```python
# data_feed\data_feed.py

import ccxt
import pandas as pd
import backtrader as bt
from datetime import datetime
import pytz

class BinanceFuturesData(bt.feeds.PandasData):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    @classmethod
    def fetch_data(cls, symbol, startdate, enddate,
binance_timeframe):
        exchange = ccxt.binanceusdm({
            'rateLimit': 1200,
            'enableRateLimit': True,
        })
        exchange.load_markets()

        if enddate is None:
            enddate = datetime.now()

        timeframe_minutes = {
                '1m': 1,
                '3m': 3,
                '5m': 5,
                '15m': 15,
                '30m': 30,
                '1h': 60,
                '2h': 120,
                '4h': 240,
                '1d': 1440,
            }

        # Convert startdate and enddate to timezone-aware datetime
objects, if they aren't already
        if not pd.to_datetime(startdate).tzinfo:
            startdate = pd.to_datetime(startdate).tz_localize('UTC')
        else:
            startdate = pd.to_datetime(startdate)

        if not pd.to_datetime(enddate).tzinfo:
            enddate = pd.to_datetime(enddate).tz_localize('UTC')
        else:
            enddate = pd.to_datetime(enddate)

        timeframe_str = binance_timeframe
```

```
        all_data = []

        start_date = startdate  # Use the timezone-aware startdate

        while True:
            since = int(start_date.timestamp() * 1000)
            ohlcv = exchange.fetch_ohlcv(
                symbol,
                timeframe=timeframe_str,
                since=since,
                limit=None,
            )

            if len(ohlcv) == 0:
                break

            df = pd.DataFrame(ohlcv, columns=['datetime', 'open',
'high', 'low', 'close', 'volume'])
            df['datetime'] = pd.to_datetime(df['datetime'], unit='ms',
utc=True)
            df.set_index('datetime', inplace=True)
            all_data.append(df)

            start_date = df.index[-1] +
pd.Timedelta(minutes=timeframe_minutes[timeframe_str])

            if start_date > enddate:  # Use the timezone-aware enddate
                break

        df = pd.concat(all_data).sort_index()
        df = df.loc[startdate:enddate]  # Use the timezone-aware
startdate and enddate for slicing
        df = df[~df.index.duplicated(keep='first')]  # Drop potential
duplicates
        return df

# indicators\heikin_ashi.py

import backtrader as bt
import talib


# Heikin Ashi Indicator
class HeikinAshi(bt.Indicator):
    lines = ('ha_open', 'ha_high', 'ha_low', 'ha_close')

    def __init__(self):
        if len(self.data) < 2:
            raise ValueError("Not enough data to compute Heikin Ashi
```

```
indicators")

        self.lines.ha_close = (self.data.close + self.data.open +
self.data.high + self.data.low) / 4
        self.lines.ha_open = (self.data.open(-1) + self.data.close(-
1)) / 2
        self.lines.ha_high = bt.Max(self.data.high,
self.lines.ha_open, self.lines.ha_close)
        self.lines.ha_low = bt.Min(self.data.low, self.lines.ha_open,
self.lines.ha_close)

# indicators\heikin_patterns.py

import backtrader as bt
from matplotlib.pyplot import plot
from indicators.heikin_ashi import HeikinAshi

class HeikinPatterns(bt.Indicator):
    lines = ('myline',)

    def __init__(self):

        self.ha = bt.indicators.HeikinAshi(self.data)
        self.ha_close = self.ha.lines.ha_close
        self.ha_open = self.ha.lines.ha_open
        self.ha_high = self.ha.lines.ha_high
        self.ha_low = self.ha.lines.ha_low
        self.lines.myline = self.ha_close - self.ha_open

    def next(self):

        self.ha_green = self.ha_close[0] > self.ha_open[0]
        self.ha_red = self.ha_close[0] < self.ha_open[0]

    def is_hammer(self, index=0):
        condition1 = self.ha_close[0] > self.ha_open[0]
        condition2 = (self.ha_close[0] - self.ha_low[0] > 2 *
(self.ha_open[0] - self.ha_close[0]))
        condition3 = (self.ha_high[0] - self.ha_close[0]) <
(self.ha_close[0] - self.ha_open[0])
        is_hammer = condition1 and condition2 and condition3
        return is_hammer

    def is_falling_star(self, index=0):
        condition1 = self.ha_close[0] < self.ha_open[0]
        condition2 = (self.ha_open[0] - self.ha_low[0] > 2 *
(self.ha_close[0] - self.ha_open[0]))
        condition3 = (self.ha_close[0] - self.ha_low[0]) <
(self.ha_high[0] - self.ha_close[0])
        is_falling_star = condition1 and condition2 and condition3
```

```
        return is_falling_star

    def bullish_engulfing(self, index=0):
        condition1 = self.ha_close[0] > self.ha_open[-1]
        condition2 = (self.ha_open[0] < self.ha_close[-1])
        condition3 = self.ha_close[0] > (self.ha_open[0] +
(self.ha_open[-1] - self.ha_close[-1]))
        bullish_engulfing = condition1 and condition2 and condition3
        return bullish_engulfing

    def bearish_engulfing(self, index=0):
        condition1 = self.ha_close[0] < self.ha_open[-1]
        condition2 = (self.ha_open[0] > self.ha_close[-1])
        condition3 = self.ha_close[0] < (self.ha_open[0] -
(self.ha_close[-1] - self.ha_open[-1]))
        bearish_engulfing = condition1 and condition2 and condition3
        return bearish_engulfing

    def bullish_harami(self, index=0):
        condition1 = self.ha_open[-1] > self.ha_close[-1]
        condition2 = (self.ha_open[0] < self.ha_close[0])
        condition3 = self.ha_close[0] <= self.ha_open[-1]
        condition4 = self.ha_close[-1] <= self.ha_open[0]
        condition5 = (self.ha_close[0] - self.ha_open[0]) <
(self.ha_open[-1] - self.ha_close[-1])
        bullish_harami = condition1 and condition2 and condition3 and
condition4 and condition5
        return bullish_harami

    def bearish_harami(self, index=0):
        condition1 = self.ha_open[-1] < self.ha_close[-1]
        condition2 = self.ha_open[0] > self.ha_close[0]
        condition3 = self.ha_close[-1] >= self.ha_open[0]
        condition4 = self.ha_close[0] >= self.ha_open[-1]
        condition5 = (self.ha_open[0] - self.ha_close[0]) >
(self.ha_close[-1] - self.ha_open[-1])
        bearish_harami = condition1 and condition2 and condition3 and
condition4 and condition5
        return bearish_harami

    def bullish_hammer(self, index=0):
        condition1 = self.ha_close[0] > self.ha_open[0]
        condition2 = self.ha_close[0] > (self.ha_high[0] +
self.ha_low[0]) / 2
        condition3 = (self.ha_high[0] - self.ha_low[0]) > 2 *
(self.ha_open[0] - self.ha_close[0])
        condition4 = (self.ha_close[0] - self.ha_open[0]) < 0.1 *
(self.ha_high[0] - self.ha_low[0])
        bullish_hammer = condition1 and condition2 and condition3 and
condition4
```

```
        return bullish_hammer

    def bearish_hanging_man(self, index=0):
        condition1 = self.ha_close[0] < self.ha_open[0]
        condition2 = self.ha_close[0] < (self.ha_high[0] +
self.ha_low[0]) / 2
        condition3 = (self.ha_high[0] - self.ha_low[0]) >= 2 *
(self.ha_open[0] - self.ha_close[0])
        condition4 = (self.ha_open[0] - self.ha_close[0]) <= 0.1 *
(self.ha_high[0] - self.ha_low[0])
        bearish_hanging_man = condition1 and condition2 and condition3
and condition4
        return bearish_hanging_man

    def inside_bar(self, index=0):
        condition1 = self.ha_high[0] < self.ha_high[-1]
        condition2 = self.ha_low[0] > self.ha_low[-1]
        inside_bar = condition1 and condition2
        return inside_bar

    def doji(self, index=0):
        doji = abs(self.ha_close[0] - self.ha_open[0]) <= 0.1 *
(self.ha_high[0] - self.ha_low[0])
        return doji

    def morning_star(self, index=0):
        condition1 = self.ha_close[-2] < self.ha_open[-2]
        condition2 = self.ha_close[-1] < self.ha_open[-1]
        condition3 = self.ha_close[0] > self.ha_open[0]
        condition4 = self.ha_close[0] > self.ha_close[-2]
        condition5 = self.ha_open[0] < self.ha_open[-2]
        morning_star = condition1 and condition2 and condition3 and
condition4 and condition5
        return morning_star

    def evening_star(self, index=0):
        condition1 = self.ha_close[-2] > self.ha_open[-2]
        condition2 = self.ha_close[-1] > self.ha_open[-1]
        condition3 = self.ha_close[0] < self.ha_open[0]
        condition4 = self.ha_close[0] < self.ha_close[-2]
        condition5 = self.ha_open[0] > self.ha_open[-2]
        evening_star = condition1 and condition2 and condition3 and
condition4 and condition5
        return evening_star

    def bullish_pattern(self):
        bullish_pattern = self.bullish_engulfing() or
self.bullish_hammer() or self.is_hammer() or self.bullish_harami() or
self.morning_star()
        return bullish_pattern
```

```
    def bearish_pattern(self):
        bearish_pattern = self.bearish_engulfing() or
self.bearish_hanging_man() or self.evening_star() or
self.bearish_harami()
        return bearish_pattern

    def pattern_buy1(self):
        condition1 = self.morning_star() or self.bullish_harami()
        condition2 = self.bullish_engulfing() or self.bullish_hammer()
or self.is_hammer()
        condition3 = self.ha_green and (self.ha_close[-1] <
self.ha_open[-1]) and (self.ha_close[-2] < self.ha_open[-2])
        condition4 = self.doji() or self.doji(-1)
        pattern_buy1 = condition1 and condition2 and condition3 and
not condition4
        return pattern_buy1

    def pattern_buy2(self):
        condition1 = (self.ha_green and (self.bullish_hammer() or
self.bullish_engulfing() or self.bullish_harami() and
self.morning_star()))
        condition2 = (self.inside_bar() or self.doji() or
self.inside_bar(-1) or self.doji(-1))
        pattern_buy2 = condition1 and not condition2
        return pattern_buy2

    def pattern_buy3(self):
        condition1 = self.ha_green and (self.ha_close[-1] <
self.ha_open[-1]) and (self.ha_close[-2] < self.ha_open[-2])
        condition2 = (self.ha_open[0] - self.ha_low[0]) <
(self.ha_high[0] - self.ha_close[0])
        condition3 = self.ha_high[0] > self.ha_high[-1]
        condition4 = self.doji() or self.inside_bar()
        pattern_buy3 = condition1 and condition2 and condition3 and
not condition4
        return pattern_buy3

    def pattern_sell1(self):
        condition1 = self.evening_star(-1) or self.bearish_harami(-1)
or self.is_falling_star(-1)
        condition2 = self.bearish_engulfing()
        condition3 = self.doji() or self.inside_bar(-1)
        pattern_sell1 = condition1 and condition2 and not condition3
        return pattern_sell1

    def pattern_sell2(self):
        condition1 = self.ha_red
        condition2 = self.bearish_hanging_man() or
self.bearish_engulfing() or self.bearish_harami() and
```

```
self.evening_star()
        condition3 = (self.inside_bar() or self.inside_bar(-1) or
self.doji() or self.doji(-1))
        pattern_sell2 = condition1 and condition2 and not condition3
        return pattern_sell2

    def pattern_sell3(self):
        condition1 = (self.ha_high[0] - self.ha_open[0]) <
(self.ha_close[0] - self.ha_low[0])
        condition2 = self.ha_high[0] <= self.ha_open[0]
        condition3 = self.ha_red
        pattern_sell3 = condition1 and condition2 and condition3
        return pattern_sell3

    def pattern_stop_buy1(self):
        condition1 = self.ha_red
        condition2 = self.bearish_engulfing() or self.bearish_harami()
or self.evening_star()
        pattern_stop_buy1 = condition1 and condition2
        return pattern_stop_buy1

    def pattern_stop_buy2(self):
        condition1 = self.ha_red
        condition2 = (self.ha_high[0] < self.ha_high[-1]) or
(self.ha_low[0] > self.ha_low[-1]) or self.doji()
        condition3 = self.evening_star()
        pattern_stop_buy2 = condition1 and condition2 and condition3
        return pattern_stop_buy2

    def pattern_stop_buy3(self):
        pattern_stop_buy3 = self.bearish_hanging_man()
        return pattern_stop_buy3

    def pattern_stop_sell1(self):
        condition1 = self.ha_green
        condition2 = self.bullish_engulfing() or self.bullish_harami()
or self.bullish_hammer()
        pattern_stop_sell1 = condition1 and condition2
        return pattern_stop_sell1

    def pattern_stop_sell2(self):
        condition1 = self.ha_green
        condition2 = (self.ha_high[0] > self.ha_high[-1]) or
(self.ha_low[0] < self.ha_low[-1]) or self.doji()
        condition3 = self.morning_star()
        pattern_stop_sell2 = condition1 and condition2 and condition3
        return pattern_stop_sell2

    def pattern_buy_signal(self):
        condition1 = self.pattern_buy1() or self.pattern_buy2() or
```

```
self.pattern_buy3()
        condition2 = self.bearish_pattern()
        pattern_buy_signal = condition1 and not condition2
        return pattern_buy_signal

    def pattern_sell_signal(self):
        condition1 = self.pattern_sell1() or self.pattern_sell2() or
self.pattern_sell3()
        condition2 = self.bullish_pattern()
        pattern_sell_signal = condition1 and not condition2
        return pattern_sell_signal

    def pattern_stopBuy_signal(self):
        condition1 = self.pattern_stop_buy1() or
self.pattern_stop_buy2() or self.pattern_stop_buy3()
        condition2 = self.bullish_pattern()
        pattern_stopBuy_signal = condition1 and not condition2
        return pattern_stopBuy_signal

    def pattern_stopSell_signal(self):
        condition1 = self.pattern_stop_sell1() or
self.pattern_stop_sell2()
        condition2 = self.bearish_pattern()
        pattern_stopSell_signal = condition1 and not condition2
        return pattern_stopSell_signal


# strategy.py

import backtrader as bt
from matplotlib.pyplot import plot
from indicators.heikin_ashi import HeikinAshi
from indicators.heikin_patterns import HeikinPatterns
import talib as ta
import datetime as dt


class HeikinAshiStrategy(bt.Strategy):
    params = {
        'fast_ema': 5,
        'slow_ema': 27,
        'hma_length': 23,
        'atr_period': 19,
        'atr_threshold': 26,
        'dmi_length': 13,
        'dmi_smooth': 48,
        'dmi_threshold': 43,
        'cmo_period': 2,
        'leverage': 1,
        'timeframe': ''
```

```
    }

    def log(self, txt, dt=None):
        ''' Logging function fot this strategy'''
        dt = dt or self.data.datetime[0]
        if isinstance(dt, float):
            dt = bt.num2date(dt)
        print('%s, %s' % (dt.isoformat(), txt))


    def __init__(self):

        self.values = []

        self.current_position_size = 0
        self.long_position = False
        self.long_entry_price = 0
        self.short_entry_price = 0
        self.entry_price = 0
        self.order = None

        self.ha = bt.indicators.HeikinAshi(self.data)  #
HeikinAshi(self.data) #
        self.patterns = HeikinPatterns(self.data, plot=False)
        self.fma =
bt.indicators.ExponentialMovingAverage(self.ha.lines.ha_close,
period=self.params.fast_ema)
        self.sma =
bt.indicators.ExponentialMovingAverage(self.ha.lines.ha_close,
period=self.params.slow_ema)
        self.hma =
bt.indicators.HullMovingAverage(self.ha.lines.ha_close,
period=self.params.hma_length)
        self.atr = bt.indicators.AverageTrueRange(self.data,
period=self.params.atr_period, plot=False) * 100
        self.dmi =
bt.indicators.AverageDirectionalMovementIndex(self.data,
period=self.params.dmi_length, plot=False)
        self.adx = self.dmi # bt.indicators.SMA(self.dmi,
period=self.params.dmi_smooth)


        ha_close_values =
self.ha.lines.ha_close.get(size=self.params.fast_ema)
        print(f"HA close values: {ha_close_values}, type:
{type(ha_close_values)}")

        # Check if all values are numeric
        all_numeric = all(isinstance(val, (int, float)) for val in
```

```
ha_close_values)
        print(f"All HA close values are numeric: {all_numeric}")

        # Check if any values are None
        any_none = any(val is None for val in ha_close_values)
        print(f"Any HA close values are None: {any_none}")

        self.ha_green = (self.ha.lines.ha_close >
self.ha.lines.ha_open)
        self.ha_red = (self.ha.lines.ha_close < self.ha.lines.ha_open)
        self.ema2_cross = bt.ind.CrossOver(self.fma, self.sma,
plot=False)
        self.ema_cross = bt.ind.CrossOver(self.ha.lines.ha_close,
self.sma, plot=False)
        self.ema_buy = bt.Or(self.ema2_cross > 0.0, self.ema_cross >
0.0)
        self.ema_sell = bt.Or(self.ema2_cross < 0.0, self.ema_cross <
0.0)
        self.hma_cross = bt.ind.CrossOver(self.ha.lines.ha_close,
self.hma, plot=False)
        self.hma_buy = bt.And(self.hma_cross > 0.0, self.data.close >
self.hma)
        self.hma_sell = bt.And(self.hma_cross < 0.0, self.data.close <
self.hma)
        self.hma_stop_buy = (self.hma_cross < 0.0)
        self.hma_stop_sell = (self.hma_cross > 0.0)
        self.cmo = bt.talib.CMO(self.data.close,
period=self.params.cmo_period, plot=False)


        self.cmo_buy = self.cmo > 0
        self.cmo_sell = self.cmo < 0
        self.high_volatility = self.atr > (self.params.atr_threshold /
100)
        self.adx_signal = bt.And(self.dmi.DIplus > self.dmi.DIminus,
self.adx > self.params.dmi_threshold)

        self.order = None  # to keep track of pending orders
        self.in_position = False
        self.entry_price = None  # to keep track of entry price

    def check_buy_condition(self):
        condition1 = self.patterns.pattern_buy_signal()
        condition2 = self.ha_green[0] and (self.ema_buy[0] or
self.hma_buy[0])
        condition3 = self.cmo_buy[0] and self.adx_signal[0]
        condition4 = self.high_volatility[0]
        condition = (condition1 or condition2) and condition3 and
condition4
        return condition
```

```
    def check_sell_condition(self):
        condition1 = self.patterns.pattern_sell_signal()
        condition2 = self.ha_red[0] and (self.ema_sell[0] or
self.hma_sell[0])
        condition3 = self.cmo_sell[0] and self.adx_signal[0]
        condition4 = self.high_volatility[0]
        condition = (condition1 or condition2) and condition3 and
condition4
        return condition

    def check_stop_buy_condition(self):
        condition1 = (self.ema_sell[0] or self.hma_stop_buy[0] or
self.patterns.pattern_stopBuy_signal())
        condition2 = self.high_volatility[0] and self.adx_signal[0]
        condition = condition1 and condition2
        return condition

    def check_stop_sell_condition(self):
        condition1 = (self.ema_buy[0] or self.hma_stop_sell[0] or
self.patterns.pattern_stopSell_signal())
        condition2 = self.high_volatility[0] and self.adx_signal[0]
        condition = condition1 and condition2
        return condition


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

            # Order has been submitted/accepted - no action required
            return

            # Check if an order has been completed
        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(f"BUY EXECUTED, Price:
{order.executed.price:.4f}, Cost: {order.executed.value:.2f}, Comm
{order.executed.comm:.2f}")
                self.current_position_size += order.executed.size
                self.long_position = True
                self.long_entry_price = order.executed.price
            else:
                self.log(f"SELL EXECUTED, Price:
{order.executed.price:.4f}, Cost: {order.executed.value:.2f}, Comm
{order.executed.comm:.2f}")
                self.current_position_size -= order.executed.size
                self.long_position = False
                self.short_entry_price = order.executed.price
```

```
            if not self.position:  # if position is closed
                closed_size = self.current_position_size
                self.current_position_size = 0
                self.in_position = False
                if self.long_position:
                    profit_loss = (self.short_entry_price -
order.executed.price) * closed_size
                    self.log(f"Closed SHORT position, Price:
{order.executed.price:.4f}, Profit/Loss = {profit_loss:.2f} $")
                    print("-" * 50)
                else:  # short position
                    profit_loss = (order.executed.price -
self.long_entry_price) * closed_size
                    self.log(f"Closed LONG position, Price:
{order.executed.price:.4f}, Profit/Loss = {profit_loss:.2f} $")
                    print("-" * 50)

        elif order.status in [order.Canceled]:
            self.log('Order Canceled')

        elif order.status in [order.Margin]:
            self.log('Order Margin')

        elif order.status in [order.Rejected]:
            self.log('Order Rejected')

        # Reset
        self.order = None


    def next(self):

        # Protect against division by zero
        price = self.data[0]
        if price == 0:
            print("Price is zero, skipping this bar")
            return

        if not self.in_position and (self.check_buy_condition() or
self.check_sell_condition()):
            self.in_position = True
            price = self.data[0]
            leverage =
self.broker.getcommissioninfo(self.data).get_leverage()
            free_money = self.broker.getcash() * 0.5
            # size =  leverage * math.floor(free_money /
self.broker.getcommissioninfo(self.data).get_margin(price))
            size =
self.broker.getcommissioninfo(self.data).getsize(price=price,
```

```
cash=free_money)
            #   bt.broker.print_function
            if self.check_buy_condition():
                self.order = self.buy(size=size,
exectype=bt.Order.Market)
                print("-" * 50)
                print(f"{bt.num2date(self.data.datetime[0])}\t ----
LONG ---- size = {size:.2f} at price = {price:.4f}")

            else:
                self.order = self.sell(size=size,
exectype=bt.Order.Market)
                print("-" * 50)
                print(f"{bt.num2date(self.data.datetime[0])}\t ----
SHORT ---- size = {size:.2f} at price = {price:.4f}")


        elif self.in_position and (
                self.check_sell_condition() or
self.check_buy_condition() or self.check_stop_buy_condition() or
self.check_stop_sell_condition()):
            self.order = self.close()
            self.in_position = False

            print("-" * 50)
            self.log('DrawDown: %.2f' % self.stats.drawdown.drawdown[-
1])
            self.log('MaxDrawDown: %.2f' %
self.stats.drawdown.maxdrawdown[-1])


        self.values.append(self.broker.getvalue())

        # print("-" * 50)
        # print('{} / {} [{}] - Open: {}, High: {}, Low: {}, Close:
{}, Volume: {}'.format(
        #     bt.num2date(self.data.datetime[0]),
        #     self.data._name,
        #     self.data._timeframe,  # ticker timeframe
        #     self.data.open[0],
        #     self.data.high[0],
        #     self.data.low[0],
        #     self.data.close[0],
        #     self.data.volume[0],
        # ))

# trade_list_analyzer.py

# Trade list similar to Amibroker output
```

```
import backtrader as bt

class trade_list(bt.Analyzer):

    def get_analysis(self):

        return self.trades


    def __init__(self):

        self.trades = []
        self.cumprofit = 0.0


    def notify_trade(self, trade):

        if trade.isclosed:

            brokervalue = self.strategy.broker.getvalue()

            dir = 'short'
            if trade.history[0].event.size > 0: dir = 'long'

            pricein = trade.history[len(trade.history)-1].status.price
            priceout = trade.history[len(trade.history)-1].event.price
            datein = bt.num2date(trade.history[0].status.dt)
            dateout = bt.num2date(trade.history[len(trade.history)-
1].status.dt)
            if trade.data._timeframe >= bt.TimeFrame.Days:
                datein = datein.date()
                dateout = dateout.date()

            pcntchange = 100 * priceout / pricein - 100
            pnl = trade.history[len(trade.history)-1].status.pnlcomm
            pnlpcnt = 100 * pnl / brokervalue
            barlen = trade.history[len(trade.history)-1].status.barlen
            pbar = pnl / barlen
            self.cumprofit += pnl

            size = value = 0.0
            for record in trade.history:
                if abs(size) < abs(record.status.size):
                    size = record.status.size
                    value = record.status.value

            highest_in_trade = max(trade.data.high.get(ago=0,
size=barlen+1))
            lowest_in_trade = min(trade.data.low.get(ago=0,
```

```
size=barlen+1))
            hp = 100 * (highest_in_trade - pricein) / pricein
            lp = 100 * (lowest_in_trade - pricein) / pricein
            if dir == 'long':
                mfe = hp
                mae = lp
            if dir == 'short':
                mfe = -lp
                mae = -hp

            self.trades.append({'ref': trade.ref, 'ticker':
trade.data._name, 'dir': dir,
                    'datein': datein, 'pricein': pricein, 'dateout':
dateout, 'priceout': priceout,
                    'chng%': round(pcntchange, 2), 'pnl': pnl, 'pnl%':
round(pnlpcnt, 2),
                    'size': size, 'value': value, 'cumpnl':
self.cumprofit,
                    'nbars': barlen, 'pnl/bar': round(pbar, 2),
                    'mfe%': round(mfe, 2), 'mae%': round(mae, 2)})

# optimizer.py

import datetime
import multiprocessing
from deap import base, creator, tools, algorithms
import random
import numpy as np
from data_feed.data_feed import BinanceFuturesData
from strategy import HeikinAshiStrategy
import backtrader as bt
import pickle


def run_backtest(fast_ema, slow_ema, hma_length, atr_period,
atr_threshold, dmi_length, dmi_smooth, dmi_threshold,
                 cmo_period, fetched_data, start_date, end_date,
timeframe, compression, bt_timeframe):
    cerebro = bt.Cerebro(quicknotify=True)

    data = BinanceFuturesData(
        dataname=fetched_data,
        fromdate=start_date,
        todate=end_date,
        timeframe=bt_timeframe,
        compression=compression,
    )
```

```
    # Add the data to the backtrader instance
    cerebro.adddata(data)

    # Set the starting cash and commission
    starting_cash = 100
    cerebro.broker.setcash(starting_cash)
    cerebro.broker.setcommission(
        automargin=True,
        leverage=10.0,
        commission=0.0004,
        commtype=bt.CommInfoBase.COMM_PERC,
        stocklike=True,
    )

    cerebro.addobserver(bt.observers.DrawDown, plot=False)

    cerebro.addanalyzer(bt.analyzers.TradeAnalyzer,
_name='trade_analyzer')
    cerebro.addanalyzer(bt.analyzers.SharpeRatio,
_name='sharpe_ratio')
    cerebro.addanalyzer(bt.analyzers.DrawDown, _name='drawdown')
    cerebro.addanalyzer(bt.analyzers.Returns, _name='returns')

    cerebro.addstrategy(
        HeikinAshiStrategy,
        fast_ema=fast_ema,
        slow_ema=slow_ema,
        hma_length=hma_length,
        atr_period=atr_period,
        atr_threshold=atr_threshold,
        dmi_length=dmi_length,
        dmi_smooth=dmi_smooth,
        dmi_threshold=dmi_threshold,
        cmo_period=cmo_period,

    )


    results = cerebro.run()
    final_value = cerebro.broker.getvalue()

    print(final_value, results)

    return final_value, results  # return both final_value and results


# Set the ranges for the parameters to optimize
fast_ema_range = range(1, 20)
slow_ema_range = range(5, 35)
```

```
hma_length_range = range(2, 30)
atr_period_range = range(1, 25)
atr_threshold_range = range(10, 100)
dmi_length_range = range(2, 20)
dmi_smooth_range = range(10, 55)
dmi_threshold_range = range(10, 85)
cmo_period_range = range(1, 25)


def evaluate(params, fetched_data, start_date, end_date, timeframe,
compression, bt_timeframe):
    assert len(params) == 9, "params should have exactly 9 elements"
    final_value, results = run_backtest(*params, fetched_data,
start_date, end_date, timeframe, compression,
                                        bt_timeframe)  # capture both
final_value and results
    drawdown = results[0].analyzers.drawdown.get_analysis()['max']
['drawdown']  # get maximum drawdown
    print(final_value, drawdown)
    return final_value, drawdown  # return both profit and drawdown


# Genetic Algorithm
creator.create("FitnessMax", base.Fitness, weights=(1.0, -1.0))  # two
objectives: maximize profit, minimize drawdown
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()

toolbox.register("attr_fast_ema", lambda: int(random.randint(1, 20)))
toolbox.register("attr_slow_ema", lambda: int(random.randint(5, 35)))
toolbox.register("attr_hma_length", lambda: int(random.randint(2,
30)))
toolbox.register("attr_atr_period", lambda: int(random.randint(1,
25)))
toolbox.register("attr_atr_threshold", lambda:
int(round(random.uniform(10, 100))))
toolbox.register("attr_dmi_length", lambda: int(random.randint(2,
20)))
toolbox.register("attr_dmi_smooth", lambda: int(random.randint(10,
55)))
toolbox.register("attr_dmi_threshold", lambda: int(random.randint(10,
85)))
toolbox.register("attr_cmo_period", lambda: int(random.randint(1,
25)))

toolbox.register("attr_int", random.randint, 1, 100)  # generates
random integers between 1 and 100
```

```
toolbox.register("individual", tools.initCycle, creator.Individual, (
    toolbox.attr_fast_ema, toolbox.attr_slow_ema,
toolbox.attr_hma_length,
    toolbox.attr_atr_period, toolbox.attr_atr_threshold,
toolbox.attr_dmi_length,
    toolbox.attr_dmi_smooth, toolbox.attr_dmi_threshold,
toolbox.attr_cmo_period), n=1)

attr_ranges = [fast_ema_range, slow_ema_range, hma_length_range,
atr_period_range, atr_threshold_range,
               dmi_length_range, dmi_smooth_range,
dmi_threshold_range, cmo_period_range]


def custom_mutate(individual, indpb):
    for i in range(len(individual)):
        if random.random() < indpb:
            individual[i] = random.choice(attr_ranges[i])  # Choose a
random value from the range of the current attribute
    return individual,

toolbox.register("population", tools.initRepeat, list,
toolbox.individual)  # creates the population
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", custom_mutate, indpb=0.2)
toolbox.register("select", tools.selNSGA2)  # use NSGA-II selection
for multi-objective optimization
toolbox.register("evaluate", evaluate)


def main(symbol, start_date, end_date, timeframe, compression,
fetched_data, bt_timeframe):
    random.seed(42)
    np.random.seed(42)
    multiprocessing.set_start_method("spawn")

    toolbox.register("evaluate", evaluate, fetched_data=fetched_data,
start_date=start_date, end_date=end_date,
                     timeframe=timeframe, compression=compression,
bt_timeframe=bt_timeframe)

    pop = toolbox.population(n=20)
    hof = tools.HallOfFame(5)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", np.mean)
    stats.register("min", np.min)
    stats.register("max", np.max)

    try:
```

```
        pop, logbook = algorithms.eaSimple(pop, toolbox, cxpb=0.7,
mutpb=0.3, ngen=10,
                                           stats=stats,
halloffame=hof, verbose=True)

        # Save best individuals and their fitness to a file
        best_individuals = [(ind, ind.fitness.values) for ind in hof]
        with open('best_individuals.pkl', 'wb') as f:
            pickle.dump(best_individuals, f)

        print("Best parameters found by GA:", hof[0])
        return hof[0]

    except Exception as e:
        print(f"An error occurred: {e}")
        import traceback
        print(traceback.format_exc())

    if len(hof) > 0:
        best_params = hof[0]
    else:
        # handle the case when the list is empty, e.g., set a default
value or raise an error
        best_params = None

    print("Best parameters found by GA:", best_params)

    return best_params

# main.py

from concurrent import futures
from doctest import debug
import backtrader as bt
from matplotlib.pyplot import plot
from deap import base, creator, tools, algorithms
import datetime as dt
from tabulate import tabulate
from trade_list_analyzer import trade_list
import numpy as np
import multiprocessing
from concurrent.futures import ProcessPoolExecutor
import quantstats as qs
import pandas as pd
from optimizer import main
import pyfolio as pf
import os
import json
from ccxtbt import CCXTStore, CCXTFeed
import time
```

```python
from data_feed.data_feed import BinanceFuturesData
from strategy import HeikinAshiStrategy


# Settings
target_coin = 'NEAR'
base_currency = 'USDT' #'BUSD' #
symbol = target_coin + base_currency
# symbol = 'GMTUSDT' # Perpetual'
start_date = dt.datetime.strptime("2023-01-01 00:00:00", "%Y-%m-%d %H:
%M:%S")
end_date = dt.datetime.strptime("2023-05-23 00:00:00", "%Y-%m-%d %H:
%M:%S")
timeframe =  'Minutes' # 'Hours' #
compression = 30
use_optimization = True


def convert_to_binance_timeframe(compression, timeframe):

    # Determine the Backtrader timeframe
    if timeframe == 'Minutes':
        bt_timeframe = bt.TimeFrame.Minutes
    elif timeframe == 'Hours':
        if compression not in [1, 2, 3, 4, 6, 8, 12]:
            raise ValueError(
                f'Invalid hourly compression for Binance:
{compression}. Supported values are 1, 2, 3, 4, 6, 8, 12.')
        bt_timeframe = bt.TimeFrame.Minutes
        compression *= 60  # Convert hours to minutes
    elif timeframe == 'Days':
        bt_timeframe = bt.TimeFrame.Days
        compression *= 24 * 60  # Convert days to minutes
    elif timeframe == 'Weeks':
        bt_timeframe = bt.TimeFrame.Weeks
        compression *= 7 * 24 * 60  # Convert weeks to minutes
    elif timeframe == 'Months':
        bt_timeframe = bt.TimeFrame.Months
        compression *= 30 * 24 * 60  # Convert months to minutes
    else:
        raise ValueError(f'Invalid timeframe: {timeframe}')

    # Determine the Binance timeframe
    binance_timeframe = str(compression)
    if timeframe == 'Minutes':
        binance_timeframe += 'm'
    elif timeframe == 'Hours':
```

```
        binance_timeframe = str(compression // 60) + 'h'  # Binance
expects the compression in hours
    elif timeframe == 'Days':
        binance_timeframe += 'd'
    elif timeframe == 'Weeks':
        binance_timeframe += 'w'
    elif timeframe == 'Months':
        binance_timeframe += 'M'

    # Validate Binance timeframe
    valid_binance_timeframes = ['1m', '3m', '5m', '15m', '30m', '1h',
'2h', '3h', '4h', '6h', '8h', '12h', '1d', '3d',
                                '1w', '1M']
    if binance_timeframe not in valid_binance_timeframes:
        raise ValueError(f'Invalid Binance timeframe:
{binance_timeframe}')

    return bt_timeframe, compression, binance_timeframe


if __name__ == '__main__':
    # Ensure multiprocessing is supported
    multiprocessing.freeze_support()

    # Convert the specified timeframe to a Binance-compatible
timeframe
    bt_timeframe, compression, binance_timeframe =
convert_to_binance_timeframe(compression, timeframe)

    # Fetch the data for the specified symbol and time range
    fetched_data = BinanceFuturesData.fetch_data(symbol, start_date,
end_date, binance_timeframe)

    if use_optimization:
        # Use the optimizer to find the best parameters for the
strategy
        best_params = main(symbol, start_date, end_date, timeframe,
compression, fetched_data, bt_timeframe)
    else:
        # Use the default parameters from the strategy
        best_params = (
            HeikinAshiStrategy.params.fast_ema,
            HeikinAshiStrategy.params.slow_ema,
            HeikinAshiStrategy.params.hma_length,
            HeikinAshiStrategy.params.atr_period,
            HeikinAshiStrategy.params.atr_threshold,
            HeikinAshiStrategy.params.dmi_length,
            HeikinAshiStrategy.params.dmi_smooth,
            HeikinAshiStrategy.params.dmi_threshold,
```

```
            HeikinAshiStrategy.params.cmo_period,
        )  # Retrieve default values from the strategy class

    # Create a new backtrader instance
    cerebro = bt.Cerebro(quicknotify=True, tradehistory=True)

    # Pass the fetched data to the BinanceFuturesData class
    data = BinanceFuturesData(
        dataname=fetched_data,
        fromdate=start_date,
        todate=end_date,
        timeframe=bt_timeframe,
        compression=compression,
    )



    # # absolute dir the script is in
    # script_dir = os.path.dirname(__file__)
    # abs_file_path = os.path.join(script_dir, '../params.json')
    # with open('.\params.json', 'r') as f:
    #     params = json.load(f)

    # # Create a CCXTStore and Data Feed
    # config = {'apiKey': params["binance"]["apikey"],
    #         'secret': params["binance"]["secret"],
    #         'enableRateLimit': True,
    #         'nonce': lambda: str(int(time.time() * 1000)),
    #         'options': { 'defaultType': 'future' },
    #         }

    dataname =
'{target_coin}/{base_currency}'.format(target_coin=target_coin,
base_currency=base_currency)

    # data = CCXTFeed(exchange='binance',
    #                           dataname=dataname,
    #                           timeframe=bt_timeframe,
    #                           fromdate=start_date,
    #                           todate=end_date,
    #                           compression=compression,
    #                           ohlcv_limit=99999,
    #                           currency=base_currency,
    #                           retries=5,

    #                           config=config)

    cerebro.adddata(data, name=dataname)

    # Set the starting cash and commission
```

```
    starting_cash = 100
    cerebro.broker.setcash(starting_cash)
    cerebro.broker.setcommission(
        automargin=True,
        leverage=10.0,
        commission=0.0004,
        commtype=bt.CommInfoBase.COMM_PERC,
        stocklike=True,
    )

    # bt.CommInfoBase.get_leverage
    cerebro.addobserver(bt.observers.DrawDown, plot=False)

    # Add the analyzers we are interested in
    # cerebro.addanalyzer(bt.analyzers.PyFolio, _name='pyfolio')

    cerebro.addanalyzer(bt.analyzers.TradeAnalyzer,
_name='tradeanalyzer')
    cerebro.addanalyzer(bt.analyzers.SQN, _name='sqn')
    cerebro.addanalyzer(bt.analyzers.Returns, _name='returns')
    cerebro.addanalyzer(bt.analyzers.DrawDown, _name='drawdown')
    cerebro.addanalyzer(trade_list, _name='trade_list')

    cerebro.addstrategy(
        HeikinAshiStrategy,
        fast_ema=best_params[0],
        slow_ema=best_params[1],
        hma_length=best_params[2],
        atr_period=best_params[3],
        atr_threshold=best_params[4],
        dmi_length=best_params[5],
        dmi_smooth=best_params[6],
        dmi_threshold=best_params[7],
        cmo_period=best_params[8],
        timeframe={binance_timeframe},

    )

    # Run the strategy and get the instance
    strat = cerebro.run(quicknotify=True, tradehistory=True)[0]

    cerebro.plot(style='candlestick', iplot=True)
    final_value = cerebro.broker.getvalue()
    profit = final_value - starting_cash # Obtain net profit
    profit_percentage = (profit / starting_cash) * 100
    max_drawdown = strat.analyzers.drawdown.get_analysis()['max']
['drawdown']
    total_trades = strat.analyzers.tradeanalyzer.get_analysis()
['total']['total']
```

```
    sqn = strat.analyzers.sqn.get_analysis()['sqn']

    trade_list = strat.analyzers.trade_list.get_analysis()
    print (tabulate(trade_list, headers="keys", tablefmt="psql",
missingval="?"))

    print()
    print()
    print("Best parameters found by GA:", best_params)
    print()
    print()
    # Print out the statistics
    print(f"Total trades:
{strat.analyzers.tradeanalyzer.get_analysis()['total']['total']}")
    print(f"SQN: {strat.analyzers.sqn.get_analysis()['sqn']:.2f}")
    print(f"Net profit: {strat.analyzers.returns.get_analysis()
['rtot']:.2f}")
    print(f"Max drawdown: {strat.analyzers.drawdown.get_analysis()
['max']['drawdown']:.2f}")
    print()
    print()
    print("$" * 77)
    print(f"Liquidation value of the portfolio: {final_value:.2f} $")
# Liquidation value of the portfolio
    print(f"Profit %: {profit_percentage:.2f} %")  # Profit %
    print(f"Profit: {profit:.2f} $")
    print("$" * 77)


    # Create the directory if it does not exist
    os.makedirs('backtests_results', exist_ok=True)

    # Save the results in a json file
    symbol_name = symbol.replace('/', '_')  # Replace '/' with '_'
    filename =
f'backtests_results/btest_results_{symbol_name}_{binance_timeframe}.js
on'

    data = {
        'symbol': symbol,
        'timeframe': binance_timeframe,
        'start_date': start_date.isoformat(),
        'end_date': end_date.isoformat(),
        'best_params': best_params,
        'profit': round(profit, 2),
        'profit_percentage': round(profit_percentage, 2),
        'max_drawdown': round(max_drawdown, 2),
        'total_trades': total_trades,
        'sqn': round(sqn, 2),
```

```
    }

    # Check if the file exists
    if os.path.isfile(filename):
        # If the file exists, open it and load the data
        with open(filename, 'r') as f:
            existing_data = json.load(f)
        print(existing_data)


        # Check if any existing data has the same symbol, timeframe,
and date range
        for entry in existing_data:
            if (entry['symbol'] == data['symbol'] and
                entry['timeframe'] == data['timeframe'] and
                entry['start_date'] == data['start_date'] and
                entry['end_date'] == data['end_date']):

                # If it does and the new profit is larger, update this
entry
                if entry['profit'] < data['profit'] and
entry['profit_percentage'] < data['profit_percentage']:
                    entry['best_params'] = data['best_params']
                    entry['profit'] = data['profit']
                    entry['total_trades'] = data['total_trades']
                    entry['profit_percentage'] =
data['profit_percentage']
                    entry['max_drawdown'] = data['max_drawdown']


                # In this case, don't append new data
                break
        else:
            # If no matching entry was found, append the new data
            existing_data.append(data)

        # Write the updated data back to the file
        with open(filename, 'w') as f:
            json.dump(existing_data, f, indent=4)
    else:
        # If the file does not exist, create it with the new data in a
list
        with open(filename, 'w') as f:
            json.dump([data], f, indent=4)
```

```python
# live_trading.py

import backtrader as bt
from ccxtbt import CCXTStore
from strategy import HeikinAshiStrategy
import api_config
import os
import json
import time
import datetime as dt
from tabulate import tabulate
from trade_list_analyzer import trade_list


# Settings
target_coin = 'GMT'
base_currency = 'USDT'
symbol = target_coin + base_currency
timeframe = bt.TimeFrame.Minutes
compression = 30  # For live trading, you typically want to use
smaller timeframes
best_params = (
    HeikinAshiStrategy.params.fast_ema,
    HeikinAshiStrategy.params.slow_ema,
    HeikinAshiStrategy.params.hma_length,
    HeikinAshiStrategy.params.atr_period,
    HeikinAshiStrategy.params.atr_threshold,
    HeikinAshiStrategy.params.dmi_length,
    HeikinAshiStrategy.params.dmi_smooth,
    HeikinAshiStrategy.params.dmi_threshold,
    HeikinAshiStrategy.params.cmo_period,
)  # Retrieve default values from the strategy class


# Create a new backtrader instance
cerebro = bt.Cerebro(quicknotify=True)

cerebro.addobserver(bt.observers.DrawDown, plot=False)

# Add the analyzers we are interested in
# cerebro.addanalyzer(bt.analyzers.PyFolio, _name='pyfolio')

cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='tradeanalyzer')
cerebro.addanalyzer(bt.analyzers.SQN, _name='sqn')
cerebro.addanalyzer(bt.analyzers.Returns, _name='returns')
cerebro.addanalyzer(bt.analyzers.DrawDown, _name='drawdown')
cerebro.addanalyzer(trade_list, _name='trade_list')

# Add your strategy
```

```
cerebro.addstrategy(
    HeikinAshiStrategy,
    fast_ema=best_params[0],
    slow_ema=best_params[1],
    hma_length=best_params[2],
    atr_period=best_params[3],
    atr_threshold=best_params[4],
    dmi_length=best_params[5],
    dmi_smooth=best_params[6],
    dmi_threshold=best_params[7],
    cmo_period=best_params[8],
    timeframe={timeframe, compression},
)

# absolute dir the script is in
script_dir = os.path.dirname(__file__)
abs_file_path = os.path.join(script_dir, '../params.json')
with open('.\params.json', 'r') as f:
    params = json.load(f)

# Create a CCXTStore and Data Feed
config = {'apiKey': params["binance"]["apikey"],
          'secret': params["binance"]["secret"],
          'enableRateLimit': True,
          'nonce': lambda: str(int(time.time() * 1000)),
          'options': { 'defaultType': 'future' },
          }

store = CCXTStore(exchange='binance', currency=base_currency,
config=config, retries=10, debug=True) #, sandbox=True)

store.exchange.setSandboxMode(True)

broker_mapping = {
    'order_types': {
        bt.Order.Market: 'market',
        bt.Order.Limit: 'limit',
        bt.Order.Stop: 'stop-loss', #stop-loss for kraken, stop for
bitmex
        bt.Order.StopLimit: 'stop limit'
    },
    'mappings':{
        'closed_order':{
            'key': 'status',
            'value':'closed'
        },
        'canceled_order':{
            'key': 'result',
            'value':1}
    }
```

```
}

broker = store.getbroker(broker_mapping=broker_mapping)
cerebro.setbroker(broker)

cerebro.broker.setcommission(leverage=10.0)

# Set the starting cash and commission
# starting_cash = 100
# cerebro.broker.setcash(starting_cash)
# cerebro.broker.setcommission(
#     automargin=True,
#     leverage=10.0,
#     commission=0.0004,
#     commtype=bt.CommInfoBase.COMM_PERC,
#     stocklike=True,
# )

hist_start_date = dt.datetime.utcnow() - dt.timedelta(days=30*1)
dataname =
'{target_coin}/{base_currency}'.format(target_coin=target_coin,
base_currency=base_currency)
data = store.getdata(dataname=dataname, name=symbol,
from_date=hist_start_date,
                     timeframe=bt.TimeFrame.Minutes, compression=1,
ohlcv_limit=500, drop_newest=True)

cerebro.resampledata(data, dataname=dataname, timeframe=timeframe,
compression=compression)

# Add the data to the backtrader instance
# cerebro.adddata(data, name=symbol)

# Run the strategy
cerebro.run()

cerebro.plot()
```