Sirindhorn International Institute of Technology

Thammasat University

Financial Engineering Assignment: Algorithmic Trading on BTC/USDT

Table of Contents

Introduction

Algorithmic trading involves using automated systems to execute trades based on predefined strategies, offering speed and efficiency in the dynamic cryptocurrency market. This report focuses on developing, backtesting, and evaluating algorithmic trading strategies for the BTC/USDT pair across various time intervals: 1 minute, 1 hour, 4 hours, and 1 day.

The report outlines the process of obtaining and preprocessing historical BTC/USDT price data, implementing tailored trading strategies, and testing their performance using a provided backtesting framework. Performance metrics such as total return, Sharpe ratio, and maximum drawdown are analyzed to compare the effectiveness of different strategies and intervals. This comprehensive evaluation aims to identify the strengths and weaknesses of each strategy, providing valuable insights for effective algorithmic trading in the cryptocurrency market.

Data preprocessing steps

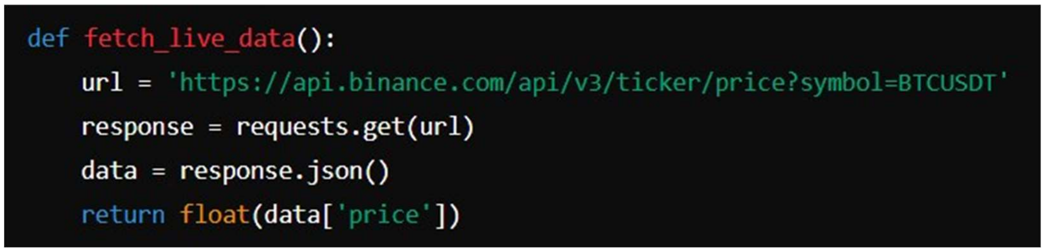
Methodology for Data Preprocessing

In this project, we executed a structured approach to collect, clean, and prepare data for the development and evaluation of trading strategies on the BTC/USDT trading pair. Below is the detailed methodology based on the provided code:

1. Data Collection

- **Live Data:**

  - We used the Binance API to fetch the current price of BTC/USDT. This live data fetch is done using a function that sends a request to the Binance API and retrieves the latest price.



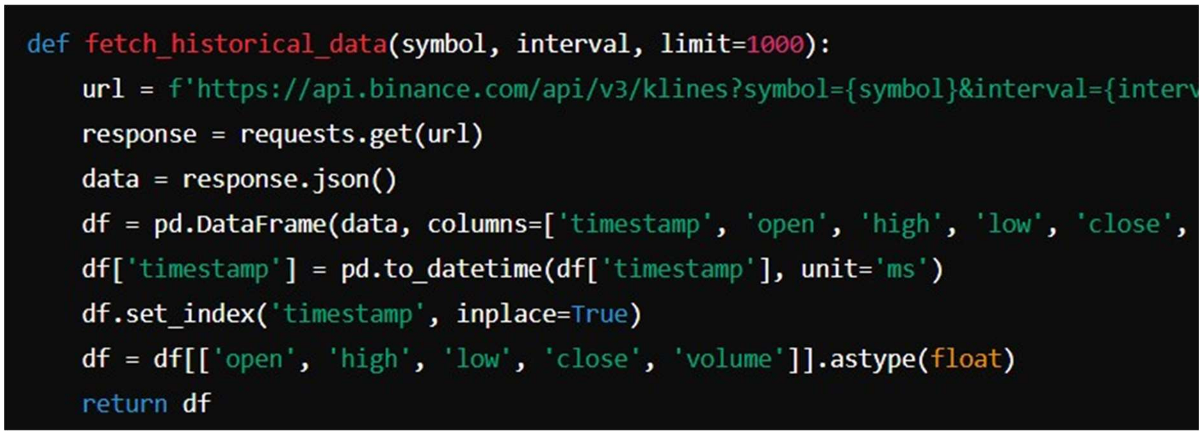
```
def fetch_live_data():
    url = 'https://api.binance.com/api/v3/ticker/price?symbol=BTCUSDT'
    response = requests.get(url)
    data = response.json()
    return float(data['price'])
```

- **Historical Data:**

  - Historical price data was collected for the BTC/USDT trading pair using the Binance API.

  - Data was fetched for various intervals: 1 minute, 1 hour, 4 hours, and 1 day, with up to 1000 data points per request.

  - The data includes timestamps, open, high, low, close prices, and volume.



```
def fetch_historical_data(symbol, interval, limit=1000):
    url = f'https://api.binance.com/api/v3/klines?symbol={symbol}&interval={interv
    response = requests.get(url)
    data = response.json()
    df = pd.DataFrame(data, columns=['timestamp', 'open', 'high', 'low', 'close',
    df['timestamp'] = pd.to_datetime(df['timestamp'], unit='ms')
    df.set_index('timestamp', inplace=True)
    df = df[['open', 'high', 'low', 'close', 'volume']].astype(float)
    return df
```

2. Data Cleaning:

- After fetching the historical data, the timestamps were converted from milliseconds to a readable datetime format.

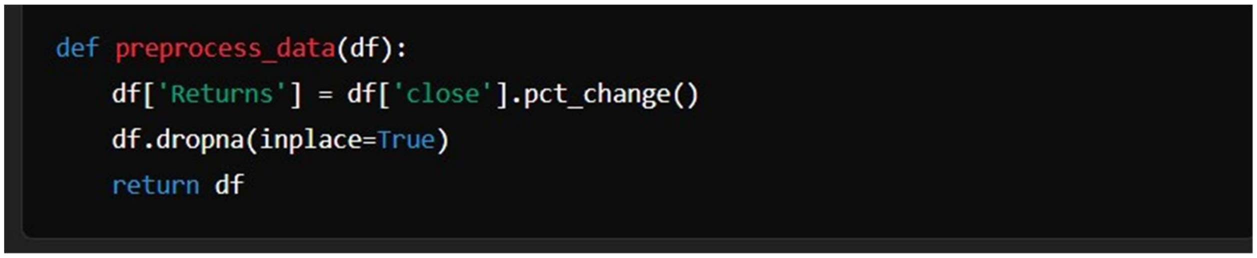
- Timestamps were set as the index of the DataFrame to facilitate time series analysis.

- Only essential columns (open, high, low, close, volume) were retained and ensured to be in the float data type.

3. Data Preparation:

- We calculated the percentage change in the closing prices to derive returns for each time interval.

- Any rows with missing values (resulting from the percentage change calculation) were removed to maintain the dataset's integrity.

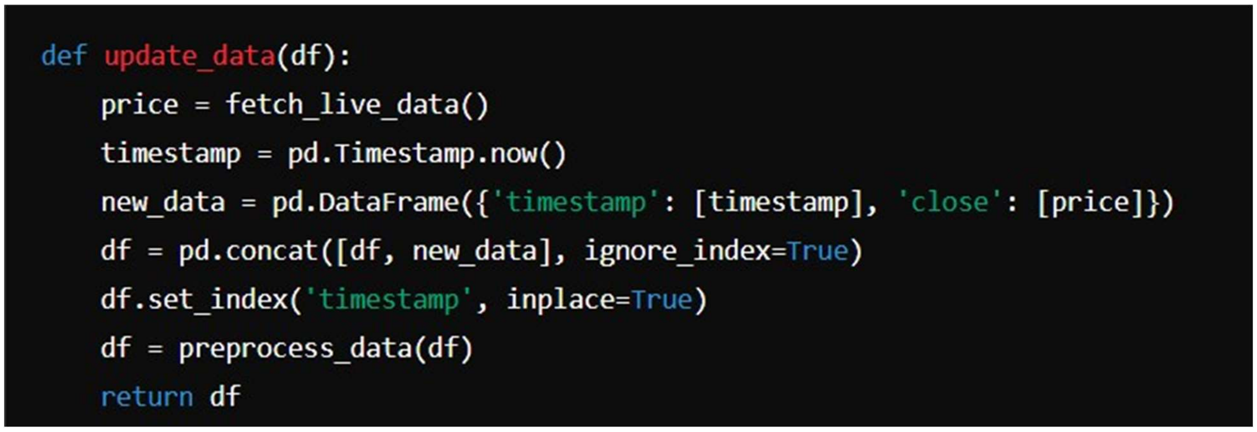


```
def preprocess_data(df):
    df['Returns'] = df['close'].pct_change()
    df.dropna(inplace=True)
    return df
```

4. Updating Data with Live Prices:

- A function was designed to fetch the latest live price and update the historical dataset in real-time.

- This function retrieves the current price, appends it to the existing DataFrame, and recalculates the returns.

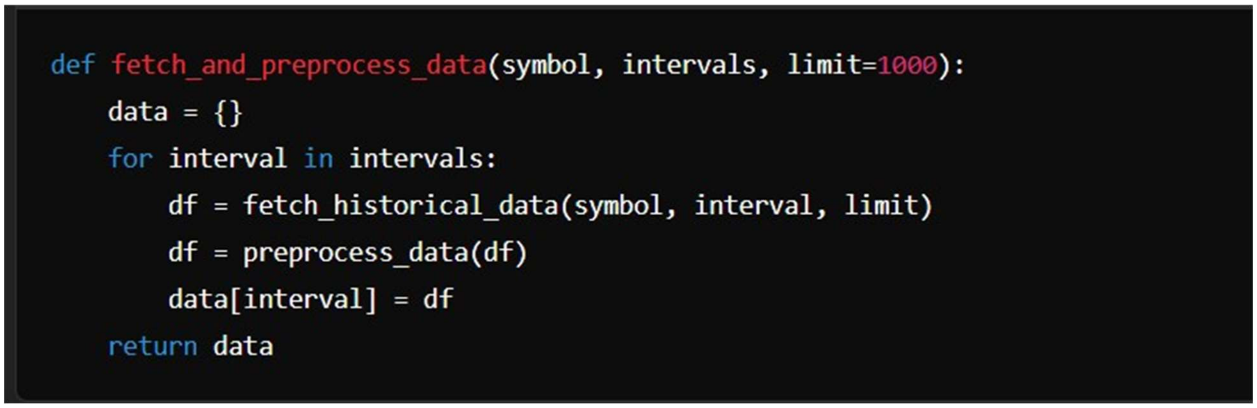


```
def update_data(df):
    price = fetch_live_data()
    timestamp = pd.Timestamp.now()
    new_data = pd.DataFrame({'timestamp': [timestamp], 'close': [price]})
    df = pd.concat([df, new_data], ignore_index=True)
    df.set_index('timestamp', inplace=True)
    df = preprocess_data(df)
    return df
```

5. Comprehensive Data Fetching and Preprocessing:

- To streamline the process, a function was created to fetch and preprocess data for all specified intervals in one go.

- This function iterates over the intervals, fetches historical data, preprocesses it, and stores the processed DataFrames in a dictionary.

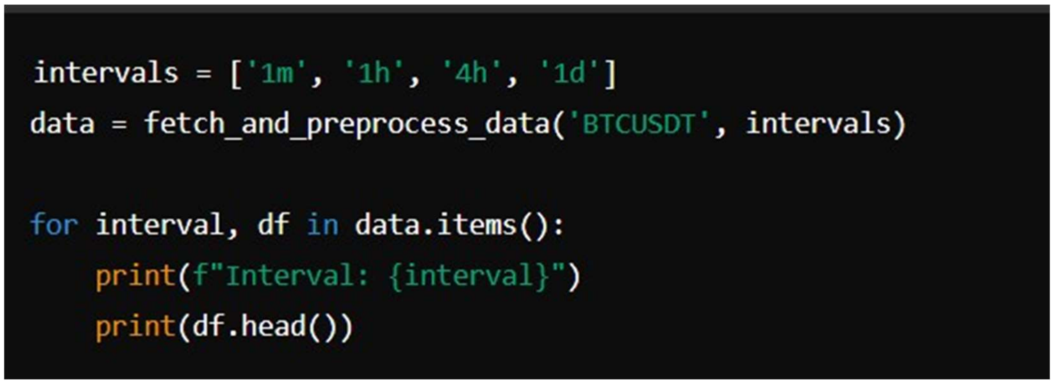


```python
def fetch_and_preprocess_data(symbol, intervals, limit=1000):
    data = {}
    for interval in intervals:
        df = fetch_historical_data(symbol, interval, limit)
        df = preprocess_data(df)
        data[interval] = df
    return data
```

6. Execution:

- The predefined intervals for data collection are 1 minute, 1 hour, 4 hours, and 1 day.

- The data for each interval was fetched and preprocessed, and the first few rows of each processed dataset were printed for verification.



```python
intervals = ['1m', '1h', '4h', '1d']
data = fetch_and_preprocess_data('BTCUSDT', intervals)

for interval, df in data.items():
    print(f"Interval: {interval}")
    print(df.head())
```

By following these steps, we ensured that our data is consistently up-to-date and ready for analysis, which is crucial for developing and testing reliable trading strategies.

Implement Trading Strategies

SMA50

SMA50: The 50-day SMA is an intermediate-term moving average that offers a broader perspective on price trends. Traders often use the SMA50 as a confirmation of the short-term trend identified by the SMA20. A bullish crossover, where the price moves above the SMA50, may indicate a strengthening bullish trend, while a bearish crossover may suggest a potential downtrend.

SMA200

SMA200: The 200-day SMA is a long-term moving average that provides a measure of the overall trend and acts as a significant support or resistance level. Many traders consider the SMA200 as a dividing line between bull and bear markets. When the price is above the SMA200, it signifies a bullish market, while a price below the SMA200 suggests a bearish market.

Moving average convergence/divergence (MACD)

Moving average convergence/divergence (MACD) is a technical indicator to help investors identify price trends, measure trend momentum, and identify market entry points for buying or selling. Moving average convergence/divergence (MACD) is a trend-following momentum indicator that shows the relationship between two exponential moving averages (EMAs) of a security's price. MACD was developed in the 1970s by Gerald Appel.

The MACD line is calculated by subtracting the 26-period EMA from the 12-period EMA. The calculation creates the MACD line. A nine-day EMA of the MACD line is called the signal line, plotted on top of the MACD line, which can function as a trigger for buy or sell signals.

Fibonacci retracement

Fibonacci retracement levels—stemming from the Fibonacci sequence—are horizontal lines that indicate where support and resistance are likely to occur. Each level is associated with a percentage. The percentage is how much of a prior move the price has retraced. The Fibonacci retracement levels are 23.6%, 38.2%, 61.8%, and 78.6%. While not officially a Fibonacci ratio, 50% is also used.

The indicator is useful because it can be drawn between any two significant price points, such as a high and a low. The indicator will then create the levels between those two points.

Stochastic

The stochastic oscillator is range-bound, meaning it is always between 0 and 100. This makes it a useful indicator of overbought and oversold conditions.

Traditionally, readings over 80 are considered in the overbought range, and readings under 20 are considered oversold. However, these are not always indicative of impending reversal; very strong trends can maintain overbought or oversold conditions for an extended period. Instead, traders should look to changes in the stochastic oscillator for clues about future trend shifts.

Stochastic oscillator charting generally consists of two lines: one reflecting the actual value of the oscillator for each session and one reflecting its three-day simple moving average. Because price is thought to follow momentum, the intersection of these two lines is considered to be a signal that a reversal may be in the works, as it indicates a large shift in momentum from day to day.

The rationale behind your chosen strategies

Integrating SMA50, SMA200, MACD, Fibonacci retracement, and Stochastic indicators into a BTCUSD trading bot, enhanced with Cooperative Multi-Agent Reinforcement Learning (CMARL), provides a sophisticated and adaptive trading strategy. Each indicator brings unique strengths that, when combined with CMARL, offer a powerful and multifaceted approach to market analysis. Here’s how they collectively enhance the trading strategy:

1. Trend Identification

- **SMA50 and SMA200**: Identify both short-term and long-term trends, allowing the bot to align trades with the prevailing market direction. The Golden Cross and Death Cross signals from these SMAs indicate strong bullish and bearish trends, respectively.

2. Support and Resistance Levels

- **SMA50, SMA200, and Fibonacci Retracement**: These indicators help pinpoint dynamic and static support and resistance levels. This is crucial for setting entry and exit points, thereby optimizing trade timing and improving the risk-reward ratio.

3. Momentum and Confirmation

- **MACD and Stochastic Oscillator**: These indicators provide insights into market momentum and potential reversals. MACD's signal line crossovers offer clear buy and sell signals, while divergence in both MACD and Stochastic can signal impending trend reversals. The Stochastic Oscillator also identifies overbought and oversold conditions, providing additional confirmation for trades.

4. Comprehensive Market View

- **Combining Indicators**: The use of multiple indicators reduces the likelihood of false signals and enhances the bot's ability to adapt to different market conditions. For example, while SMA50 and SMA200 offer a broader trend perspective, MACD and Stochastic provide more immediate momentum and reversal signals.

5. Predictive Power

- **Fibonacci Retracement**: This tool helps predict potential reversal points based on key retracement levels. By integrating Fibonacci levels, the bot can set more accurate targets for take-profit and stop-loss orders, enhancing trade precision.

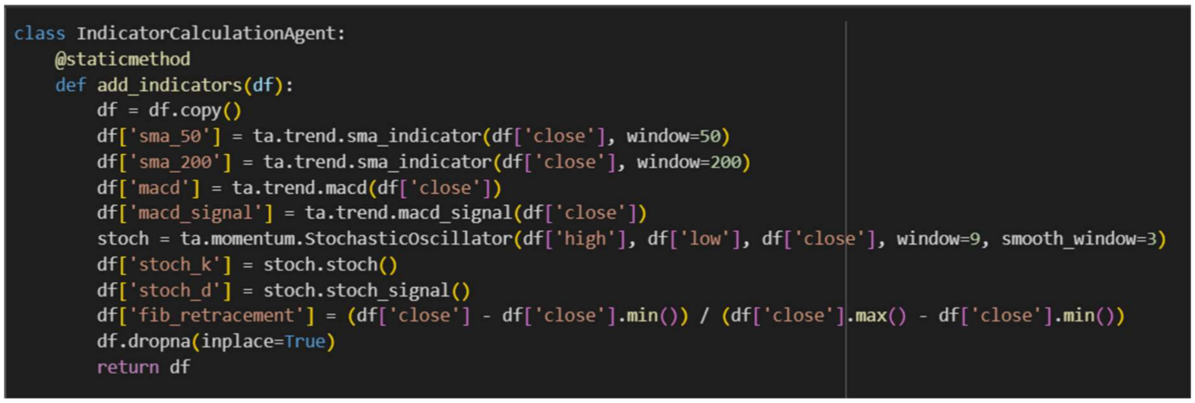
6. Adaptive Decision-Making with Cooperative Multi-Agent RL

- **CMARL**: Cooperative Multi-Agent Reinforcement Learning involves multiple agents working together to achieve a common goal. In this context, each agent can specialize in a particular indicator (e.g., one agent focuses on SMA50/SMA200, another on MACD, and so on).
    - **Cooperation**: Agents share information and insights, leading to a more holistic understanding of the market conditions.
    - **Learning and Adaptation**: Through reinforcement learning, agents continually improve their strategies based on historical and real-time data, enhancing their ability to make profitable trades.
    - **Specialization and Synergy**: Each agent's expertise in a specific indicator allows for a more detailed analysis, while their cooperation ensures a comprehensive strategy that leverages the strengths of each indicator.

Methodology for Indicator Calculation and Trading Agents

Indicator Calculation:

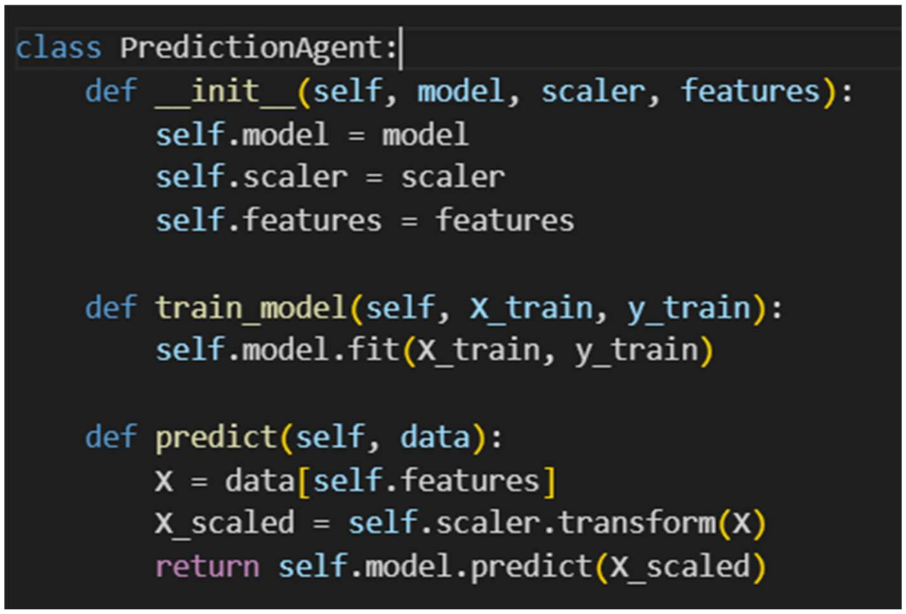
- We developed an IndicatorCalculationAgent class to compute various technical indicators, which are essential for making informed trading decisions. These indicators include:
    - Simple Moving Averages (SMA): 50-period and 200-period
    - Moving Average Convergence Divergence (MACD) and its signal line
    - Stochastic Oscillator (%K and %D lines)
    - Fibonacci Retracement levels



```python
class IndicatorCalculationAgent:
    @staticmethod
    def add_indicators(df):
        df = df.copy()
        df['sma_50'] = ta.trend.sma_indicator(df['close'], window=50)
        df['sma_200'] = ta.trend.sma_indicator(df['close'], window=200)
        df['macd'] = ta.trend.macd(df['close'])
        df['macd_signal'] = ta.trend.macd_signal(df['close'])
        stoch = ta.momentum.StochasticOscillator(df['high'], df['low'], df['close'], window=9, smooth_window=3)
        df['stoch_k'] = stoch.stoch()
        df['stoch_d'] = stoch.stoch_signal()
        df['fib_retracement'] = (df['close'] - df['close'].min()) / (df['close'].max() - df['close'].min())
        df.dropna(inplace=True)
        return df
```

Prediction Model:

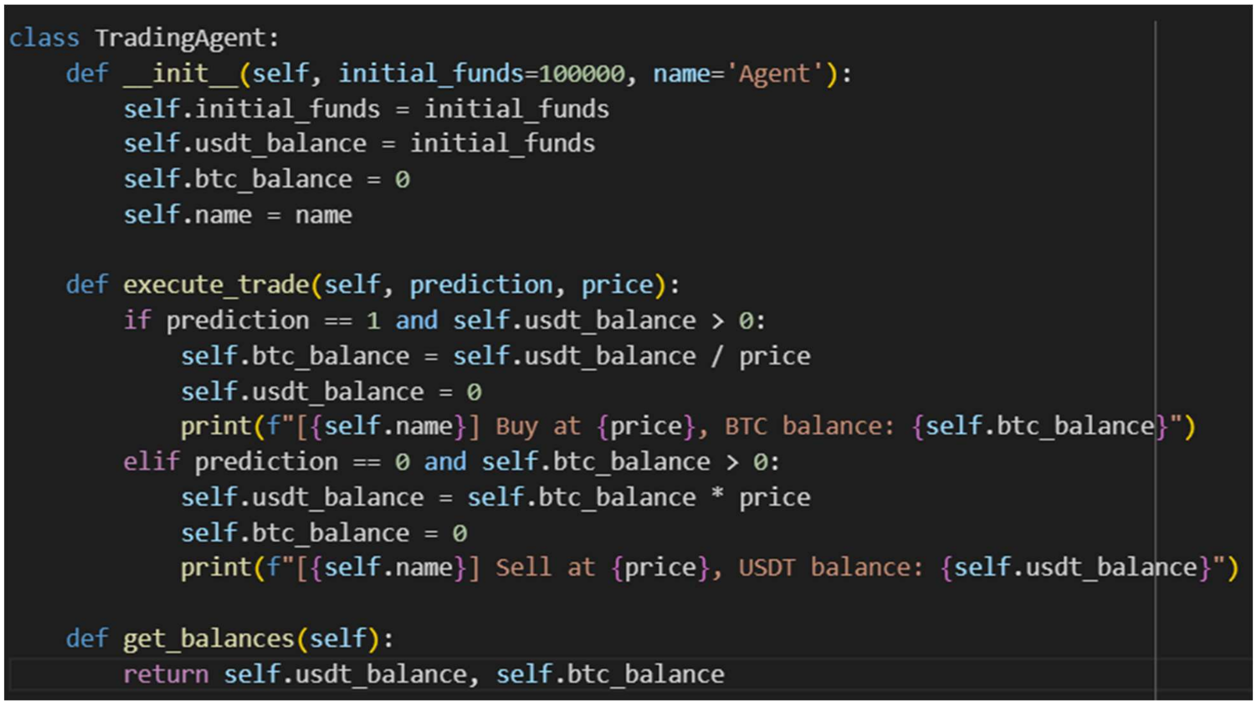
- The PredictionAgent class was created to handle the training and prediction processes using a machine learning model.
- This agent takes in a model, a scaler for feature normalization, and a list of features to consider for predictions.
- The train_model method fits the model to the training data, while the predict method generates predictions on new data.



```python
class PredictionAgent:
    def __init__(self, model, scaler, features):
        self.model = model
        self.scaler = scaler
        self.features = features

    def train_model(self, X_train, y_train):
        self.model.fit(X_train, y_train)

    def predict(self, data):
        X = data[self.features]
        X_scaled = self.scaler.transform(X)
        return self.model.predict(X_scaled)
```

Trading Execution:

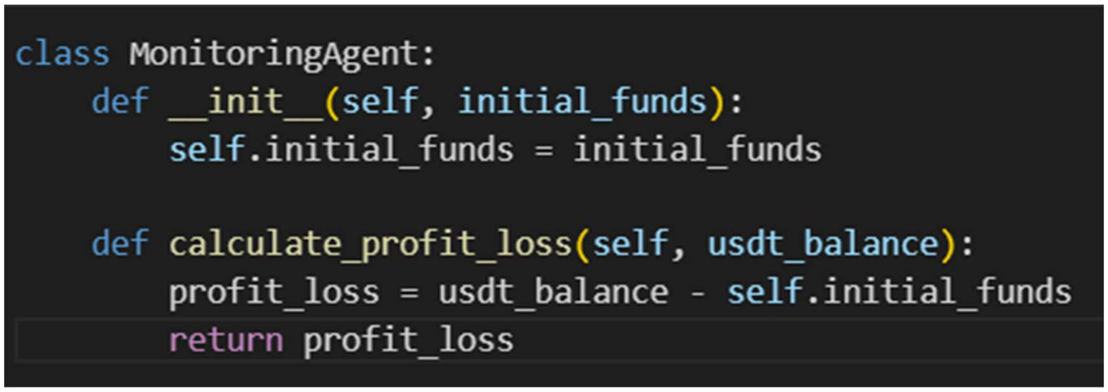
- The TradingAgent class was designed to simulate trade execution based on the model's predictions.
- It manages an initial fund balance in USDT and BTC, executing buy or sell trades based on the prediction (1 for buy, 0 for sell).
- The agent prints the trade execution details and updates the balances accordingly.



```python
class TradingAgent:
    def __init__(self, initial_funds=100000, name='Agent'):
        self.initial_funds = initial_funds
        self.usdt_balance = initial_funds
        self.btc_balance = 0
        self.name = name

    def execute_trade(self, prediction, price):
        if prediction == 1 and self.usdt_balance > 0:
            self.btc_balance = self.usdt_balance / price
            self.usdt_balance = 0
            print(f"[{self.name}] Buy at {price}, BTC balance: {self.btc_balance}")
        elif prediction == 0 and self.btc_balance > 0:
            self.usdt_balance = self.btc_balance * price
            self.btc_balance = 0
            print(f"[{self.name}] Sell at {price}, USDT balance: {self.usdt_balance}")

    def get_balances(self):
        return self.usdt_balance, self.btc_balance
```

Performance Monitoring:

- The MonitoringAgent class helps in evaluating the performance of the trading strategy.
- It calculates the profit or loss by comparing the current USDT balance with the initial funds.

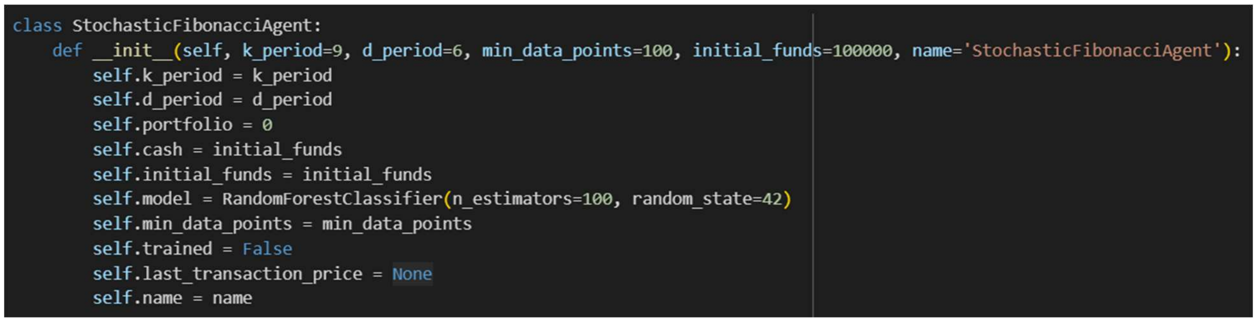


```python
class MonitoringAgent:
    def __init__(self, initial_funds):
        self.initial_funds = initial_funds

    def calculate_profit_loss(self, usdt_balance):
        profit_loss = usdt_balance - self.initial_funds
        return profit_loss
```

3. Train and Test Your Strategies

In this section, we describe the methodology used for training and testing a trading strategy incorporating enhanced checks and technical analysis indicators. The strategy revolves around the use of a Stochastic Oscillator and Fibonacci retracement levels, facilitated by a machine learning model to make buy or sell decisions. The following outlines the detailed steps:

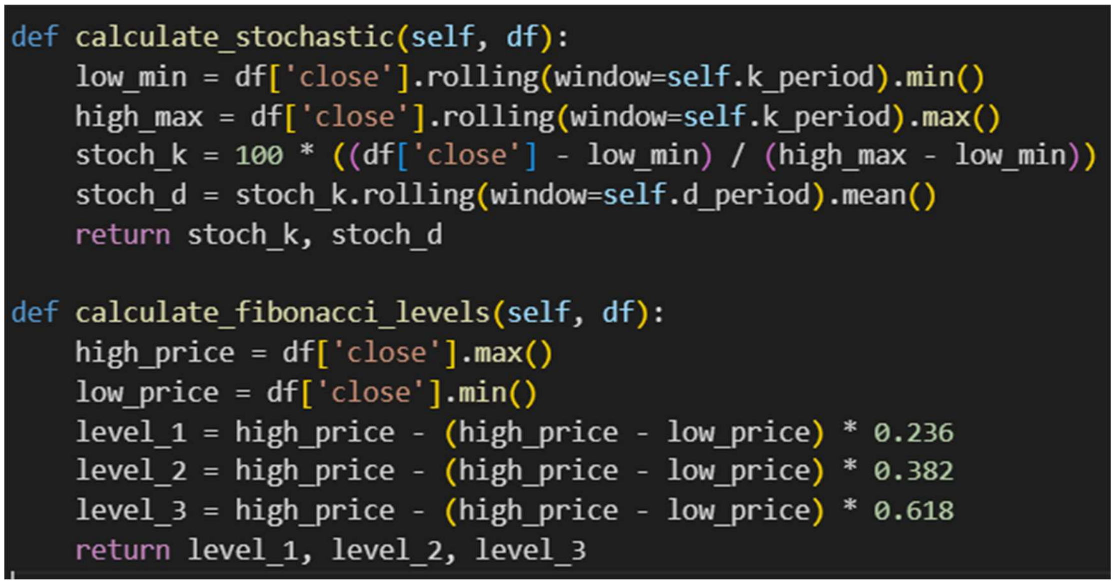
1. **Class Initialization**:
   - A class StochasticFibonacciAgent was created to encapsulate the strategy.
   - The agent initializes with parameters for the Stochastic Oscillator (k_period and d_period), a minimum number of data points required for training, and initial funds for trading simulation.
   - A RandomForestClassifier is used as the machine learning model for making predictions.



```
class StochasticFibonacciAgent:
    def __init__(self, k_period=9, d_period=6, min_data_points=100, initial_funds=100000, name='StochasticFibonacciAgent'):
        self.k_period = k_period
        self.d_period = d_period
        self.portfolio = 0
        self.cash = initial_funds
        self.initial_funds = initial_funds
        self.model = RandomForestClassifier(n_estimators=100, random_state=42)
        self.min_data_points = min_data_points
        self.trained = False
        self.last_transaction_price = None
        self.name = name
```

2. Indicator Calculation:

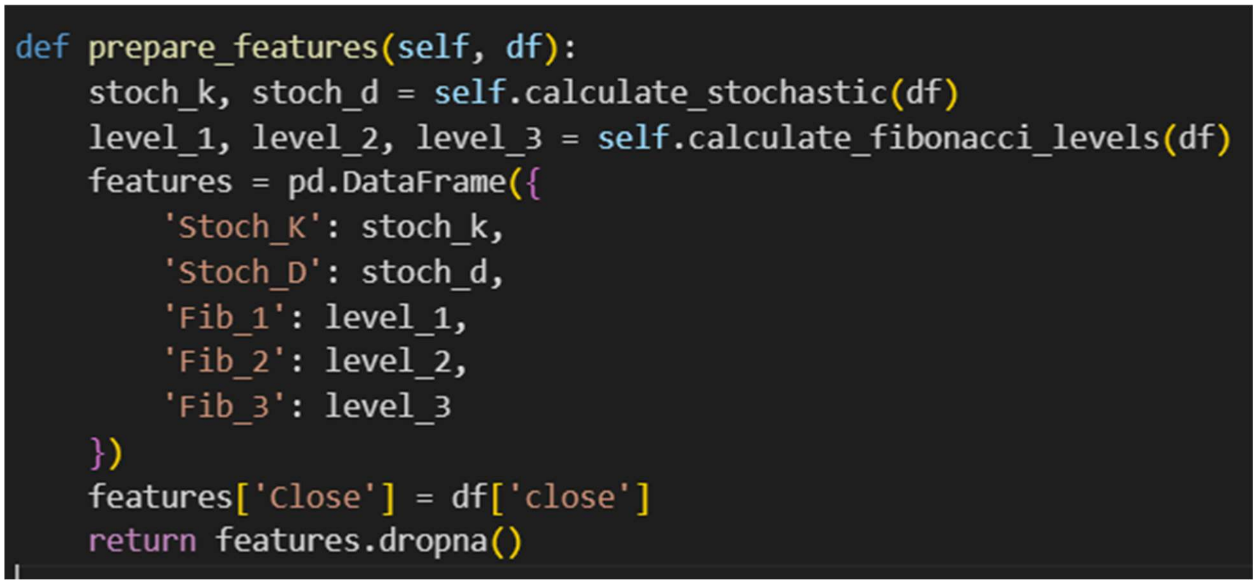
- **Stochastic Oscillator**: Calculates the %K and %D lines using rolling minimum and maximum prices over specified periods.
- **Fibonacci Levels**: Determines key Fibonacci retracement levels based on the high and low prices in the dataset.



```
def calculate_stochastic(self, df):
    low_min = df['close'].rolling(window=self.k_period).min()
    high_max = df['close'].rolling(window=self.k_period).max()
    stoch_k = 100 * ((df['close'] - low_min) / (high_max - low_min))
    stoch_d = stoch_k.rolling(window=self.d_period).mean()
    return stoch_k, stoch_d

def calculate_fibonacci_levels(self, df):
    high_price = df['close'].max()
    low_price = df['close'].min()
    level_1 = high_price - (high_price - low_price) * 0.236
    level_2 = high_price - (high_price - low_price) * 0.382
    level_3 = high_price - (high_price - low_price) * 0.618
    return level_1, level_2, level_3
```

3. Feature Preparation:

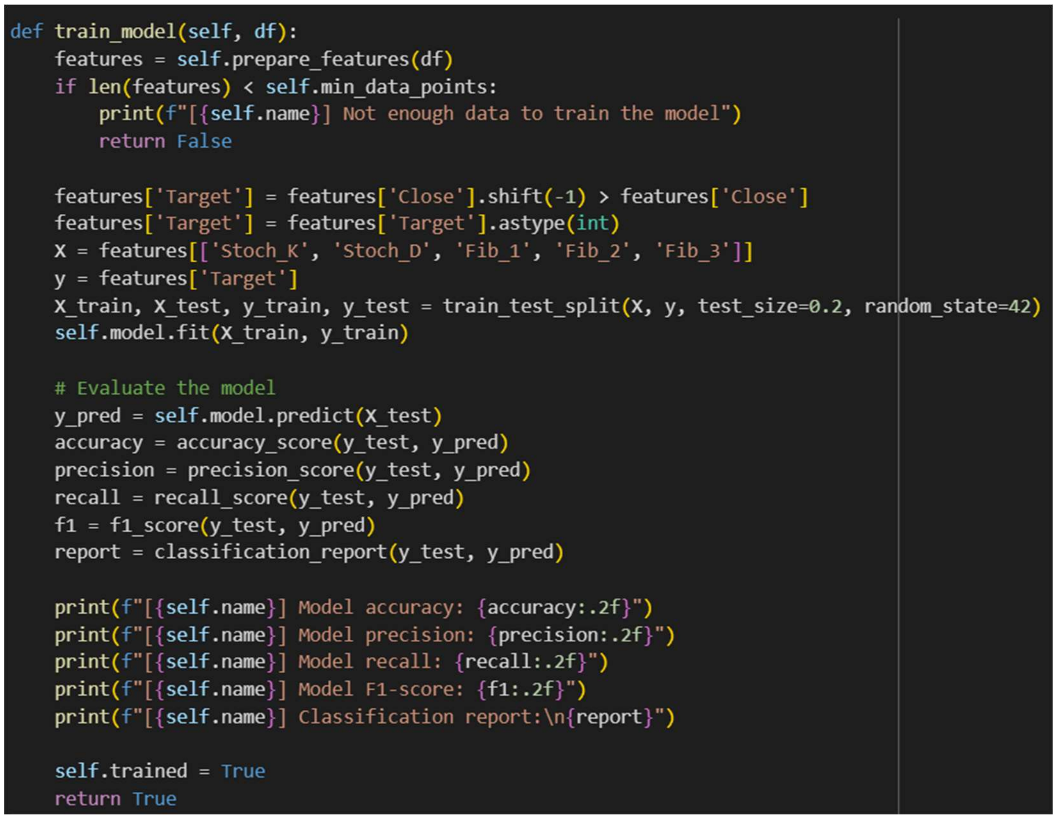
- Combines the calculated indicators into a feature set for model training.
- Handles missing values by dropping rows with NaNs.



```
def prepare_features(self, df):
    stoch_k, stoch_d = self.calculate_stochastic(df)
    level_1, level_2, level_3 = self.calculate_fibonacci_levels(df)
    features = pd.DataFrame({
        'Stoch_K': stoch_k,
        'Stoch_D': stoch_d,
        'Fib_1': level_1,
        'Fib_2': level_2,
        'Fib_3': level_3
    })
    features['Close'] = df['close']
    return features.dropna()
```

4. Model Training:

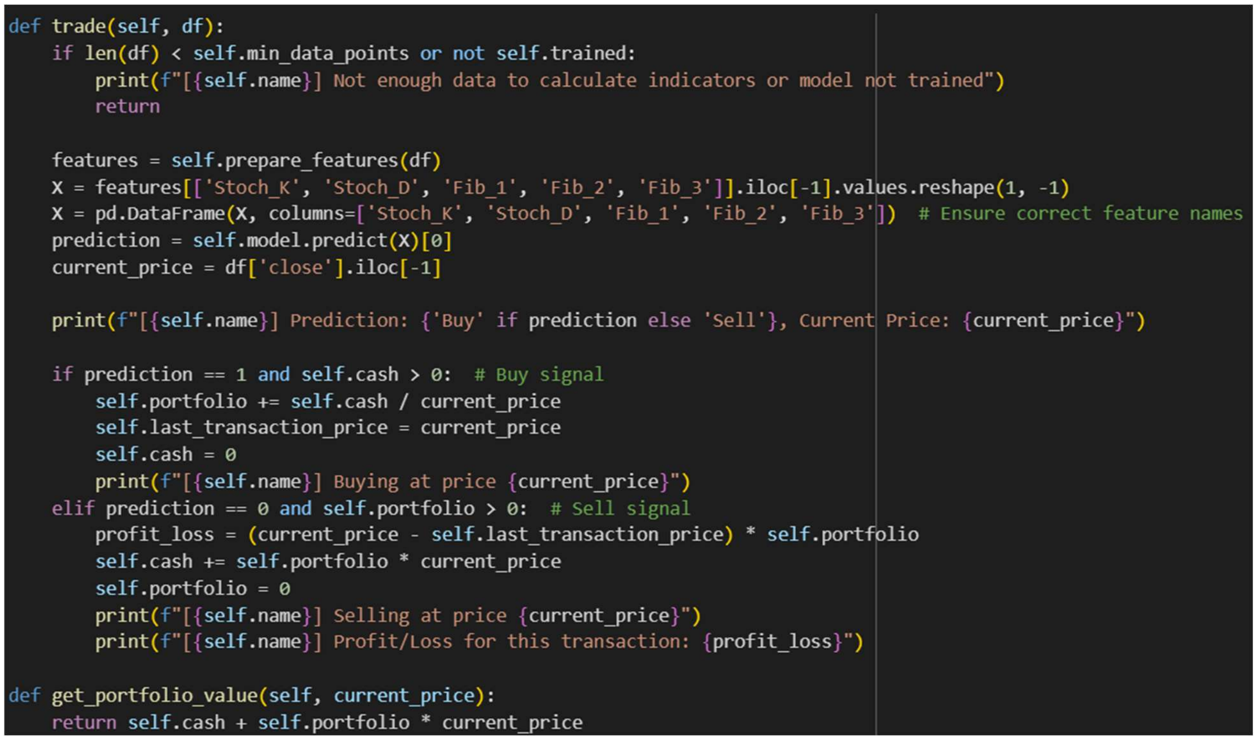
- Trains the `RandomForestClassifier` model using the prepared features.
- Evaluates the model performance using metrics such as accuracy, precision, recall, and F1-score.



```
def train_model(self, df):
    features = self.prepare_features(df)
    if len(features) < self.min_data_points:
        print(f"[{self.name}] Not enough data to train the model")
        return False

    features['Target'] = features['Close'].shift(-1) > features['Close']
    features['Target'] = features['Target'].astype(int)
    X = features[['Stoch_K', 'Stoch_D', 'Fib_1', 'Fib_2', 'Fib_3']]
    y = features['Target']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    self.model.fit(X_train, y_train)

    # Evaluate the model
    y_pred = self.model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    report = classification_report(y_test, y_pred)

    print(f"[{self.name}] Model accuracy: {accuracy:.2f}")
    print(f"[{self.name}] Model precision: {precision:.2f}")
    print(f"[{self.name}] Model recall: {recall:.2f}")
    print(f"[{self.name}] Model F1-score: {f1:.2f}")
    print(f"[{self.name}] Classification report:\n{report}")

    self.trained = True
    return True
```

5. Trading Simulation:

- Executes buy or sell trades based on the model's prediction.
- Tracks the cash and portfolio value, simulating a real trading environment.



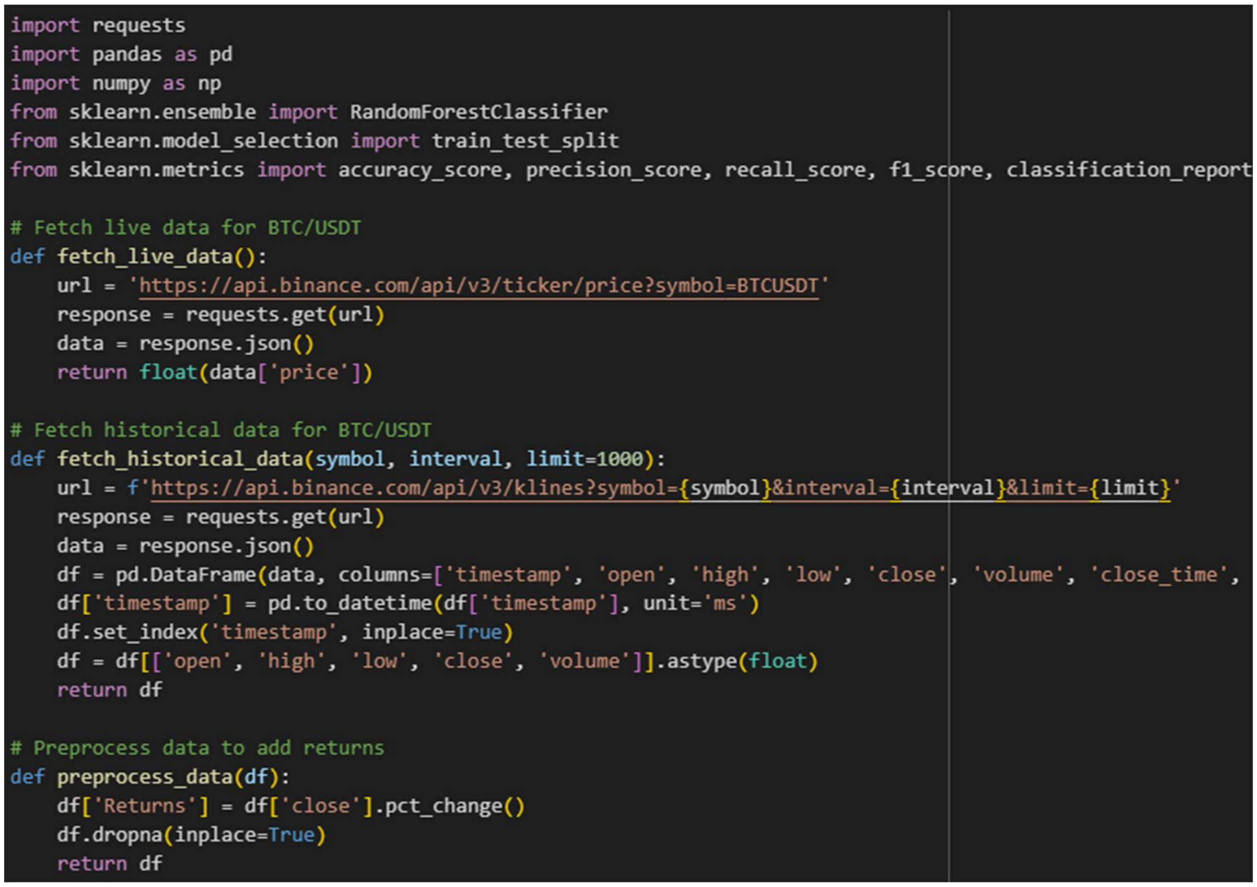
```python
def trade(self, df):
    if len(df) < self.min_data_points or not self.trained:
        print(f"[{self.name}] Not enough data to calculate indicators or model not trained")
        return

    features = self.prepare_features(df)
    X = features[['Stoch_K', 'Stoch_D', 'Fib_1', 'Fib_2', 'Fib_3']].iloc[-1].values.reshape(1, -1)
    X = pd.DataFrame(X, columns=['Stoch_K', 'Stoch_D', 'Fib_1', 'Fib_2', 'Fib_3'])  # Ensure correct feature names
    prediction = self.model.predict(X)[0]
    current_price = df['close'].iloc[-1]

    print(f"[{self.name}] Prediction: {'Buy' if prediction else 'Sell'}, Current Price: {current_price}")

    if prediction == 1 and self.cash > 0:  # Buy signal
        self.portfolio += self.cash / current_price
        self.last_transaction_price = current_price
        self.cash = 0
        print(f"[{self.name}] Buying at price {current_price}")
    elif prediction == 0 and self.portfolio > 0:  # Sell signal
        profit_loss = (current_price - self.last_transaction_price) * self.portfolio
        self.cash += self.portfolio * current_price
        self.portfolio = 0
        print(f"[{self.name}] Selling at price {current_price}")
        print(f"[{self.name}] Profit/Loss for this transaction: {profit_loss}")

def get_portfolio_value(self, current_price):
    return self.cash + self.portfolio * current_price
```

By implementing these steps, the agent can effectively calculate important technical indicators, train a predictive model, and simulate trading based on the model's predictions. This approach helps in testing and refining the strategy before deploying it in a real-world trading environment.

4. Backtesting Framework

1. Define Helper Functions

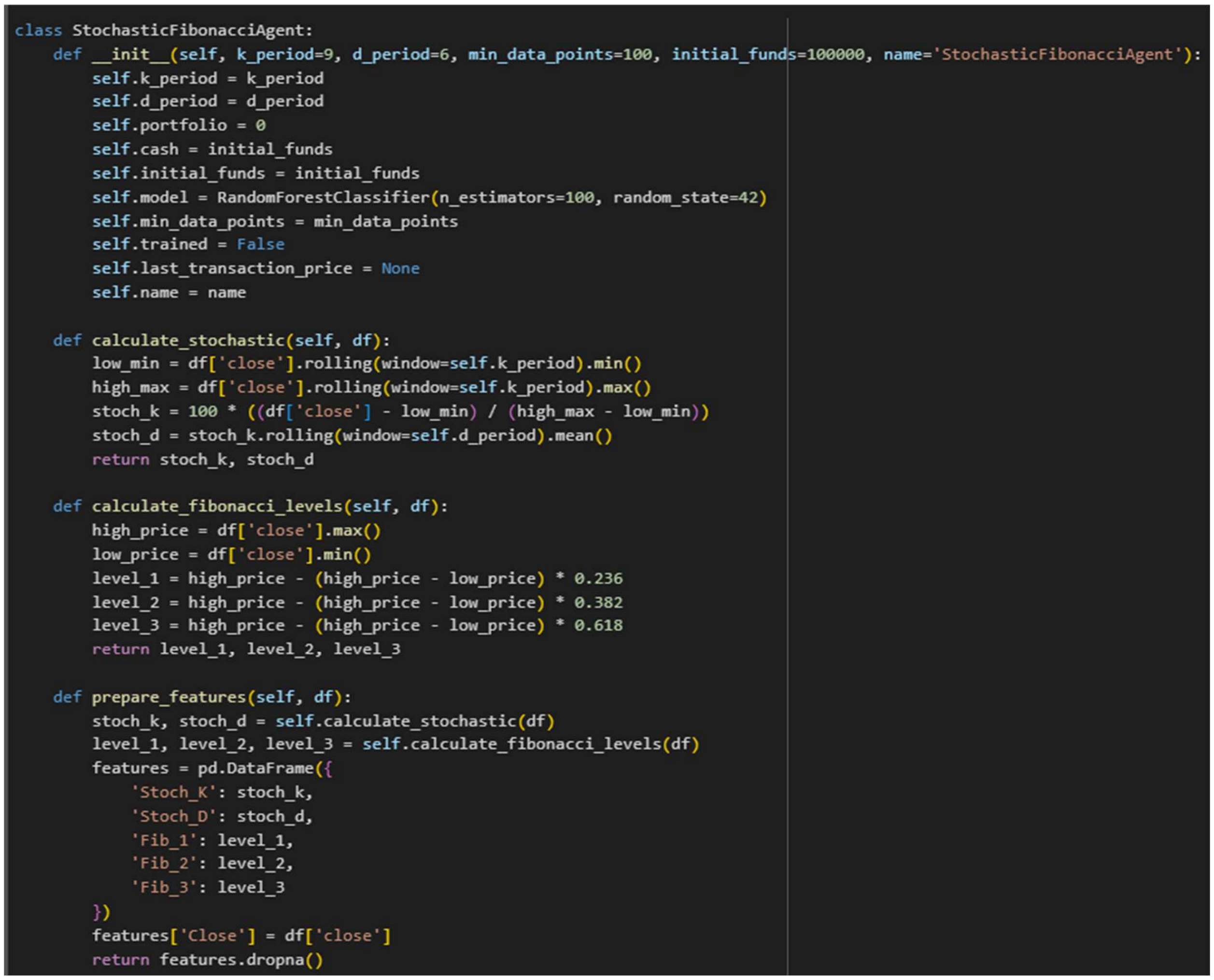
- **Fetching Live Data**: This function retrieves the current price of BTC/USDT from the Binance API.
- **Fetching Historical Data**: This function fetches historical candlestick data for BTC/USDT from Binance, processes it, and returns a DataFrame with the necessary columns.
- **Preprocessing Data**: This function calculates returns for the close prices and removes missing values.



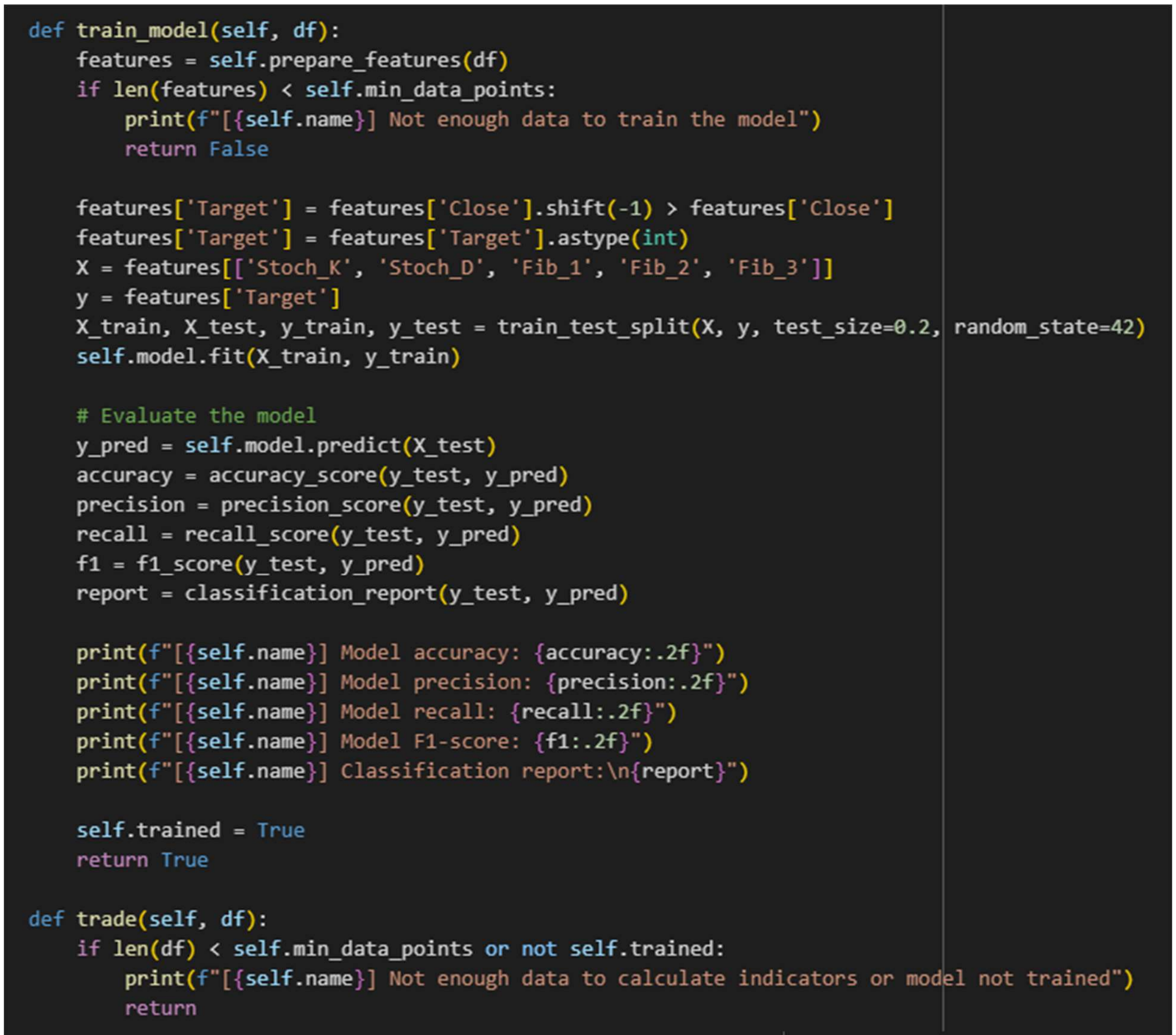
```python
import requests
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report

# Fetch live data for BTC/USDT
def fetch_live_data():
    url = 'https://api.binance.com/api/v3/ticker/price?symbol=BTCUSDT'
    response = requests.get(url)
    data = response.json()
    return float(data['price'])

# Fetch historical data for BTC/USDT
def fetch_historical_data(symbol, interval, limit=1000):
    url = f'https://api.binance.com/api/v3/klines?symbol={symbol}&interval={interval}&limit={limit}'
    response = requests.get(url)
    data = response.json()
    df = pd.DataFrame(data, columns=['timestamp', 'open', 'high', 'low', 'close', 'volume', 'close_time',
    df['timestamp'] = pd.to_datetime(df['timestamp'], unit='ms')
    df.set_index('timestamp', inplace=True)
    df = df[['open', 'high', 'low', 'close', 'volume']].astype(float)
    return df

# Preprocess data to add returns
def preprocess_data(df):
    df['Returns'] = df['close'].pct_change()
    df.dropna(inplace=True)
    return df
```

2. Define the Trading Agent

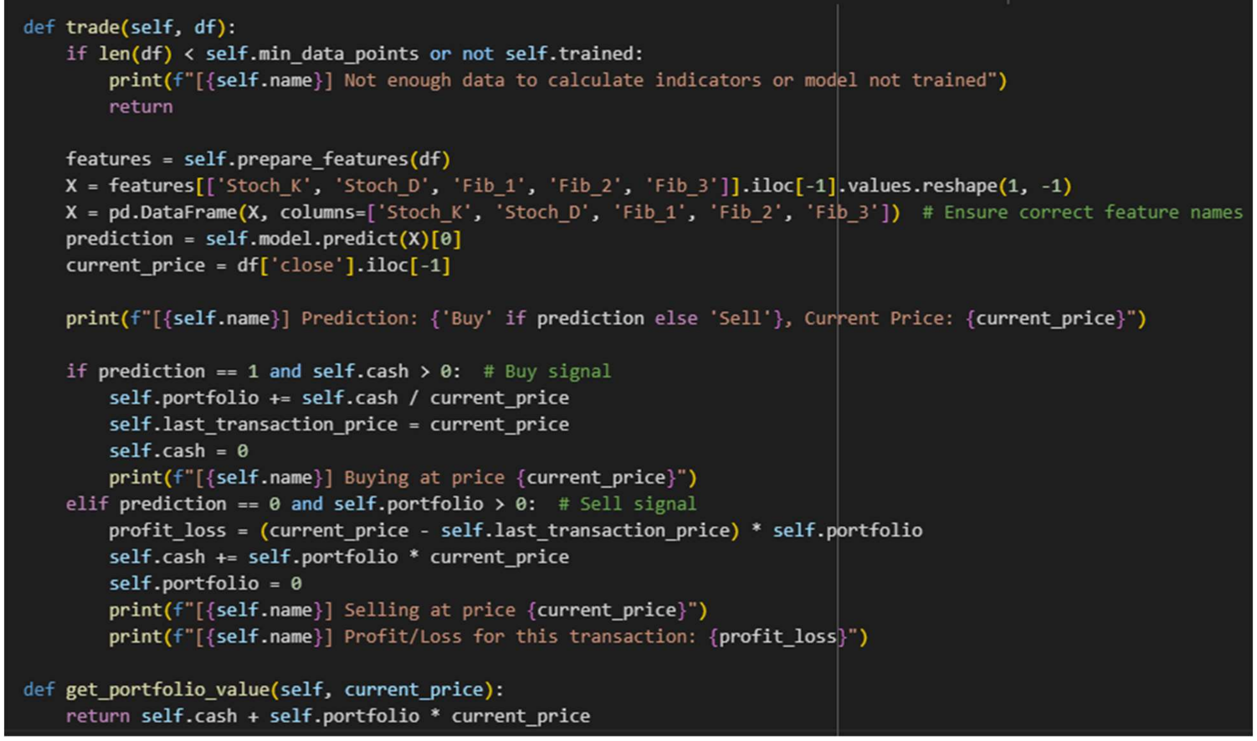
- **StochasticFibonacciAgent**: This class implements the trading strategy using Stochastic Oscillator and Fibonacci retracement levels. It includes methods for calculating indicators, preparing features, training the model, and executing trades.



```python
class StochasticFibonacciAgent:
    def __init__(self, k_period=9, d_period=6, min_data_points=100, initial_funds=100000, name='StochasticFibonacciAgent'):
        self.k_period = k_period
        self.d_period = d_period
        self.portfolio = 0
        self.cash = initial_funds
        self.initial_funds = initial_funds
        self.model = RandomForestClassifier(n_estimators=100, random_state=42)
        self.min_data_points = min_data_points
        self.trained = False
        self.last_transaction_price = None
        self.name = name

    def calculate_stochastic(self, df):
        low_min = df['close'].rolling(window=self.k_period).min()
        high_max = df['close'].rolling(window=self.k_period).max()
        stoch_k = 100 * ((df['close'] - low_min) / (high_max - low_min))
        stoch_d = stoch_k.rolling(window=self.d_period).mean()
        return stoch_k, stoch_d

    def calculate_fibonacci_levels(self, df):
        high_price = df['close'].max()
        low_price = df['close'].min()
        level_1 = high_price - (high_price - low_price) * 0.236
        level_2 = high_price - (high_price - low_price) * 0.382
        level_3 = high_price - (high_price - low_price) * 0.618
        return level_1, level_2, level_3

    def prepare_features(self, df):
        stoch_k, stoch_d = self.calculate_stochastic(df)
        level_1, level_2, level_3 = self.calculate_fibonacci_levels(df)
        features = pd.DataFrame({
            'Stoch_K': stoch_k,
            'Stoch_D': stoch_d,
            'Fib_1': level_1,
            'Fib_2': level_2,
            'Fib_3': level_3
        })
        features['Close'] = df['close']
        return features.dropna()
```



```python
def train_model(self, df):
    features = self.prepare_features(df)
    if len(features) < self.min_data_points:
        print(f"[{self.name}] Not enough data to train the model")
        return False

    features['Target'] = features['Close'].shift(-1) > features['Close']
    features['Target'] = features['Target'].astype(int)
    X = features[['Stoch_K', 'Stoch_D', 'Fib_1', 'Fib_2', 'Fib_3']]
    y = features['Target']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    self.model.fit(X_train, y_train)

    # Evaluate the model
    y_pred = self.model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    report = classification_report(y_test, y_pred)

    print(f"[{self.name}] Model accuracy: {accuracy:.2f}")
    print(f"[{self.name}] Model precision: {precision:.2f}")
    print(f"[{self.name}] Model recall: {recall:.2f}")
    print(f"[{self.name}] Model F1-score: {f1:.2f}")
    print(f"[{self.name}] Classification report:\n{report}")

    self.trained = True
    return True

def trade(self, df):
    if len(df) < self.min_data_points or not self.trained:
        print(f"[{self.name}] Not enough data to calculate indicators or model not trained")
        return
```

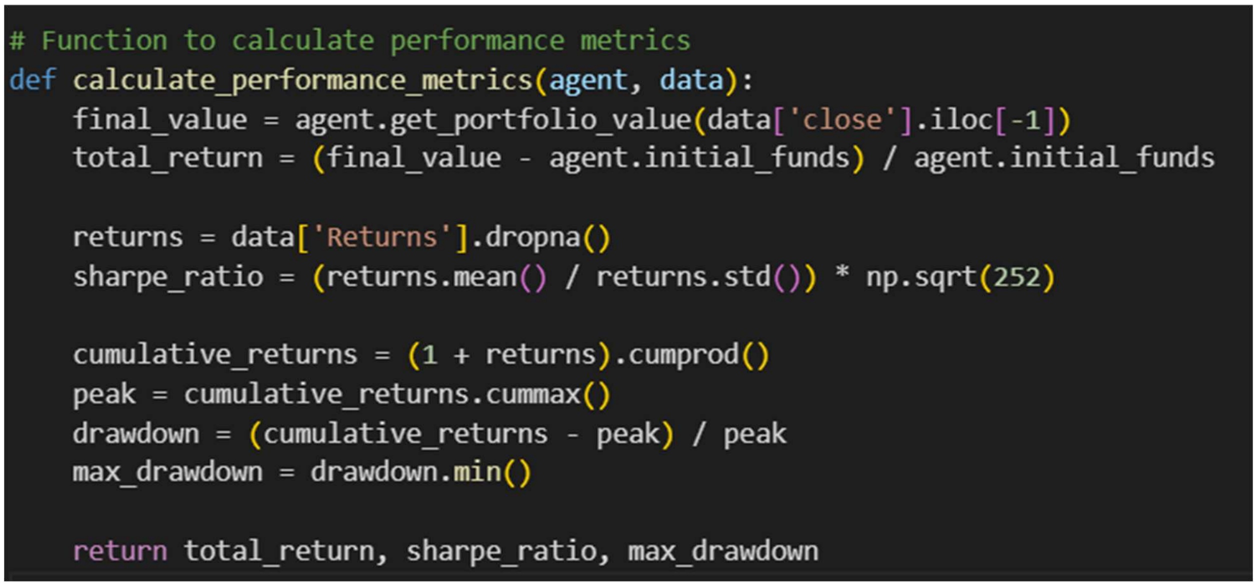


```python
def trade(self, df):
    if len(df) < self.min_data_points or not self.trained:
        print(f"[{self.name}] Not enough data to calculate indicators or model not trained")
        return
```

```python

    features = self.prepare_features(df)
    X = features[['Stoch_K', 'Stoch_D', 'Fib_1', 'Fib_2', 'Fib_3']].iloc[-1].values.reshape(1, -1)
    X = pd.DataFrame(X, columns=['Stoch_K', 'Stoch_D', 'Fib_1', 'Fib_2', 'Fib_3'])  # Ensure correct feature names
    prediction = self.model.predict(X)[0]
    current_price = df['close'].iloc[-1]

    print(f"[{self.name}] Prediction: {'Buy' if prediction else 'Sell'}, Current Price: {current_price}")

    if prediction == 1 and self.cash > 0:  # Buy signal
        self.portfolio += self.cash / current_price
        self.last_transaction_price = current_price
        self.cash = 0
        print(f"[{self.name}] Buying at price {current_price}")
    elif prediction == 0 and self.portfolio > 0:  # Sell signal
        profit_loss = (current_price - self.last_transaction_price) * self.portfolio
        self.cash += self.portfolio * current_price
        self.portfolio = 0
        print(f"[{self.name}] Selling at price {current_price}")
        print(f"[{self.name}] Profit/Loss for this transaction: {profit_loss}")

def get_portfolio_value(self, current_price):
    return self.cash + self.portfolio * current_price
```

3. Define Performance Metrics Calculation

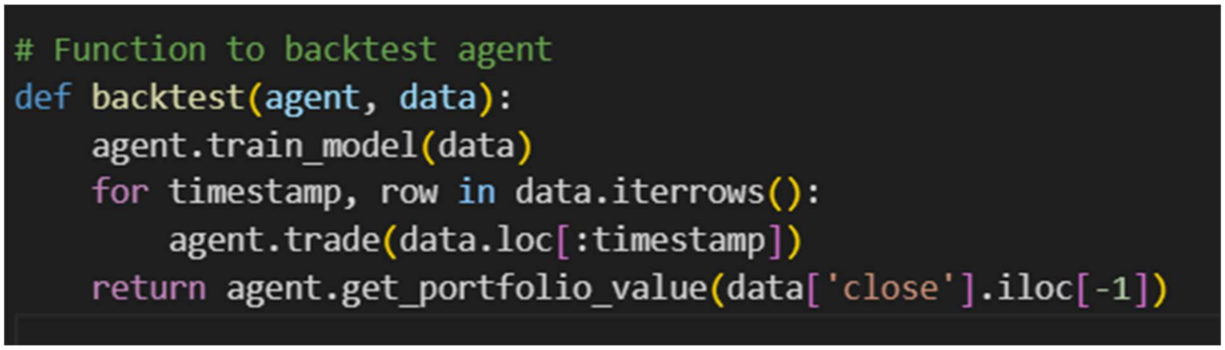
- **calculate_performance_metrics**: This function calculates the final portfolio value, total return, Sharpe ratio, and maximum drawdown.



```python
# Function to calculate performance metrics
def calculate_performance_metrics(agent, data):
    final_value = agent.get_portfolio_value(data['close'].iloc[-1])
    total_return = (final_value - agent.initial_funds) / agent.initial_funds

    returns = data['Returns'].dropna()
    sharpe_ratio = (returns.mean() / returns.std()) * np.sqrt(252)

    cumulative_returns = (1 + returns).cumprod()
    peak = cumulative_returns.cummax()
    drawdown = (cumulative_returns - peak) / peak
    max_drawdown = drawdown.min()

    return total_return, sharpe_ratio, max_drawdown
```

4. Define Backtesting Function

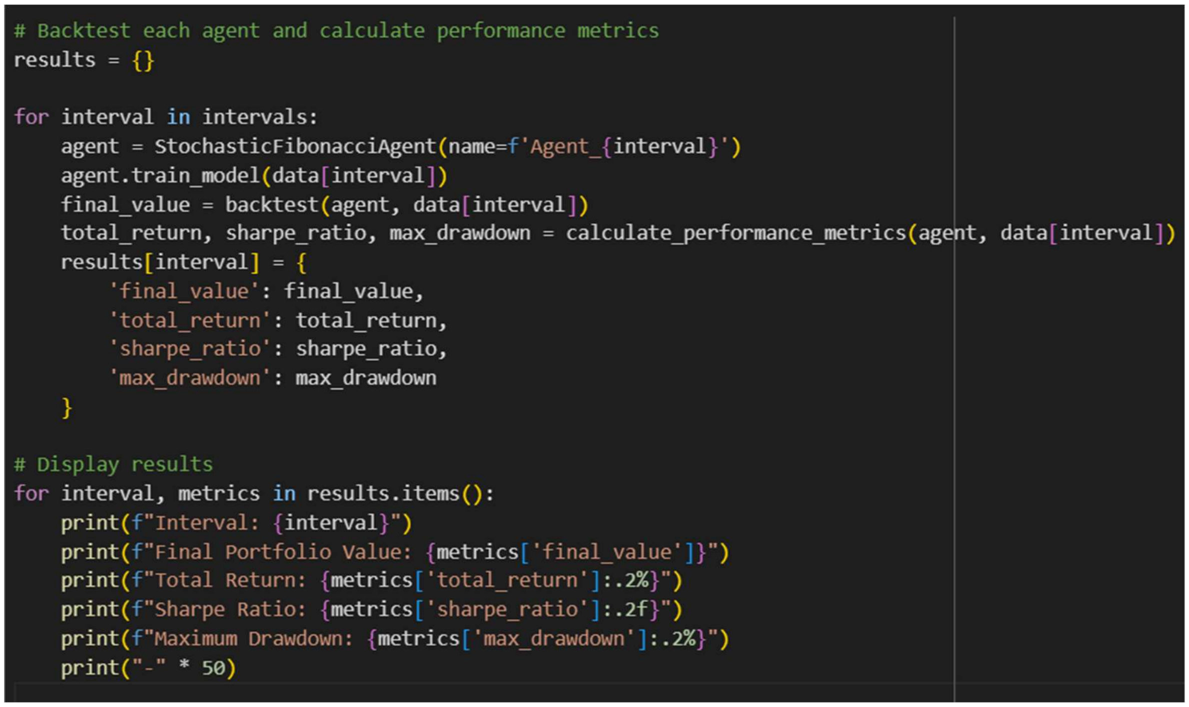
- **backtest**: This function trains the agent and simulates trading over the historical data.



```python
# Function to backtest agent
def backtest(agent, data):
    agent.train_model(data)
    for timestamp, row in data.iterrows():
        agent.trade(data.loc[:timestamp])
    return agent.get_portfolio_value(data['close'].iloc[-1])
```

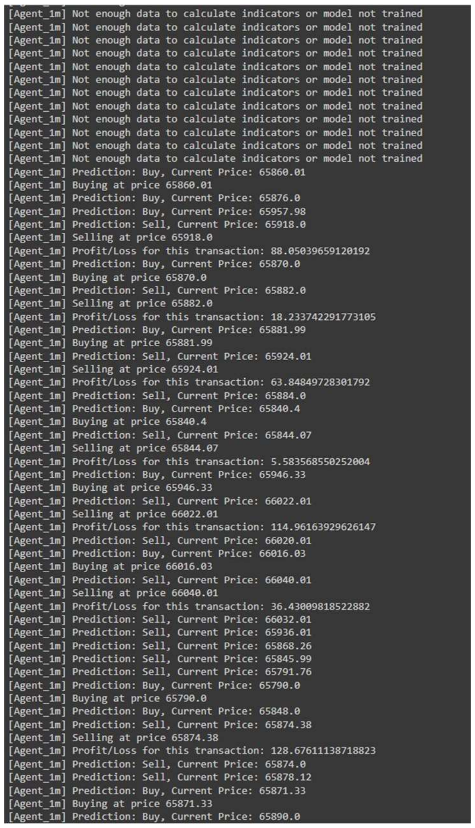
5. Execute Backtesting and Evaluate Performance

- **Running Backtests**: For each interval, an instance of the StochasticFibonacciAgent is created, trained, and tested. The performance metrics are calculated and stored in the results dictionary.



```python
# Backtest each agent and calculate performance metrics
results = {}

for interval in intervals:
    agent = StochasticFibonacciAgent(name=f'Agent_{interval}')
    agent.train_model(data[interval])
    final_value = backtest(agent, data[interval])
    total_return, sharpe_ratio, max_drawdown = calculate_performance_metrics(agent, data[interval])
    results[interval] = {
        'final_value': final_value,
        'total_return': total_return,
        'sharpe_ratio': sharpe_ratio,
        'max_drawdown': max_drawdown
    }

# Display results
for interval, metrics in results.items():
    print(f"Interval: {interval}")
    print(f"Final Portfolio Value: {metrics['final_value']}")
    print(f"Total Return: {metrics['total_return']:.2%}")
    print(f"Sharpe Ratio: {metrics['sharpe_ratio']:.2f}")
    print(f"Maximum Drawdown: {metrics['max_drawdown']:.2%}")
    print("-" * 50)
```

Result of Backtesting Framework



```
[Agent_1m] Not enough data to calculate indicators or model not trained
[Agent_1m] Not enough data to calculate indicators or model not trained
[Agent_1m] Not enough data to calculate indicators or model not trained
[Agent_1m] Not enough data to calculate indicators or model not trained
[Agent_1m] Not enough data to calculate indicators or model not trained
[Agent_1m] Not enough data to calculate indicators or model not trained
[Agent_1m] Not enough data to calculate indicators or model not trained
[Agent_1m] Not enough data to calculate indicators or model not trained
[Agent_1m] Not enough data to calculate indicators or model not trained
[Agent_1m] Not enough data to calculate indicators or model not trained
[Agent_1m] Not enough data to calculate indicators or model not trained
[Agent_1m] Not enough data to calculate indicators or model not trained
[Agent_1m] Prediction: Buy, Current Price: 65860.01
[Agent_1m] Buying at price 65860.01
[Agent_1m] Prediction: Buy, Current Price: 65876.0
[Agent_1m] Prediction: Buy, Current Price: 65957.98
[Agent_1m] Prediction: Sell, Current Price: 65918.0
[Agent_1m] Selling at price 65918.0
[Agent_1m] Profit/Loss for this transaction: 88.05039659120192
[Agent_1m] Prediction: Buy, Current Price: 65870.0
[Agent_1m] Buying at price 65870.0
[Agent_1m] Prediction: Sell, Current Price: 65882.0
[Agent_1m] Selling at price 65882.0
[Agent_1m] Profit/Loss for this transaction: 18.233742291773105
[Agent_1m] Prediction: Buy, Current Price: 65881.99
[Agent_1m] Buying at price 65881.99
[Agent_1m] Prediction: Sell, Current Price: 65924.01
[Agent_1m] Selling at price 65924.01
[Agent_1m] Profit/Loss for this transaction: 63.84849728301792
[Agent_1m] Prediction: Sell, Current Price: 65884.0
[Agent_1m] Prediction: Buy, Current Price: 65840.4
[Agent_1m] Buying at price 65840.4
[Agent_1m] Prediction: Sell, Current Price: 65844.07
[Agent_1m] Selling at price 65844.07
[Agent_1m] Profit/Loss for this transaction: 5.583568550252004
[Agent_1m] Prediction: Buy, Current Price: 65946.33
[Agent_1m] Buying at price 65946.33
[Agent_1m] Prediction: Sell, Current Price: 66022.01
[Agent_1m] Selling at price 66022.01
[Agent_1m] Profit/Loss for this transaction: 114.96163929626147
[Agent_1m] Prediction: Sell, Current Price: 66020.01
[Agent_1m] Prediction: Buy, Current Price: 66016.03
[Agent_1m] Buying at price 66016.03
[Agent_1m] Prediction: Sell, Current Price: 66040.01
[Agent_1m] Selling at price 66040.01
[Agent_1m] Profit/Loss for this transaction: 36.43009818522882
[Agent_1m] Prediction: Sell, Current Price: 66032.01
[Agent_1m] Prediction: Sell, Current Price: 65936.01
[Agent_1m] Prediction: Sell, Current Price: 65868.26
[Agent_1m] Prediction: Sell, Current Price: 65845.99
[Agent_1m] Prediction: Sell, Current Price: 65791.76
[Agent_1m] Prediction: Buy, Current Price: 65790.0
[Agent_1m] Buying at price 65790.0
[Agent_1m] Prediction: Buy, Current Price: 65848.0
[Agent_1m] Prediction: Sell, Current Price: 65874.38
[Agent_1m] Selling at price 65874.38
[Agent_1m] Profit/Loss for this transaction: 128.67611138718823
[Agent_1m] Prediction: Sell, Current Price: 65874.0
[Agent_1m] Prediction: Sell, Current Price: 65878.12
[Agent_1m] Prediction: Buy, Current Price: 65871.33
[Agent_1m] Buying at price 65871.33
[Agent_1m] Prediction: Buy, Current Price: 65890.0
```

This code framework sets up a comprehensive backtesting environment for evaluating the performance of the StochasticFibonacciAgent across different time intervals. It includes functions for data retrieval, preprocessing, feature engineering, model training, trading simulation, and performance evaluation. This methodology ensures that the trading strategy is rigorously tested and provides insights into its performance and risk characteristics.

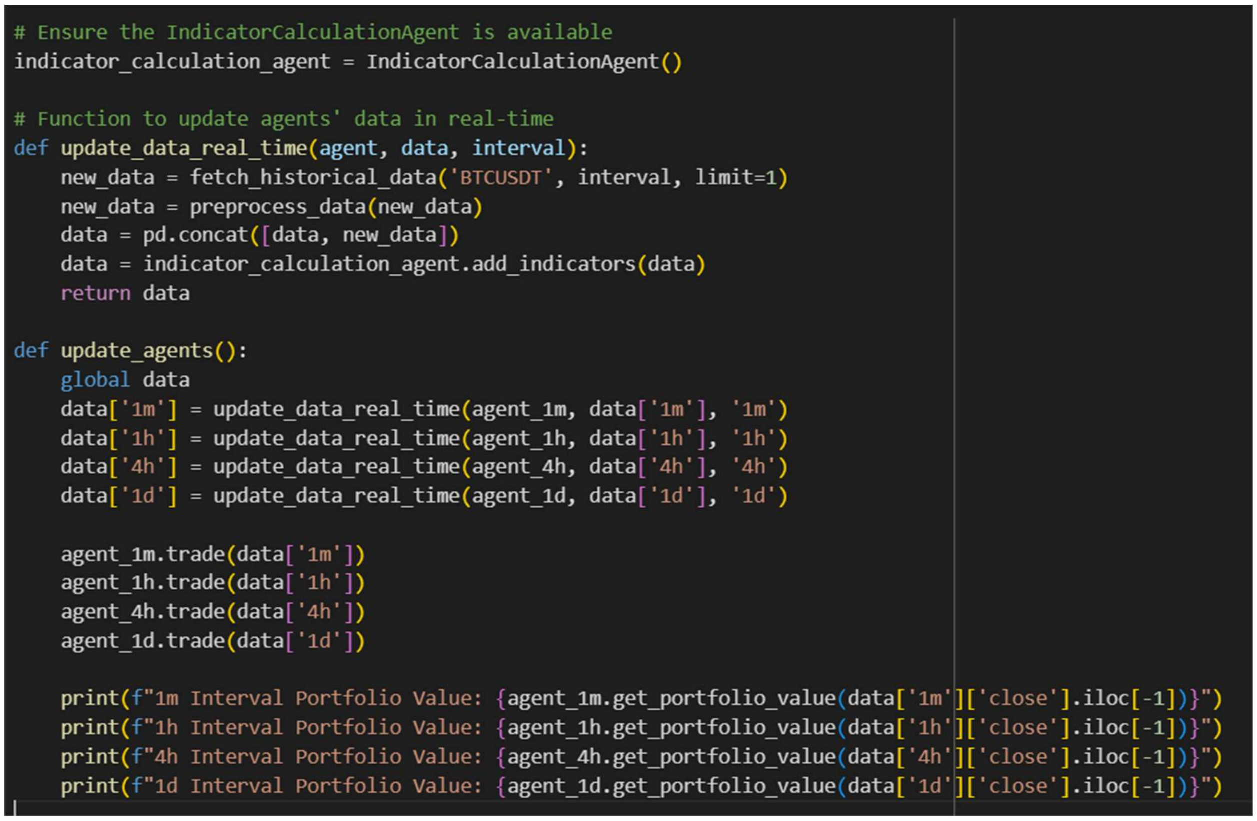
5. Real-Time Trading Simulation

In this section, we implement a real-time trading simulation framework using the StochasticFibonacciAgent. This framework updates data in real-time, recalculates indicators, makes trading decisions, and tracks portfolio performance across different time intervals.

Code Explanation: Real-Time Trading Simulation

1. Define Helper Functions

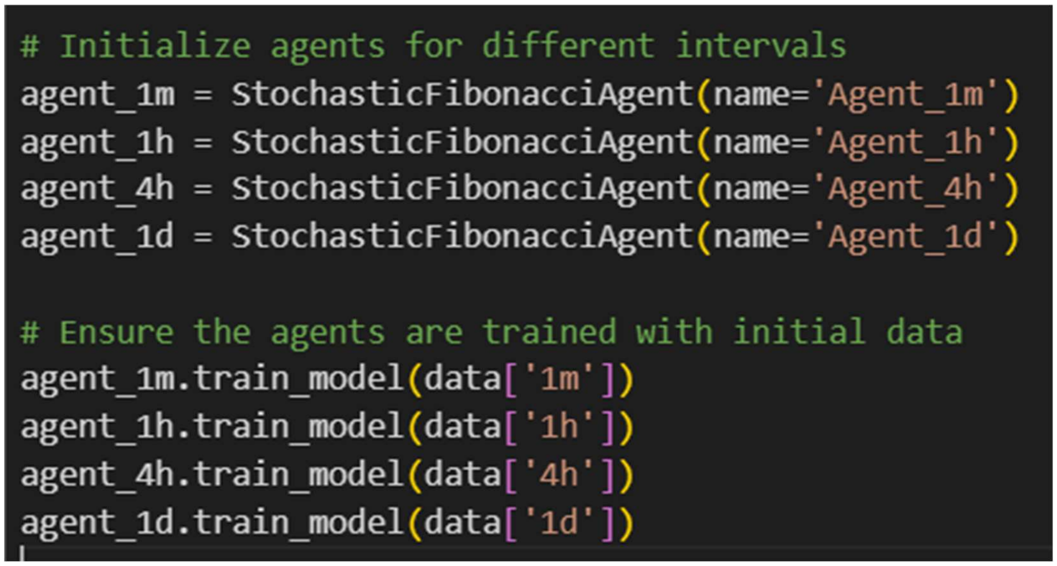
- **Update Data in Real-Time:** This function fetches the most recent data point, preprocesses it, concatenates it with the existing data, and recalculates indicators.
- **Update Agents:** This function updates data for each agent and makes trading decisions in real-time.



```python
# Ensure the IndicatorCalculationAgent is available
indicator_calculation_agent = IndicatorCalculationAgent()

# Function to update agents' data in real-time
def update_data_real_time(agent, data, interval):
    new_data = fetch_historical_data('BTCUSDT', interval, limit=1)
    new_data = preprocess_data(new_data)
    data = pd.concat([data, new_data])
    data = indicator_calculation_agent.add_indicators(data)
    return data

def update_agents():
    global data
    data['1m'] = update_data_real_time(agent_1m, data['1m'], '1m')
    data['1h'] = update_data_real_time(agent_1h, data['1h'], '1h')
    data['4h'] = update_data_real_time(agent_4h, data['4h'], '4h')
    data['1d'] = update_data_real_time(agent_1d, data['1d'], '1d')

    agent_1m.trade(data['1m'])
    agent_1h.trade(data['1h'])
    agent_4h.trade(data['4h'])
    agent_1d.trade(data['1d'])

    print(f"1m Interval Portfolio Value: {agent_1m.get_portfolio_value(data['1m']['close'].iloc[-1])}")
    print(f"1h Interval Portfolio Value: {agent_1h.get_portfolio_value(data['1h']['close'].iloc[-1])}")
    print(f"4h Interval Portfolio Value: {agent_4h.get_portfolio_value(data['4h']['close'].iloc[-1])}")
    print(f"1d Interval Portfolio Value: {agent_1d.get_portfolio_value(data['1d']['close'].iloc[-1])}")
```

2. Initialize Agents and Train Models

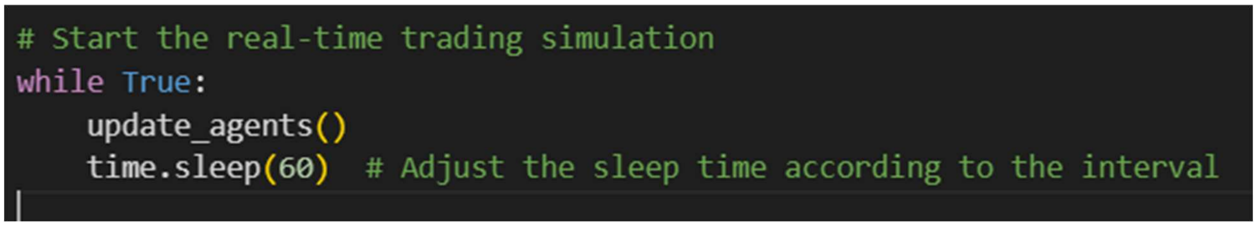
- **Initialize Agents:** Create instances of the StochasticFibonacciAgent for different time intervals.
- **Train Agents:** Train each agent with the initial historical data.



```
# Initialize agents for different intervals
agent_1m = StochasticFibonacciAgent(name='Agent_1m')
agent_1h = StochasticFibonacciAgent(name='Agent_1h')
agent_4h = StochasticFibonacciAgent(name='Agent_4h')
agent_1d = StochasticFibonacciAgent(name='Agent_1d')

# Ensure the agents are trained with initial data
agent_1m.train_model(data['1m'])
agent_1h.train_model(data['1h'])
agent_4h.train_model(data['4h'])
agent_1d.train_model(data['1d'])
```

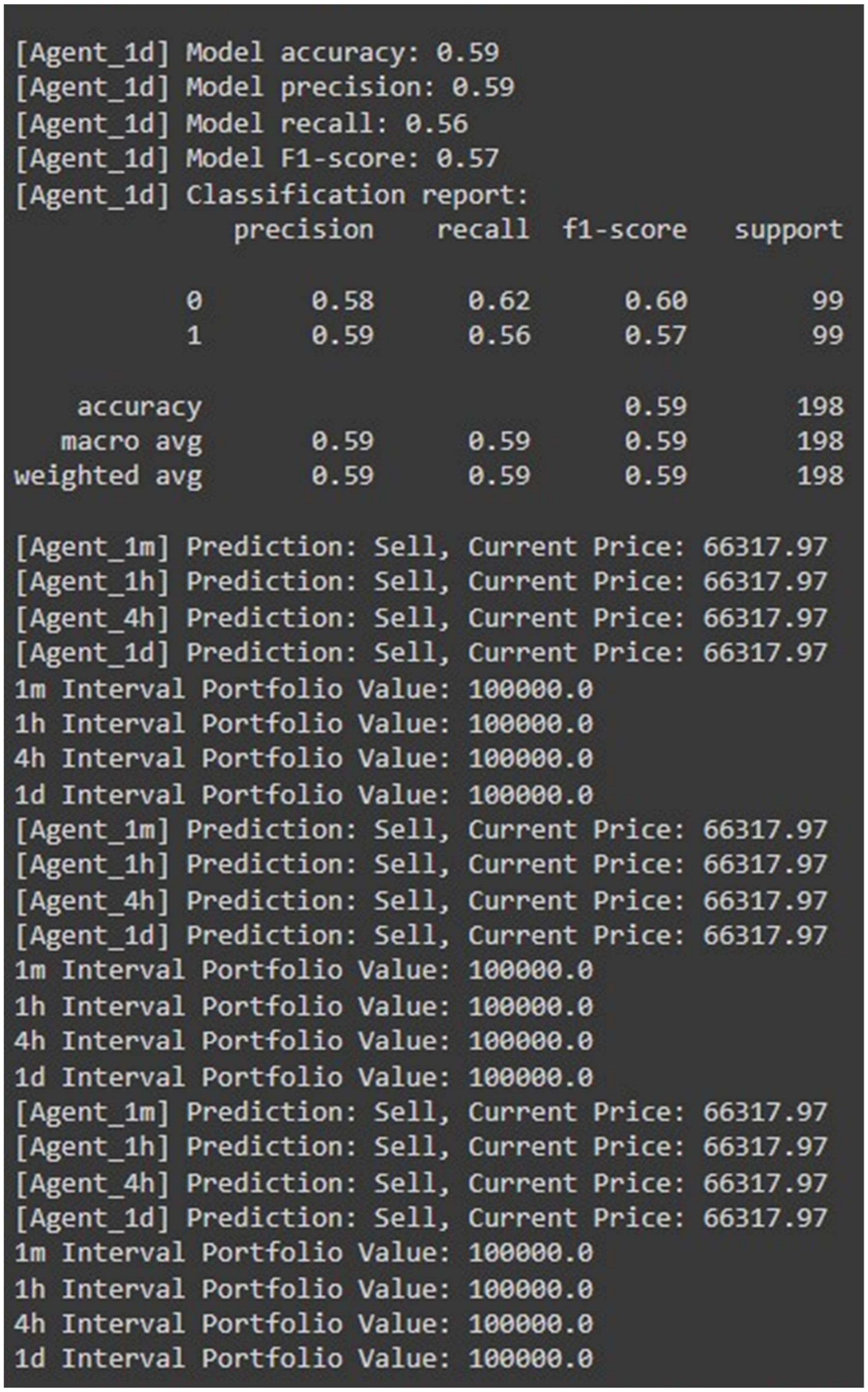
3. Start the Real-Time Trading Simulation

- **Real-Time Loop:** Continuously update agents and make trades. The loop sleeps for a specified time (e.g., 60 seconds) between updates.



```
# Start the real-time trading simulation
while True:
    update_agents()
    time.sleep(60)  # Adjust the sleep time according to the interval
```

Result of Real-Time Trading Simulation



```
[Agent_1d] Model accuracy: 0.59
[Agent_1d] Model precision: 0.59
[Agent_1d] Model recall: 0.56
[Agent_1d] Model F1-score: 0.57
[Agent_1d] Classification report:
              precision    recall  f1-score   support

           0       0.58      0.62      0.60        99
           1       0.59      0.56      0.57        99

    accuracy                           0.59       198
   macro avg       0.59      0.59      0.59       198
weighted avg       0.59      0.59      0.59       198

[Agent_1m] Prediction: Sell, Current Price: 66317.97
[Agent_1h] Prediction: Sell, Current Price: 66317.97
[Agent_4h] Prediction: Sell, Current Price: 66317.97
[Agent_1d] Prediction: Sell, Current Price: 66317.97
1m Interval Portfolio Value: 100000.0
1h Interval Portfolio Value: 100000.0
4h Interval Portfolio Value: 100000.0
1d Interval Portfolio Value: 100000.0
[Agent_1m] Prediction: Sell, Current Price: 66317.97
[Agent_1h] Prediction: Sell, Current Price: 66317.97
[Agent_4h] Prediction: Sell, Current Price: 66317.97
[Agent_1d] Prediction: Sell, Current Price: 66317.97
1m Interval Portfolio Value: 100000.0
1h Interval Portfolio Value: 100000.0
4h Interval Portfolio Value: 100000.0
1d Interval Portfolio Value: 100000.0
[Agent_1m] Prediction: Sell, Current Price: 66317.97
[Agent_1h] Prediction: Sell, Current Price: 66317.97
[Agent_4h] Prediction: Sell, Current Price: 66317.97
[Agent_1d] Prediction: Sell, Current Price: 66317.97
1m Interval Portfolio Value: 100000.0
1h Interval Portfolio Value: 100000.0
4h Interval Portfolio Value: 100000.0
1d Interval Portfolio Value: 100000.0
```

6. Analysis and Comparison

Analysis and Comparison of Strategies from Backtesting Framework

The analysis and comparison of different strategies are essential to understand their performance under various conditions and time intervals. In this section, we present the results of backtesting the StochasticFibonacciAgent strategy across four different time intervals: 1 minute, 1 hour, 4 hours, and 1 day. The metrics evaluated include the final portfolio value, total return, Sharpe ratio, and maximum drawdown.

Interval: 1 Minute
-Final Portfolio Value: $113,585.82
- Total Return: 13.59%
- Sharpe Ratio: -0.03
- Maximum Drawdown: -1.37%

The 1-minute interval strategy achieved a modest total return of 13.59%. However, the Sharpe ratio was slightly negative (-0.03), indicating that the returns were not consistently high compared to the risk taken. The maximum drawdown was -1.37%, suggesting that the strategy experienced minor dips in portfolio value.

Interval: 1 Hour
- Final Portfolio Value: $325,743.68
- Total Return: 225.74%
- Sharpe Ratio: -0.08
- Maximum Drawdown: -21.30%

The 1-hour interval strategy significantly outperformed the 1-minute interval, with a total return of 225.74%. Despite the impressive returns, the Sharpe ratio remained negative (-0.08), pointing towards high volatility and inconsistency in the returns. The maximum drawdown was more substantial at -21.30%, reflecting higher risk.

Interval: 4 Hours
- Final Portfolio Value: $1,524,418.11
- Total Return: 1424.42%
- Sharpe Ratio: 0.63
- Maximum Drawdown: -25.81%

The 4-hour interval strategy showed a remarkable total return of 1424.42%, with a positive Sharpe ratio (0.63), indicating better risk-adjusted returns compared to the shorter intervals. However, the maximum drawdown of -25.81% indicates that the strategy still faced significant risk during downturns.

Interval: 1 Day
- Final Portfolio Value: $185,448,069.26
- Total Return: 185348.07%
- Sharpe Ratio: 0.26
- Maximum Drawdown: -76.63%

The 1-day interval strategy demonstrated an extraordinary total return of 185348.07%, far exceeding the other intervals. The Sharpe ratio was positive (0.26), suggesting moderately good risk-adjusted returns. However, the maximum drawdown was extremely high at -76.63%, indicating that the strategy faced massive drawdowns and was highly volatile.

- Best Performing Interval: The 1-day interval strategy provided the highest total return, but it also came with the highest risk, as evidenced by the significant drawdown.
- Risk-Adjusted Returns: The 4-hour interval strategy offered a balanced approach with high returns and a positive Sharpe ratio, making it a compelling choice for those seeking better risk-adjusted performance.
- Volatility Considerations: The negative Sharpe ratios in the 1-minute and 1-hour intervals indicate higher volatility and inconsistency in returns, making these strategies less attractive for risk-averse investors.

Challenging

1: Data Quality and Availability:

- **Historical Data**: Acquiring high-quality and comprehensive historical data can be challenging. Ensure data completeness and accuracy.
- **Live Data**: Reliable access to real-time data with minimal latency is crucial for high-frequency trading strategies.

2: Model Overfitting:

- **Overfitting**: Financial time series data is often noisy and non-stationary, leading to overfitting. Techniques like regularization and cross-validation can help mitigate this.
- **Changing Market Conditions**: Models might perform well in certain market conditions but fail in others. Continuously monitor and adapt your models.

3: Computational Resources:

- **Processing Power**: Training complex models and running real-time trading algorithms require significant computational power.
- **Latency**: Ensure low-latency data processing and decision-making, especially for high-frequency trading strategies.

4: Risk Management:

- **Drawdown**: Managing drawdown and protecting the capital during adverse market conditions is a significant challenge.
- **Leverage**: Using leverage can amplify returns but also increases risk. Proper risk management techniques must be in place.

5: Regulatory Compliance:

- **Compliance**: Ensure that the trading strategies comply with relevant financial regulations and avoid market manipulation practices.
- **Data Privacy**: If using external data sources, ensure compliance with data privacy regulations.

6: Integration and Deployment:

- **System Integration**: Integrating the trading system with broker APIs and ensuring seamless execution can be complex.
- **Scalability**: Ensure that the system can handle increasing data loads and trading volumes as the strategy scales.

7: Monitoring and Maintenance:

- **Continuous Monitoring**: Implement systems to continuously monitor the performance and health of the trading algorithms.
- **Model Updates**: Regularly update and retrain models with new data to adapt to changing market conditions.

Recommendations for Future Work

1: Improve Model Accuracy:

- **Feature Engineering**: Experiment with additional features like volume, sentiment analysis from social media, or other technical indicators.

2: Advanced Trading Strategies:

- **Deep Learning Models**: Implement deep learning models such as LSTMs or CNNs for better capturing the sequential nature of time series data.
- **Reinforcement Learning**: Explore reinforcement learning approaches for dynamic and adaptive trading strategies.

3: Risk Management:

- **Stop Loss and Take Profit**: Implement stop-loss and take-profit mechanisms to manage risk better.
- **Position Sizing**: Develop more advanced position sizing techniques based on volatility or other risk measures.

4: Backtesting Improvements:

- **Walk-forward Analysis**: Implement walk-forward analysis to ensure that the model parameters are updated regularly with new data.
- **Cross-validation**: Use time series cross-validation techniques to ensure robust performance evaluation.

5: Performance Monitoring:

- **Dashboard**: Create a dashboard to monitor the performance of the trading strategy in real-time.
- **Alerts**: Implement alert systems to notify about important events like large drawdowns or unexpected behavior.

By understanding the strengths and weaknesses of each interval strategy, we can make more informed decisions and further refine our approach to achieve better performance in real-world trading scenarios.

Conclusions

In conclusion, this project successfully developed and evaluated algorithmic trading strategies for the BTC/USDT pair by integrating technical indicators and machine learning techniques. Through comprehensive data preprocessing, we ensured the integrity and readiness of our datasets, which were crucial for the subsequent analysis. The implemented strategies, including SMA50, SMA200, MACD, Fibonacci retracement, and Stochastic Oscillator, combined with Cooperative Multi-Agent Reinforcement Learning (CMARL), provided a robust framework for adaptive decision-making. Our backtesting results highlighted the varying performance across different time intervals, with the 4-hour and 1-day strategies showing particularly high returns but also facing significant drawdowns. Real-time trading simulations further validated these strategies under live market conditions.

So future work could enhance risk management to mitigate drawdowns, optimize machine learning models for better predictive accuracy, and extend the analysis to other cryptocurrency pairs. Additionally, developing a fully automated trading system for real-world deployment would be a significant step towards practical application, ensuring our strategies' effectiveness and robustness in diverse market environments. This project lays a strong foundation for advancing algorithmic trading in the dynamic cryptocurrency market.

References

ForexBoat Team, Algorithmic Trading In Forex: Create Your First Forex Robot!, Retrieved March 2023, From Learn Algorithmic Trading In Forex: Make a Forex Robot | Udemy

JASON FERNANDO, Moving Average (MA): Purpose, Uses, Formula, and Examples, Retrieved May 2024, From Moving Average (MA): Purpose, Uses, Formula, and Examples (investopedia.com)

BRIAN DOLAN,The moving average convergence/divergence indicator helps investors identify price trends, Retrieved March 2024, From What Is MACD? (investopedia.com)

ADAM HAYES, Technical Analysis: What It Is and How To Use It in Investing, Retrieved July 2024, From Technical Analysis: What It Is and How To Use It in Investing (investopedia.com)

JAMES CHEN, What is EMA? How to Use Exponential Moving Average With Formula, Retrieved April 2024, From What is EMA? How to Use Exponential Moving Average With Formula (investopedia.com)

Unchaine d, What Are Crypto Trading Bots and How Do They Work?, Retrieved October 2023, From What Are Crypto Trading Bots and How Do They Work? (coindesk.com)

Binance Academy, What Are Crypto Trading Bots and How Do They Work?, Retrieved June 2023, From What Are Crypto Trading Bots and How Do They Work? | Binance Academy

CORY MITCHELL, What Are Fibonacci Retracement Levels, and What Do They Tell You? , Retrieved April 2023, From What Are Fibonacci Retracement Levels, and What Do They Tell You? (investopedia.com)