It will show you a "Thinking..." message and then present a plan.
   - Respond with `Y` (or Enter) to approve, `n` to reject, or `edit` to provide adjustment notes.
   - If approved, it will generate and test the code, showing you script outputs and success/failure status.
   - If successful, it will ask a follow-up question.

Example Interaction:

```
🔮 AI Agent: Plan ▶ Confirm ▶ Generate ▶ Debug ▶ Follow-up 🔮

🤖 LLM 'gemma3:12b' initialized.
Enter your Python-script request (or type 'exit' to quit): get financial
statements for tesla from yahoo finance and store them in csv files.

🧠 Phase: Proposing Plan (Attempt 1/2 for current request)
🧠 Thinking...

📝 Here's the proposed plan:
```

```
# 1. Define functions: fetch_financial_data(ticker) to retrieve data from
Yahoo Finance API, and save_to_csv(data, filename) to store it.
# 2. Initialize ticker symbol (e.g., "TSLA") and a list of financial
statement types (e.g., ["income_stmt", "balance_sheet", "cash_flow"]).
# 3. Iterate through the list of financial statement types, calling
fetch_financial_data() for each, and then save_to_csv() to store the
retrieved data as CSV files.
# 4. Implement error handling within the loop to manage potential API issues
or data retrieval failures (e.g., try-except blocks).
# 5. Add a main execution block to run the process only when the script is
run directly, ensuring reusability.

Is this plan OK? (Y/n/edit) y
✅ Plan approved by user.

🧠 Phase: Generating and Debugging Code...
🔄 Code Generation/Debug Attempt 1/5
🧠 Thinking...
💾 The followig script generated and saved to 'temp_script.py':

 fimport yfinance as yf
import pandas as pd

def fetch_financial_data(ticker):
    try:
        data = yf.Ticker(ticker).financials
        return data
    except Exception as e:
        print(f"Error fetching data for {ticker}: {e}")
        return None

def save_to_csv(data, filename):
    try:
        if data is not None:
            data.to_csv(filename)
            print(f"Data saved to {filename}")
        else:
            print(f"No data to save to {filename}")
    except Exception as e:
        print(f"Error saving to {filename}: {e}")

if __name__ == "__main__":
    ticker = "TSLA"
    financial_statements = ["income_stmt", "balance_sheet", "cash_flow"]

    for statement_type in financial_statements:
```

```
        data = fetch_financial_data(ticker)
        if data is not None:
            filename = f"{ticker}_{statement_type}.csv"
            save_to_csv(data, filename).

 Running…
  ▶ Script Return code: 0
  📋 Script Output:
Data saved to TSLA_income_stmt.csv
Data saved to TSLA_balance_sheet.csv
Data saved to TSLA_cash_flow.csv


✅🎉 Success! Script ran cleanly for the current request.

🧠 Phase: Follow-up
🧠 Thinking...

🤖 Assistant: The code retrieves financial statements (income statement,
balance sheet, and cash flow) for Tesla (TSLA) from Yahoo Finance using the
`yfinance` library and saves each statement as a separate CSV file. Error
handling is included to manage potential issues during data fetching or
saving.

Would you like to modify the script to retrieve data for a different ticker
symbol?
Your response (or type 'new' for a new unrelated task, 'exit' to quit): exit
👋 Exiting agent.
```

5. Key Concepts Demonstrated

- **LLM as a Multi-Role Tool:** Used for planning, code generation, debugging, and even generating conversational follow-ups.
- **Prompt Engineering:** The script uses different, carefully crafted prompts for each distinct task (planning, initial code generation, debugging, follow-up). The quality of these prompts heavily influences the LLM's performance.
- **Iterative Refinement:** The debugging loop is a prime example of iterative refinement, where the agent learns from failures.
- **User-in-the-Loop:** The plan confirmation stage ensures human oversight and alignment before significant computation (code generation) occurs.
- **Local and Private AI:** By using Ollama, the entire process can run locally, keeping your requests and code private.

6. Potential Improvements & Customization

This agent is a strong foundation. Here are some ideas to extend it:

- **Advanced Plan Refinement:** Instead of just one adjustment, allow a multi-turn dialogue to refine the plan.
- **Persistent Memory for Follow-ups:** Use LangChain's `ConversationChain` and memory modules if you want the follow-up interaction to be a longer, stateful conversation.
- **Tool Usage:** For more complex tasks, explore LangChain Agents that can use tools (e.g., web search for API docs, file system access).
- **GUI/Web Interface:** Create a more user-friendly interface instead of the command line.
- **Saving Successful Scripts:** Automatically save successfully generated scripts with meaningful names instead of just `temp_script.py`.
- **More Sophisticated Error Analysis:** Instead of just regex patterns, use the LLM to analyze the `stderr` more deeply to understand the root cause of errors during debugging.
- **Cost/Token Management:** If using paid LLM APIs (not the case with local Ollama here, but for future reference), tracking token usage would be important.

7. Conclusion

You've now explored the architecture of an AI Coding Assistant that goes beyond simple code generation. By incorporating planning, user confirmation, and robust iterative debugging, this agent provides a more intelligent and collaborative approach to leveraging LLMs for development tasks. The ability to run this locally with Ollama opens up many possibilities for customization and private, powerful AI assistance. Experiment with different models, refine the prompts, and happy coding!

Can Kalman Filters Improve Your Trading Signals

Kalman filters offer an advanced technique for signal processing, often used to extract underlying states, like trend or velocity, from noisy data. Applying this to financial markets allows us to estimate price movements potentially more adaptively than standard indicators.

This article details a `backtrader` strategy using a Kalman filter (via a custom `KalmanFilterIndicator`) to estimate price velocity. The strategy enters trades based on the *sign* of this estimated velocity and relies exclusively on a trailing stop-loss for exits.

Strategy Logic Overview:

1. **Filtering:** A `KalmanFilterIndicator` estimates the underlying price and its velocity based on closing prices.

2. **Entry Signal:** Enter long if the estimated velocity turns positive (`> 0`). Enter short if the velocity turns negative (`< 0`). Entries only happen when flat.
3. **Exit Signal:** A percentage-based trailing stop-loss (`trail_percent`) manages exits. Once a position is open, the Kalman velocity sign is ignored for exiting.

The Supporting Indicator: `KalmanFilterIndicator`

(The code for `KalmanFilterIndicator` as you provided it is assumed here. It calculates `kf_price` and `kf_velocity` and is set to plot on the main chart panel using `plotinfo = dict(subplot=False)`).

The Strategy Class: `KalmanFilterTrendWithTrail`

This class orchestrates the trading logic using the indicator's output.

1. Parameters (`params`)

These allow configuration of the filter and the trailing stop.

```
# --- Inside KalmanFilterTrendWithTrail class ---
    params = (
        # Parameters passed to the Kalman Filter Indicator
        ('process_noise', 1e-5),    # Filter parameter: Assumed noise in the
price model (Q)
        ('measurement_noise', 1e-1),# Filter parameter: Assumed noise in the
price data (R)

        # Strategy-specific parameter
        ('trail_percent', 0.05),    # Trailing stop loss percentage (e.g.,
0.05 = 5%)
        ('printlog', True),         # Enable logging output
    )
```

- `process_noise` & `measurement_noise`: Control the Kalman filter's behavior. Finding good values requires testing and optimization specific to the asset and timeframe.
- `trail_percent`: Determines the percentage drawdown from the peak price (for longs) or trough price (for shorts) that triggers the stop-loss.

2. Initialization (`__init__`)

Sets up the strategy by creating the indicator instance.

```
# --- Inside KalmanFilterTrendWithTrail class ---
    def __init__(self):
        # Instantiate the Kalman Filter Indicator, passing relevant
parameters
        self.kf = KalmanFilterIndicator(
            process_noise=self.p.process_noise,
            measurement_noise=self.p.measurement_noise
```

```
        )

        # Create convenient references to the indicator's output lines
        self.kf_price = self.kf.lines.kf_price
        self.kf_velocity = self.kf.lines.kf_velocity

        # Initialize order trackers
        self.order = None # Tracks pending entry orders
        self.stop_order = None # Tracks pending stop orders

        if self.params.printlog:
            # Log the parameters being used
            print(f"Strategy Parameters: Process
Noise={self.params.process_noise}, "
                  f"Measurement Noise={self.params.measurement_noise}, "
                  f"Trail Percent={self.params.trail_percent * 100:.2f}%")
```

3. Entry Logic (next)

The next method contains the core logic executed on each bar. For entries, it checks the position status and the Kalman velocity sign.

```
# --- Inside KalmanFilterTrendWithTrail class ---
    def next(self):
        # If an entry order is pending, do nothing
        if self.order:
            return

        # Get the estimated velocity from the indicator
        # Need to check length because indicator might need warmup
        if len(self.kf_velocity) == 0:
             return # Indicator not ready yet

        estimated_velocity = self.kf_velocity[0]
        current_position_size = self.position.size
        current_close = self.data.close[0] # For logging

        # --- Trading Logic ---
        # Only evaluate entries if FLAT
        if current_position_size == 0:
            if self.stop_order: # Safety check - cancel any stray stop orders
if flat
                self.log("Warning: Position flat but stop order exists.
Cancelling.", doprint=True)
                self.cancel(self.stop_order)
                self.stop_order = None

            # --- Entry Signal Check ---
            if estimated_velocity > 0:
```

```
                # Positive velocity -> Go Long
                self.log(f'BUY CREATE (KF Vel > 0),
Close={current_close:.2f}, KF Vel={estimated_velocity:.4f}', doprint=True)
                self.order = self.buy() # Place buy order and track it
            elif estimated_velocity < 0:
                # Negative velocity -> Go Short
                self.log(f'SELL CREATE (KF Vel < 0 - Short Entry),
Close={current_close:.2f}, KF Vel={estimated_velocity:.4f}', doprint=True)
                self.order = self.sell() # Place sell order and track it
        else:
             # If already in a position, do nothing here.
             # The trailing stop placed via notify_order handles the exit.
             pass
```

This logic is straightforward: if flat, buy on positive velocity, sell on negative velocity. If already in a position, it relies entirely on the trailing stop.

4. Exit Logic (`notify_order`)

Exits are handled by placing a `StopTrail` order immediately after an entry order is successfully filled. This logic resides within the `notify_order` method.

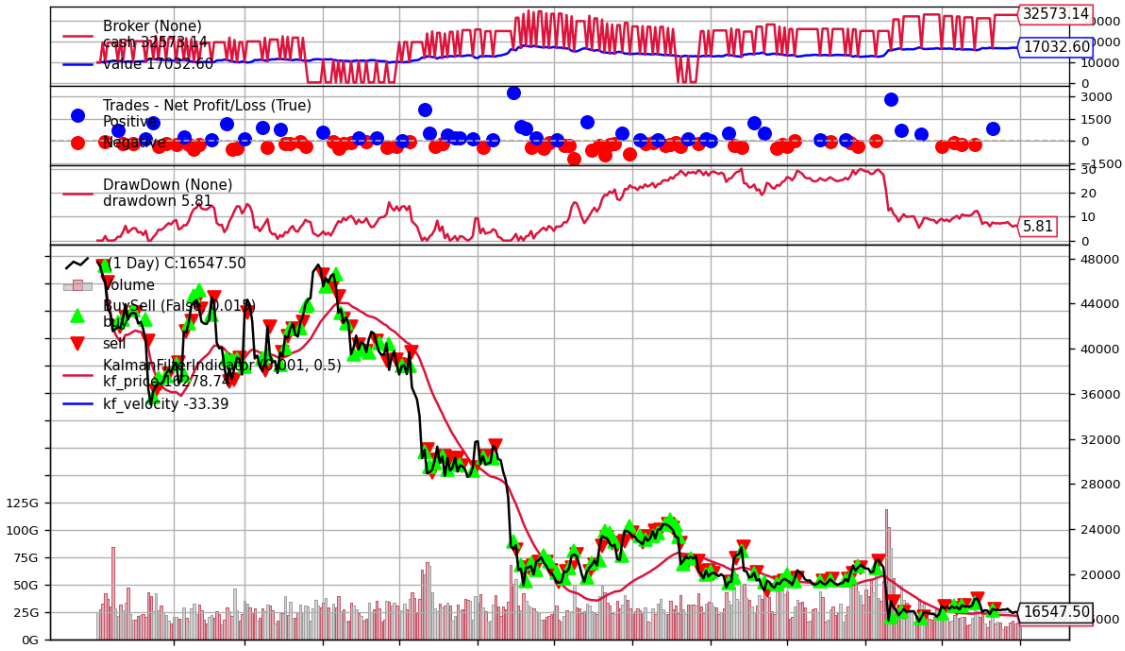
```
# --- Inside KalmanFilterTrendWithTrail class ---
    def notify_order(self, order):
        # (Initial checks for Submitted/Accepted status omitted for brevity)
        ...
        if order.status == order.Completed:
            # Check if it's the ENTRY order we were waiting for
            if self.order and order.ref == self.order.ref:
                entry_type = "BUY" if order.isbuy() else "SELL"
                exit_func = self.sell if order.isbuy() else self.buy #
Determine exit order type

                # Log entry execution (code omitted for brevity)
                ...

                # Place the TRAILING STOP order if trail_percent is valid
                if self.p.trail_percent and self.p.trail_percent > 0.0:
                    self.stop_order = exit_func(exectype=bt.Order.StopTrail,

trailpercent=self.p.trail_percent)
                    self.log(f'Trailing Stop Placed for {entry_type} order
ref {self.stop_order.ref} at {self.p.trail_percent * 100:.2f}% trail',
doprint=True)
                else:
                     self.log(f'No Trailing Stop Placed
(trail_percent={self.p.trail_percent})', doprint=True)

                self.order = None # Reset entry order tracker
```

```
            # Check if it's the STOP order that completed
            elif self.stop_order and order.ref == self.stop_order.ref:
                # Log stop execution (code omitted for brevity)
                ...
                self.stop_order = None # Reset stop order tracker
                self.order = None # Reset entry tracker too
        # Handle Failed orders (code omitted for brevity)
        ...
```

This ensures that as soon as an entry trade is confirmed, the trailing stop is activated to manage the exit.

Running the Backtest

The `__main__` block in your provided code sets up `cerebro`, fetches data (BTC-USD, 2021-2023), configures the broker/sizer/analyzers, and runs the strategy with specific parameters (`process_noise=0.001`, `measurement_noise=0.5`, `trail_percent=0.02`). It then prints performance metrics and attempts to plot the results, including the Kalman Filter price overlayed on the main chart.



Tuning and Considerations

- **Parameter Sensitivity:** This strategy's performance is highly dependent on the `process_noise`, `measurement_noise`, and `trail_percent` parameters. The values used (`0.001`, `0.5`, `0.02`) are specific examples and likely require optimization for different market conditions or assets.

- **Whipsaws:** Using only the sign of the velocity can lead to frequent entries and exits (whipsaws) in non-trending or choppy markets, potentially hurting performance even with a trailing stop.
- **Model Limitations:** The constant velocity model is a simplification. Real market dynamics are more complex.
- **Optimization:** Thorough backtesting and optimization across various parameter combinations are essential to evaluate the strategy's potential robustness.

Conclusion

This `backtrader` strategy demonstrates using a Kalman filter's velocity estimate for trend direction signals, combined with a trailing stop for risk management. While conceptually interesting, its practical effectiveness hinges critically on careful parameter tuning and understanding its limitations, particularly the sensitivity to noise when using only the velocity sign for entries.

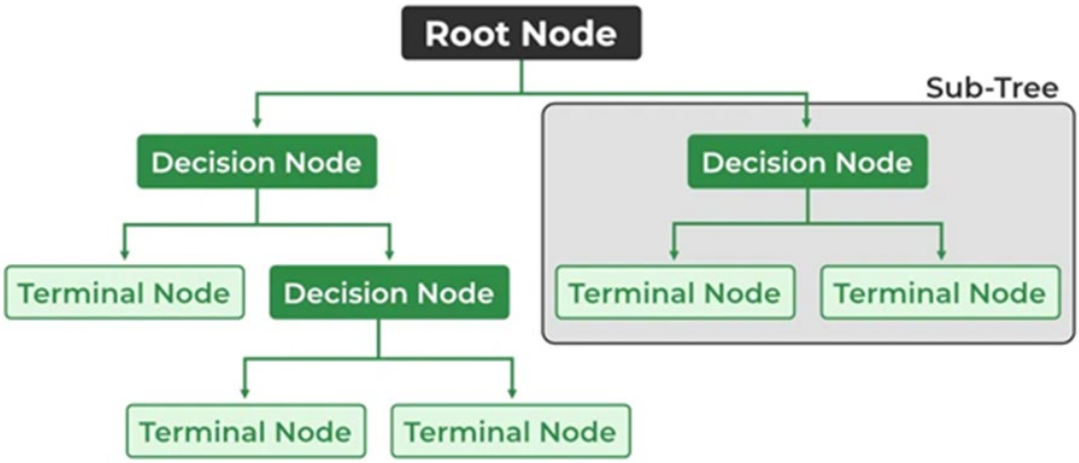
Decision Tree Learning

Decision Tree Learning

Decision tree learning is a supervised machine learning method used in statistics and data mining. It involves creating a predictive model in the form of a classification or regression tree based on a set of observations. Classification trees are used when the target variable has discrete values, representing class labels, while regression trees are employed for continuous values. Decision trees are popular for their simplicity and intelligibility. They visually represent decisions and decision-making processes, making them useful in decision analysis. In data mining, decision trees describe data, and the resulting classification tree can be used for decision-making.

Decision tree learning is widely used in data mining is aiming to create a predictive model for a target variable based on multiple input variables. In this context, a decision tree is a straightforward representation used for classifying examples. Assuming finite discrete domains for input features and a single target feature called "classification," the decision tree comprises nodes labeled with input features and branches labeled with possible

values of the target



feature.

The tree-building process involves recursively splitting the source set into subsets based on classification features, creating internal nodes and decision paths. This top-down induction of decision trees is a greedy algorithm, where each node is split to maximize predictive value. The recursion stops when subsets have uniform values for the target variable, or further splitting adds minimal predictive value.

The data consists of records in the form ((, Y)=(x_1, x_2, x_3, , x_k, Y)), where (Y) is the target variable, and () is the feature vector used for the task.

Types of Decision Trees

Decision trees used in data mining can be categorized into two main types:

· **Classification Tree:** This type predicts the class (discrete) to which the data belongs. Each leaf node in the tree represents a class label, and the branches represent conjunctions of features leading to those labels.

· **Regression Tree:** This type predicts outcomes that can be considered real numbers, such as the price of a house or a patient's length of stay in a hospital.

The term **Classification and Regression Tree (CART)** refers to both procedures, and it was introduced by Breiman et al. in 1984. While classification and regression trees share similarities, there are differences, particularly in the procedure used to determine where to split.

Ensemble methods, known as boosted trees and bagged decision trees, construct multiple decision trees for improved performance:

· **Boosted Trees:** Incrementally builds an ensemble by training each new instance to emphasize the training instances previously mis-modeled. AdaBoost is a typical example, applicable to both regression and classification problems.

· **Bootstrap Aggregated (Bagged) Decision Trees:** Builds multiple decision trees by repeatedly resampling training data with replacement, and the trees vote for a consensus prediction.

· **Random Forest Classifier:** A specific type of bootstrap-aggregated decision trees.

· **Rotation Forest:** Each decision tree is trained by first applying principal component analysis (PCA) on a random subset of the input features.

Notable decision tree algorithms include ID3 (Iterative Dichotomiser 3), C4.5 (a successor of ID3), and CART (Classification And Regression Tree). These algorithms were developed independently but follow a similar approach for learning decision trees from training tuples.

Additionally, concepts from fuzzy set theory have been proposed for a special version of a decision tree known as Fuzzy Decision Tree (FDT), where an input vector is associated with multiple classes, each having a different confidence value. Boosted ensembles of FDTs have been suggested for improved performance.

Let's take a brief look at two algorithms in the following.

ID3 Algorithm

The ID3 algorithm begins with the original set (S) as as the root node. On each iteration of the algorithm, it iterates through every unused attribute of the set (S) and calculates the entropy (H(S)) or the information gain (IG(S)) of that attribute. It then selects the attribute which has the smallest entropy (or largest information gain) value. The set (S) is then split or partitioned by the selected attribute to produce subsets of the data.

In summary:

1. Take the Entire dataset as an input.
2. Calculate the Entropy of the target variable, As well as the predictor attributes
3. Calculate the information gain of all attributes.
4. Choose the attribute with the highest information gain as the Root Node
5. Repeat the same procedure on every branch until the decision node of each branch is finalized.

Entropy is a measure of impurity or disorder in a set of examples. In the context of decision trees, it is used to quantify the uncertainty associated with a given set of data.

((S)=_{x X}-p(x) _2 p(x))

Information gain measures the effectiveness of an attribute in reducing uncertainty (entropy) about the classification of the data.

(I G(S, A)=(S)-_{t T} p(t) (t)=(S)-(S A))

Classification And Regression Tree Algorithm

The CART (Classification And Regression Tree) algorithm is a decision tree algorithm that can be used for both classification and regression tasks. It was introduced by Leo Breiman and his colleagues in 1984. CART is widely used due to its simplicity, effectiveness, and versatility in handling different types of data. Here's an overview of how the CART algorithm works:

Basic Steps of the CART Algorithm:

· **Binary Splitting:** The CART algorithm builds a binary tree, where each node represents a binary decision based on the value of a particular attribute. At each node, the algorithm selects the attribute and a corresponding threshold that best splits the data into two subsets.

· **Objective Function for Splitting:** For classification tasks, the Gini impurity is commonly used as an objective function to measure the impurity of a node. For regression tasks, the mean squared error (MSE) is used to evaluate the variance of the target variable within a node.

· **Node Splitting:** The algorithm evaluates all possible splits for each attribute and selects the split that minimizes the Gini impurity (for classification) or the mean squared error (for regression). The selected attribute and threshold are used to create two child nodes, and the data is divided accordingly.

· **Recursion:** The process is repeated recursively for each child node until a stopping criterion is met. This could be a predefined tree depth, a minimum number of samples in a node, or other criteria.

· **Leaf Node Prediction:** When a stopping criterion is met, a leaf node is created. For classification, the majority class in the node is assigned to the leaf. For regression, the mean or median of the target values in the node is used.

Gini Impurity

Gini impurity is used by the CART (classification and regression tree) algorithm for classification trees. Gini impurity measures how often a randomly chosen element of a set would be incorrectly labeled if it were labeled randomly and independently according to the distribution of labels in the set. It reaches its minimum (zero) when all cases in the node fall into a single target category.

(*G(p)*={i=1}^J(p_i *{k i}* p_k)={i=1}^J p_i(1-p_i)=*{i=1}*$^{J(p\_i-p\_i 2)}$={i=1}^J p_i-*{i=1}^J p_i^2=1*-{i=1}^J p_i^2)

Python Implementation

We will use the following model from sklearn library to predict the price direction for bitcoin.

sklearn.tree.DecisionTreeClassifier

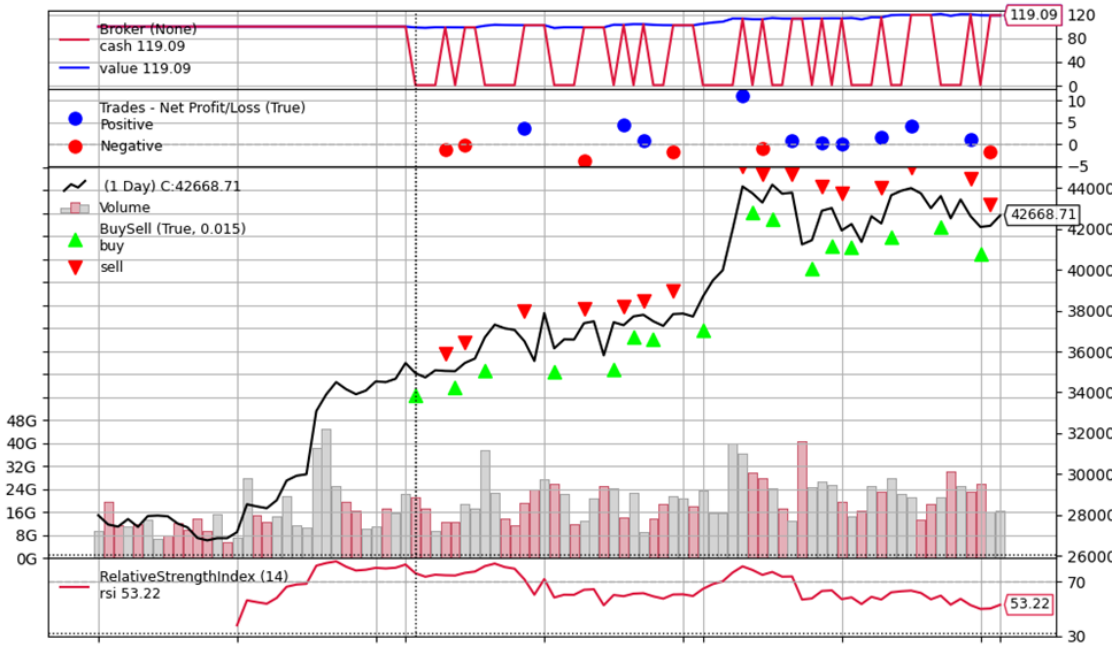
class sklearn.tree.**DecisionTreeClassifier**(*_*_, *criterion='gini'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *class_weight=None*, *ccp_alpha=0.0*)

The following Python code implements a basic trading strategy using Backtrader, a financial analysis library. The strategy employs a Decision Tree classifier to make buy and sell decisions based on historical price data for Bitcoin (BTC-USD). The strategy's features include the price difference between close and open, Relative Strength Index (RSI), and trading volume over a specified lookback period. The Decision Tree model is trained on these features, and predictions are made to determine whether to buy, sell, or hold. The strategy uses a simple position sizing approach and executes trades accordingly. The Backtrader engine is configured to handle the strategy, and the code concludes by printing the initial and final values of the brokerage account and plotting the strategy's performance over the specified historical data period.

```
import backtrader as bt
from sklearn.tree import DecisionTreeClassifier
import numpy as np
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt

class MLStrategy(bt.Strategy):
    params = (
        ("lookback_period", 30),
        ("decision_tree_model", DecisionTreeClassifier())
    )

    def __init__(self):
        self.data_close = self.datas[0].close
        self.data_open = self.datas[0].open
        self.decision_tree_model = self.params.decision_tree_model
        self.lookback_period = self.params.lookback_period
        self.rsi = bt.indicators.RelativeStrengthIndex(self.data_close,
period=14)
        self.volume = self.datas[0].volume
        self.order = None

    def next(self):
        if len(self) > self.lookback_period:
            # Convert array.array to NumPy arrays for subtraction
            close_prices =
```

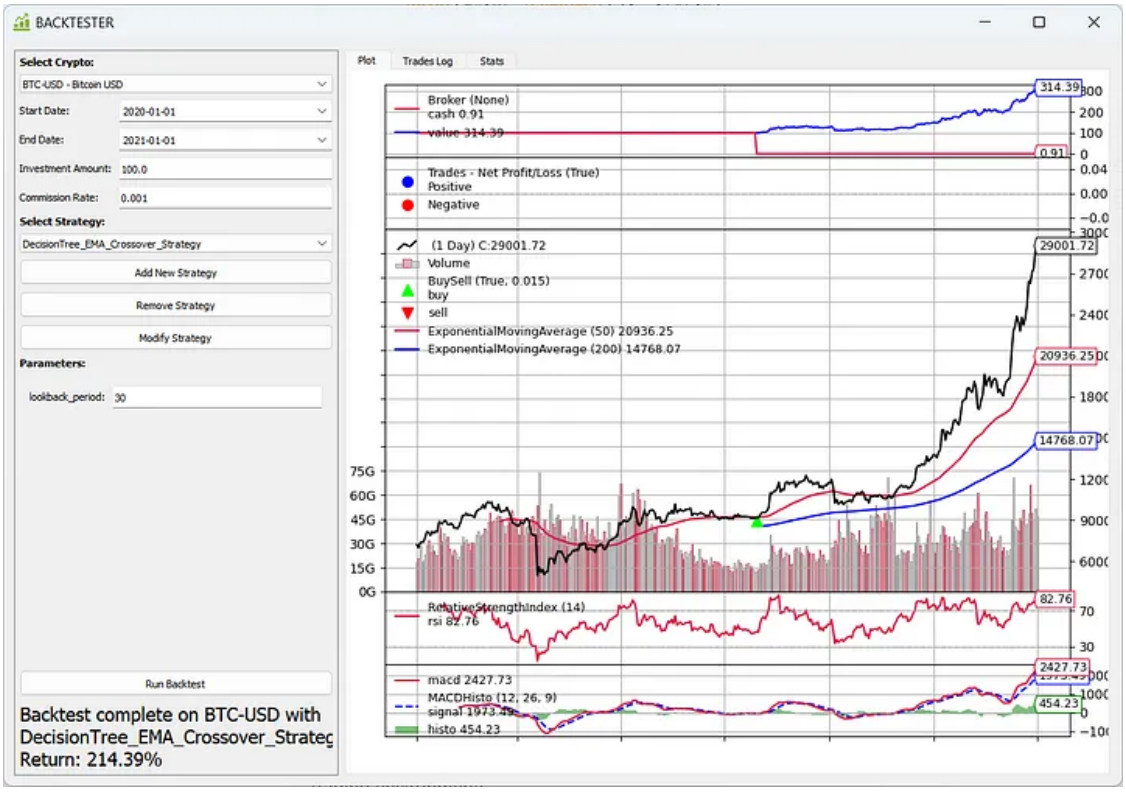
```
np.array(self.data_close.get(size=self.lookback_period))
            open_prices =
np.array(self.data_open.get(size=self.lookback_period))
            price_diff = close_prices - open_prices
            rsi_values = np.array(self.rsi.get(size=self.lookback_period))
            volume_values =
np.array(self.volume.get(size=self.lookback_period))

            # Feature generation
            features = np.column_stack((price_diff, rsi_values,
volume_values))

            # Decision tree input
            X = features.reshape(-1, 3)

            # price directions
            y = np.sign(np.diff(self.data_close.get(size=self.lookback_period
+ 1)))

            # Train the decision tree model
            self.decision_tree_model.fit(X[:-1], y[1:])

            # Predict using decision tree
            prediction = self.decision_tree_model.predict(X[-1:])

            # Check if there is no open position
            if not self.position:
                cash = self.broker.get_cash()
                asset_price = self.data.close[0]
                position_size = cash / asset_price * 0.99

                # Make trading decision based on prediction
                if prediction[-1] == 1:
                    self.buy(size=position_size)
            else:
                if prediction[-1] == -1:
                    self.close()

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return

        # Check if an order has been completed
        if order.status == order.Completed:
            if order.isbuy():
                self.log(f"Buy executed: {order.executed.price:.2f}")
            elif order.issell():
                self.log(f"Sell executed: {order.executed.price:.2f}")
```

```
        # Reset order
        self.order = None

    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0)
        print(f"{dt.isoformat()}, {txt}")

if __name__ == '__main__':
    # Create a Cerebro engine
    cerebro = bt.Cerebro()

    # Add data
    data = bt.feeds.PandasData(dataname=yf.download('BTC-USD',
                                                    period='3mo',
                                                    ))

    cerebro.adddata(data)

    # Add the strategy
    cerebro.addstrategy(MLStrategy)

    # Set the initial cash amount
    cerebro.broker.setcash(100.)
    cerebro.broker.setcommission(.001)

    print('<START> Brokerage account: $%.2f' % cerebro.broker.getvalue())
    cerebro.run()
    print('<FINISH> Brokerage account: $%.2f' % cerebro.broker.getvalue())

    # Plot the strategy
    plt.rcParams["figure.figsize"] = (10, 6)
    cerebro.plot()
```



Decision Trees and EMA Crossover 50% Average Annual Returns

I am working on trading strategies that blend traditional technical indicators with machine learning to generate buy and sell signals. In this article I try mixing a simple EMA crossover strategy with Decision Trees. First I will explain a bit on the theory of the methods and then share the Python implementation of the strategy with some backtest results. You can read this article on my website as well: https://www.aliazary.com/. You will find more articles and resources as well. You can also subscribe with your email address so that you get my newsletter and don't miss out on anything, especially my new backtesting app that I am working on. You can add your own strategies, modify the strategies and change the parameters and the asset and dates for backtesting the strategies to find the best strategies for trading:



1_2Wv9CPpkmEQFwszVpkcv2g 1.webp

1_2Wv9CPpkmEQFwszVpkcv2g 1.webp

---

1. Theoretical Foundations

1.1 Exponential Moving Average (EMA)

The **Exponential Moving Average (EMA)** is a weighted moving average that gives more importance to recent prices, making it more responsive to new information. The formula is:

[*t = P_t + (1 -)* {t-1}]

where:

- (P_t) is the current price,
- (=) is the smoothing factor, and
- (n) is the number of periods.

In our strategy, we use two EMAs:

- **Short-term EMA** with a period of 50.

- **Long-term EMA** with a period of 200.

A bullish signal is generated when the short-term EMA crosses above the long-term EMA, while a bearish signal occurs when it crosses below.

1.2 Relative Strength Index (RSI)

The **Relative Strength Index (RSI)** is a momentum oscillator that measures the speed and change of price movements. Its formula is:

[= 100 -]

with

[RS =]

Typically, an RSI above 70 suggests that an asset may be overbought, while an RSI below 30 indicates oversold conditions.

1.3 Moving Average Convergence Divergence (MACD)

The **MACD** is a trend-following momentum indicator that shows the relationship between two EMAs of a security's price. It is calculated as:

[= *{}* - {}]

Usually, the short-term EMA is taken over 12 periods and the long-term EMA over 26 periods. A signal line, typically a 9-period EMA of the MACD, is also computed. In this strategy, the MACD histogram (the difference between the MACD line and its signal line) is used to capture momentum changes.

---

2. Decision Trees: Theory and Equations

2.1 Introduction to Decision Trees

A **Decision Tree** is a non-parametric supervised learning method used for both classification and regression. In classification, the goal is to assign a class label to a given input by learning decision rules inferred from the features.

2.2 Structure of a Decision Tree

A decision tree is composed of:

- **Root Node:** Represents the entire dataset.
- **Internal Nodes:** Each node represents a test on an attribute (feature).

- **Branches:** The outcome of the test.
- **Leaf Nodes:** Represent class labels or outcomes.

2.3 Splitting Criteria

To split the data at each node, decision trees typically use measures of impurity such as **Entropy** or the **Gini Index**.

Entropy

Entropy is a measure of the randomness or impurity in the data. For a binary classification, the entropy (H) is calculated as:

[H(p) = -p _2(p) - (1-p) _2(1-p)]

where (p) is the proportion of positive examples. A perfectly pure node (all examples of one class) has an entropy of 0.

Information Gain

Information Gain (IG) is used to measure the effectiveness of a split. It is defined as the difference between the entropy of the parent node and the weighted average of the entropies of the child nodes:

[= H() - _{i=1}^{k} H(_i)]

where:

- (N) is the total number of samples in the parent node,
- (N_i) is the number of samples in child (i), and
- (H(_i)) is the entropy of child (i).

Gini Index

The Gini Index is another measure of impurity:

[(p) = 1 - _{i=1}^{C} p_i^2]

where (p_i) is the probability of class (i) in the node. Lower values indicate higher purity.

2.4 Decision Trees in the Trading Strategy

In our strategy, the **Decision Tree Classifier** is used to predict whether the price will go up (represented by 1) or not (represented by 0). The steps include:

1. **Feature Extraction:**
   The classifier uses features derived from technical indicators (e.g., EMA values, RSI, MACD, signal values) over a defined lookback window.

2. **Training:**
   The decision tree is trained on historical data from the lookback window. The training involves splitting the data based on the feature values to minimize impurity (using either entropy or Gini Index).

3. **Prediction:**
   The latest feature vector is passed to the trained decision tree, which predicts the class label (up or down). This prediction is then used as one of the signals for trade execution.

4. **Model Adaptation:**
   The model is retrained continuously using a rolling window, ensuring that it adapts to new market conditions.

---

3. Strategy Implementation

3.1 Feature Engineering

In this strategy, features are generated from a lookback window (default 30 periods) including:

- Short-term EMA values (50 periods)
- Long-term EMA values (200 periods)
- RSI values (14 periods)
- MACD values and its signal line

These features are stacked into a matrix (X) for the decision tree to process. The target variable (y) is defined based on whether the price increased in the lookback window.

3.2 Training and Prediction Process

- **Training Data:**
  The feature matrix (X) is constructed from historical data (all rows except the last) and aligned with the target variable (y) (shifted by one period to maintain causality).

- **Prediction:**
  The most recent feature vector (last row of (X)) is fed into the decision tree to predict whether the price will increase.

3.3 Trade Execution Logic

The strategy combines the machine learning prediction with the EMA crossover condition:

- **Entry Signal:**
  If the decision tree predicts a price increase (1) and the short-term EMA is above the long-term EMA, a buy order is executed.
- **Position Sizing:**
  The size of the position is calculated based on available cash and the asset price, with a minor adjustment factor (0.99) for risk management.
- **Exit Signal:**
  If the short-term EMA falls below the long-term EMA, any open positions are closed, signaling a potential trend reversal.

3.4 Code Walkthrough

Below is a Python implementation of the the strategy for use with backtrader library (or the **BACKTESTER** app) that integrates these concepts:

```
class DecisionTree_EMA_Crossover_Strategy(bt.Strategy):
    params = (("lookback_period", 30),)

    def __init__(self):
        # Data series and lookback window
        self.data_close = self.datas[0].close
        self.window = self.params.lookback_period

        # Decision Tree Classifier initialization
        self.model = DecisionTreeClassifier(random_state=42)

        # Technical indicators initialization
        self.emas = bt.indicators.ExponentialMovingAverage(self.data_close,
period=50)
        self.emal = bt.indicators.ExponentialMovingAverage(self.data_close,
period=200)
        self.rsi = bt.indicators.RelativeStrengthIndex(self.data_close,
period=14)
        self.macd = bt.indicators.MACDHisto(self.data_close,
                                            period_me1=12,
                                            period_me2=26,
                                            period_signal=9)
        self.order = None  # Track pending orders

    def next(self):
        # Ensure sufficient data is available for the lookback period
        if len(self) > self.window:
```

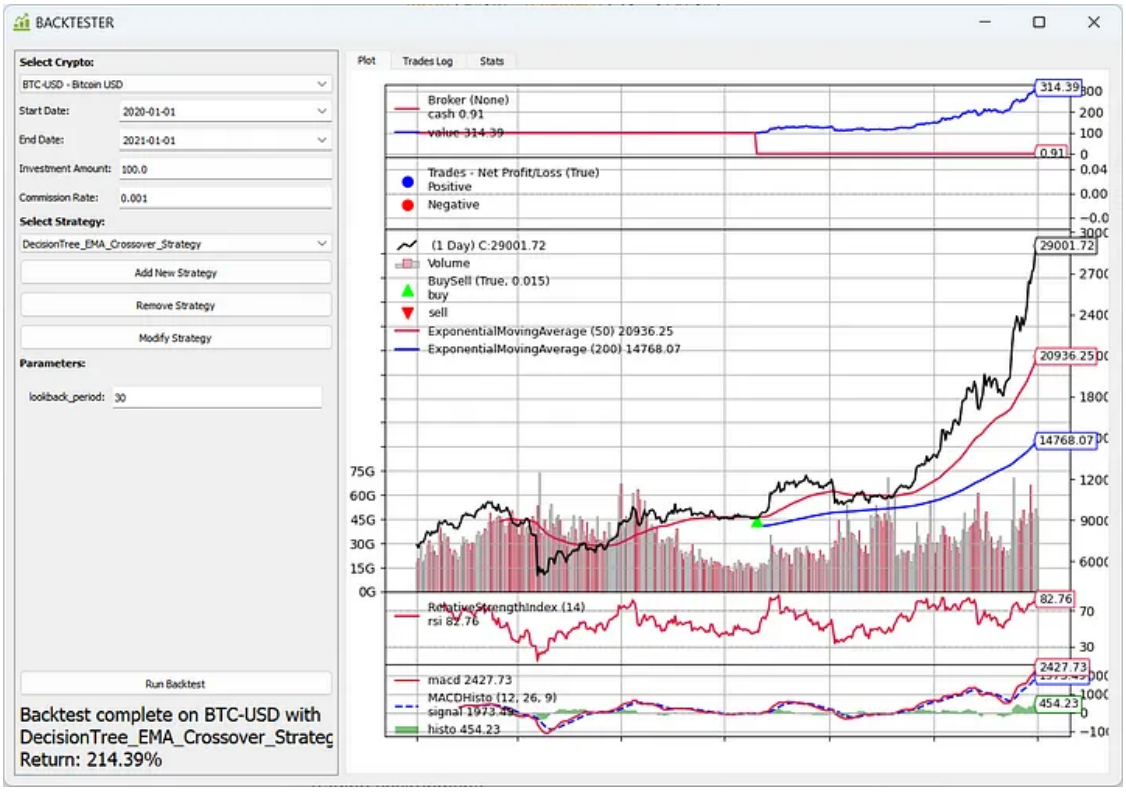
```
            # Extract indicator values over the lookback window
            emas_values = np.array(self.emas.get(size=self.window))
            emal_values = np.array(self.emal.get(size=self.window))
            rsi_values = np.array(self.rsi.get(size=self.window))
            macd_values = np.array(self.macd.macd.get(size=self.window))
            signal_values = np.array(self.macd.signal.get(size=self.window))

            # Construct feature matrix X
            X = np.column_stack((emas_values, emal_values, rsi_values,
macd_values, signal_values))

            # Define target variable: 1 if price increased, 0 otherwise
            prices = np.array(self.data_close.get(size=self.window + 1))
            y = np.where(np.diff(prices) > 0, 1, 0)

            # Prepare training and testing data
            X_train = X[:-1]
            y_train = y[1:]  # Shift target by one period to align with
features
            X_test = X[-1]

            # Train the decision tree on historical lookback data
            self.model.fit(X_train, y_train)

            # Predict the next move using the most recent features
            prediction = self.model.predict(X_test.reshape(1, -1))

            # Trade execution: enter position if conditions are met
            if not self.position:
                cash = self.broker.get_cash()
                asset_price = self.data_close[0]
                position_size = cash / asset_price * 0.99

                # Buy if prediction is 1 and the EMA crossover is bullish
                if prediction[0] == 1 and self.emas[0] > self.emal[0]:
                    self.buy(size=position_size)
                    self.log(f"Buy order placed at price: {asset_price:.2f}")
            else:
                # Close position if the EMA crossover indicates a bearish
trend
                if self.emas[0] < self.emal[0]:
                    self.close()
                    self.log(f"Position closed at price:
{self.data_close[0]:.2f}")

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
```

```
        # Log order execution details
        if order.status == order.Completed:
            if order.isbuy():
                self.log(f"Buy executed: {order.executed.price:.2f}")
            elif order.issell():
                self.log(f"Sell executed: {order.executed.price:.2f}")
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log("Order canceled/margin/rejected")
        self.order = None

    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0)
        print(f"{dt.isoformat()}, {txt}")
```

---

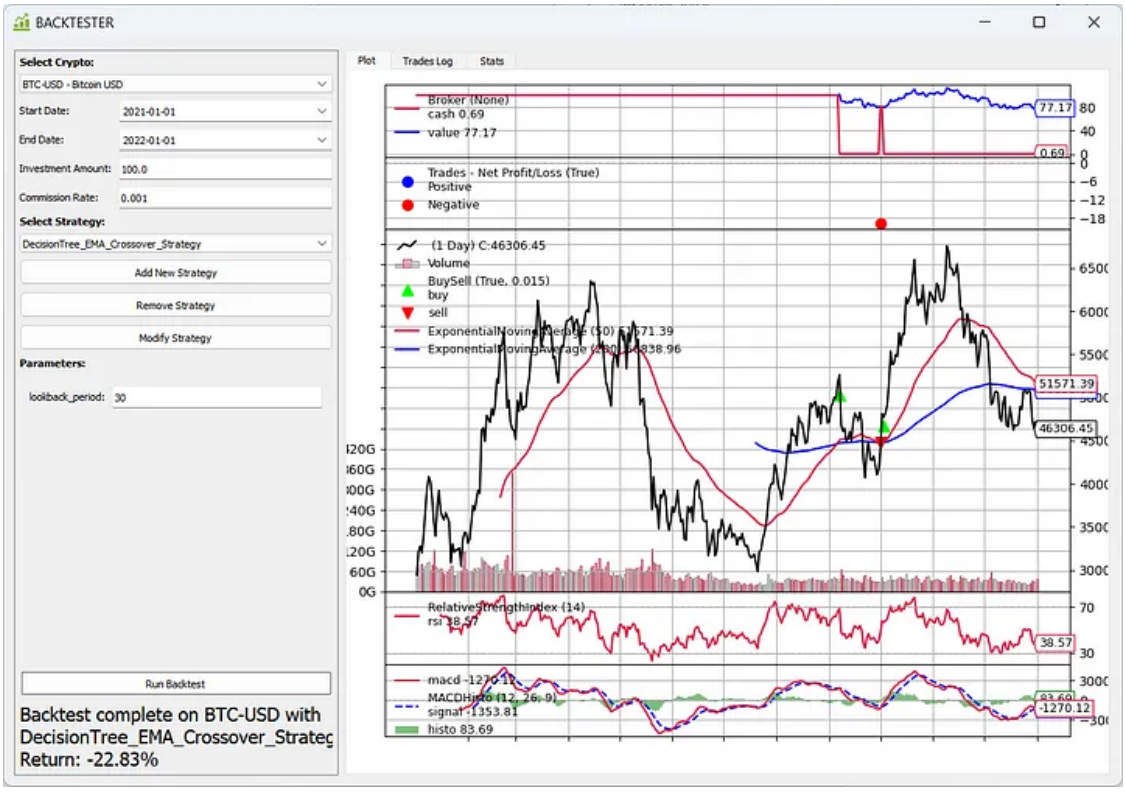
5. Backtests

let's see the backtest results for trading Bitcoin for 5 consecutive years from 2020 to 2025. Since the strategy only takes long positions, it won't make money in bearish markets. However, we can easily add short positions using opposite conditions so that we can make money in any market regime. If you are trading using a margin or futures account you can take short positions as well:



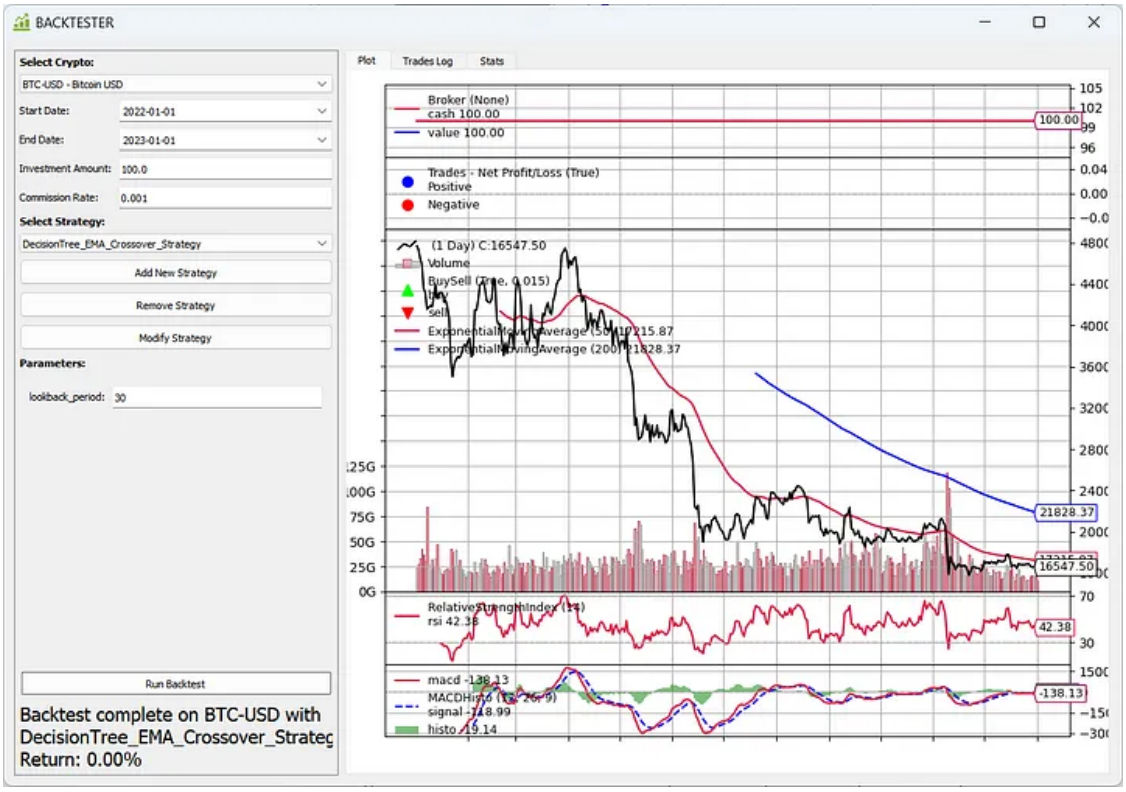
1_2Wv9CPpkmEQFwszVpkcv2g.webp

1_2Wv9CPpkmEQFwszVpkcv2g.webp



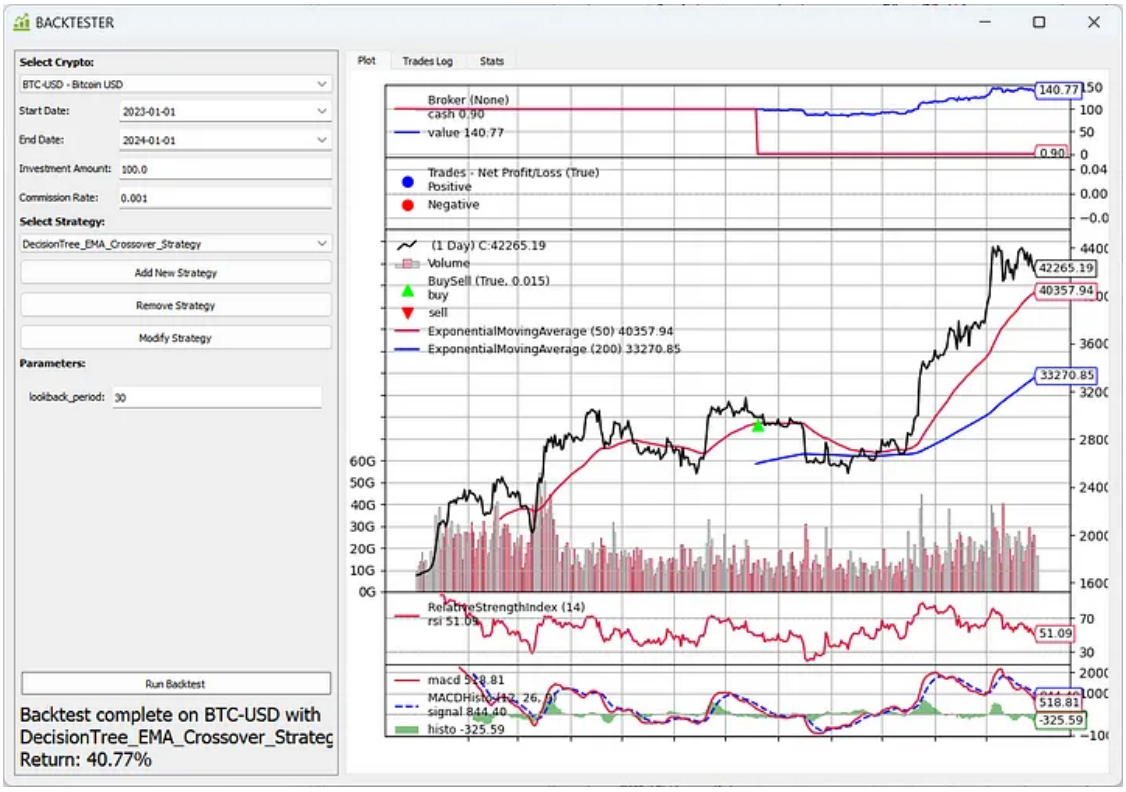
1_ZhRVxa7aCOYNzouXc3pBCQ.webp

1_ZhRVxa7aCOYNzouXc3pBCQ.webp



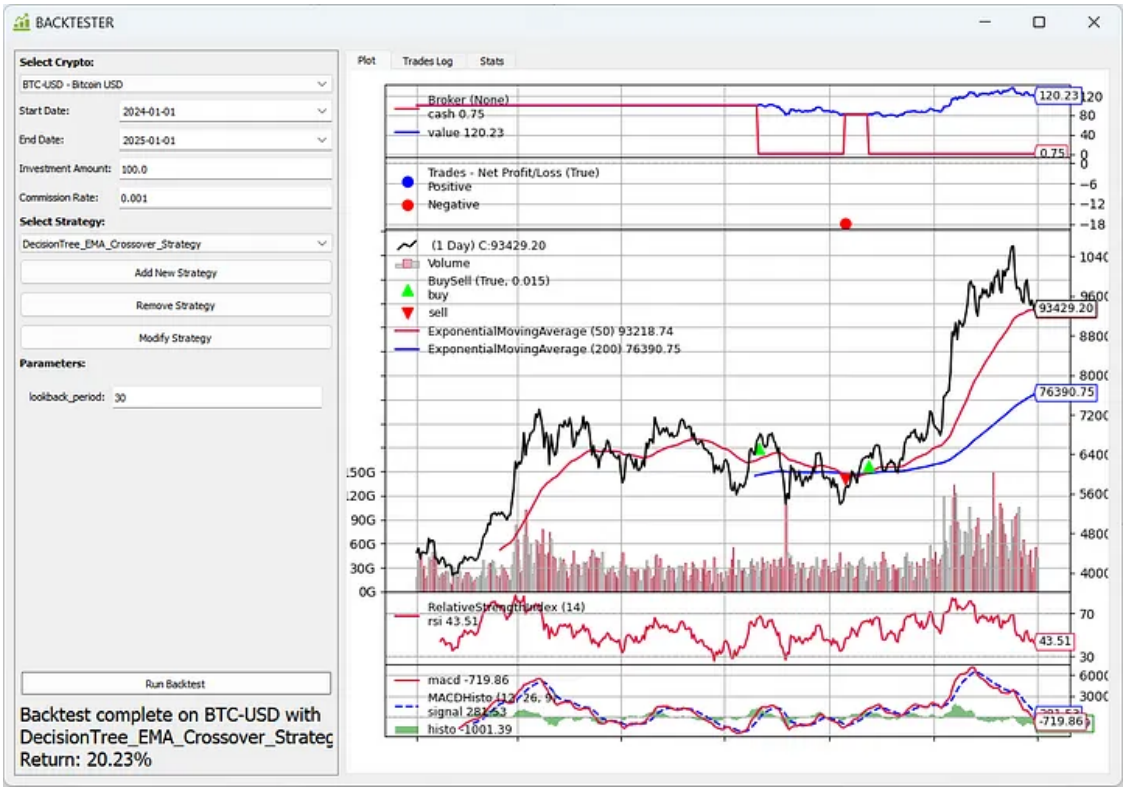
1_IrSuMTr6fSl9BAbzhcbJNQ.webp

1_IrSuMTr6fSl9BAbzhcbJNQ.webp



1_NsD8a7tzNxEnXahla5Nakg.webp

1_NsD8a7tzNxEnXahla5Nakg.webp



1_kQHBhBLCMohnNUWJrBUXbQ.webp

1_kQHBhBLCMohnNUWJrBUXbQ.webp

Overall the results seem promising. In the ranging market of 2021 we lost about 20%, which can be easily avoided with a stop-loss. The best case was the bullish market of 2020 where we made more than 200%. For a long-only strategy its performance is not so bad even for ranging or bearish markets. If we implement short selling and also put in place stop-loss conditions or any other risk management strategy, it has great potential as a consistently profitable strategy. In the end, please make sure to backtest it thoroughly for different periods and different assets to make sure its performance is what you expect. Also please make sure to go over the code carefully so that there are no mistakes. Always be careful, and try with a small account for real trading, so you make sure the real-life performance is good enough and you don't risk losing your money.

---

5. Conclusion

The **DecisionTree_EMA_Crossover_Strategy** represents a hybrid approach that integrates machine learning with traditional technical analysis. By employing technical indicators such as EMA, RSI, and MACD, the strategy gathers rich features that are fed into a decision tree classifier. The decision tree uses well-established splitting criteria—grounded in entropy, information gain, or the Gini Index—to predict future price movements. Coupled

with the EMA crossover condition, this strategy aims to enhance trade execution by confirming machine-generated signals with trend-based indicators. As I mentioned before, you can make it even better adding short selling and implementing a simple risk management strategy like a stop-loss and end up with a very profitable trading bot that makes you money consistently.

This comprehensive overview provides both the theoretical background and the practical implementation details, offering a robust framework for adapting machine learning to dynamic trading environments. I hope you find it useful and I would also appreciate your ideas and comments if you have any.

Forecasting Bitcoin Autocorrelation with 74% Directional Accuracy using LSTMs

Financial time series, like Bitcoin prices, are notoriously complex and volatile. While directly predicting price is challenging, analyzing and predicting underlying statistical properties can offer valuable insights. This article walks through a Python implementation that builds, trains, and evaluates a Long Short-Term Memory (LSTM) neural network to forecast the **rolling autocorrelation** of Bitcoin's closing price. Autocorrelation measures the persistence of trends, and predicting it could potentially inform trading strategies or market analysis.

We'll cover fetching data, calculating the target feature, preparing data for the LSTM, building and training the model with regularization, and finally evaluating its predictive performance.

1. Setting the Stage: Imports and Parameters

First, we import the necessary libraries: `numpy` and `pandas` for data manipulation, `yfinance` to fetch market data, `matplotlib` for plotting, `sklearn` for evaluation metrics and scaling (optional), `math` for calculations, and `tensorflow.keras` for building the LSTM model.

Python

```
import numpy as np
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from math import sqrt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, BatchNormalization
```

```
from tensorflow.keras.callbacks import EarlyStopping
import datetime
```

We then define key parameters for data fetching, feature calculation, and the LSTM model:

Python

```
# Data and Feature Parameters
ticker = 'BTC-USD'
start_date = '2023-01-01'
end_date = datetime.datetime.now().strftime('%Y-%m-%d')
rolling_window = 30 # Window for calculating autocorrelation
lag = 1             # Lag for autocorrelation (day-over-day)

# Model Hyperparameters
num_lags = 90       # How many past autocorrelation values to use as input
train_test_split = 0.80 # 80% for training, 20% for testing
num_neurons_in_hidden_layers = 128 # LSTM layer size
num_epochs = 100    # Max training epochs
batch_size = 20     # Samples per gradient update
dropout_rate = 0.1  # Regularization rate
```

2. Data Acquisition and Feature Engineering

We use `yfinance` to download historical Bitcoin price data.

Python

```
print(f"Fetching {ticker} data from {start_date} to {end_date}...")
data = yf.download(ticker, start=start_date, end=end_date)
# Clean up potential multi-level columns from yfinance
if isinstance(data.columns, pd.MultiIndex):
    data.columns = data.columns.droplevel(1)
data = data['Close'] # We only need closing prices
data = data.dropna()
print(f"Data fetched successfully. Shape: {data.shape}")
```

The core feature we want to predict is the rolling autocorrelation. This measures how correlated the price change on one day is with the price change on the previous day, calculated over the specified `rolling_window`.

Python

```
print(f"Calculating {rolling_window}-day rolling autocorrelation
(lag={lag})...")
rolling_autocorr_series = data.rolling(
    window=rolling_window
).apply(lambda x: x.autocorr(lag=lag), raw=False) # Use pandas Series method

rolling_autocorr = rolling_autocorr_series.dropna().values # Drop initial
```

```
NaNs
rolling_autocorr = np.reshape(rolling_autocorr, (-1)) # Ensure 1D shape
print(f"Rolling autocorrelation calculated. Shape: {rolling_autocorr.shape}")
```

Note: We use `raw=False` to ensure the `apply` function receives a pandas Series, which has the `.autocorr()` method.

3. Preparing Data for the LSTM

LSTMs require input data in a specific format: sequences of past observations (features) paired with the next observation (target). We define a helper function `data_preprocessing` for this:

Python

```
def data_preprocessing(data_series, n_lags, train_split_ratio):
    """
    Prepares time series data into lags for supervised learning and splits.
    """
    X, y = [], []
    # Create sequences: Use 'n_lags' points to predict the next point
    for i in range(n_lags, len(data_series)):
        X.append(data_series[i-n_lags:i])
        y.append(data_series[i])
    X, y = np.array(X), np.array(y)

    # Split into training and testing sets
    split_index = int(len(X) * train_split_ratio)
    x_train = X[:split_index]
    y_train = y[:split_index]
    x_test = X[split_index:]
    y_test = y[split_index:]

    print(f"Data shapes: X_train={x_train.shape}, y_train={y_train.shape},
X_test={x_test.shape}, y_test={y_test.shape}")
    return x_train, y_train, x_test, y_test

# Create the datasets
x_train, y_train, x_test, y_test = data_preprocessing(
    rolling_autocorr, num_lags, train_test_split
)
```

This function iterates through the autocorrelation series, creating input sequences (`X`) of length `num_lags` and corresponding target values (`y`). It then splits these into training and testing sets.

LSTMs expect a 3D input shape: `(samples, timesteps, features)`.
Our `timesteps` dimension is `num_lags`, and we have 1 feature (the autocorrelation value).

Python

```
# Reshape Input for LSTM [samples, time steps, features]
x_train = x_train.reshape((-1, num_lags, 1))
x_test = x_test.reshape((-1, num_lags, 1))
print(f"Data reshaped for LSTM: x_train={x_train.shape},
x_test={x_test.shape}")
```

4. Building the LSTM Model with Regularization

We use Keras' `Sequential` API to define the model architecture. Key components include:

- `LSTM` layer: The core recurrent layer that learns temporal dependencies.
- `BatchNormalization`: Normalizes activations between layers, often leading to faster and more stable training.
- `Dropout`: Randomly sets a fraction (`dropout_rate`) of input units to 0 during training, helping prevent overfitting.
- `Dense` layer: A standard fully connected layer with one output neuron for our single predicted value.

Python

```
print("Building LSTM model...")
model = Sequential()
model.add(LSTM(units=num_neurons_in_hidden_layers, input_shape=(num_lags,
1)))
model.add(BatchNormalization()) # Regularization / Stability
model.add(Dropout(dropout_rate)) # Regularization
model.add(Dense(units=1))       # Output layer

# Compile: Define loss function and optimizer
model.compile(loss='mean_squared_error', optimizer='adam')
model.summary() # Display model structure
```

5. Training the Model with Early Stopping

To prevent overfitting and avoid unnecessary training time, we use `EarlyStopping`. This callback monitors a specified metric (here, the training `loss`) and stops training if it doesn't improve for a set number of epochs (`patience`). `restore_best_weights=True` ensures the model weights from the best epoch are kept.

Python

```
# Early stopping implementation
early_stopping = EarlyStopping(monitor='loss', patience=15,
                               restore_best_weights=True, verbose=1)

print("Training model...")
history = model.fit(x_train, y_train,
```

```
                    epochs=num_epochs,
                    batch_size=batch_size,
                    callbacks=[early_stopping],
                    verbose=1,
                    shuffle=False) # Keep temporal order if needed

print("Training finished.")
if early_stopping.stopped_epoch > 0:
    print(f"Early stopping triggered at epoch {early_stopping.stopped_epoch +
1}")
```

Note: Using `shuffle=False` is often recommended for time series to maintain temporal sequence, although its impact might be less critical when using long input sequences (`num_lags`).

6. Prediction and Evaluation

With the model trained, we generate predictions on both the training data (in-sample) and the unseen test data (out-of-sample).

Python

```
print("Predicting...")
y_predicted_train = model.predict(x_train).flatten()
y_predicted_test = model.predict(x_test).flatten()

# Prepare actual values (flatten)
y_train_flat = y_train.flatten()
y_test_flat = y_test.flatten()
```

We evaluate performance using several metrics:

- **RMSE (Root Mean Squared Error):** Measures the average magnitude of prediction errors. Lower is better.
- **Correlation:** Measures how well the predicted values track the actual values (ranging from -1 to +1). Higher (closer to 1) is better.
- **Directional Accuracy:** Measures the percentage of times the model correctly predicted whether the autocorrelation would increase or decrease compared to the previous day. Higher is better (> 50% suggests predictive ability).

Python

```
print("Evaluating performance...")
# Calculate Metrics
rmse_train = sqrt(mean_squared_error(y_train_flat, y_predicted_train))
rmse_test = sqrt(mean_squared_error(y_test_flat, y_predicted_test))

# (Assuming calculate_directional_accuracy function is defined as above)
accuracy_train = calculate_directional_accuracy(y_train_flat,
```

```
y_predicted_train)
accuracy_test = calculate_directional_accuracy(y_test_flat, y_predicted_test)

min_len_train = min(len(y_train_flat), len(y_predicted_train))
min_len_test = min(len(y_test_flat), len(y_predicted_test))
correlation_train = np.corrcoef(y_train_flat[:min_len_train],
y_predicted_train[:min_len_train])[0, 1]
correlation_test = np.corrcoef(y_test_flat[:min_len_test],
y_predicted_test[:min_len_test])[0, 1]

# Print Results
print("\n--- Results ---")
# ... (print statements for metrics) ...
print("---------------\n")
```

Comparing the test metrics to the train metrics is crucial. If test performance is significantly worse, it indicates overfitting. Similar performance suggests the model generalizes well.

6. Analysis of Results

The evaluation metrics provide quantitative insights into the model's performance:

```
--- Results ---
Directional Accuracy Train = 72.96 %
Directional Accuracy Test  = 73.61 %
RMSE Train = 0.10346005
RMSE Test  = 0.07769025
Correlation In-Sample Predicted/Train = 0.971
Correlation Out-of-Sample Predicted/Test = 0.967
---------------
```

Let's break down what these numbers tell us:

- **Correlation (Train: 0.971, Test: 0.967):** These are exceptionally high correlation coefficients, very close to 1.0. This indicates that the model's predictions track the actual movements (ups and downs, general shape) of the rolling autocorrelation extremely well, both on the data it was trained on and, more importantly, on the unseen test data. The minimal drop between train and test correlation signifies excellent generalization.

- **RMSE (Train: 0.103, Test: 0.078):** The Root Mean Squared Error measures the typical magnitude of the prediction error. Given that autocorrelation ranges from -1 to +1, these RMSE values are relatively low. Crucially, the **Test RMSE is significantly *lower* than the Train RMSE**. This is a strong positive sign, suggesting that the regularization techniques (Batch Normalization, Dropout, and especially Early Stopping) were highly effective in preventing overfitting. The model performs even better on unseen data according to this metric.

- **Directional Accuracy (Train: 72.96%, Test: 73.61%):** Both values are well above 50%, indicating the model is considerably better than random chance at predicting whether the autocorrelation will *increase* or *decrease* in the next time step. Similar to RMSE, the test accuracy is slightly higher than the train accuracy, further reinforcing the conclusion that the model generalizes well.

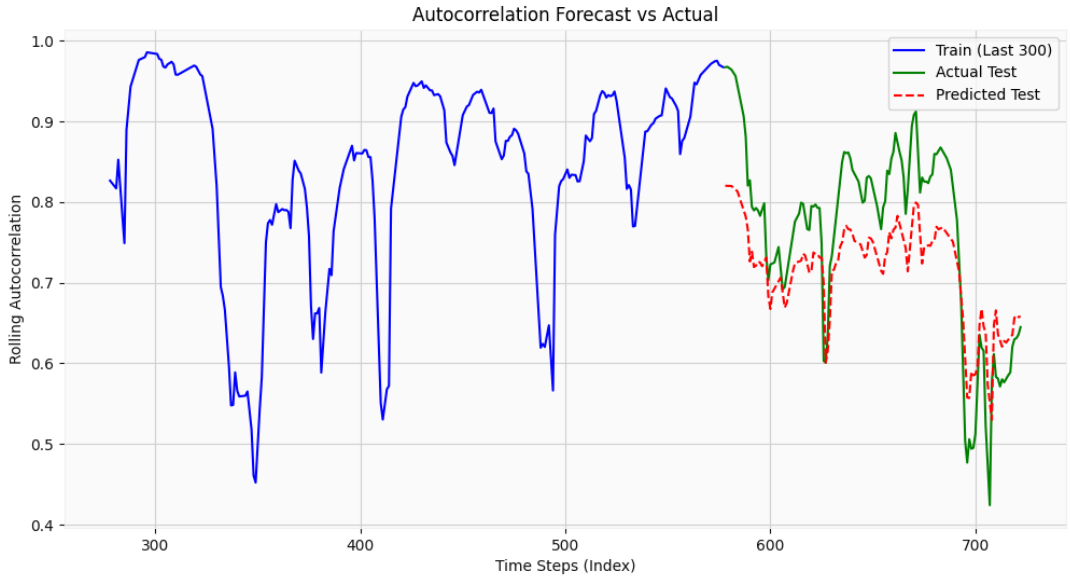
Synthesis: Overall, these metrics paint a very positive picture. The LSTM model learned to predict the one-step-ahead 30-day rolling autocorrelation with high fidelity (high correlation), relatively low error magnitude (low RMSE), and good directional correctness. Most importantly, the model demonstrates excellent generalization to unseen test data, avoiding the common pitfall of overfitting.

7. Visualizing the Forecast

While metrics provide quantitative scores, a visual inspection helps confirm the model's behavior.

Python

```
print("Plotting results...")
# (Assuming plot_train_test_values function is defined as above)
plot_train_test_values(n_train_plot=300, n_test_plot=len(y_test_flat),
                       y_train=y_train_flat,
                       y_test=y_test_flat,
                       y_predicted=y_predicted_test)
```



Plot Interpretation:

The plot visually confirms the strong performance indicated by the metrics.

- The red dashed line (Predicted Test values) closely follows the overall pattern and major fluctuations of the green line (Actual Test values).
- This visual alignment corroborates the high correlation score (0.967).
- While the prediction captures the general trend and turning points well, it doesn't perfectly match every peak and trough, which is expected and reflected in the non-zero RMSE (0.078). The predictions appear slightly smoother in some sections compared to the actual data.

This visual confirmation reinforces our confidence that the model has successfully learned the underlying short-term dynamics of the rolling autocorrelation series in this dataset.

Conclusion

This article demonstrated the complete workflow for building, training, and evaluating an LSTM model to forecast the rolling autocorrelation of Bitcoin prices. Key steps included fetching data, calculating the autocorrelation feature, preparing sequences for the LSTM, defining a regularized model architecture, training with early stopping, and evaluating using relevant metrics like RMSE, correlation, and directional accuracy.

While this model predicts a statistical feature rather than price directly, understanding and forecasting market persistence through autocorrelation could be a valuable component in developing more sophisticated trading algorithms or market analysis tools. Further work could involve hyperparameter tuning, exploring different model architectures, or integrating these predictions into a full backtesting framework like `backtrader`.

Market Regime Detection using Hidden Markov Models

This article explores a Python script that leverages Hidden Markov Models (HMMs) to identify distinct market regimes (specifically strong bull and strong bear phases) within financial time series data. It then utilizes the `backtrader` library to visualize these regime shifts on a price chart.

Core Concepts:

1. **Hidden Markov Models (HMMs):** HMMs are statistical models assuming a system transitions through a sequence of *unobservable* ("hidden") states. Each state has a probability distribution governing the *observable* outputs (or features). In finance, we can think of market regimes (bull, bear, ranging) as hidden states, and price movements (like returns and volatility) as observable features.
2. **Backtrader:** A popular Python framework for backtesting trading strategies and creating financial visualizations. It handles data loading, indicator calculations, strategy logic, and plotting.
3. **Market Regimes:** Distinct periods in the market characterized by different price behavior (e.g., strong upward trend, sharp downward trend, low-volatility sideways

movement). Identifying the current regime can be crucial for adjusting trading strategies.

Prerequisites:

You'll need the following Python libraries installed:

Bash

```
pip install backtrader yfinance numpy pandas hmmlearn matplotlib
```

Code Breakdown

Let's dissect the provided script section by section.

1. Imports:

Python

```
import backtrader as bt
import yfinance as yf
import numpy as np
import pandas as pd
import warnings
from hmmlearn import hmm
import matplotlib.pyplot as plt
# Optional: Configure matplotlib backend if needed
# %matplotlib qt5

warnings.filterwarnings("ignore") # Suppress common warnings
```

- **`backtrader (bt)`:** The core backtesting and plotting engine.
- **`yfinance (yf)`:** Used to download historical stock/crypto data from Yahoo Finance.
- **`numpy (np)`:** For numerical operations, especially array manipulations.
- **`pandas (pd)`:** For data manipulation using DataFrames.
- **`warnings`:** To control how warnings are handled (here, they are suppressed).
- **`hmmlearn.hmm`:** Provides the `GaussianHMM` class for implementing Hidden Markov Models with Gaussian emissions.
- **`matplotlib.pyplot (plt)`:** Used by `backtrader` (and potentially directly) for plotting.

2. HMM Training and State Identification (`train_hmm_and_identify_states`):

This is the heart of the regime detection logic.

Python

```
# --- MODIFICATIONS ONLY WITHIN THIS FUNCTION ---
def train_hmm_and_identify_states(df, n_states=5, n_iter=500, tol=1e-4,
vol_window=20):
    """
    Train an HMM on [Log Return, Volatility of Log Return], label each bar
with its state,
    and identify strong/weak bull & bear plus ranging regimes based on mean
log return.
    # ... (docstring continues) ...
    """
    df_hmm = df.copy() # Work on a copy to avoid modifying the original
DataFrame

    # --- Feature Calculation using Log Returns ---
    # Log returns are often preferred in finance as they are additive over
time
    # and approximate percentage changes for small values.
    df_hmm['Log Return'] = np.log(df_hmm['Close'] / df_hmm['Close'].shift(1))
    df_hmm['Log Return'].fillna(0, inplace=True) # Handle the first NaN value

    # Calculate rolling standard deviation of log returns as a measure of
volatility
    df_hmm['Volatility'] = df_hmm['Log Return'].rolling(vol_window).std()
    df_hmm['Volatility'].fillna(0, inplace=True) # Handle initial NaNs from
rolling window

    # --- Select features for HMM ---
    # The HMM will learn hidden states based on these observable features.
    # More features could potentially improve state differentiation.
    X = df_hmm[['Log Return', 'Volatility']].values

    # Handle potential numerical issues before fitting
    if np.any(np.isnan(X)) or np.any(np.isinf(X)):
        print("Warning: NaNs or Infs detected in HMM features. Replacing with
0.")
        X = np.nan_to_num(X, nan=0.0, posinf=0.0, neginf=0.0)

    # --- HMM Training ---
    # GaussianHMM assumes the features within each hidden state follow a
Gaussian distribution.
    # 'n_components': The number of hidden states to find (a key tuning
parameter).
    # 'covariance_type="diag"': Assumes features are independent within a
state (simpler, less prone to overfitting).
    # 'n_iter', 'tol': Control the convergence of the training algorithm.
    model = hmm.GaussianHMM(
        n_components=n_states,
```

```
        covariance_type='diag',
        n_iter=n_iter,
        tol=tol,
        random_state=42, # For reproducibility
        verbose=False
    )

    print(f"\nFitting HMM with {n_states} states...")
    try:
        # Fit the HMM model to the feature data (X)
        with warnings.catch_warnings(): # Suppress specific warnings during
fitting
            warnings.filterwarnings("ignore", category=DeprecationWarning)
            warnings.filterwarnings("ignore", category=RuntimeWarning)
            model.fit(X)
    except ValueError as e:
        print(f"Error fitting HMM: {e}")
        print("Check input data X for issues.")
        raise e

    if not model.monitor_.converged:
        print(f"Warning: HMM did not converge after {n_iter} iterations.")

    # Predict the most likely hidden state for each data point
    states = model.predict(X)
    df_hmm['HMM_State'] = states # Add the predicted states back to the
DataFrame

    # --- State Interpretation ---
    # Analyze the characteristics of each predicted state
    stats = []
    for i in range(n_states):
        mask = (states == i)
        if mask.sum() == 0: # Check if a state was even predicted
            print(f"Warning: State {i} was not predicted for any data
point.")
            continue
        # Calculate average log return and volatility for data points
belonging to this state
        stats.append({
            'State': i,
            'Mean Log Return': df_hmm.loc[mask, 'Log Return'].mean(),
            'Mean Volatility': df_hmm.loc[mask, 'Volatility'].mean(),
            'Count': mask.sum() # How many data points belong to this state
        })

    if not stats:
        raise ValueError("HMM training resulted in no predictable states.")
```

```
    # Sort states by their average log return (descending)
    # Assumption: Highest mean return = Strong Bull, Lowest mean return =
Strong Bear
    stats_df = pd.DataFrame(stats).sort_values('Mean Log Return',
ascending=False).reset_index(drop=True)

    print("\nHMM State Summary (sorted by Mean Log Return):")
    print(stats_df.to_string(index=False, float_format='{:.6f}'.format))

    # Assign regimes based on sorted order (assuming 5 states initially)
    state_indices = stats_df['State'].tolist()
    s_bull_strong = -1 # Initialize with invalid index
    s_bear_strong = -1

    # Adjust assignment based on how many distinct states were actually found
    if len(state_indices) > 0:
        s_bull_strong = state_indices[0]      # State with highest mean log
return
        s_bear_strong = state_indices[-1]     # State with lowest mean log
return
    # (The code handles cases with < 5 states by only assigning strong
bull/bear)

    print(f"\nRegime mapping (based on Mean Log Return sort):")
    print(f"  Strong Bull State = {s_bull_strong} (Highest Mean Log Return)")
    print(f"  Strong Bear State = {s_bear_strong} (Lowest Mean Log Return)")

    # Basic check for valid state assignment
    if s_bull_strong < 0 or s_bear_strong < 0:
          print("\nError: Could not reliably assign Strong Bull or Strong
Bear state index.")
          # The indicator initialization will later catch these invalid
indices

    print("\nReturning states for Strong Bull and Strong Bear signals.")
    # Return the DataFrame with HMM states and the identified indices for
strong bull/bear
    return df_hmm, s_bull_strong, s_bear_strong
# --- END OF MODIFICATIONS ---
```

- **Feature Engineering:** Calculates log returns and rolling volatility of log returns. These serve as the observable inputs for the HMM.
- **HMM Instantiation:** Creates a `GaussianHMM` model. `n_states=5` is a crucial parameter – it dictates how many distinct market patterns the model should try to find.

- **Training:** The `model.fit(X)` method trains the HMM using the Expectation-Maximization algorithm to find the parameters (transition probabilities between states, emission probabilities for each state) that best explain the observed feature data (`X`).
- **State Prediction:** `model.predict(X)` determines the most likely hidden state for each time step based on the trained model and the observed features.
- **State Interpretation:** After training, the script analyzes the average log return and volatility associated with each identified state. It sorts the states based on mean log return, assuming the state with the highest mean return corresponds to a "Strong Bull" regime and the state with the lowest mean return corresponds to a "Strong Bear" regime.
- **Return Values:** The function returns the original DataFrame augmented with the `HMM_State` column, and the integer indices corresponding to the identified strong bull and strong bear states.

3. Custom Backtrader Data Feed (`HMMData`):

Python

```
class HMMData(bt.feeds.PandasData):
    """Custom PandasData that carries the HMM_State column through as
`hmm_state`."""
    lines = ('hmm_state',) # Declare the new data line
    params = (
        # Map standard OHLCV columns
        ('datetime', None), # Use index for datetime
        ('open', 'Open'),
        ('high', 'High'),
        ('low', 'Low'),
        ('close', 'Close'),
        ('volume', 'Volume'),
        ('openinterest', None), # Not used here
        # Map our custom column 'HMM_State' from the DataFrame to the
'hmm_state' line
        ('hmm_state', 'HMM_State'),
    )
```

- This class inherits from `bt.feeds.PandasData`.
- It tells `backtrader` how to read the Pandas DataFrame prepared earlier.
- Crucially, it adds a custom data line `hmm_state` and maps it to the `HMM_State` column created by the HMM function. This makes the HMM state available within `backtrader` strategies and indicators.

4. Custom Backtrader Indicator (`HMMRegimeStartSignal`):

Python

```
class HMMRegimeStartSignal(bt.Indicator):
    """
    Signals the first bar of each new strong bull or strong bear regime
    by comparing the current HMM state to the prior bar.
    """
    lines = ('bull_start', 'bear_start',) # Output lines for signals
    params = (
        ('bull_state_idx', None), # Parameter to receive the strong bull
state index
        ('bear_state_idx', None), # Parameter to receive the strong bear
state index
    )
    plotinfo = dict(subplot=False) # Plot directly on the price chart
    plotlines = dict(
        # Define how the signals should be plotted (green up triangles, red
down triangles)
        bull_start=dict(marker='^', markersize=8, color='green',
linestyle='None'),
        bear_start=dict(marker='v', markersize=8, color='red',
linestyle='None'),
    )

    def __init__(self):
        # Validate that valid state indices were passed from the main script
        if self.p.bull_state_idx is None or self.p.bear_state_idx is None or
\
           self.p.bull_state_idx < 0 or self.p.bear_state_idx < 0:
            raise ValueError("Must pass valid non-negative bull_state_idx and
bear_state_idx to HMMRegimeStartSignal")
        # Access the custom hmm_state line from the data feed
        self.hmm_state = self.data.hmm_state

    def next(self):
        # Called for each bar of data (once enough data is available)
        if len(self.data) < 2: # Need at least two bars to compare current
and previous state
            return

        # Default signal values to NaN (no signal)
        self.lines.bull_start[0] = float('nan')
        self.lines.bear_start[0] = float('nan')

        # Get current and previous HMM state
        curr = int(self.data.hmm_state[0])
        prev = int(self.data.hmm_state[-1])
        b = self.p.bull_state_idx # Convenience alias for bull state index
        r = self.p.bear_state_idx # Convenience alias for bear state index
(renamed from 'r' for clarity)
```

```
        # --- Signal Logic ---
        # Strong bull entry: Current state is strong bull, previous was not.
        if curr == b and prev != b:
            # Place a green marker slightly below the low of the current bar
            self.lines.bull_start[0] = self.data.low[0] * 0.99

        # Exit strong bull: Previous state was strong bull, current is not.
        # This is treated as a potential sell/bearish signal.
        elif prev == b and curr != b:
            # Place a red marker slightly above the high of the current bar
            self.lines.bear_start[0] = self.data.high[0] * 1.01

        # Strong bear entry: Current state is strong bear, previous was not.
        elif curr == r and prev != r:
            # Place a red marker slightly above the high of the current bar
            self.lines.bear_start[0] = self.data.high[0] * 1.01
```

- This class inherits from `bt.Indicator`.
- It takes the identified `bull_state_idx` and `bear_state_idx` as parameters.
- The `__init__` method validates these parameters and gets access to the `hmm_state` data line.
- The `next` method contains the core logic:
    - It compares the `hmm_state` of the *current* bar (`[0]`) with the `hmm_state` of the *previous* bar (`[-1]`).
    - It generates a `bull_start` signal (plots a green marker) only on the *first* bar where the state transitions *into* the `bull_state_idx`.
    - It generates a `bear_start` signal (plots a red marker) on the *first* bar where the state transitions *into* the `bear_state_idx` OR when the state transitions *out of* the `bull_state_idx`. This treats both entering a bear state and exiting a bull state as potentially bearish signals for visualization.
- The `plotinfo` and `plotlines` dictionaries configure how these signals appear on the chart.

5. Main Execution Block (`if __name__ == '__main__':`)

Python

```
if __name__ == '__main__':
    ticker, start, end = 'BTC-USD', '2022-01-01', '2023-12-31'

    print(f"\nDownloading {ticker} data from {start} to {end}...")
    df = yf.download(ticker, start=start, end=end, progress=False)
    if df.empty:
        raise ValueError(f"No data downloaded for {ticker}.")

    # Optional: Flatten MultiIndex columns if yfinance returns them
```

```
    if isinstance(df.columns, pd.MultiIndex):
        df.columns = df.columns.droplevel(1)

    # --- Run HMM ---
    # Call the function to train HMM and get the state-augmented data +
regime indices
    data_with_hmm, bull_state, bear_state = train_hmm_and_identify_states(df)

    # --- Backtrader Setup ---
    cerebro = bt.Cerebro(stdstats=False) # Create the main backtrader engine
instance

    # --- Add Data ---
    # Ensure DataFrame index is DatetimeIndex (usually true for yfinance)
    if not isinstance(data_with_hmm.index, pd.DatetimeIndex):
         data_with_hmm.index = pd.to_datetime(data_with_hmm.index)
    # Create the custom data feed using the HMM-augmented DataFrame
    data_feed = HMMData(dataname=data_with_hmm)
    cerebro.adddata(data_feed) # Add the data feed to Cerebro

    # --- Add Indicators ---
    # Add the custom HMM signal indicator, passing the identified state
indices
    cerebro.addindicator(HMMRegimeStartSignal,
                         bull_state_idx=bull_state,
                         bear_state_idx=bear_state)
    # Add standard Moving Average indicators for context
    cerebro.addindicator(bt.indicators.SimpleMovingAverage, period=30)
    cerebro.addindicator(bt.indicators.SimpleMovingAverage, period=90)

    # --- Run and Plot ---
    print("\n--- Running Cerebro (for plotting) ---")
    cerebro.run() # Run the engine (calculates indicators)

    # Configure plot appearance
    plt.rcParams['figure.figsize'] = (10, 6)
    plt.rcParams['figure.dpi'] = 100
    print("\n--- Generating Plot ---")
    # Generate the plot: includes price, volume, SMAs, and HMM signals
    cerebro.plot(style='line', volume=True, iplot=False) # iplot=False for
static plot
```

- **Data Download:** Fetches historical data for `BTC-USD` using `yfinance`.
- **HMM Training:** Calls the `train_hmm_and_identify_states` function.
- **Cerebro Initialization:** Creates a `backtrader` Cerebro engine.
- **Data Addition:** Adds the data (including HMM states) to Cerebro using the custom `HMMData` feed.

- **Indicator Addition:** Adds the custom `HMMRegimeStartSignal` indicator (providing it with the `bull_state` and `bear_state` indices) and two standard Simple Moving Averages (SMAs) for visual context.
- **Execution:** `cerebro.run()` processes the data and calculates the indicator values. Note that no trading *strategy* is added here; Cerebro is used primarily for its indicator calculation and plotting capabilities in this script.
- **Plotting:** `cerebro.plot()` generates the final chart, displaying the price, volume, SMAs, and the HMM regime start signals (green and red markers).

How it Works - Summary

1. Download historical price data (e.g., BTC-USD).
2. Calculate features relevant to market behavior (log returns, volatility).
3. Train a Gaussian Hidden Markov Model on these features to identify a predefined number of hidden market states (`n_states`).
4. Analyze the characteristics (mean return, mean volatility) of each state found by the HMM.
5. Designate the state with the highest average return as the “Strong Bull” regime and the state with the lowest average return as the “Strong Bear” regime.
6. Feed the price data and the corresponding predicted HMM state for each bar into `backtrader` using a custom data feed.
7. Use a custom `backtrader` indicator to detect when the market transitions *into* the strong bull state (plot green marker) or transitions *into* the strong bear state / *out of* the strong bull state (plot red marker).

8. Display the price chart with standard indicators (like SMAs) and the HMM regime transition markers overlaid.

This script provides a powerful way to visualize potential market regime shifts identified by an HMM, which could be a valuable input for discretionary trading or the development of regime-aware automated strategies.

Neural Networks with Kalman Filter for Trading

In quantitative finance, combining statistical filtering techniques with machine learning can provide robust insights into market dynamics. In this article, we explore two powerful tools—**Neural Networks** and the **Kalman Filter**—and show how they can be used together to predict the direction of asset price movements. We then outline a trading strategy that uses these predictions, backtests its performance, and compares it to a simple buy-and-hold approach.

---

1. Theoretical Background

1.1 Neural Networks

Neural networks are a class of machine learning models inspired by biological neural structures. They consist of layers of interconnected nodes (neurons) that transform inputs into outputs through a series of linear and nonlinear operations.

Feed-Forward Neural Network Model

A basic multilayer perceptron (MLP) can be mathematically described as follows:

1. **Input Layer:**
   The network receives an input vector: [= [x_1, x_2, , x_n]^T]
2. **Hidden Layers:**
   Each hidden layer performs a linear transformation followed by a nonlinear activation function (e.g., ReLU or sigmoid). For layer (l): [^{(l)} = (^{(l)} ^{(l-1)} + ^{(l)})]

   where:

   - (^{(l)}) is the weight matrix.
   - (^{(l)}) is the bias vector.
   - () is an activation function.
   - For (l = 1), (^{(0)} =).
3. **Output Layer:**
   The final layer computes the output: [= (^{(L)} ^{(L-1)} + ^{(L)})]

The softmax function is often used for classification tasks to convert raw scores into probabilities.

4. **Training via Backpropagation:**
   The network parameters ({^{(l)}, ^{(l)}}) are optimized by minimizing a loss function (L(,)) (e.g., cross-entropy for classification) using gradient descent: [-]

   where () represents all the parameters and () is the learning rate.

In our code, we use Python's `MLPClassifier` from scikit-learn, which implements a multilayer perceptron with hidden layers (in our case, with sizes 32 and 16 neurons) to predict the direction of asset price movements.

1.2 Kalman Filter

The Kalman filter is a recursive algorithm used for estimating the state of a dynamic system from noisy observations. It is especially useful in financial applications where price signals are noisy.

Kalman Filter Equations

The filter works in two main steps: **prediction** and **update**.

1. **Prediction Step:**
   - **State Prediction:** [*{k|k-1}* = {k-1|k-1}]
   - **Error Covariance Prediction:** [*{k|k-1}* = {k-1|k-1} ^T +]

   Here, () is the state transition model, and () is the process noise covariance.

2. **Update Step:**
   - **Kalman Gain:** [*k* = {k|k-1} ^T (_{k|k-1} ^T +)^{-1}]
   - **State Update:** [*{k|k}* = {k|k-1} + _k (*k* - {k|k-1})]
   - **Error Covariance Update:** [_{k|k} = (- *k*) {k|k-1}]

   In these equations, () is the observation model, () is the measurement noise covariance, and (_k) is the measurement at time (k).

In our code, we use a custom `KalmanFilter` class to smooth the price series. The filter produces two outputs: a **smoothed price** and an **estimated rate of change**, which serve as features for the neural network.

---

2. The Trading Strategy

The core idea is to predict the price direction for the next week (7 days) using the neural network. The target is defined as:

[=

]

Trading Signal Generation

- **Long Position (+1):**
  If the model predicts a 1, the strategy goes long by buying at the current close and selling 7 days later.
- **Short Position (-1):**
  If the model predicts -1, the strategy goes short by selling (or taking a short position) at the current close and buying back 7 days later.
- **No Trade (0):**
  If the model's confidence is below a threshold (e.g., 80%), the signal is set to 0, meaning no position is taken.

Backtesting the Strategy

For backtesting:

- **7-Day Returns:**
  The asset's 7-day return is computed as: [_t = - 1]
- **Strategy Return:**
  The trading return is given by: [_t = _t _t]

The backtest aggregates these returns, computes cumulative performance (equity curve), and then compares the strategy to a buy-and-hold approach.

---

3. Code Walkthrough

Below we break down key sections of the code, explaining how each component contributes to the overall strategy.

3.1 Data Acquisition and Preprocessing

```
from binance.client import Client
import pandas as pd
import numpy as np
import ta
```

```
# Download price data from Binance
client = Client()
pair = 'ETHUSDC'
data = pd.DataFrame(client.get_historical_klines(pair, '1d', '1 year ago'))
data.columns = ['timestamp', 'open', 'high', 'low', 'close', 'volume',
                'close_time', 'quote_asset_volume', 'trades',
                'taker_buy_base', 'taker_buy_quote', 'ignore']
data['timestamp'] = pd.to_datetime(data['timestamp'], unit='ms')
ohlcv_columns = ['open', 'high', 'low', 'close', 'volume']
data[ohlcv_columns] = data[ohlcv_columns].astype(float)
data.set_index('timestamp', inplace=True)

# Shift data to avoid lookahead bias in indicator calculations
data = data.shift()
```

Explanation:

- Data is fetched using the Binance API and converted into a DataFrame with proper datetime indexing.
- The `.shift()` function is used to avoid using current day data for calculations that would normally be computed using past data.

3.2 Smoothing with the Kalman Filter

```
from KalmanFilter import KalmanFilter
kf = KalmanFilter(delta_t=1, process_var=1e-7, measurement_var=1e-1)
data[['kalman_price', 'kalman_rate']] = kf.filter(data['close'])
```

Explanation:

- The Kalman filter is applied to the close price to produce a smoothed price and an estimated rate of change (velocity).
- These filtered outputs are later used as features for the neural network.

3.3 Rolling Neural Network Training and Prediction

```
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from tqdm import tqdm

# Features used by the neural network (in this case, the Kalman outputs)
features = ['kalman_price', 'kalman_rate']
rolling_window = 30
nn_predictions = []
nn_probabilities = []
actuals = []
prediction_dates = []

# Set up scaler and MLP neural network
scaler = StandardScaler()
```

```
mlp = MLPClassifier(hidden_layer_sizes=(32, 16), max_iter=500,
random_state=42)

# Rolling window loop: Train and predict
for i in tqdm(range(rolling_window, len(data) - 1)):
    # Training data for past window
    X_train = data[features].iloc[i - rolling_window:i]
    y_train = data['direction'].iloc[i - rolling_window:i]
    X_train_scaled = scaler.fit_transform(X_train)

    # Train the network on the rolling window
    mlp.fit(X_train_scaled, y_train)

    # Predict for the next interval
    X_next = data[features].iloc[i].values.reshape(1, -1)
    X_next_scaled = scaler.transform(X_next)
    nn_pred = mlp.predict(X_next_scaled)[0]
    nn_prob = mlp.predict_proba(X_next_scaled)[0]
    nn_predictions.append(nn_pred)
    nn_probabilities.append(nn_prob)

    # Save the actual direction and prediction time
    actuals.append(data['direction'].iloc[i])
    prediction_dates.append(data.index[i])
```

Explanation:

- **Rolling Window:** The neural network is retrained on a moving window (30 days) to adapt to recent market behavior.
- **Scaling:** Data is standardized using `StandardScaler` to ensure that features are on the same scale.
- **Prediction:** The network predicts the next interval's direction. The probabilities are stored for later confidence filtering.

Adjusting Predictions Based on Confidence

```
# Use probabilities to adjust predictions
for i in range(len(nn_predictions)):
    prob = nn_probabilities[i]
    nn_predictions[i] = -1 if prob[0] > prob[1] else 1

# Only accept predictions with high confidence (>= 80%)
for i in range(len(nn_predictions)):
    pred = nn_predictions[i]
    confidence = nn_probabilities[i][0] if pred == -1 else
nn_probabilities[i][1]
    if confidence < 0.8:
        nn_predictions[i] = 0
```

```
data['nn_predictions'] = 0
data.loc[prediction_dates, 'nn_predictions'] = nn_predictions
```

Explanation:

- The prediction is chosen based on the higher probability between -1 and 1.
- A confidence threshold is applied—if the probability is less than 80%, the model issues no trade (signal 0).

3.4 Performance Evaluation and 7-Day Return Calculation

```
# Calculate 7-day returns for the base asset
data['7d_return'] = (data['close'].shift(-7) / data['close']) - 1

# Calculate strategy returns based on NN predictions
data['nn_7d_return'] = data['nn_predictions'] * data['7d_return']

# Filter rows with valid predictions and returns
predicted_data = data[(data['nn_predictions'] != 0) &
(data['7d_return'].notna())]

# Print success rate of NN predictions
predictions = data[['nn_predictions', 'direction']][data['nn_predictions'] !=
0]
success_rate = np.where(predictions['nn_predictions'] ==
predictions['direction'], 1, 0).mean() * 100
print("Neural Network Success Rate: {:.2f}%".format(success_rate))
```

Explanation:

- **7-Day Returns:** The asset's return over the next 7 days is calculated.
- **Strategy Return:** The NN signal is multiplied by the 7-day return. A positive signal captures the asset return for a long position, and a negative signal inverses the return for a short position.
- **Success Rate:** The percentage of correct predictions is computed.

3.5 Constructing and Plotting the Equity Curves

NN Strategy Equity Curve

```
# Initialize an equity column and starting capital
data['nn_equity'] = np.nan
equity = 1.0   # Starting capital
i = 0

# Simulate non-overlapping trades (skip 8 days after each trade)
while i < len(data) - 7:
    idx_entry = data.index[i]
    data.at[idx_entry, 'nn_equity'] = equity
```

```
    signal = data['nn_predictions'].iloc[i]
    entry_price = data['close'].iloc[i]
    exit_price  = data['close'].iloc[i + 7]

    if signal == 1:
        trade_return = (exit_price - entry_price) / entry_price
    elif signal == -1:
        trade_return = (entry_price - exit_price) / entry_price
    else:
        trade_return = 0.0

    equity *= (1 + trade_return)
    idx_exit = data.index[i + 7]
    data.at[idx_exit, 'nn_equity'] = equity
    i += 8

# Fill missing equity values
data['nn_equity'].ffill(inplace=True)
data['nn_equity'].bfill(inplace=True)

# Convert equity to percent profit
data['nn_equity_pct'] = (data['nn_equity'] - 1.0) * 100
```

Explanation:

- **Trade Simulation:** The code simulates entering a trade when a signal is generated, holds the position for 7 days, and then updates the equity.
- **Non-Overlapping Trades:** After closing a trade, the index is advanced by 8 days to ensure trades do not overlap.
- **Equity Curve:** The cumulative equity is forward- and back-filled across the entire date range and then converted to percent profit.

Buy-and-Hold Equity Curve

```
# Buy and hold strategy: calculate daily returns and cumulative product
data['bh_return'] = data['close'].pct_change()
data['bh_equity'] = (1 + data['bh_return']).cumprod()
data['bh_equity_pct'] = (data['bh_equity'] - 1.0) * 100
```

Explanation:

- This simple benchmark strategy simulates buying the asset at the beginning and holding it throughout the period.
- The cumulative return is calculated by taking the cumulative product of daily returns.

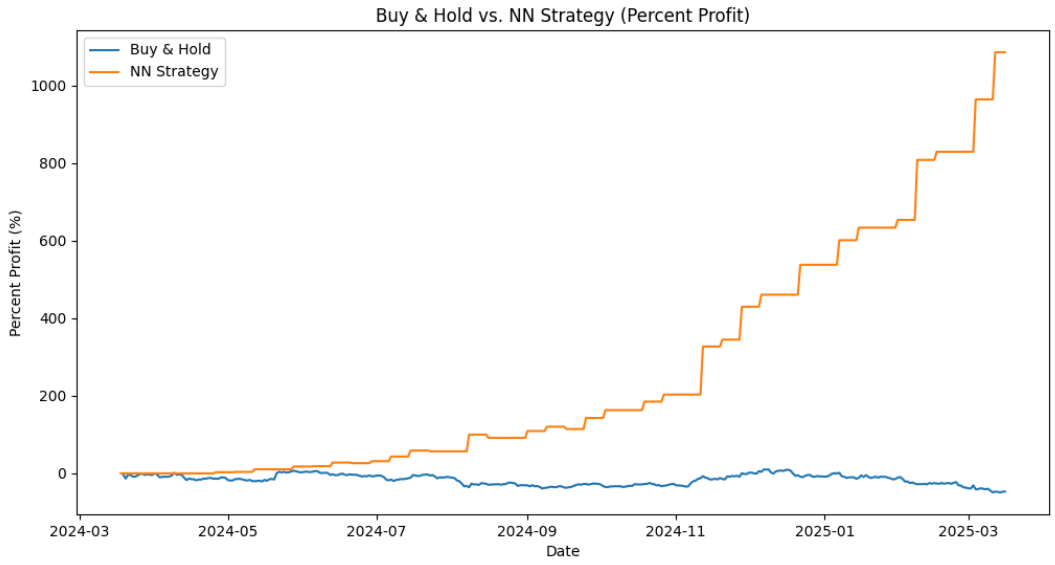
Plotting the Comparison

```
import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(12,6))
plt.plot(data.index, data['bh_equity_pct'], label='Buy & Hold')
plt.plot(data.index, data['nn_equity_pct'], label='NN Strategy')
plt.title('Buy & Hold vs. NN Strategy (Percent Profit)')
plt.xlabel('Date')
plt.ylabel('Percent Profit (%)')
plt.legend()
plt.show()
```

Explanation:

- The equity curves of the NN strategy and the buy-and-hold approach are plotted on the same time axis.
- The y-axis is in percent profit, allowing for an intuitive comparison of overall performance.



4. Conclusion

In this article, we explored how neural networks and the Kalman filter can be integrated into a trading strategy:

- **Neural Networks** provide a way to learn complex, nonlinear relationships from historical data, using layers of weighted transformations and activation functions.
- **Kalman Filters** help smooth out noisy price data and estimate underlying trends, producing additional features that can improve prediction accuracy.
- By training a neural network on a rolling window of past data and using its predictions to determine trading signals (long, short, or no trade), we can simulate a trading strategy that captures 7-day returns.

- The code further demonstrates how to backtest this strategy by constructing an equity curve, comparing it to a benchmark buy-and-hold strategy.

This framework is a starting point for further research and refinement. Future enhancements might include improved feature engineering, more sophisticated risk management, and alternative model architectures. As always, caution is advised when applying these techniques to live trading due to the challenges of market dynamics and overfitting.

Predicting Bitcoin's Weekly Moves with 68% Accuracy using Random Forests in Python

Predicting the direction of volatile assets like Bitcoin is a central challenge in quantitative finance. While daily noise can make short-term predictions resemble random walks, analyzing trends over slightly longer horizons, like a week, might offer more traction. This article details a Python-based approach using a Random Forest classifier and a rolling forecast methodology to predict whether Bitcoin's price will be higher or lower seven days from the present, leveraging a pre-selected set of technical indicators. We'll cover the theory, the implementation with code snippets, and how to interpret the results.

1. Theoretical Background

Before diving into the code, let's understand the core concepts:

a) Random Forest Classifier

- **Ensemble Learning:** Random Forest is an ensemble machine learning method primarily used for classification and regression. It operates by constructing a multitude of individual decision trees during training.
- **How it Works:**
    1. **Bagging (Bootstrap Aggregating):** It creates multiple random subsets of the original training data (with replacement). A separate decision tree is trained on each subset.
    2. **Feature Randomness:** When splitting a node in a decision tree, the algorithm considers only a random subset of the available features, rather than all of them. This decorrelates the trees.
    3. **Voting:** For classification, the final prediction is determined by a majority vote among all the individual trees in the forest. The class predicted by the most trees wins.
- **Advantages:**
    - Handles non-linear relationships between features and the target well.
    - Generally robust to overfitting compared to individual decision trees, especially when well-tuned.

    - Can handle high-dimensional data (many features).
    - Provides useful estimates of **Feature Importance**, indicating which features contributed most to the model's decisions.
- **Equations:** While the implementation is complex, the core idea relies on aggregating simple decision trees. The prediction () for an input (x) is often represented conceptually as: (= _{b=1}^{B}{ _b(x) })where B is the number of trees and (_b(x)) is the prediction of the (b^{th}) tree trained on a bootstrap sample and considering random feature subsets.

b) Feature Selection (Context)

This script assumes that a preliminary analysis has been performed to identify potentially predictive features. In our development process, Mutual Information scores were used to rank ~30 technical indicators based on their statistical relationship with the 1-day price direction. We will use the top 15 features identified in that analysis as inputs to our Random Forest model, assuming they might also hold relevance for the 7-day horizon.

c) Rolling Forecast Evaluation

- **Why Use It:** Financial markets evolve. A model trained on data from years ago might not perform well today. A simple train-test split doesn't capture this dynamic. A rolling forecast provides a more realistic simulation of how a model might perform when periodically retrained on recent data and used to predict the near future.
- **How it Works:**
    1. Define a fixed-size training window (e.g., the last 30 days).
    2. Train the model on the data within this window.
    3. Make a prediction for the target period (e.g., 7 days ahead).
    4. Slide the window forward by one time step (e.g., one day).
    5. Repeat steps 2-4 until the end of the dataset is reached.
    6. Evaluate the model based on the aggregated predictions made across all windows.

d) Classification Metrics

Since we're predicting direction (Up=1, Down=0), we use classification metrics:

- **Accuracy:** Overall percentage of correct predictions. Accuracy=TP+TN+FP+FNTP+TN
- **Precision (for class 1):** Of the times the model predicted 'Up', how often was it right? Minimizes False Positives (FP). Precision=TP+FPTP
- **Recall (Sensitivity, for class 1):** Of all the actual 'Up' movements, how many did the model catch? Minimizes False Negatives (FN). Recall=TP+FNTP
- **F1-Score (for class 1):** Harmonic mean of Precision and Recall, useful for imbalanced datasets. F1=2×Precision+RecallPrecision×Recall

- **AUC-ROC:** Area Under the Receiver Operating Characteristic Curve. Measures the model's ability to distinguish between classes across1 different probability thresholds (0.5 = random, 1.0 = perfect).
- **Confusion Matrix:** A table visualizing performance:

	Predicted Down (0)	**Predicted Up (1)**
Actual Down (0)	True Negative (TN)	False Positive(FP)
Actual Up (1)	False Negative(FN)	True Positive (TP)

2. Python Implementation Details

Let's walk through the key parts of the Python script.

a) Setup and Configuration

Import libraries and set up parameters. Critically, set `PREDICTION_HORIZON = 7` and define the `TRAINING_WINDOW_DAYS` and the list of `TOP_FEATURES` derived from previous analysis.

Python

```
#
============================================================================
=
# Imports
#
============================================================================
=
import pandas as pd
import numpy as np
import yfinance as yf
import talib # Make sure TA-Lib is installed
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, confusion_matrix,
ConfusionMatrixDisplay,
                             roc_auc_score)
import warnings
# ... (warnings configuration) ...

#
============================================================================
=
# Configuration
#
============================================================================
=
```

```
TICKER = 'BTC-USD'
START_DATE = '2021-01-01' # Needs enough data for rolling
END_DATE = None
INTERVAL = '1d' # Daily data

# --- Rolling Window Parameters ---
TRAINING_WINDOW_DAYS = 30 # Approx 1 month training window
PREDICTION_HORIZON = 7    # Predict direction 7 days ahead

# --- Feature Selection ---
# Using Top 15 features identified previously from MI analysis
TOP_FEATURES = [
    'ROC_10', 'STOCHRSI_d', 'ADX_14', 'STOCHRSI_k', 'RSI_14',
    'STOCH_k', 'ATR_14', 'EMA_20', 'STOCH_d', 'MACD',
    'ULTOSC', 'BB_upper', 'SAR', 'Open_Close', 'MACD_hist'
]

# --- Random Forest Model Parameters ---
N_ESTIMATORS = 150
MAX_DEPTH = 8
MIN_SAMPLES_LEAF = 5
CLASS_WEIGHT = 'balanced'
RANDOM_STATE = 42
```

b) Data Loading and Indicator Calculation

Standard functions using `yfinance` and `talib` are used to fetch OHLCV data and compute the full set of ~30 technical indicators.

Python

```
# Function definitions for load_data and calculate_indicators
# (Use the full function definitions from the previous script response)

# In main execution block:
data = load_data(TICKER, START_DATE, END_DATE, INTERVAL)
if data is not None:
    data_indicators = calculate_indicators(data.copy())
```

c) Target Variable and Feature Preparation

The 7-day target variable (1 if price is higher 7 days later, 0 otherwise) is created. The data is cleaned of NaNs, and only the `TOP_FEATURES` columns are selected into the `X_all_features` DataFrame, while the `Target` column becomes `Y_all`.

Python

```
# Function definition for create_target (horizon=PREDICTION_HORIZON)
# (Use the function definition from the previous script response)
```

```
# In main execution block:
data_target = create_target(data_indicators, horizon=PREDICTION_HORIZON)
data_processed = data_target.dropna()

available_features = [f for f in TOP_FEATURES if f in data_processed.columns]
# ... (Error handling if features are missing) ...

X_all_features = data_processed[available_features]
Y_all = data_processed['Target']
Dates_all = data_processed.index # Keep dates for plotting results
```

d) The Rolling Forecast Loop

This is the core logic change from a simple train/test split.

Python

```
# --- Rolling Forecast Loop ---
all_predictions = []
all_actuals = []
all_predict_dates = []
all_probabilities = []

start_index = TRAINING_WINDOW_DAYS
end_index = len(X_all_features) - PREDICTION_HORIZON

print(f"\nStarting rolling forecast from index {start_index} to {end_index-
1}...")

for i in range(start_index, end_index):
    # 1. Define window boundaries
    train_start_idx = i - TRAINING_WINDOW_DAYS
    train_end_idx = i
    predict_feature_idx = i
    actual_target_idx = i

    # 2. Extract current window data
    X_train_window = X_all_features.iloc[train_start_idx:train_end_idx]
    Y_train_window = Y_all.iloc[train_start_idx:train_end_idx]
    X_predict_point = X_all_features.iloc[[predict_feature_idx]]
    Y_actual_point = Y_all.iloc[actual_target_idx]

    # 3. Scale features WITHIN the loop
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train_window)
    X_predict_scaled = scaler.transform(X_predict_point)

    # 4. Build and Train Model WITHIN the loop
```

```
    rf_model = RandomForestClassifier(
        n_estimators=N_ESTIMATORS,
        max_depth=MAX_DEPTH,
        min_samples_leaf=MIN_SAMPLES_LEAF,
        random_state=RANDOM_STATE,
        n_jobs=-1,
        class_weight=CLASS_WEIGHT
    )
    rf_model.fit(X_train_scaled, Y_train_window)

    # 5. Predict and Store Results
    prediction = rf_model.predict(X_predict_scaled)[0]
    probability = rf_model.predict_proba(X_predict_scaled)[0, 1] # Robust
extraction might be needed here too

    all_predictions.append(prediction)
    all_actuals.append(Y_actual_point)
    all_probabilities.append(probability)
    all_predict_dates.append(Dates_all[actual_target_idx])

    # ... (Optional progress print) ...

print("Rolling forecast complete.")
```

Crucially, the `StandardScaler` and `RandomForestClassifier` are initialized and fitted inside the loop on each window's data.

e) Aggregated Evaluation

After the loop completes, the collected predictions and actual values are used to calculate the overall performance metrics.

Python

```
# --- Evaluate Aggregated Results ---
if not all_actuals:
    print("No predictions were made.")
else:
    print("\n--- Aggregated Rolling Forecast Metrics ---")
    accuracy = accuracy_score(all_actuals, all_predictions)
    precision = precision_score(all_actuals, all_predictions,
zero_division=0)
    recall = recall_score(all_actuals, all_predictions, zero_division=0)
    f1 = f1_score(all_actuals, all_predictions, zero_division=0)
    try:
        roc_auc = roc_auc_score(all_actuals, all_probabilities)
    except ValueError:
        roc_auc = float('nan')
        # ... (print warning) ...
```

```
    print(f"Accuracy:         {accuracy:.4f}")
    print(f"Precision (for 1):{precision:.4f}")
    # ... (print other metrics) ...

    # Baseline comparison
    majority_class_overall = Y_all.value_counts().idxmax()
    baseline_accuracy = accuracy_score(all_actuals, np.full(len(all_actuals),
majority_class_overall))
    print(f"\nBaseline Accuracy (...): {baseline_accuracy:.4f}")

    # Confusion Matrix Plotting
    print("\n--- Confusion Matrix (Aggregated Rolling Forecast) ---")
    cm = confusion_matrix(all_actuals, all_predictions)
    print(cm)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0, 1])
    # ... (Plotting code for CM) ...
    plt.show()

    # Optional: Plot actual vs predicted directions over time
    # ... (Plotting code for results_df) ...
    plt.show()
```

3. Results and Interpretation (Based on Your Last Run)

Your last run with this rolling Random Forest approach yielded:

- **Accuracy:** ~0.6793 (vs. Baseline ~0.5136)
- **Precision (Up):** ~0.6919
- **Recall (Up):** ~0.6772
- **F1-Score (Up):** ~0.6845
- **AUC-ROC:** ~0.7524
- **Confusion Matrix:** `[[122 57] / [ 61 128]]`

Interpretation:

These results show a clear improvement over random chance and the baseline of simply predicting the majority class. The model achieved ~68% accuracy in predicting the 7-day direction over the rolling test period. Precision and Recall are reasonably balanced (around 68-69%), indicating the model identifies 'Up' moves moderately well without excessively predicting 'Up' incorrectly. The AUC of ~0.75 suggests a decent discriminatory ability. While not perfect, these results indicate that the combination of selected features, the Random Forest model, and the rolling approach captured a statistically significant predictive signal in the historical data tested.

4. How to Use the Code

1. **Install Prerequisites:** Ensure Python, `pandas`, `numpy`, `yfinance`, `matplotlib`, `seaborn`, `scikit-learn`, and crucially, `TA-Lib` (C library + Python wrapper) are installed.
2. **Save:** Save the complete code as a Python file (e.g., `rolling_rf_btc.py`).
3. **Configure:** Modify settings like `TICKER`, `START_DATE`, `TRAINING_WINDOW_DAYS`, `PREDICTION_HORIZON`, or Random Forest parameters if desired.
4. **Run:** Execute from your terminal: `python rolling_rf_btc.py`. It will take some time as the model retrains repeatedly.
5. **Analyze:** Review the printed metrics and the confusion matrix plot. Compare accuracy to the baseline. Assess if the Precision/Recall/F1/AUC meet your requirements for considering a signal potentially useful.

5. Limitations and Conclusion

- **Historical Performance:** Success on past data doesn't guarantee future results. Markets change.
- **Not a Trading Strategy:** This analyzes predictive accuracy ONLY. It lacks entry/exit rules, risk management, cost simulation, etc.
- **Need for Tuning/Testing:** Results depend heavily on the chosen features, hyperparameters, and time period. Extensive testing and tuning are required for any real application.
- **Feature Stability:** The selected `TOP_FEATURES` might lose predictive power over time.

In conclusion, this script provides a robust framework for evaluating the predictive power of technical indicators for Bitcoin's weekly direction using a Random Forest model and a realistic rolling forecast method. The results achieved (~68% accuracy, ~0.75 AUC historically) demonstrate a potential edge worthy of further investigation, but require critical interpretation and significant further development before any practical trading application.

Trading Using Neural Networks

In this article, we explore the development of a trading strategy for Bitcoin using a neural network model and various technical indicators. By leveraging 15-minute interval data, we aim to predict short-term price movements and compare the returns from our strategy against a traditional buy-and-hold approach.

Introduction to Neural Networks

Neural networks are a class of machine learning models inspired by the structure and functioning of the human brain. They are designed to recognize patterns, make decisions, and learn from data through a process of training and optimization. Here's a brief overview

of how neural networks work and their significance in modern machine learning and artificial intelligence.

1. Basic Structure

A neural network consists of layers of interconnected nodes or neurons. The basic components are:

- **Input Layer:** The first layer that receives the raw data. Each neuron in this layer represents a feature or input variable.
- **Hidden Layers:** Intermediate layers between the input and output layers where computation and transformation of data occur. Neural networks can have multiple hidden layers, which allows them to learn complex patterns and features.
- **Output Layer:** The final layer that produces the prediction or classification result. The number of neurons in this layer corresponds to the number of possible outputs.

2. Neurons and Activation Functions

Each neuron in a neural network receives inputs, applies a weighted sum, and passes the result through an activation function. The activation function introduces non-linearity into the model, enabling it to learn complex patterns. Common activation functions include:

- **Sigmoid:** Maps inputs to a value between 0 and 1.
- **ReLU (Rectified Linear Unit):** Outputs the input directly if it is positive; otherwise, it outputs zero.
- **Tanh:** Maps inputs to a value between -1 and 1.

3. Training Neural Networks

Training a neural network involves adjusting its weights and biases to minimize the error between the predicted output and the actual output. This is typically done using:

- **Forward Propagation:** The process of passing input data through the network to obtain predictions.
- **Loss Function:** A measure of the difference between the predicted output and the actual output. Common loss functions include mean squared error (MSE) and cross-entropy loss.
- **Backpropagation:** An algorithm used to update the weights and biases based on the error. It involves computing the gradient of the loss function with respect to each weight using the chain rule and adjusting the weights to reduce the error.
- **Optimizer:** An algorithm that adjusts the weights to minimize the loss function. Popular optimizers include Stochastic Gradient Descent (SGD), Adam, and RMSprop.

4. Types of Neural Networks

- **Feedforward Neural Networks:** The simplest type, where connections between nodes do not form cycles. Used for straightforward prediction tasks.
- **Convolutional Neural Networks (CNNs):** Designed for processing grid-like data, such as images. They use convolutional layers to detect spatial hierarchies.
- **Recurrent Neural Networks (RNNs):** Suitable for sequential data, such as time series or natural language. They have connections that form cycles, allowing them to maintain context and memory.
- **Generative Adversarial Networks (GANs):** Consist of two networks—a generator and a discriminator—that compete against each other, used for generating realistic synthetic data.

Step 1: Downloading Bitcoin Price Data

We start by pulling 15-minute interval Bitcoin price data from Yahoo Finance using the yfinance library. The data spans the most recent month.

import yfinance as yf

btc_data = yf.download('BTC-USD', interval='15m', period='1mo')

Step 2: Calculating Technical Indicators

Next, we compute several key technical indicators using the TA-Lib library:

- **EMA_12**: 12-period Exponential Moving Average.
- **EMSD_12**: 12-period Exponential Moving Standard Deviation.
- **RSI_14**: 14-period Relative Strength Index.

These indicators serve as inputs to the neural network model.

```
import talib as ta
btc_data['EMA_12'] = ta.EMA(btc_data['Close'], timeperiod=12)
btc_data['EMSD_12'] = ta.STDDEV(btc_data['Close'], timeperiod=12)
btc_data['RSI_14'] = ta.RSI(btc_data['Close'], timeperiod=14)
btc_data.dropna(inplace=True)
```

Step 3: Preparing Input Features and Target

We standardize the features using MinMaxScaler and prepare the target variable as a binary outcome: whether the next period's close price is higher than the current period's close price.

```
from sklearn.preprocessing import MinMaxScaler
import numpy as np

features = btc_data[['EMA_12', 'EMSD_12', 'RSI_14']]
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_features = scaler.fit_transform(features)
```

```
btc_data['Target'] = np.where(btc_data['Close'].shift(-1) >
btc_data['Close'], 1, 0)
btc_data.dropna(inplace=True)
target = btc_data['Target']
```

Step 4: Building the Neural Network Model

The neural network is constructed using Keras, with the architecture consisting of an input layer, five hidden layers with ReLU activation, and an output layer using the softmax function. The model is trained using the Adam optimizer.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam

model = Sequential()
model.add(Dense(12, input_dim=3, activation='relu'))
model.add(Dense(40, activation='relu'))
model.add(Dense(30, activation='relu'))
model.add(Dense(20, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(5, activation='relu'))
model.add(Dense(4, activation='softmax'))

model.compile(optimizer=Adam(learning_rate=0.001),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(scaled_features, target, epochs=400, batch_size=500, verbose=2)
```

Step 5: Training and Evaluating the Model

We split the data into training and test sets, retrain the model on the training data, and then evaluate its performance on the test data. We calculate accuracy and generate a classification report.

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score

X_train, X_test, y_train, y_test = train_test_split(scaled_features, target,
test_size=0.2, random_state=42)
model.fit(X_train, y_train, epochs=400, batch_size=500, verbose=2)

scores = model.evaluate(X_test, y_test)
y_pred = np.argmax(model.predict(X_test), axis=1)

accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

print(f"Test Accuracy: {accuracy}")
```

```
print("Classification Report:")
print(report)
```

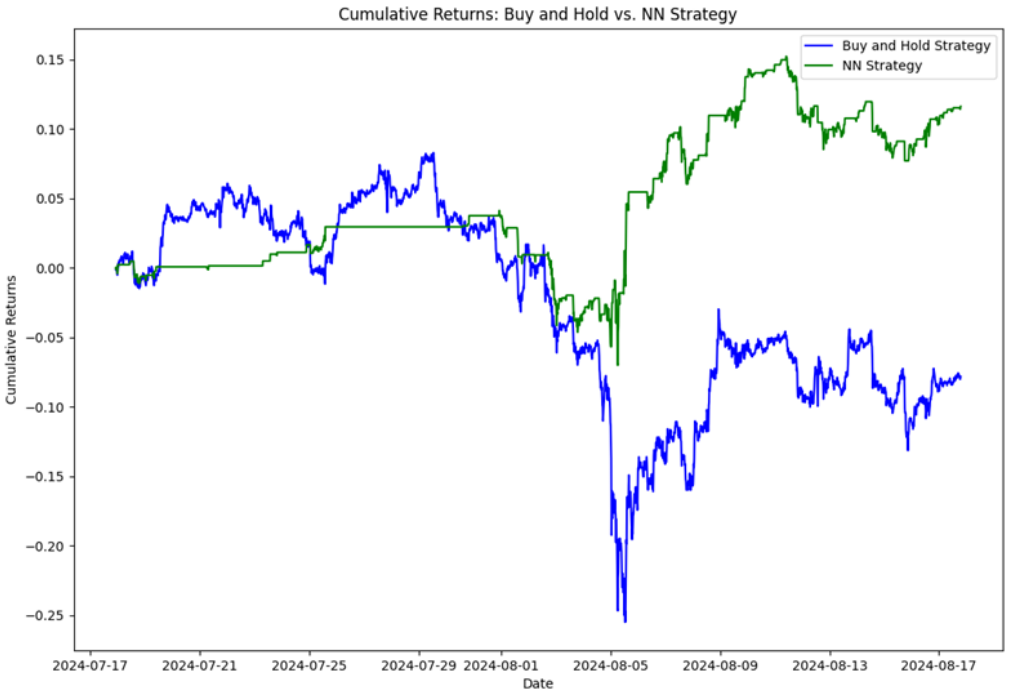
Step 6: Comparing Strategy Returns with Buy-and-Hold

To assess the effectiveness of our neural network strategy, we compare the cumulative returns from both the buy-and-hold strategy and our model's predictions.

- **Buy-and-Hold Returns**: Calculated as the cumulative sum of logarithmic returns.
- **Strategy Returns**: Determined by the model's predicted signals.

We then plot both cumulative returns on the same graph.

```
import matplotlib.pyplot as plt

btc_data['Buy_Hold_Returns'] = np.log(btc_data['Close'] /
btc_data['Close'].shift(1))
btc_data['Buy_Hold_Cumulative'] = btc_data['Buy_Hold_Returns'].cumsum()

btc_data['Signal'] = model.predict(scaled_features).argmax(axis=1)
btc_data['Strategy_Returns'] = btc_data['Signal'].shift(1) *
btc_data['Buy_Hold_Returns']
btc_data['Strategy_Cumulative'] = btc_data['Strategy_Returns'].cumsum()

plt.figure(figsize=(14, 7))
plt.plot(btc_data.index, btc_data['Buy_Hold_Cumulative'], label='Buy and Hold
Strategy', color='blue')
plt.plot(btc_data.index, btc_data['Strategy_Cumulative'], label='NN
Strategy', color='green')
plt.title('Cumulative Returns: Buy and Hold vs. NN Strategy')
plt.xlabel('Date')
plt.ylabel('Cumulative Returns')
plt.legend()
plt.show()
```



Conclusion

The graph comparing the cumulative returns of the buy-and-hold strategy with those of our neural network-based strategy reveals the potential of using machine learning for short-term trading. While the buy-and-hold strategy offers steady returns, the neural network model can potentially capture more significant price movements, leading to better overall performance during volatile periods.

This exercise demonstrates the power of combining technical analysis with machine learning to create trading strategies that adapt to market conditions. As always, further tuning and validation are essential before deploying such strategies in live trading environments.

What if Darwin Traded Crypto An Experiment with Evolutionary AI & Neural Nets

Algorithmic trading, the use of computer programs to execute trading strategies, has revolutionized financial markets. Designing profitable strategies, however, remains a significant challenge. It often involves navigating complex market dynamics, identifying predictive patterns, and managing risk effectively. Machine learning and optimization techniques offer powerful tools to tackle this complexity.

This article delves into one such approach: using an **Evolution Strategy (ES)**, a type of optimization algorithm inspired by natural evolution, to train a simple **Neural Network (NN)** based trading agent. We will explore the underlying theory of ES and NNs in this context, walk through a Python implementation using `yfinance` for Bitcoin data, and emphasize the importance of realistic backtesting with train/test splits.

Background Theory

1. Evolution Strategies (ES)

Evolution Strategies are a class of optimization algorithms belonging to the broader field of Evolutionary Computation. Unlike Genetic Algorithms (GAs) which often work with discrete representations (like binary strings) and rely heavily on crossover, ES typically operates directly on real-valued parameter vectors (like the weights of a neural network) and primarily uses mutation (often Gaussian noise) and selection to guide the search towards optimal solutions.

- **Core Idea:** ES maintains a "population" of candidate solutions (parameter vectors). In each generation (iteration), it creates new candidate solutions by adding random perturbations (mutations) to the current best solution(s). It then evaluates the "fitness" (performance) of these new solutions using an objective function (in our case, a trading simulation reward). Finally, it updates the central solution vector by taking a weighted average of the perturbations, where the weights are determined by the fitness scores of the corresponding perturbed solutions. Solutions that yield better fitness contribute more to the direction of the update.
- **Simplified ES Update:** A common, basic form of ES update rule for a parameter vector (or weight matrix) W can be expressed conceptually as:

  [W_{t+1}=W_t+_{k=1}^{N}R_k_k]

  Where:

  - Wt is the parameter vector at iteration t.
  - α is the learning rate (step size).
  - N is the population size.
  - σ is the standard deviation of the Gaussian noise (mutation strength).
  - ϵk is the random Gaussian noise vector added to create the kth population member (Wt+σϵk).
  - Rk is the fitness (reward) obtained by the kth population member, often normalized (e.g., converted to standard scores).

  This update essentially moves the current parameters Wt in a direction that is positively correlated with the perturbations that led to higher rewards.

- **Advantages:** ES can be very effective for optimizing parameters of complex, non-differentiable systems where gradients are hard or impossible to compute (like the overall profit of a multi-step trading simulation). It's a black-box optimization technique.

2. Neural Networks (NNs) for Policy Representation

In our agent, the neural network acts as the "brain" or the *policy*. It maps the current market *state* to a preferred *action*.

- **Function:** It's a function approximator. Given an input vector representing the market state, it outputs scores indicating the desirability of each possible action (Buy, Sell, Hold).
- **Simple Structure:** We use a basic feedforward network with one hidden layer:
    - Input Layer: Receives the state vector (e.g., recent price changes).
    - Hidden Layer: Performs a linear transformation (Input·Whidden+Biashidden) followed potentially by a non-linear activation (though our implementation uses an implicit linear activation here). This layer learns intermediate features.
    - Output Layer: Performs another linear transformation (HiddenOutput·Woutput) to produce the final action scores.
- **Parameters:** The network's behavior is determined by its weights (Whidden ,Woutput) and biases (Biashidden). These are the parameters that the Evolution Strategy optimizes.

3. Trading Agent Framework

We can frame the trading problem in terms similar to Reinforcement Learning (RL), although ES optimizes differently:

- **Agent:** The program making trading decisions.
- **Environment:** The financial market (Bitcoin price time series).
- **State (St):** A representation of the market at time t. Choosing a good state representation is crucial. Using raw prices can be problematic due to non-stationarity. Price *changes* or returns over a lookback window are often preferred, as used in our implementation.
- **Action (At):** The decision made by the agent at time t (e.g., Buy, Sell, Hold).
- **Reward (Rt):** A measure of how good the outcome was after taking actions. In ES applied to trading, the reward is typically sparse – calculated only at the *end* of a simulation episode (e.g., the total percentage profit/loss over the training period).

The Implemented Method: ES Optimizing NN Weights via Simulated Trading

Our approach uses the Evolution Strategy to directly optimize the weights of the neural network policy.

1. The ES generates variations (population members) of the current NN weights.
2. For each set of weights, the `_calculate_reward_on_train` function is called. This function simulates the agent trading over the entire **training dataset** using the NN with those specific weights to decide actions (Buy/Sell/Hold) at each step.
3. The simulation result (final percentage profit/loss on the training data) is returned as the fitness score (reward) for that set of weights.
4. The ES uses these rewards to update the central NN weights according to its update rule, aiming to find weights that maximize the simulated profit on the training data.

Implementation Details (Python)

Let's look at the key parts of the Python code (using the version with the train/test split).

1. Data Handling and Splitting

We fetch historical Bitcoin data using `yfinance` and then split it chronologically into training and testing sets. This ensures we train the agent on one period and evaluate it on a completely separate, later period.

Python

```
import yfinance as yf
import numpy as np
import pandas as pd

ticker = 'BTC-USD'
try:
    # Fetch 3 years data for a reasonable split
    df = yf.download(ticker, period='3y')
    if df.empty:
        raise ValueError(f"No data fetched for {ticker}.")
    print(f"Fetched {len(df)} rows of data for {ticker}")
    df = df.sort_index()
    all_prices = df['Close'].values
    all_dates = df.index
except Exception as e:
    print(f"Error fetching data: {e}")
    exit()

# Split data: 80% train, 20% test
test_size_percentage = 0.20
split_index = int(len(all_prices) * (1 - test_size_percentage))

train_prices = all_prices[:split_index]
test_prices = all_prices[split_index:]
train_dates = all_dates[:split_index]
test_dates = all_dates[split_index:]
```

```
print(f"Data split: {len(train_prices)} training samples, {len(test_prices)}
testing samples.")
```

Explanation: We get 3 years of daily closing prices for BTC-USD. We calculate an index (`split_index`) to divide the data, assigning the earlier 80% to `train_prices` and the later 20% to `test_prices`. Corresponding dates are also separated.

2. Neural Network Model (`SimpleModel`)

This class defines the structure and prediction logic of our simple neural network.

Python

```
class SimpleModel:
    """ A simple neural network model with one hidden layer. """
    def __init__(self, input_size, layer_size, output_size):
        # Initialize weights randomly with small values
        self.weights = [
            np.random.randn(input_size, layer_size) * 0.1,  # Input -> Hidden
            np.random.randn(layer_size, output_size) * 0.1, # Hidden ->
Output
            np.random.randn(1, layer_size) * 0.1            # Hidden layer
bias
        ]

    def predict(self, inputs):
        """ Makes a prediction based on the inputs and current weights. """
        if inputs.ndim == 1: inputs = inputs.reshape(1, -1) # Ensure input is
2D
        # Linear transformation for hidden layer + bias
        hidden_input = np.dot(inputs, self.weights[0]) + self.weights[2]
        # Linear activation (no non-linearity applied in this version)
        hidden_output = hidden_input
        # Linear transformation for output layer
        final_output = np.dot(hidden_output, self.weights[1])
        return final_output # Returns raw scores for actions

    def get_weights(self):
        return [w.copy() for w in self.weights] # Return copies

    def set_weights(self, weights):
        self.weights = [w.copy() for w in weights] # Use copies
```

Explanation: The model stores weights for input-to-hidden, hidden-to-output layers, and a bias for the hidden layer. The `predict` method performs matrix multiplications to calculate output scores based on the input state. `get_weights` and `set_weights` are used by the ES and Agent.

3. Evolution Strategy (`EvolutionStrategy`)

This class implements the optimization algorithm.

Python

```
class EvolutionStrategy:
    # ... (init, _get_perturbed_weights) ...

    def train(self, iterations=100, print_every=10):
        # ... (setup) ...
        for i in range(iterations):
            # 1. Generate population noise vectors (epsilon_k)
            population_noise = []
            rewards = np.zeros(self.population_size)
            for _ in range(self.population_size):
                member_noise = [np.random.randn(*w.shape) for w in
self.weights]
                population_noise.append(member_noise)

            # 2. Evaluate population fitness (R_k)
            for k in range(self.population_size):
                perturbed_weights = self._get_perturbed_weights(self.weights,
population_noise[k])
                # This calls Agent._calculate_reward_on_train
                rewards[k] = self.reward_function(perturbed_weights)

            # 3. Normalize rewards
            if np.std(rewards) > 1e-7:
                 rewards = (rewards - np.mean(rewards)) / np.std(rewards)
            else:
                 rewards = np.zeros_like(rewards)

            # 4. Calculate weighted sum of noise
            weighted_noise_sum = [np.zeros_like(w) for w in self.weights]
            for k in range(self.population_size):
                member_noise = population_noise[k]
                for j in range(len(self.weights)):
                    # Summing R_k * epsilon_k for each weight matrix/vector
                    weighted_noise_sum[j] += member_noise[j] * rewards[k]

            # 5. Update central weights (W_t+1 = W_t + update)
            update_factor = self.learning_rate / (self.population_size *
self.sigma)
            for j in range(len(self.weights)):
                self.weights[j] += update_factor * weighted_noise_sum[j]

            # ... (logging) ...
        # ... (end timing) ...
```

```
    def get_weights(self):
        return self.weights
```

Explanation: The `train` method implements the ES loop: generate random noise (`population_noise`), create perturbed weights, evaluate them using the `reward_function` (which simulates trading on the training set), normalize rewards, compute the weighted sum of noise based on rewards, and finally update the central weights using the learning rate and population parameters.

4. Trading Agent (`TradingAgent`)

This class orchestrates the process, connecting the model, the ES, and the environment simulation.

Python

```
class TradingAgent:
    # ... (constants, __init__) ...

    def _get_state(self, t):
        """ Returns the state (price changes) at index t using all_prices.
"""
        start_index = max(0, t - self.window_size)
        end_index = t + 1
        window_prices = self.all_prices[start_index : end_index]
        # Calculate price differences (returns)
        price_diffs = np.diff(window_prices)
        # Pad if needed to ensure fixed size
        padded_diffs = np.zeros(self.window_size)
        if len(price_diffs) > 0:
           padded_diffs[-len(price_diffs):] = price_diffs
        return padded_diffs.reshape(1, -1)

    def _decide_action(self, state):
        """ Uses the model to decide action (0=hold, 1=buy, 2=sell). """
        prediction_scores = self.model.predict(state)
        return np.argmax(prediction_scores[0]) # Action with highest score

    def _calculate_reward_on_train(self, weights):
        """ Fitness function for ES: Simulates trading ONLY on training data.
"""
        self.model.set_weights(weights) # Use candidate weights
        money = self.initial_money
        inventory = 0.0
        # Simulate only within the training data indices
        start_sim_index = self.window_size
        end_sim_index = self.train_end_index
        for t in range(start_sim_index, end_sim_index, self.skip):
            state = self._get_state(t)
```

```
            action = self._decide_action(state)
            price_now = self.all_prices[t]
            # Simplified fractional buy/sell logic
            if action == 1 and money > self.min_order_size * price_now:
                 buy_units = (money * 0.5) / price_now # Example: invest 50%
cash
                 if buy_units >= self.min_order_size:
                    inventory += buy_units; money -= buy_units * price_now
            elif action == 2 and inventory >= self.min_order_size:
                 sell_units = inventory * 0.5 # Example: sell 50% inventory
                 if sell_units >= self.min_order_size:
                    money += sell_units * price_now; inventory -= sell_units
        # Calculate final value based on last training price
        final_value = money + inventory * self.all_prices[end_sim_index -1]
        reward = ((final_value - self.initial_money) / self.initial_money) *
100
        return reward

    def train_agent(self, iterations, checkpoint):
        """ Trains the agent using ES on the training data. """
        self.es.train(iterations, print_every=checkpoint)
        self.model.set_weights(self.es.get_weights()) # Use the final weights

    def run_test_simulation(self, test_dates_param, return_logs=True):
        """ Evaluates the TRAINED agent ONLY on the test data. """
        print("\nRunning final simulation on UNSEEN TEST DATA...")
        money = self.initial_money
        inventory = 0.0
        states_buy_test, states_sell_test, log = [], [], []
        # Simulate only within the test data indices
        start_test_sim_index = self.train_end_index
        end_test_sim_index = len(self.all_prices) - 1
        for t in range(start_test_sim_index, end_test_sim_index, self.skip):
            state = self._get_state(t)
            action = self._decide_action(state) # Use trained model
            price_now = self.all_prices[t]
            test_set_index = t - start_test_sim_index
            timestamp = test_dates_param[test_set_index]
            # ... (Execute buy/sell logic as in _calculate_reward) ...
            # ... (Logging actions) ...
        # Calculate final value based on last test price
        final_value = money + inventory * self.all_prices[-1]
        total_gains = final_value - self.initial_money
        invest_percent = ((final_value - self.initial_money) /
self.initial_money) * 100
        # ... (Print results) ...
        return states_buy_test, states_sell_test, total_gains,
invest_percent, log
```

Explanation: The agent manages the overall process. `_get_state` prepares the NN input. `_decide_action` gets the NN prediction. `_calculate_reward_on_train` simulates trading *only* on the training price range to provide the fitness score for the ES. `train_agent` runs the ES optimization. `run_test_simulation` uses the final, trained weights to simulate trading *only* on the unseen test price range, providing a realistic performance evaluation.

5. Main Execution and Plotting

This part sets up the parameters, creates the objects, runs the training, runs the test simulation, and plots the results focusing on the test period.

Python

```
# --- Main Execution ---
WINDOW_SIZE = 30
SKIP = 1
INITIAL_MONEY = 10000
LAYER_SIZE = 128
OUTPUT_SIZE = 3
ITERATIONS = 200
CHECKPOINT = 20

# Create Model and Agent
model = SimpleModel(input_size=WINDOW_SIZE, layer_size=LAYER_SIZE,
output_size=OUTPUT_SIZE)
agent = TradingAgent(model=model,
                     all_prices=all_prices,
                     train_end_index=split_index, # Pass split index
                     window_size=WINDOW_SIZE,
                     initial_money=INITIAL_MONEY,
                     skip=SKIP)

# Train the Agent (uses training data internally)
agent.train_agent(iterations=ITERATIONS, checkpoint=CHECKPOINT)

# Evaluate the Agent (uses test data internally)
states_buy_test, states_sell_test, total_gains_test, invest_percent_test,
logs_test = agent.run_test_simulation(test_dates_param=test_dates)

# --- Plotting (Focus on Test Set Performance) ---
fig, ax = plt.subplots(figsize=(15, 7))
# Plot train data (grayed out)
ax.plot(train_dates, train_prices, color='gray', lw=1.0, label='Train Price',
alpha=0.5)
# Plot test data
ax.plot(test_dates, test_prices, color='lightblue', lw=1.5, label='Test
Price')
```

```
# Plot buy/sell markers on test data
buy_marker_dates = test_dates[states_buy_test]
# ... (rest of plotting code) ...
plt.show()
```

Explanation: We define hyperparameters, instantiate the model and agent (passing the full price list and the training end index). We call `train_agent`, then `run_test_simulation`. The plot visualizes both price series but highlights the trades made during the test period.

Sample Results The results using the 3-year period Bitcoin data, 80% of which we used for training the model and the remaining recent 20% for testing its performance are as follows:

```
Data split: 877 training samples, 220 testing samples.
Training data from 2022-05-04 to 2024-09-26
Testing data from 2024-09-27 to 2025-05-04
Starting Evolution Strategy training for 200 iterations...
Iteration 20/200. Current Reward (on train set): 432.9548
Iteration 40/200. Current Reward (on train set): 625.9546
Iteration 60/200. Current Reward (on train set): 920.6560
Iteration 80/200. Current Reward (on train set): 1004.2816
Iteration 100/200. Current Reward (on train set): 1125.9427
Iteration 120/200. Current Reward (on train set): 1108.8394
Iteration 140/200. Current Reward (on train set): 1231.4112
Iteration 160/200. Current Reward (on train set): 1212.2319
Iteration 180/200. Current Reward (on train set): 1282.9231
Iteration 200/200. Current Reward (on train set): 1377.4133
Training finished in 87.46 seconds.
Final Reward on training set: 1377.4133

Running final simulation on UNSEEN TEST DATA...
Test Day 2 (2024-09-29): Buy 0.076179 units at $65,635.30, Bal: $5,000.00,
Inv: 0.076179
Test Day 3 (2024-09-30): Buy 0.039476 units at $63,329.50, Bal: $2,500.00,
Inv: 0.115655
Test Day 4 (2024-10-01): Sell 0.057827 units at $60,837.01, Bal: $6,018.04,
Inv: 0.057827
Test Day 5 (2024-10-02): Buy 0.049627 units at $60,632.79, Bal: $3,009.02,
Inv: 0.107454
Test Day 9 (2024-10-06): Buy 0.023950 units at $62,818.95, Bal: $1,504.51,
Inv: 0.131404
Test Day 10 (2024-10-07): Buy 0.012087 units at $62,236.66, Bal: $752.25,
Inv: 0.143491
Test Day 11 (2024-10-08): Sell 0.071746 units at $62,131.97, Bal: $5,209.95,
Inv: 0.071746
Test Day 12 (2024-10-09): Sell 0.035873 units at $60,582.10, Bal: $7,383.20,
Inv: 0.035873
Test Day 14 (2024-10-11): Buy 0.059118 units at $62,445.09, Bal: $3,691.60,
Inv: 0.094990
Test Day 16 (2024-10-13): Sell 0.047495 units at $62,851.38, Bal: $6,676.74,
```

```
Inv: 0.047495
Test Day 17 (2024-10-14): Buy 0.050546 units at $66,046.12, Bal: $3,338.37,
Inv: 0.098041
Test Day 18 (2024-10-15): Buy 0.024898 units at $67,041.11, Bal: $1,669.18,
Inv: 0.122939
Test Day 19 (2024-10-16): Buy 0.012344 units at $67,612.72, Bal: $834.59,
Inv: 0.135283
Test Day 20 (2024-10-17): Buy 0.006191 units at $67,399.84, Bal: $417.30,
Inv: 0.141474
Test Day 27 (2024-10-24): Sell 0.070737 units at $68,161.05, Bal: $5,238.81,
Inv: 0.070737
Test Day 30 (2024-10-27): Buy 0.038561 units at $67,929.30, Bal: $2,619.40,
Inv: 0.109298
Test Day 31 (2024-10-28): Buy 0.018735 units at $69,907.76, Bal: $1,309.70,
Inv: 0.128033
Test Day 32 (2024-10-29): Buy 0.009005 units at $72,720.49, Bal: $654.85,
Inv: 0.137038
Test Day 33 (2024-10-30): Buy 0.004526 units at $72,339.54, Bal: $327.43,
Inv: 0.141564
Test Day 34 (2024-10-31): Sell 0.070782 units at $70,215.19, Bal: $5,297.39,
Inv: 0.070782
Test Day 35 (2024-11-01): Sell 0.035391 units at $69,482.47, Bal: $7,756.44,
Inv: 0.035391
Test Day 36 (2024-11-02): Sell 0.017695 units at $69,289.27, Bal: $8,982.55,
Inv: 0.017695
Test Day 37 (2024-11-03): Sell 0.008848 units at $68,741.12, Bal: $9,590.75,
Inv: 0.008848
Test Day 38 (2024-11-04): Buy 0.070716 units at $67,811.51, Bal: $4,795.38,
Inv: 0.079564
Test Day 39 (2024-11-05): Buy 0.034569 units at $69,359.56, Bal: $2,397.69,
Inv: 0.114133
Test Day 40 (2024-11-06): Buy 0.015850 units at $75,639.08, Bal: $1,198.84,
Inv: 0.129982
Test Day 41 (2024-11-07): Buy 0.007897 units at $75,904.86, Bal: $599.42,
Inv: 0.137880
Test Day 42 (2024-11-08): Buy 0.003915 units at $76,545.48, Bal: $299.71,
Inv: 0.141795
Test Day 43 (2024-11-09): Buy 0.001952 units at $76,778.87, Bal: $149.86,
Inv: 0.143747
Test Day 47 (2024-11-13): Sell 0.071873 units at $90,584.16, Bal: $6,660.45,
Inv: 0.071873
Test Day 48 (2024-11-14): Buy 0.038169 units at $87,250.43, Bal: $3,330.22,
Inv: 0.110042
Test Day 49 (2024-11-15): Sell 0.055021 units at $91,066.01, Bal: $8,340.76,
Inv: 0.055021
Test Day 50 (2024-11-16): Buy 0.046052 units at $90,558.48, Bal: $4,170.38,
Inv: 0.101073
Test Day 52 (2024-11-18): Buy 0.023030 units at $90,542.64, Bal: $2,085.19,
Inv: 0.124103
```

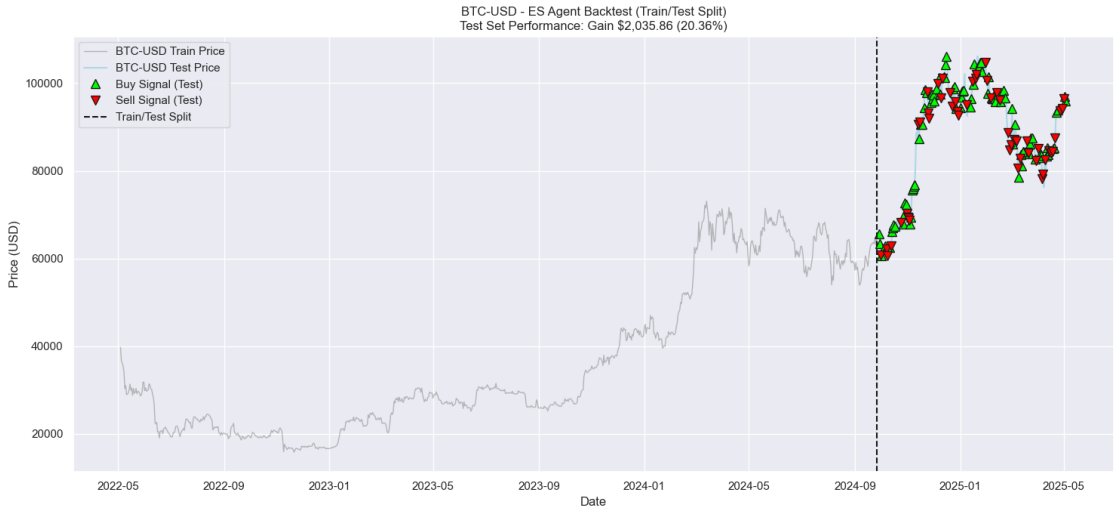
```
Test Day 54 (2024-11-20): Buy 0.011052 units at $94,339.49, Bal: $1,042.60,
Inv: 0.135154
Test Day 55 (2024-11-21): Buy 0.005292 units at $98,504.73, Bal: $521.30,
Inv: 0.140446
Test Day 57 (2024-11-23): Buy 0.002666 units at $97,777.28, Bal: $260.65,
Inv: 0.143112
Test Day 58 (2024-11-24): Sell 0.071556 units at $98,013.82, Bal: $7,274.13,
Inv: 0.071556
Test Day 59 (2024-11-25): Sell 0.035778 units at $93,102.30, Bal: $10,605.14,
Inv: 0.035778
Test Day 60 (2024-11-26): Sell 0.017889 units at $91,985.32, Bal: $12,250.67,
Inv: 0.017889
Test Day 62 (2024-11-28): Buy 0.064037 units at $95,652.47, Bal: $6,125.34,
Inv: 0.081926
Test Day 63 (2024-11-29): Buy 0.031424 units at $97,461.52, Bal: $3,062.67,
Inv: 0.113351
Test Day 64 (2024-11-30): Buy 0.015877 units at $96,449.05, Bal: $1,531.33,
Inv: 0.129228
Test Day 65 (2024-12-01): Buy 0.007871 units at $97,279.79, Bal: $765.67,
Inv: 0.137099
Test Day 66 (2024-12-02): Buy 0.003993 units at $95,865.30, Bal: $382.83,
Inv: 0.141092
Test Day 68 (2024-12-04): Buy 0.001938 units at $98,768.53, Bal: $191.42,
Inv: 0.143030
Test Day 70 (2024-12-06): Sell 0.071515 units at $99,920.71, Bal: $7,337.25,
Inv: 0.071515
Test Day 72 (2024-12-08): Buy 0.036238 units at $101,236.02, Bal: $3,668.63,
Inv: 0.107753
Test Day 73 (2024-12-09): Buy 0.018826 units at $97,432.72, Bal: $1,834.31,
Inv: 0.126580
Test Day 74 (2024-12-10): Sell 0.063290 units at $96,675.43, Bal: $7,952.90,
Inv: 0.063290
Test Day 75 (2024-12-11): Sell 0.031645 units at $101,173.03, Bal:
$11,154.52, Inv: 0.031645
Test Day 77 (2024-12-13): Buy 0.054970 units at $101,459.26, Bal: $5,577.26,
Inv: 0.086615
Test Day 78 (2024-12-14): Buy 0.027509 units at $101,372.97, Bal: $2,788.63,
Inv: 0.114124
Test Day 79 (2024-12-15): Buy 0.013368 units at $104,298.70, Bal: $1,394.31,
Inv: 0.127492
Test Day 80 (2024-12-16): Buy 0.006575 units at $106,029.72, Bal: $697.16,
Inv: 0.134068
Test Day 84 (2024-12-20): Sell 0.067034 units at $97,755.93, Bal: $7,250.11,
Inv: 0.067034
Test Day 87 (2024-12-23): Sell 0.033517 units at $94,686.24, Bal: $10,423.70,
Inv: 0.033517
Test Day 88 (2024-12-24): Buy 0.052818 units at $98,676.09, Bal: $5,211.85,
Inv: 0.086335
Test Day 89 (2024-12-25): Buy 0.026243 units at $99,299.20, Bal: $2,605.92,
```

```
Inv: 0.112578
Test Day 90 (2024-12-26): Sell 0.056289 units at $95,795.52, Bal: $7,998.15,
Inv: 0.056289
Test Day 91 (2024-12-27): Buy 0.042469 units at $94,164.86, Bal: $3,999.07,
Inv: 0.098758
Test Day 93 (2024-12-29): Sell 0.049379 units at $93,530.23, Bal: $8,617.49,
Inv: 0.049379
Test Day 94 (2024-12-30): Sell 0.024689 units at $92,643.21, Bal: $10,904.80,
Inv: 0.024689
Test Day 96 (2025-01-01): Buy 0.057746 units at $94,419.76, Bal: $5,452.40,
Inv: 0.082436
Test Day 97 (2025-01-02): Buy 0.028138 units at $96,886.88, Bal: $2,726.20,
Inv: 0.110574
Test Day 98 (2025-01-03): Buy 0.013894 units at $98,107.43, Bal: $1,363.10,
Inv: 0.124468
Test Day 99 (2025-01-04): Buy 0.006938 units at $98,236.23, Bal: $681.55,
Inv: 0.131406
Test Day 100 (2025-01-05): Buy 0.003466 units at $98,314.96, Bal: $340.78,
Inv: 0.134872
Test Day 103 (2025-01-08): Sell 0.067436 units at $95,043.52, Bal: $6,750.12,
Inv: 0.067436
Test Day 108 (2025-01-13): Buy 0.035709 units at $94,516.52, Bal: $3,375.06,
Inv: 0.103145
Test Day 109 (2025-01-14): Buy 0.017481 units at $96,534.05, Bal: $1,687.53,
Inv: 0.120626
Test Day 110 (2025-01-15): Sell 0.060313 units at $100,504.49, Bal:
$7,749.25, Inv: 0.060313
Test Day 111 (2025-01-16): Buy 0.038841 units at $99,756.91, Bal: $3,874.62,
Inv: 0.099154
Test Day 112 (2025-01-17): Buy 0.018546 units at $104,462.04, Bal: $1,937.31,
Inv: 0.117699
Test Day 114 (2025-01-19): Sell 0.058850 units at $101,089.61, Bal:
$7,886.39, Inv: 0.058850
Test Day 115 (2025-01-20): Sell 0.029425 units at $102,016.66, Bal:
$10,888.21, Inv: 0.029425
Test Day 117 (2025-01-22): Buy 0.052522 units at $103,653.07, Bal: $5,444.10,
Inv: 0.081947
Test Day 118 (2025-01-23): Buy 0.026184 units at $103,960.17, Bal: $2,722.05,
Inv: 0.108131
Test Day 119 (2025-01-24): Buy 0.012984 units at $104,819.48, Bal: $1,361.03,
Inv: 0.121115
Test Day 120 (2025-01-25): Buy 0.006499 units at $104,714.65, Bal: $680.51,
Inv: 0.127614
Test Day 121 (2025-01-26): Buy 0.003314 units at $102,682.50, Bal: $340.26,
Inv: 0.130928
Test Day 125 (2025-01-30): Sell 0.065464 units at $104,735.30, Bal:
$7,196.63, Inv: 0.065464
Test Day 127 (2025-02-01): Sell 0.032732 units at $100,655.91, Bal:
$10,491.29, Inv: 0.032732
```

```
Test Day 128 (2025-02-02): Buy 0.053697 units at $97,688.98, Bal: $5,245.64,
Inv: 0.086429
Test Day 129 (2025-02-03): Buy 0.025865 units at $101,405.42, Bal: $2,622.82,
Inv: 0.112294
Test Day 131 (2025-02-05): Sell 0.056147 units at $96,615.45, Bal: $8,047.49,
Inv: 0.056147
Test Day 132 (2025-02-06): Buy 0.041657 units at $96,593.30, Bal: $4,023.75,
Inv: 0.097804
Test Day 133 (2025-02-07): Buy 0.020842 units at $96,529.09, Bal: $2,011.87,
Inv: 0.118646
Test Day 134 (2025-02-08): Buy 0.010426 units at $96,482.45, Bal: $1,005.94,
Inv: 0.129072
Test Day 135 (2025-02-09): Buy 0.005212 units at $96,500.09, Bal: $502.97,
Inv: 0.134284
Test Day 136 (2025-02-10): Buy 0.002581 units at $97,437.55, Bal: $251.48,
Inv: 0.136865
Test Day 137 (2025-02-11): Buy 0.001313 units at $95,747.43, Bal: $125.74,
Inv: 0.138178
Test Day 138 (2025-02-12): Sell 0.069089 units at $97,885.86, Bal: $6,888.59,
Inv: 0.069089
Test Day 140 (2025-02-14): Buy 0.035323 units at $97,508.97, Bal: $3,444.29,
Inv: 0.104412
Test Day 142 (2025-02-16): Sell 0.052206 units at $96,175.03, Bal: $8,465.20,
Inv: 0.052206
Test Day 143 (2025-02-17): Buy 0.044194 units at $95,773.38, Bal: $4,232.60,
Inv: 0.096400
Test Day 146 (2025-02-20): Buy 0.021522 units at $98,333.94, Bal: $2,116.30,
Inv: 0.117921
Test Day 148 (2025-02-22): Buy 0.010956 units at $96,577.76, Bal: $1,058.15,
Inv: 0.128878
Test Day 151 (2025-02-25): Sell 0.064439 units at $88,736.17, Bal: $6,776.22,
Inv: 0.064439
Test Day 153 (2025-02-27): Sell 0.032219 units at $84,704.23, Bal: $9,505.34,
Inv: 0.032219
Test Day 155 (2025-03-01): Sell 0.016110 units at $86,031.91, Bal:
$10,891.30, Inv: 0.016110
Test Day 156 (2025-03-02): Buy 0.057780 units at $94,248.35, Bal: $5,445.65,
Inv: 0.073890
Test Day 157 (2025-03-03): Buy 0.031637 units at $86,065.67, Bal: $2,722.82,
Inv: 0.105526
Test Day 158 (2025-03-04): Sell 0.052763 units at $87,222.20, Bal: $7,324.93,
Inv: 0.052763
Test Day 159 (2025-03-05): Buy 0.040414 units at $90,623.56, Bal: $3,662.47,
Inv: 0.093177
Test Day 161 (2025-03-07): Sell 0.046589 units at $86,742.67, Bal: $7,703.68,
Inv: 0.046589
Test Day 163 (2025-03-09): Sell 0.023294 units at $80,601.04, Bal: $9,581.22,
Inv: 0.023294
Test Day 164 (2025-03-10): Buy 0.061002 units at $78,532.00, Bal: $4,790.61,
```

```
Inv: 0.084296
Test Day 165 (2025-03-11): Sell 0.042148 units at $82,862.21, Bal: $8,283.10,
Inv: 0.042148
Test Day 166 (2025-03-12): Buy 0.049468 units at $83,722.36, Bal: $4,141.55,
Inv: 0.091616
Test Day 167 (2025-03-13): Buy 0.025544 units at $81,066.70, Bal: $2,070.78,
Inv: 0.117160
Test Day 168 (2025-03-14): Buy 0.012331 units at $83,969.10, Bal: $1,035.39,
Inv: 0.129491
Test Day 169 (2025-03-15): Buy 0.006138 units at $84,343.11, Bal: $517.69,
Inv: 0.135628
Test Day 173 (2025-03-19): Sell 0.067814 units at $86,854.23, Bal: $6,407.65,
Inv: 0.067814
Test Day 174 (2025-03-20): Sell 0.033907 units at $84,167.20, Bal: $9,261.51,
Inv: 0.033907
Test Day 175 (2025-03-21): Sell 0.016954 units at $84,043.24, Bal:
$10,686.35, Inv: 0.016954
Test Day 176 (2025-03-22): Buy 0.063736 units at $83,832.48, Bal: $5,343.17,
Inv: 0.080690
Test Day 177 (2025-03-23): Buy 0.031045 units at $86,054.38, Bal: $2,671.59,
Inv: 0.111735
Test Day 178 (2025-03-24): Buy 0.015266 units at $87,498.91, Bal: $1,335.79,
Inv: 0.127002
Test Day 179 (2025-03-25): Buy 0.007636 units at $87,471.70, Bal: $667.90,
Inv: 0.134637
Test Day 183 (2025-03-29): Buy 0.004043 units at $82,597.59, Bal: $333.95,
Inv: 0.138680
Test Day 184 (2025-03-30): Sell 0.069340 units at $82,334.52, Bal: $6,043.03,
Inv: 0.069340
Test Day 186 (2025-04-01): Sell 0.034670 units at $85,169.17, Bal: $8,995.85,
Inv: 0.034670
Test Day 188 (2025-04-03): Buy 0.054125 units at $83,102.83, Bal: $4,497.93,
Inv: 0.088795
Test Day 189 (2025-04-04): Buy 0.026823 units at $83,843.80, Bal: $2,248.96,
Inv: 0.115618
Test Day 190 (2025-04-05): Buy 0.013466 units at $83,504.80, Bal: $1,124.48,
Inv: 0.129084
Test Day 191 (2025-04-06): Sell 0.064542 units at $78,214.48, Bal: $6,172.61,
Inv: 0.064542
Test Day 192 (2025-04-07): Sell 0.032271 units at $79,235.34, Bal: $8,729.62,
Inv: 0.032271
Test Day 194 (2025-04-09): Sell 0.016136 units at $82,573.95, Bal:
$10,061.99, Inv: 0.016136
Test Day 196 (2025-04-11): Buy 0.060320 units at $83,404.84, Bal: $5,031.00,
Inv: 0.076456
Test Day 197 (2025-04-12): Buy 0.029494 units at $85,287.11, Bal: $2,515.50,
Inv: 0.105950
Test Day 198 (2025-04-13): Buy 0.015030 units at $83,684.98, Bal: $1,257.75,
Inv: 0.120980
```

```
Test Day 199 (2025-04-14): Sell 0.060490 units at $84,542.39, Bal: $6,371.71,
Inv: 0.060490
Test Day 201 (2025-04-16): Sell 0.030245 units at $84,033.87, Bal: $8,913.31,
Inv: 0.030245
Test Day 203 (2025-04-18): Sell 0.015122 units at $84,450.80, Bal:
$10,190.41, Inv: 0.015122
Test Day 204 (2025-04-19): Buy 0.059899 units at $85,063.41, Bal: $5,095.21,
Inv: 0.075021
Test Day 205 (2025-04-20): Buy 0.029910 units at $85,174.30, Bal: $2,547.60,
Inv: 0.104932
Test Day 206 (2025-04-21): Sell 0.052466 units at $87,518.91, Bal: $7,139.36,
Inv: 0.052466
Test Day 207 (2025-04-22): Buy 0.038202 units at $93,441.89, Bal: $3,569.68,
Inv: 0.090668
Test Day 209 (2025-04-24): Buy 0.018999 units at $93,943.80, Bal: $1,784.84,
Inv: 0.109667
Test Day 212 (2025-04-27): Sell 0.054834 units at $93,754.84, Bal: $6,925.75,
Inv: 0.054834
Test Day 214 (2025-04-29): Sell 0.027417 units at $94,284.79, Bal: $9,510.74,
Inv: 0.027417
Test Day 216 (2025-05-01): Sell 0.013708 units at $96,492.34, Bal:
$10,833.49, Inv: 0.013708
Test Day 217 (2025-05-02): Buy 0.055895 units at $96,910.07, Bal: $5,416.75,
Inv: 0.069603
Test Day 218 (2025-05-03): Buy 0.028244 units at $95,891.80, Bal: $2,708.37,
Inv: 0.097847

Test Set Simulation Results:
Total Gains: $2,035.86
Total Investment Return: 20.36%
Ending Cash: $2,708.37
Ending Inventory: 0.097847 units (@ $95,327.29 = $9,327.49)
Final Portfolio Value: $12,035.86
```



Realistic Backtesting: The Importance of Train/Test Split

As highlighted previously, testing a trading strategy on the same data used to optimize it leads to inflated and unrealistic performance metrics due to overfitting. The agent learns the specific patterns *of the training data*, including its noise.

By splitting the data:

1. **Training Set:** Used exclusively by the Evolution Strategy (`_calculate_reward_on_train`) to find the optimal neural network weights.
2. **Test Set:** A completely separate period used only once (`run_test_simulation`) to evaluate how well the strategy, optimized on past data, performs on new, unseen data.

This mimics real-world trading where strategies are developed on historical data and deployed on future, unknown data. The performance on the test set gives a much more reliable (though still not guaranteed) indication of potential real-world viability.

Further Considerations and Limitations

Even with a train/test split, this implementation is still simplified:

- **Transaction Costs:** Real trading involves commissions and potential slippage (difference between expected and execution price), which are ignored here but reduce profits.
- **Market Regimes:** The strategy's performance might vary drastically depending on whether the market is trending, ranging, or volatile. The train/test split helps, but longer periods or walk-forward analysis might be needed for more robustness.
- **Parameter Sensitivity:** The performance heavily depends on `WINDOW_SIZE`, `LAYER_SIZE`, ES hyperparameters (`POPULATION_SIZE`, `SIGMA`, `LEARNING_RATE`), and the number of `ITERATIONS`. These

often require careful tuning (hyperparameter optimization), potentially using a *third* dataset (validation set) separate from train and test.

- **Trading Logic:** The buy/sell logic (e.g., "invest 50% cash") is arbitrary. More sophisticated position sizing and risk management rules are essential in real trading.
- **Feature Engineering:** Using only price changes is basic. Incorporating volume, volatility measures, or other indicators could potentially improve performance.

Conclusion

This article demonstrated how Evolution Strategies can be combined with a simple neural network to optimize a trading agent. We implemented this approach in Python, emphasizing the crucial step of separating training and testing data for realistic backtesting. While ES provides a powerful method for optimizing complex strategies where gradients are unavailable, building a consistently profitable trading bot requires careful consideration of data handling, model complexity, realistic simulation (including costs), robust validation techniques, and rigorous risk management. This example serves as an educational foundation for exploring these advanced concepts in algorithmic trading.