Lumina - Reinforcement Learning for Portfolio Management

Version: 1.0

Author: Krish Garg

Table of Contents

Introduction

Lumina is a reinforcement learning (RL) framework designed for portfolio management. It leverages advanced machine learning techniques to construct and manage investment portfolios by learning optimal trading strategies. The project integrates market data analysis, regime detection using Hidden Markov Models (HMM), training of a Proximal Policy Optimization (PPO) agent within a custom Gym environment, and backtesting to evaluate performance.

Features

- **Market Data Acquisition:** Downloads historical market data using Yahoo Finance.
- **Regime Detection:** Identifies market regimes using Hidden Markov Models.
- **Technical Indicators:** Calculates moving averages and volatility indicators.
- **Custom Gym Environment:** Defines a tailored environment for portfolio management tasks.
- **Reinforcement Learning:** Trains a PPO agent to optimize portfolio allocations.
- **Backtesting:** Evaluates the trained model's performance on historical data.
- **Performance Metrics:** Computes Sharpe Ratio and Maximum Drawdown.

- **Visualization:** Generates performance plots for analysis.
- **Logging:** Provides comprehensive logging for monitoring and debugging.

Disclaimer

> **DISCLAIMER:**
> This code is provided for educational purposes only and does not constitute financial advice. Use at your own risk.

Installation

Prerequisites

- **Python 3.8+**
- **pip** package manager

Required Libraries

Ensure you have the following Python libraries installed:

- `numpy`
- `pandas`
- `yfinance`
- `hmmlearn`
- `scikit-learn`
- `matplotlib`
- `gymnasium`
- `stable-baselines3`
- `tqdm`

Installation Steps

1. **Clone the Repository**

```bash
git clone https://github.com/krish1905/lumina.git
cd lumina
```

2. **Create a Virtual Environment (Optional but Recommended)**

```bash
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. **Install Dependencies**

```bash
pip install -r requirements.txt
```

If `requirements.txt` is not provided, install manually:

```bash
pip install numpy pandas yfinance hmmlearn scikit-learn matplotlib gymnasium
stable-baselines3 tqdm
```

Project Structure

```bash
lumina/
├── logs/                   # Folder for model checkpoints
├── lumina.py               # Main script containing the project code
├── README.md               # Project documentation
├── requirements.txt        # Python dependencies
```

Usage Guide

The primary script `lumina.py` orchestrates the entire workflow from data acquisition to model evaluation. Below is a detailed explanation of each component.

Configuration Parameters

At the beginning of the script, several user-configurable parameters are defined. These parameters allow customization of the model's behavior and the data used.

```python
# User-Configurable Parameters
TICKERS = ["GS", "NVDA", "BRK-B", "C", "JPM", "^VIX"]  # Assets to include in the
portfolio
START_DATE = "2010-01-01"                             # Start date for historical
data
END_DATE = "2024-12-15"                               # End date for historical data
N_COMPONENTS_HMM = 3                                   # Number of hidden states in
HMM
WINDOW_SIZE = 20                                       # Window size for
observations
TRANSACTION_COST_RATE = 0.001                          # Transaction cost rate per
trade
TRAINING_TIMESTEPS = 500000                            # Total training timesteps
for PPO
RISK_FREE_RATE = 0.02                                  # Annual risk-free rate for
Sharpe Ratio
MODEL_NAME = "ppo_portfolio_agent"                     # Filename to save the
trained model
```

Parameter Descriptions

- **TICKERS:** A list of stock symbols and indices to include in the portfolio. For example, `"^VIX"` represents the CBOE Volatility Index.
- **START_DATE & END_DATE:** Define the period for which historical market data is fetched.
- **N_COMPONENTS_HMM:** Specifies the number of hidden states for the HMM, representing different market regimes.
- **WINDOW_SIZE:** The number of past days' data used to form the observation for the RL agent.

- **TRANSACTION_COST_RATE:** Represents the cost incurred for each trade, expressed as a percentage.
- **TRAINING_TIMESTEPS:** The number of iterations the PPO agent will undergo during training.
- **RISK_FREE_RATE:** Annualized risk-free rate used in the calculation of the Sharpe Ratio.
- **MODEL_NAME:** The name under which the trained PPO model will be saved.

Data Loading and Preprocessing

Downloading Historical Data

The script uses the `yfinance` library to download adjusted closing prices for the specified tickers.

```python
logging.info("Downloading historical data...")
data = yf.download(TICKERS, start=START_DATE, end=END_DATE)["Adj Close"].dropna(how='all')
if data.empty:
    raise ValueError("No data was downloaded. Please check your tickers or date ranges.")
```

- **Data Retrieval:** Fetches adjusted close prices, which account for corporate actions like dividends and stock splits.
- **Data Cleaning:** Drops any rows where all ticker data is missing to ensure data integrity.

Calculating Returns and Scaling

Returns are calculated as daily percentage changes in adjusted closing prices. These returns are then scaled between 0 and 1 using `MinMaxScaler`.

```python
logging.info("Calculating returns and scaling...")
returns = data.pct_change().dropna()
returns = returns.replace([np.inf, -np.inf], 0).fillna(0)  # Replace infinities and NaNs
scaler = MinMaxScaler()
scaled_returns = scaler.fit_transform(returns)  # scaled_returns is now a numpy array
```

```
# Convert back to DataFrame for convenience
scaled_returns_df = pd.DataFrame(scaled_returns, index=returns.index,
columns=returns.columns)
```

- **Returns Calculation:** Daily returns are essential for understanding asset performance.
- **Data Cleaning:** Replaces infinite and NaN values with zeros to prevent computational issues.
- **Scaling:** Normalizes the data to a consistent range, facilitating model training.

Market Regime Detection with Hidden Markov Model (HMM)

Overview

Market regimes represent distinct phases in financial markets, such as bullish, bearish, or volatile periods. Identifying these regimes helps in tailoring investment strategies to current market conditions.

Implementation

The script employs a Gaussian Hidden Markov Model to classify different market states based on scaled returns.

```python
logging.info("Fitting HMM to identify market regimes...")
hmm_model = hmm.GaussianHMM(n_components=N_COMPONENTS_HMM, covariance_type="full")
hmm_model.fit(scaled_returns)
hidden_states = hmm_model.predict(scaled_returns)
```

- **Model Choice:** Gaussian HMM assumes that the observations are generated from a mixture of Gaussian distributions.
- **Number of States:** Defined by `N_COMPONENTS_HMM`, allowing flexibility in capturing market complexities.
- **Fitting the Model:** The HMM is trained on the scaled returns to learn the transition probabilities between states.
- **State Prediction:** Assigns a hidden state (regime) to each time step in the dataset.

Technical Indicators Calculation

Technical indicators provide additional insights into market behavior, aiding the RL agent in making informed decisions.

Moving Average and Volatility

```python
def calculate_technical_indicators(returns, window=20):
    moving_avg = returns.rolling(window=window).mean()
    volatility = returns.rolling(window=window).std()
    return moving_avg, volatility

moving_avg, volatility = calculate_technical_indicators(returns, WINDOW_SIZE)
```

- **Moving Average (MA):** Smoothens price data to identify trends over a specified window.
- **Volatility:** Measures the degree of variation in returns, indicating market stability or turbulence.

Custom Gym Environment

The `TradingEnv` class defines a custom environment compatible with Gymnasium, tailored for portfolio management tasks.

Environment Components

- **Observation Space:**
  - **Past Scaled Returns:** Historical returns over the specified `WINDOW_SIZE`.
  - **Current Market Regime:** Current hidden state from the HMM.
  - **Technical Indicators:** Moving average and volatility for each asset.
- **Action Space:**
  - **Portfolio Allocations:** Continuous actions representing the weight distribution across assets.
- **Reward Function:**
  - **Portfolio Return:** The return achieved by the portfolio.
  - **Transaction Costs:** Costs incurred from adjusting portfolio allocations.

Environment Definition

python

```python
class TradingEnv(gym.Env):
    """
    A trading environment for RL agents.
    Observation:
        - Past scaled returns for a specified window.
        - Current market regime (from HMM).
        - Moving average and volatility indicators (unscaled).

    Action:
        - Continuous actions mapping to portfolio allocations across multiple
assets.

    Reward:
        - Portfolio return minus transaction costs.
    """

    metadata = {'render_modes': ['human']}

    def __init__(self, scaled_returns_df, returns, hidden_states, window_size=20,
transaction_cost_rate=0.001):
        super().__init__()
        self.returns = returns
        self.scaled_returns = scaled_returns_df
        self.hidden_states = hidden_states
        self.window_size = window_size
        self.n_assets = returns.shape[1]
        self.current_step = window_size
        self.transaction_cost_rate = transaction_cost_rate

        # Continuous action space: Portfolio weights for each asset
        self.action_space = spaces.Box(low=0, high=1, shape=(self.n_assets,),
dtype=np.float32)

        # Observation space: [past_scaled_returns, regime, ma, vol]
        observation_space_size = self.n_assets * window_size + 1 + (self.n_assets *
2)
        self.observation_space = spaces.Box(
            low=-np.inf,
            high=np.inf,
            shape=(observation_space_size,),
            dtype=np.float32
        )
        self.portfolio = np.zeros(self.n_assets)

    def reset(self, *, seed=None, options=None):
        super().reset(seed=seed)
        self.current_step = self.window_size
        self.portfolio = np.zeros(self.n_assets)
        obs = self._get_observation()
        return obs, {}

    def step(self, action):
        action_vector = np.clip(action, 0, 1)  # Ensure actions are within bounds
        if np.sum(action_vector) == 0:
            action_vector = np.ones(self.n_assets) / self.n_assets  # Avoid division
by zero
        else:
            action_vector /= np.sum(action_vector)  # Normalize weights

        day_returns = self.returns.iloc[self.current_step].values
        portfolio_return = np.dot(day_returns, action_vector)

        transaction_cost = np.sum(np.abs(self.portfolio - action_vector)) *
self.transaction_cost_rate

        reward = portfolio_return - transaction_cost

        self.portfolio = action_vector
        self.current_step += 1
        terminated = self.current_step >= len(self.returns) - 1
        truncated = False

        obs = self._get_observation()
        return obs, reward, terminated, truncated, {}

    def _get_observation(self):
        if self.current_step < self.window_size:
            return np.zeros(self.observation_space.shape[0], dtype=np.float32)
        past_scaled_returns = self.scaled_returns.iloc[self.current_step -
self.window_size: self.current_step].values.flatten()
        current_regime = np.array([self.hidden_states[self.current_step]],
dtype=np.float32)
        recent_returns = self.returns.iloc[self.current_step - self.window_size:
```

```
self.current_step]
        ma = recent_returns.mean(axis=0).fillna(0).values  # Replace NaN with 0
        vol = recent_returns.std(axis=0).fillna(0).values  # Replace NaN with 0
        observation = np.concatenate((past_scaled_returns, current_regime, ma, vol))
        return observation.astype(np.float32)
```

Key Components

- **Initialization (`__init__`):**
  - Sets up the action and observation spaces.
  - Initializes the portfolio weights.
- **Reset (`reset`):**
  - Resets the environment to the initial state.
  - Resets the portfolio to an equal-weighted distribution or zeros.
- **Step (`step`):**
  - Receives an action (portfolio allocation) from the agent.
  - Calculates the portfolio return based on current asset returns.
  - Applies transaction costs for changing portfolio allocations.
  - Computes the reward as the net return after costs.
  - Advances the environment to the next time step.
- **Observation (`_get_observation`):**
  - Constructs the observation vector combining past returns, current regime, moving averages, and volatility.

Training the PPO Agent

The PPO agent is trained using the Stable Baselines3 library within the defined trading environment.

Train-Test Split

The dataset is split into training and testing periods to evaluate the model's performance on unseen data.

```python
```

```
# Train-Test Split
train_start, train_end = "2010-01-01", "2018-12-31"
test_start, test_end = "2019-01-01", "2024-12-15"

train_returns = returns[train_start:train_end]
test_returns = returns[test_start:test_end]

train_hidden_states = hidden_states[:len(train_returns)]
test_hidden_states = hidden_states[len(train_returns):]

train_scaled_returns = scaled_returns_df[train_start:train_end]
test_scaled_returns = scaled_returns_df[test_start:test_end]
```

Environment Setup

```python
# Training Environment
train_env = TradingEnv(
    train_scaled_returns,
    train_returns,
    train_hidden_states,
    window_size=WINDOW_SIZE,
    transaction_cost_rate=TRANSACTION_COST_RATE
)
train_env = DummyVecEnv([lambda: train_env])
```

- **DummyVecEnv:** Wraps the environment for vectorized operations, facilitating efficient training.

PPO Model Initialization

```python
model = PPO(
    "MlpPolicy",
    train_env,
    verbose=1,
    learning_rate=1e-5,  # Smaller learning rate
    batch_size=256,
    n_steps=2048,
    gamma=0.99,
    ent_coef=0.001,
    vf_coef=0.5,
)
```

- **Policy:** Multi-layer Perceptron (MLP) policy suitable for continuous action spaces.
- **Hyperparameters:**
  - **Learning Rate:** Set to a small value for stable convergence.
  - **Batch Size:** Number of samples per gradient update.
  - **n_steps:** Number of steps to run for each environment per update.
  - **Gamma:** Discount factor for future rewards.
  - **Entropy Coefficient (ent_coef):** Encourages exploration by adding entropy to the loss.
  - **Value Function Coefficient (vf_coef):** Weights the value function loss in the overall loss.

Training Process

```python
logging.info("Training the PPO model...")
checkpoint_callback = CheckpointCallback(save_freq=10000, save_path="./logs/")
model.learn(total_timesteps=TRAINING_TIMESTEPS, callback=checkpoint_callback)
model.save(MODEL_NAME)
```

- **Checkpointing:** Saves model checkpoints every 10,000 timesteps for recovery and analysis.
- **Training:** The PPO agent learns over the specified number of timesteps, optimizing its policy based on rewards.

Backtesting the Trained Model

Backtesting evaluates the performance of the trained model on historical data not seen during training.

Backtest Function

```python

```

```python
def backtest_ppo(model, env, returns, window_size):
    obs = env.reset()[0]
    portfolio_values = [1.0]
    portfolio_returns = []

    for _ in tqdm(range(len(returns) - window_size - 1), desc="Backtesting PPO 
Strategy"):
        action, _ = model.predict(obs, deterministic=True)
        obs, reward, done, info = env.step(action)
        obs = obs[0]
        reward = reward[0]
        done = done[0]

        portfolio_values.append(portfolio_values[-1] * (1 + reward))
        portfolio_returns.append(reward)

        if done:
            break

    return np.array(portfolio_values), np.array(portfolio_returns)
```

- **Initialization:** Starts with a portfolio value of 1.0 (representing initial capital).
- **Loop Through Test Data:**
  - **Action Prediction:** The model predicts portfolio allocations based on the current observation.
  - **Environment Step:** Applies the action to the environment, obtaining the reward.
  - **Portfolio Update:** Updates the portfolio value based on the reward (return).
  - **Termination:** Stops if the end of the test data is reached.
- **Output:** Returns the portfolio values and daily returns for analysis.

Running Backtest

```python
# Run backtest
portfolio_values, portfolio_returns = backtest_ppo(model, test_env, test_returns, 
WINDOW_SIZE)
```

Evaluation Metrics

Sharpe Ratio

The Sharpe Ratio measures the risk-adjusted return of the portfolio.

```python
def calculate_sharpe_ratio(portfolio_returns, risk_free_rate=0.02):
    portfolio_returns = np.array(portfolio_returns)
    excess_returns = portfolio_returns - (risk_free_rate / 252)
    return (np.sqrt(252) * np.mean(excess_returns)) / (np.std(excess_returns) + 1e-
8)
```

- **Calculation:**
    - **Excess Returns:** Daily returns minus the daily risk-free rate.
    - **Annualization:** Multiplies by the square root of 252 (approximate trading days in a year).
    - **Risk-Adjusted Return:** Mean of excess returns divided by the standard deviation.

Maximum Drawdown

Maximum Drawdown quantifies the largest peak-to-trough decline in portfolio value.

```python
def calculate_max_drawdown(portfolio_values):
    peak = np.maximum.accumulate(portfolio_values)
    drawdowns = (peak - portfolio_values) / peak
    return np.max(drawdowns)
```

- **Calculation:**
    - **Peak Accumulation:** Tracks the highest portfolio value achieved up to each time step.
    - **Drawdowns:** Measures the percentage decline from the peak.
    - **Maximum:** Identifies the largest drawdown experienced.

Displaying Metrics

```python
```

```
sharpe_ratio = calculate_sharpe_ratio(portfolio_returns, RISK_FREE_RATE)
max_drawdown = calculate_max_drawdown(portfolio_values)

print(f"PPO Strategy - Sharpe Ratio: {sharpe_ratio:.2f}, Max Drawdown:
{max_drawdown:.2%}")
```

Financial Concepts

Portfolio Management

Portfolio management involves selecting and overseeing a collection of investments to achieve specific financial objectives. Key aspects include:

- **Asset Allocation:** Distributing investments across various asset classes (e.g., stocks, bonds) to balance risk and return.
- **Diversification:** Spreading investments to reduce exposure to any single asset's performance.
- **Rebalancing:** Adjusting portfolio allocations to maintain desired asset distribution over time.

Transaction Costs

Transaction costs are fees incurred when buying or selling assets. They impact the net returns of a trading strategy.

- **Impact:** High transaction costs can erode profits, especially in strategies with frequent trading.
- **Model Consideration:** Incorporating transaction costs into the reward function encourages the RL agent to minimize unnecessary trades.

Sharpe Ratio

The Sharpe Ratio evaluates the performance of an investment by adjusting for its risk.

- **Formula:**

$$\text{Sharpe Ratio} = \frac{E[R_p - R_f]}{\sigma_p}$$

  - $E[R_p - R_f]$: Expected excess return of the portfolio over the risk-free rate.
  - σ_p: Standard deviation of the portfolio's excess returns.
- **Interpretation:**
  - **Higher Ratio:** Better risk-adjusted performance.
  - **Usage:** Compares different investment strategies to identify superior risk-adjusted returns.

Max Drawdown

Maximum Drawdown (Max DD) measures the largest single drop from peak to trough in the portfolio value.

- **Significance:** Indicates the potential downside risk of an investment strategy.
- **Usage:** Helps investors understand the worst-case scenario for their portfolio.

Algorithms and Techniques

Hidden Markov Model (HMM)

Overview

A Hidden Markov Model is a statistical model that represents systems with unobservable (hidden) states. It is widely used for modeling sequential data where the system's state transitions are not directly observable.

Application in Lumina

- **Purpose:** Identifies distinct market regimes based on historical returns.
- **States:** Each hidden state corresponds to a different market condition (e.g., bull, bear, volatile).
- **Observation:** Scaled daily returns serve as the observable data used to infer the hidden states.

Implementation Details

- **Library:** `hmmlearn`

- **Model Type:** Gaussian HMM, assuming that observations are generated from Gaussian distributions.
- **Fitting:** The model is trained on scaled returns to learn state distributions and transition probabilities.
- **Prediction:** After training, the model predicts the most likely hidden state for each time step.

Proximal Policy Optimization (PPO)

Overview

Proximal Policy Optimization is a reinforcement learning algorithm that optimizes the policy directly while ensuring stable updates. It is part of the policy gradient family of methods.

Advantages

- **Stability:** PPO introduces mechanisms to prevent large, destabilizing policy updates.
- **Sample Efficiency:** Requires fewer samples to achieve good performance compared to some other RL algorithms.
- **Ease of Implementation:** PPO is simpler to implement and tune.

Application in Lumina

- **Objective:** Learns the optimal portfolio allocation strategy that maximizes cumulative returns while considering transaction costs.
- **Policy:** The agent's policy outputs continuous portfolio weights for each asset.
- **Training:** The agent interacts with the custom Gym environment, receiving rewards based on portfolio performance.

Implementation Details

- **Library:** `stable-baselines3`
- **Policy Network:** Multi-layer Perceptron (MLP) suited for continuous action spaces.
- **Hyperparameters:**
  - **Learning Rate:** Controls the step size during optimization.
  - **Batch Size:** Determines the number of samples used for each gradient update.
  - **n_steps:** Number of steps to collect before updating the policy.
  - **Gamma:** Discount factor for future rewards.

- **Entropy Coefficient:** Balances exploration and exploitation.
- **Value Function Coefficient:** Balances the value function loss in the overall loss.

Output and Visualization

After training and backtesting, the script generates visualizations to assess the strategy's performance.

Portfolio Performance Plot

```python
plt.figure(figsize=(12, 6))
plt.plot(portfolio_values, label="PPO Strategy")
plt.xlabel("Time Step")
plt.ylabel("Portfolio Value")
plt.title("PPO Strategy Portfolio Performance")
plt.legend()
plt.show()
```

- **Description:** Plots the evolution of the portfolio value over the backtesting period.
- **Interpretation:** Helps visualize the growth or decline of the portfolio, identifying periods of strong or weak performance.

Saving Plots

All generated plots are saved in the `outputs` folder for future reference and analysis.

Note: The `outputs` folder has been removed as per user request. Ensure to adjust the script accordingly if necessary.

Logging and Monitoring

Comprehensive logging is implemented to monitor the training and evaluation processes.

```python
# Set up logging
logging.basicConfig(stream=sys.stdout, level=logging.INFO, format='[%(asctime)s] %(levelname)s: %(message)s')
```

- **Purpose:** Provides real-time feedback on the script's execution, including data loading, model training, and any potential issues.
- **Log Levels:** INFO level logs general progress and significant events.

Checkpointing

Model checkpoints are saved periodically during training to allow recovery and analysis of intermediate models.

```python
checkpoint_callback = CheckpointCallback(save_freq=10000, save_path="./logs/")
model.learn(total_timesteps=TRAINING_TIMESTEPS, callback=checkpoint_callback)
```

- **Frequency:** Every 10,000 timesteps.
- **Save Path:** Stored in the `logs` folder.
- **Usage:** Enables resuming training from the last checkpoint in case of interruptions.

Conclusion

Lumina leverages cutting-edge reinforcement learning techniques to navigate the complexities of portfolio management. By integrating market regime detection, technical indicators, and advanced RL algorithms, it provides a robust framework for developing and evaluating trading strategies. While the project offers a solid foundation, further enhancements and optimizations can be explored to improve performance and adapt to evolving market conditions.

For any questions or support, please refer to the project's repository or contact the author.