UMASS BOSTON

CS410 Software Engineering

Trend Sensitive Trailing Stop Loss Bot

[All code relating to this project as well as access to the served web app will be openly and publicly available, free of charge.]

Introduction

Trading is fun and trading through programmatic means is even funner. We are interested in developing a completely automated trading algorithm that relies on Technical Analysis (TA) and, conditional on simulation results, a Deep Learning model, to the end of handling its own asset wallets as well as trade orders and their responses.

Problem Approach

Predicting the future price of an asset is a difficult problem generally not modeled well by neural networks. We instead are interested in estimating the trend direction, spread, and momentum associated with the price action of dozens of assets. We further seek to automate the process of 'locking-in' profit, thereby protecting against loss from volatility.

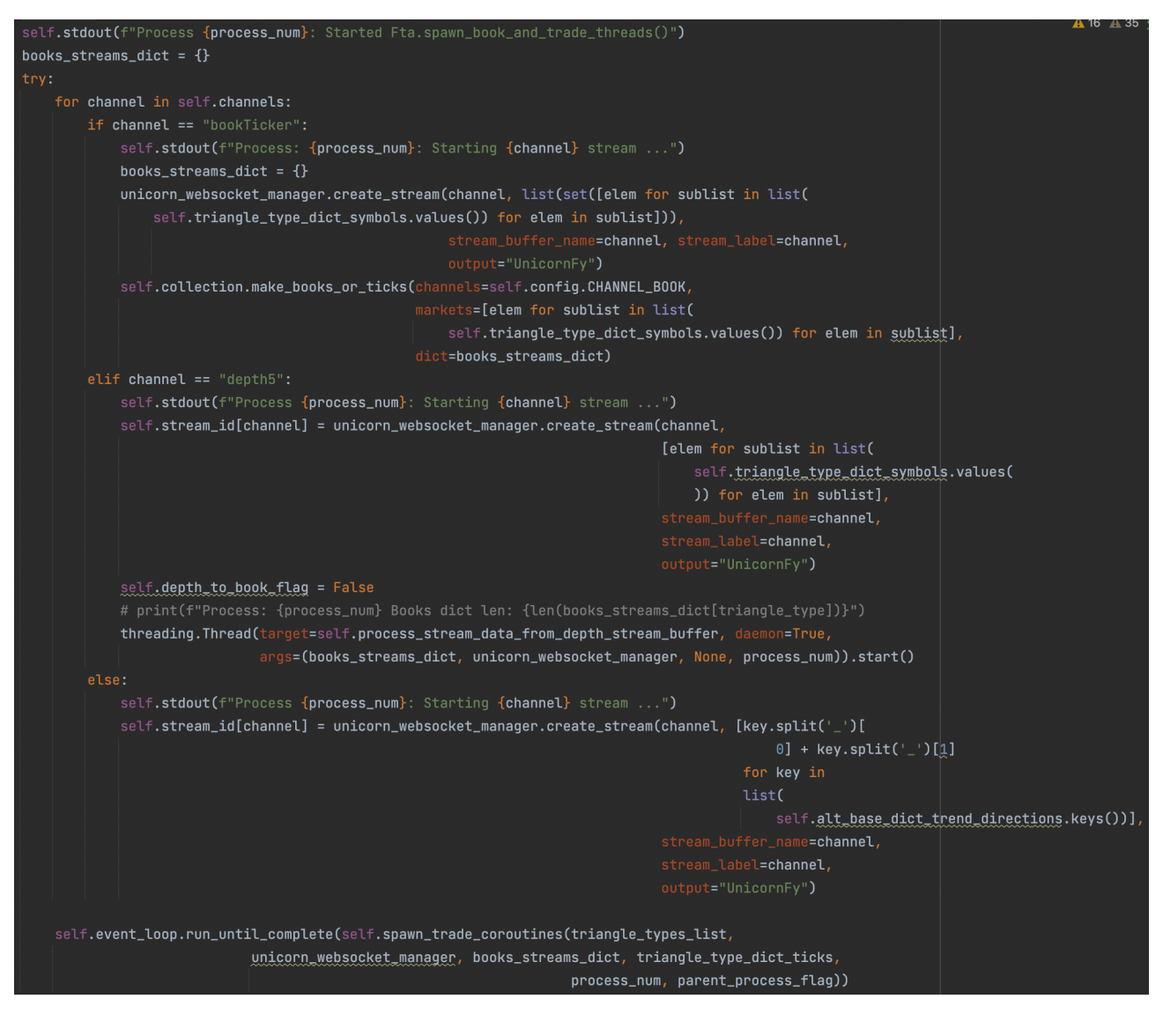
In this project, you will explore the effects of combining TA indicators with deep learning model building to produce a successful trading product. You will implement a simulation class to both generate training data and gauge model performance, and you will develop an awesome Python web app to showcase your results.

Objectives

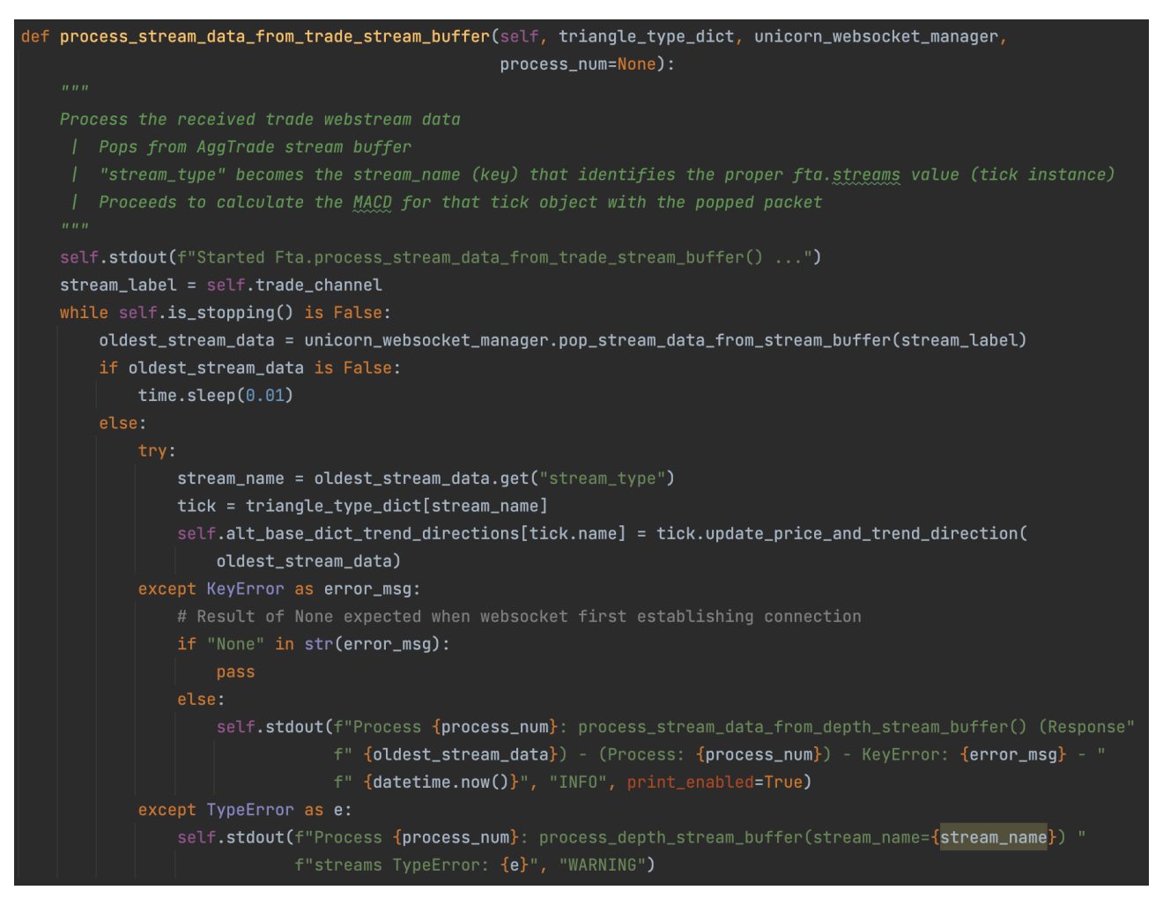
1. **Python Websocket Programming and Trading Indicator implementation.**

   Rely on LUCIT Systems and Development's UNICORN Binance Suite for high level persistent TCP connections to trade data streams.

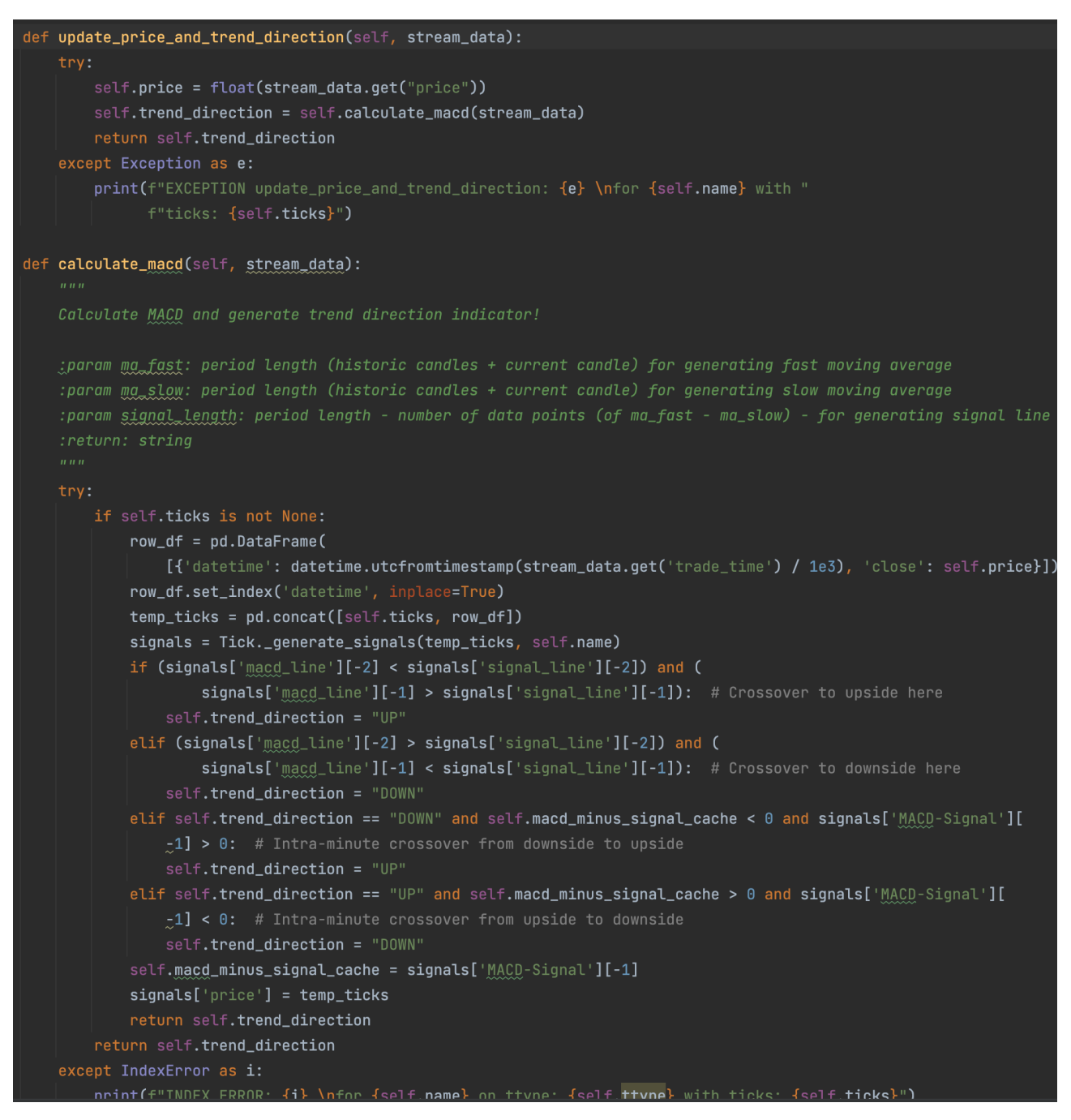
   Investigate and implement trading indicators with fast updates. The Moving Average Convergence Divergence (MACD) oscillating indicator has already been implemented for you.



```python
self.stdout(f"Process {process_num}: Started Fta.spawn_book_and_trade_threads()")
books_streams_dict = {}
try:
    for channel in self.channels:
        if channel == "bookTicker":
            self.stdout(f"Process: {process_num}: Starting {channel} stream ...")
            books_streams_dict = {}
            unicorn_websocket_manager.create_stream(channel, list(set([elem for sublist in list(
                self.triangle_type_dict_symbols.values()) for elem in sublist])),
                                                    stream_buffer_name=channel, stream_label=channel,
                                                    output="UnicornFy")
            self.collection.make_books_or_ticks(channels=self.config.CHANNEL_BOOK,
                                                markets=[elem for sublist in list(
                                                    self.triangle_type_dict_symbols.values()) for elem in sublist],
                                                dict=books_streams_dict)
        elif channel == "depth5":
            self.stdout(f"Process {process_num}: Starting {channel} stream ...")
            self.stream_id[channel] = unicorn_websocket_manager.create_stream(channel,
                                                                              [elem for sublist in list(
                                                                                  self.triangle_type_dict_symbols.values(
                                                                                  )) for elem in sublist],
                                                                              stream_buffer_name=channel,
                                                                              stream_label=channel,
                                                                              output="UnicornFy")

            self.depth_to_book_flag = False
            # print(f"Process: {process_num} Books dict len: {len(books_streams_dict[triangle_type])}")
            threading.Thread(target=self.process_stream_data_from_depth_stream_buffer, daemon=True,
                             args=(books_streams_dict, unicorn_websocket_manager, None, process_num)).start()
        else:
            self.stdout(f"Process {process_num}: Starting {channel} stream ...")
            self.stream_id[channel] = unicorn_websocket_manager.create_stream(channel, [key.split('_')[
                                                                                  0] + key.split('_')[1]
                                                                                  for key in
                                                                                  list(
                                                                                      self.alt_base_dict_trend_directions.keys())],
                                                                              stream_buffer_name=channel,
                                                                              stream_label=channel,
                                                                              output="UnicornFy")

    self.event_loop.run_until_complete(self.spawn_trade_coroutines(triangle_types_list,
                                       unicorn_websocket_manager, books_streams_dict, triangle_type_dict_ticks,
                                                                   process_num, parent_process_flag))
```

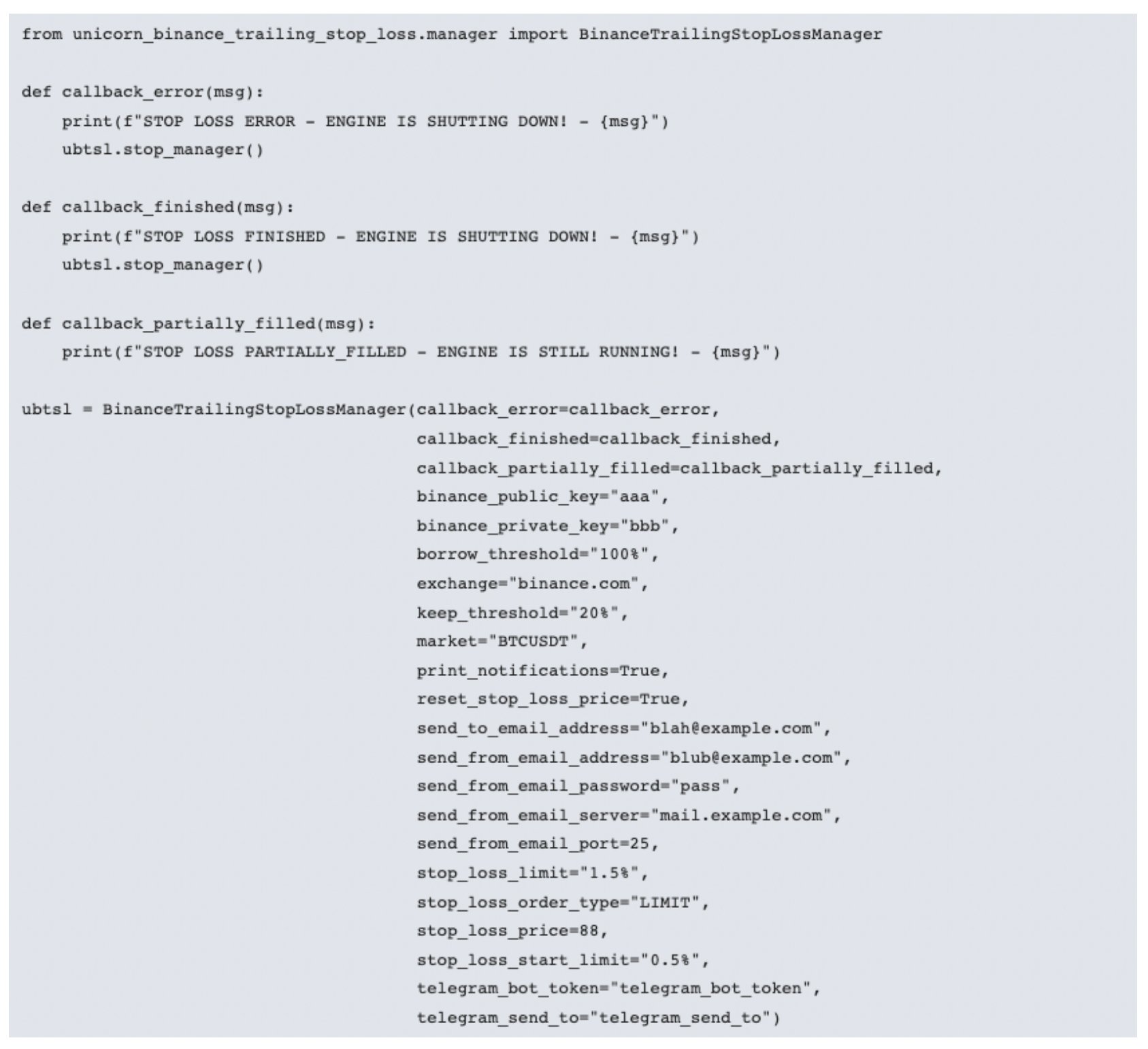


```python
def process_stream_data_from_trade_stream_buffer(self, triangle_type_dict, unicorn_websocket_manager,
                                                 process_num=None):
    """
    Process the received trade webstream data
     |  Pops from AggTrade stream buffer
     |  "stream_type" becomes the stream_name (key) that identifies the proper fta.streams value (tick instance)
     |  Proceeds to calculate the MACD for that tick object with the popped packet
    """
    self.stdout(f"Started Fta.process_stream_data_from_trade_stream_buffer() ...")
    stream_label = self.trade_channel
    while self.is_stopping() is False:
        oldest_stream_data = unicorn_websocket_manager.pop_stream_data_from_stream_buffer(stream_label)
        if oldest_stream_data is False:
            time.sleep(0.01)
        else:
            try:
                stream_name = oldest_stream_data.get("stream_type")
                tick = triangle_type_dict[stream_name]
                self.alt_base_dict_trend_directions[tick.name] = tick.update_price_and_trend_direction(
                    oldest_stream_data)
            except KeyError as error_msg:
                # Result of None expected when websocket first establishing connection
                if "None" in str(error_msg):
                    pass
                else:
                    self.stdout(f"Process {process_num}: process_stream_data_from_depth_stream_buffer() (Response"
                                f" {oldest_stream_data}) - (Process: {process_num}) - KeyError: {error_msg} - "
                                f" {datetime.now()}", "INFO", print_enabled=True)
            except TypeError as e:
                self.stdout(f"Process {process_num}: process_depth_stream_buffer(stream_name={stream_name}) "
                            f"streams TypeError: {e}", "WARNING")
```



```python
def update_price_and_trend_direction(self, stream_data):
    try:
        self.price = float(stream_data.get("price"))
        self.trend_direction = self.calculate_macd(stream_data)
        return self.trend_direction
    except Exception as e:
        print(f"EXCEPTION update_price_and_trend_direction: {e} \nfor {self.name} with "
              f"ticks: {self.ticks}")

def calculate_macd(self, stream_data):
    """
    Calculate MACD and generate trend direction indicator!

    :param ma_fast: period length (historic candles + current candle) for generating fast moving average
    :param ma_slow: period length (historic candles + current candle) for generating slow moving average
    :param signal_length: period length - number of data points (of ma_fast - ma_slow) - for generating signal line
    :return: string
    """
    try:
        if self.ticks is not None:
            row_df = pd.DataFrame(
                [{'datetime': datetime.utcfromtimestamp(stream_data.get('trade_time') / 1e3), 'close': self.price}])
            row_df.set_index('datetime', inplace=True)
            temp_ticks = pd.concat([self.ticks, row_df])
            signals = Tick._generate_signals(temp_ticks, self.name)
            if (signals['macd_line'][-2] < signals['signal_line'][-2]) and (
                    signals['macd_line'][-1] > signals['signal_line'][-1]):  # Crossover to upside here
                self.trend_direction = "UP"
            elif (signals['macd_line'][-2] > signals['signal_line'][-2]) and (
                    signals['macd_line'][-1] < signals['signal_line'][-1]):  # Crossover to downside here
                self.trend_direction = "DOWN"
            elif self.trend_direction == "DOWN" and self.macd_minus_signal_cache < 0 and signals['MACD-Signal'][
                -1] > 0:  # Intra-minute crossover from downside to upside
                self.trend_direction = "UP"
            elif self.trend_direction == "UP" and self.macd_minus_signal_cache > 0 and signals['MACD-Signal'][
                -1] < 0:  # Intra-minute crossover from upside to downside
                self.trend_direction = "DOWN"
            self.macd_minus_signal_cache = signals['MACD-Signal'][-1]
            signals['price'] = temp_ticks
            return self.trend_direction
        return self.trend_direction
    except IndexError as i:
        print(f"INDEX ERROR: {i} \nfor {self.name} on ttype: {self.ttype} with ticks: {self.ticks}")
```

1. **Associate Indicator values with starting, stopping, and re-initializing the UNICORN Binance trailing stop loss engine.**



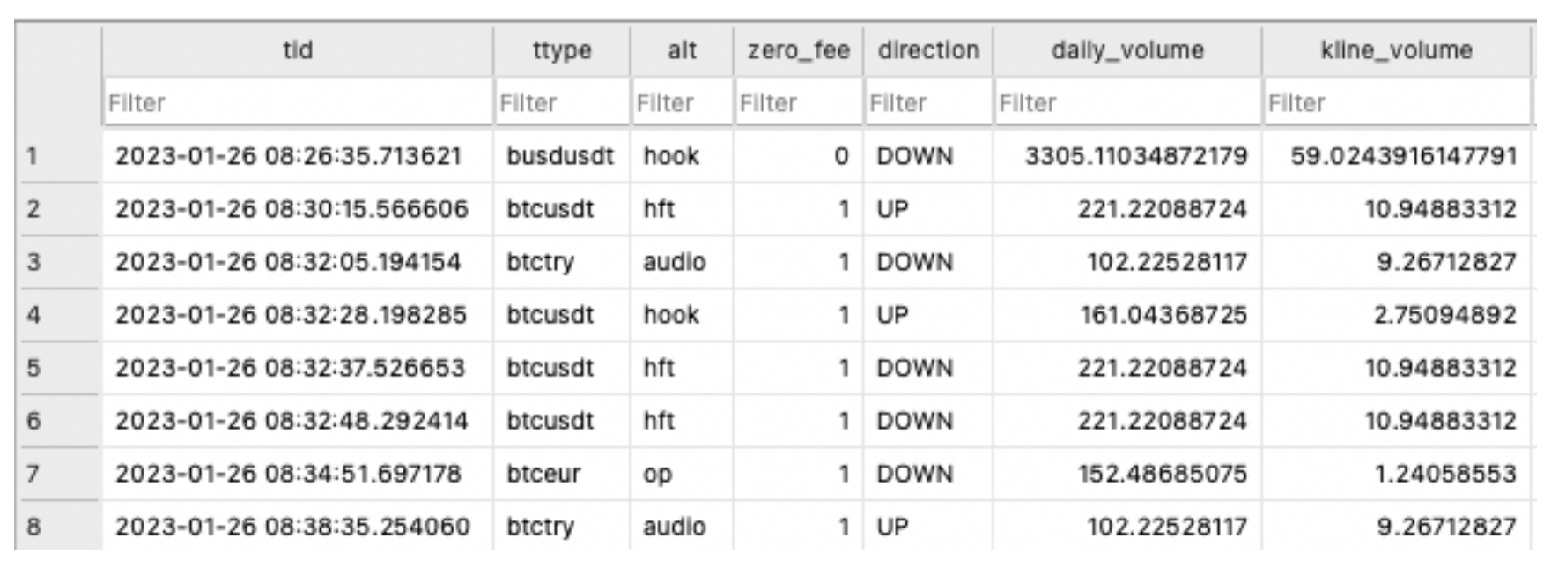
```python
from unicorn_binance_trailing_stop_loss.manager import BinanceTrailingStopLossManager

def callback_error(msg):
    print(f"STOP LOSS ERROR - ENGINE IS SHUTTING DOWN! - {msg}")
    ubtsl.stop_manager()

def callback_finished(msg):
    print(f"STOP LOSS FINISHED - ENGINE IS SHUTTING DOWN! - {msg}")
    ubtsl.stop_manager()

def callback_partially_filled(msg):
    print(f"STOP LOSS PARTIALLY_FILLED - ENGINE IS STILL RUNNING! - {msg}")

ubtsl = BinanceTrailingStopLossManager(callback_error=callback_error,
                                       callback_finished=callback_finished,
                                       callback_partially_filled=callback_partially_filled,
                                       binance_public_key="aaa",
                                       binance_private_key="bbb",
                                       borrow_threshold="100%",
                                       exchange="binance.com",
                                       keep_threshold="20%",
                                       market="BTCUSDT",
                                       print_notifications=True,
                                       reset_stop_loss_price=True,
                                       send_to_email_address="blah@example.com",
                                       send_from_email_address="blub@example.com",
                                       send_from_email_password="pass",
                                       send_from_email_server="mail.example.com",
                                       send_from_email_port=25,
                                       stop_loss_limit="1.5%",
                                       stop_loss_order_type="LIMIT",
                                       stop_loss_price=88,
                                       stop_loss_start_limit="0.5%",
                                       telegram_bot_token="telegram_bot_token",
                                       telegram_send_to="telegram_send_to")
```

2. **Automate stable bot cycles that in principle persist indefinitely.**

   You must implement bag management and disbursement, ensure the absence of memory leaks, proper error handling—especially for unknown errors (i.e., graceful shutdowns with notifications), and the inclusion of fail safes surrounding equity use and amount.

3. **Simulate indicator and engine use, gather data, and train networks.**

   Will require the use of parallel jobs from UMASS Boston's Gibbs compute cluster to control for market conditions.



	tid	ttype	alt	zero_fee	direction	daily_volume	kline_volume
	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	2023-01-26 08:26:35.713621	busdusdt	hook	0	DOWN	3305.11034872179	59.0243916147791
2	2023-01-26 08:30:15.566606	btcusdt	hft	1	UP	221.22088724	10.94883312
3	2023-01-26 08:32:05.194154	btctry	audio	1	DOWN	102.22528117	9.26712827
4	2023-01-26 08:32:28.198285	btcusdt	hook	1	UP	161.04368725	2.75094892
5	2023-01-26 08:32:37.526653	btcusdt	hft	1	DOWN	221.22088724	10.94883312
6	2023-01-26 08:32:48.292414	btcusdt	hft	1	DOWN	221.22088724	10.94883312
7	2023-01-26 08:34:51.697178	btceur	op	1	DOWN	152.48685075	1.24058553
8	2023-01-26 08:38:35.254060	btctry	audio	1	UP	102.22528117	9.26712827

4. **Develop and serve Python web app to showcase results and provide settled-upon model.**

Contact

Pablo Bendiksen, p.bendiksen001@umb.edu

Resources

`https://github.com/LUCIT-Systems-and-Development/unicorn-binance-suite`

`https://binance-docs.github.io/apidocs/futures/en/#change-log`

`https://www.umb.edu/rc/hpc/gibbs/gibbs_scheduler`