```python
# IE598 Project - Joseph Loss
#
# tmp_hft_model.py
# Modification of original hft_model file to suit needs
#
# THIS IS A WORK IN PROGRESS FILE - T HERE MAY BE (AND LIKELY IS) SEVERAL BUGS
#
import datetime as dt
import sys
import threading
import time

import numpy as np
import pandas as pd
from ib.opt import ibConnection, Connection
from ibapi.wrapper import EWrapper

import ib_data_types as datatype
from chart import Chart
from ib_util import IBUtil
from contract_data import ContractData
from strategy_parameters import StrategyParameters


# from ibapi import client, wrapper, common, ticktype, message, utils, reader

class HFTModel:

    def __init__(self,
                 host = 'localhost',
                 port = 4002,
                 # port = 7497,
                 client_id = 1, is_use_gateway = True,
                 evaluation_time_secs = 20,
                 resample_interval_secs = '30s',
                 moving_window_period = dt.timedelta(seconds = 90)):

        self.moving_window_period = moving_window_period
        self.chart = Chart()
        self.ib_util = IBUtil()

        # Store parameters for this model
        self.strategy_params = StrategyParameters(evaluation_time_secs, resample_interval_sec

        self.stocks_data = { }  # Dictionary storing StockData objects.
        self.symbols = None  # List of current symbols
        self.account_code = ""
        self.prices = None  # Store last prices in a DataFrame
        self.trade_qty = 100
        self.order_id = 4751
        self.lock = threading.Lock()

        # Use ibConnection() for TWS, or create connection for API Gateway
        self.conn = ibConnection() if is_use_gateway else Connection.create(host = host, port
client_id)
```

```
        self.__register_data_handlers(self.__on_tick_event, self.__event_handler)


    statistics_params = (
        ('m', 0.564),  # slope
        ('b', 5.597244),  # intercept
        ('std', 1.355027),
        ('avg', 0.5),
        ('muti', 1),  # muti*std+avg
        ('size', 1000)
    )


    def __perform_trade_logic(self):
        volatility_ratio = self.strategy_params.get_volatility_ratio()

        is_up_trend, is_down_trend = volatility_ratio > 1.0, volatility_ratio < 1.0
        is_overbought, is_oversold = self.__is_overbought_or_oversold()

        # Our final trade signals
        is_buy_signal, is_sell_signal = (is_up_trend and is_oversold), (is_down_trend and is_

        # Use account position details
        symbol_a = self.symbols[0]
        position = self.stocks_data[symbol_a].position

        is_position_closed, is_short, is_long = (position == 0), (position < 0), (position > (

        upnl, rpnl = self.__calculate_pnls()

        # Display to terminal dynamically
        signal_text = "BUY" if is_buy_signal else "SELL" if is_sell_signal else "NONE"
        console_output = '\r[%s] signal=%s, position=%s UPnL=%s RPnL=%s\r' % (dt.datetime.now(
position, upnl, rpnl)
        sys.stdout.write(console_output)
        sys.stdout.flush()

        # Buy/Sell Signals:
        if is_position_closed and is_sell_signal:
            print("==================================")
            print("OPEN SHORT POSIITON: SELL A BUY B")
            print("==================================")
            self.__place_spread_order(-self.trade_qty)

        elif is_position_closed and is_buy_signal:
            print("==================================")
            print("OPEN LONG POSIITON: BUY A SELL B")
            print("==================================")
            self.__place_spread_order(self.trade_qty)

        elif is_short and is_buy_signal:
            print("==================================")
            print("CLOSE SHORT POSITION: BUY A SELL B")
            print("==================================")
            self.__place_spread_order(self.trade_qty)
```

```
109
110         elif is_long and is_sell_signal:
111             print("================================")
112             print("CLOSE LONG POSITION: SELL A BUY B")
113             print("================================")
114             self.__place_spread_order(-self.trade_qty)
115
116
117     def __recalculate_strategy_parameters_at_interval(self):
118         """
119         Consider re-evaluation of parameters on:
120         - regime shifts
121         - structural breaks
122         """
123         if self.strategy_params.is_evaluation_time_elapsed():
124             self.__calculate_strategy_params()
125             self.strategy_params.set_new_evaluation_time()
126             print('[%s] === Beta re-evaluated ===' % dt.datetime.now())
127
128
129     def __calculate_strategy_params(self):
130         """
131         Here, we are calculating beta and volatility ratio
132         for our signal indicators.
133
134         Consider calculating other statistics here:
135         - stddevs of errs
136         - correlations
137         - co-integration
138         """
139         [symbol_a, symbol_b] = self.symbols
140         filled_prices = self.prices.fillna(method = 'ffill')
141         resampled = filled_prices.resample(self.strategy_params.resample_interval_secs).ffill(
142
143         mean = np.mean(resampled)
144         beta = mean[symbol_a] / mean[symbol_b]
145
146         # self.symbols[symbol_b]
147
148         stddevs = np.std(resampled.pct_change().dropna())
149         volatility_ratio = stddevs[symbol_a] / stddevs[symbol_b]
150         # import random
151         # volatility_ratio = random.normalvariate(0,1)
152
153         self.strategy_params.add_indicators(beta, volatility_ratio)
154
155
156     def __register_data_handlers(self,
157                                  tick_event_handler,
158                                  universal_event_handler):
159         self.conn.registerAll(universal_event_handler)
160         self.conn.unregister(universal_event_handler,
161                              EWrapper.tickSize,
162                              EWrapper.tickPrice,
163                              EWrapper.tickString,
```

```
                              EWrapper.tickGeneric,
                              EWrapper.tickOptionComputation)
        self.conn.register(tick_event_handler,
                           EWrapper.tickPrice,
                           EWrapper.tickSize)


    def __init_stocks_data(self, symbols):
        self.symbols = symbols
        self.prices = pd.DataFrame(columns = symbols)  # Init price storage
        for stock_symbol in symbols:
            contract = self.ib_util.create_stock_contract(stock_symbol)
            self.stocks_data[stock_symbol] = ContractData(contract)


    # Request Streaming Market Data
    def __request_streaming_data(self, ib_conn):
        for index, (key, stock_data) in enumerate(self.stocks_data.items()):
            ib_conn.reqMktData(index,
                               stock_data.contract,
                               datatype.GENERIC_TICKS_NONE,
                               datatype.SNAPSHOT_NONE)
            time.sleep(1)

        #
-------------------------------------------------------------------------------------------
        # Requesting Bid/Ask Tick Data:
        # This is proving extremely difficult to implement; work in progress.

        #     def __request_bidask_data(self, ib_conn):
        #       for index, (key, stock_data) in enumerate(self.stocks_data.items()):
        #           client.EClient.reqTickByTickData(self, index, stock_data.contract, "BidAs
        #           time.sleep(1)


        # Stream account updates
        ib_conn.reqAccountUpdates(True, self.account_code)

    # Request Historical Data
    def __request_historical_data(self, ib_conn):
        self.lock.acquire()
        try:
            for index, (key, stock_data) in enumerate(
                    self.stocks_data.items()):
                stock_data.is_storing_data = True
                ib_conn.reqHistoricalData(
                        index,
                        stock_data.contract,
                        time.strftime(datatype.DATE_TIME_FORMAT),
                        datatype.DURATION_1_HR,
                        datatype.BAR_SIZE_5_SEC,
                        datatype.WHAT_TO_SHOW_TRADES,
                        datatype.RTH_ALL,
                        datatype.DATEFORMAT_STRING)
                time.sleep(1)
```

```
        finally:
            self.lock.release()


    def __wait_for_download_completion(self):
        is_waiting = True
        while is_waiting:
            is_waiting = False
            self.lock.acquire()
            try:
                for symbol in self.stocks_data.keys():
                    if self.stocks_data[symbol].is_storing_data:
                        is_waiting = True
            finally:
                self.lock.release()
            if is_waiting:
                time.sleep(1)


    def __place_spread_order(self, qty):
        [symbol_a, symbol_b] = self.symbols
        self.__send_order(symbol_a, qty)
        self.__send_order(symbol_b, -qty)


    def __send_order(self, symbol, qty):
        stock_data = self.stocks_data[symbol]
        order = self.ib_util.create_stock_order(abs(qty), qty > 0)
        self.conn.placeOrder(self.__generate_order_id(),
                             stock_data.contract,
                             order)
        stock_data.add_to_position(qty)


    def __generate_order_id(self):
        next_order_id = self.order_id
        self.order_id += 1
        return next_order_id


    def __on_portfolio_update(self, msg):
        for key, stock_data in self.stocks_data.items():
            if stock_data.contract.m_symbol == msg.contract.m_symbol:
                stock_data.update_position(msg.position,
                                           msg.marketPrice,
                                           msg.marketValue,
                                           msg.averageCost,
                                           msg.unrealizedPNL,
                                           msg.realizedPNL,
                                           msg.accountName)
                return


    # ------------------------------------------------------------------------------
    # Overbought / Oversold Trading Signal
```

```
273     # This code is necessary to run the original trade logic above.
274     # I commented it out so I could create my own strategy.
275
276     def __is_overbought_or_oversold(self):
277         [symbol_a, symbol_b] = self.symbols
278         # leg_a_last_price = np.mean(self.prices[symbol_a].values[-25:])
279         # leg_b_last_price = np.mean(self.prices[symbol_b].values[-25:])
280
281
282         # limit loss, we will not trade unless  -(2*std + 0.5) <= (BP-m*RSDA-b) <= (2*std + 0
283         if (((self.prices[symbol_a].values[0] - self.statistics_params.m *
284               self.prices[symbol_b].values[0] - self.statistics_params.b) <=
285              (2 * self.statistics_params.std * self.statistics_params.muti + self.statistics_
286                 ((self.prices[symbol_a].values[0] - self.statistics_params.m *
287                   self.prices[symbol_b].values[0] - self.statistics_params.b) >=
288                  -(2 * self.statistics_params.std * self.statistics_params.muti + self.statis
289
290             # Buy BP-m * RSDA-b > 1*std + 0.5
291             if ((self.self.prices[symbol_a].values[0] - self.statistics_params.m *
self.self.prices[symbol_b].values[
292                 0] - self.statistics_params.b) > (self.statistics_params.std * self.statistic
self.statistics_params.avg)):
293                 self.log('SELL PORTFOLIO,SELL{},BUY{}'.format(self.self.prices[symbol_a].valu
self.self.prices[symbol_b].values[0]))
294                 # Keep track of the created order to avoid a 2nd order
295                 self.order = self.sell(self.datas[0], size = self.statistics_params.size)
296                 self.order = self.buy(self.datas[1], size = int(self.statistics_params.m *
self.statistics_params.size))
297
298             # BP-m*RSDA-b <-(1*std + 0.5)
299             elif ((self.self.prices[symbol_a].values[0] - self.statistics_params.m *
self.self.prices[symbol_b].values[
300                 0] - self.statistics_params.b) < -(self.statistics_params.std * self.statistic
self.statistics_params.avg)):
301                 self.log('BUY PORTFOLIO,BUY{},SELL{}'.format(self.self.prices[symbol_a].value
self.self.prices[symbol_b].values[0]))
302                 # Keep track of the created order to avoid a 2nd order
303                 self.order = self.buy(self.datas[0], size = self.statistics_params.size)
304                 self.order = self.sell(self.datas[1], size = int(self.statistics_params.m *
self.statistics_params.size))
305
306         # this part is not exactly what I want but till now it is fine
307         else:
308             # if the price is too high we need to buy our portfolio back incase of risk(buy B
309             if ((self.self.prices[symbol_a].values[0] - self.statistics_params.m *
self.self.prices[symbol_b].values[
310                 0] - self.statistics_params.b) >= (
311                     2 * self.statistics_params.std * self.statistics_params.muti + self.stati
312                 self.log('CLOSE THE SELL LIMIT LOSS,BUY{},SELL{}'.format(self.self.prices[syml
313                                                                          self.self.prices[syml
314                 # Keep track of the created order to avoid a 2nd order
315                 self.order = self.buy(self.datas[0], size = self.statistics_params.size)
316                 self.order = self.sell(self.datas[1], size = int(self.statistics_params.m *
self.statistics_params.size))
317
318             # if the price is too low we need to sell our portfolio back incase of risk(sell
```

```
319             elif ((self.self.prices[symbol_a].values[0] - self.statistics_params.m *
    self.self.prices[symbol_b].values[
320                 0] - self.statistics_params.b) <= -(
321                     2 * self.statistics_params.std * self.statistics_params.muti + self.statis
322                 self.log('CLOSE THE BUY LIMIT LOSS,SELL{},BUY{}'.format(self.self.prices[symbo
323                                                                         self.self.prices[symbo
324                 # Keep track of the created order to avoid a 2nd order
325                 self.order = self.sell(self.datas[0], size = self.statistics_params.size)
326                 self.order = self.buy(self.datas[1], size = int(self.statistics_params.m *
    self.statistics_params.size))
327
328             else:
329                 # SELL
330                 # -0.5 < BP-m*RSDA-b < 0.5
331                 if (((self.self.prices[symbol_a].values[0] - self.statistics_params.m *
    self.self.prices[symbol_b].values[
332                     0] - self.statistics_params.b) < self.statistics_params.avg) and
333                         ((self.self.prices[symbol_a].values[0] - self.statistics_params.m *
    self.self.prices[symbol_b].values[
334                             0] - self.statistics_params.b) > -self.statistics_params.avg)):
335
336                     # comes from higher portfolio value
337                     if ((self.self.prices[symbol_a].values[0] - self.statistics_params.m *
    self.self.prices[symbol_b].values[
338                         0] - self.statistics_params.b) > 0):
339                         self.log('CLOSE THE SELL,BUY{},SELL{}'.format(self.self.prices[symbol_
340                                                                       self.self.prices[symbol_
341                         # Keep track of the created order to avoid a 2nd order
342                         self.order = self.buy(self.datas[0], size = self.statistics_params.si
343                         self.order = self.sell(self.datas[1], size = int(self.statistics_para
    self.statistics_params.size))
344
345                     # comes from lower portfolio value
346                     else:
347                         self.log(
348                             'CLOSE THE BUY,SELL{},BUY{}'.format(self.self.prices[symbol_a].va
    self.self.prices[symbol_b].values[0]))
349                         # Keep track of the created order to avoid a 2nd order
350                         self.order = self.sell(self.datas[0], size = self.statistics_params.s
351                         self.order = self.buy(self.datas[1], size = int(self.statistics_param
    self.statistics_params.size))
352
353
354         # expected_leg_a_price = leg_b_last_price * self.strategy_params.get_beta()
355         #
356         # is_overbought = leg_a_last_price < expected_leg_a_price  # Cheaper than expected
357         # is_oversold = leg_a_last_price > expected_leg_a_price  # Higher than expected
358         # return is_overbought, is_oversold
359
360
361     def __calculate_pnls(self):
362         upnl, rpnl = 0, 0
363         for key, stock_data in self.stocks_data.items():
364             upnl += stock_data.unrealized_pnl
365             rpnl += stock_data.realized_pnl
366         return upnl, rpnl
```

```
367
368
369     def __event_handler(self, msg):
370         if msg.typeName == datatype.MSG_TYPE_HISTORICAL_DATA:
371             self.__on_historical_data(msg)
372
373         elif msg.typeName == datatype.MSG_TYPE_UPDATE_PORTFOLIO:
374             self.__on_portfolio_update(msg)
375
376         elif msg.typeName == datatype.MSG_TYPE_MANAGED_ACCOUNTS:
377             self.account_code = msg.accountsList
378
379         elif msg.typeName == datatype.MSG_TYPE_NEXT_ORDER_ID:
380             self.order_id = msg.orderId
381         else:
382             print(msg)
383
384
385     def __on_historical_data(self, msg):
386         print(msg)
387         ticker_index = msg.reqId
388         if msg.WAP == -1:
389             self.__on_historical_data_completed(ticker_index)
390         else:
391             self.__add_historical_data(ticker_index, msg)
392
393
394     def __on_historical_data_completed(self, ticker_index):
395         self.lock.acquire()
396         try:
397             symbol = self.symbols[ticker_index]
398             self.stocks_data[symbol].is_storing_data = False
399         finally:
400             self.lock.release()
401
402
403     def __add_historical_data(self, ticker_index, msg):
404         timestamp = dt.datetime.strptime(msg.date, datatype.DATE_TIME_FORMAT)
405         self.__add_market_data(ticker_index, timestamp, msg.close)
406
407
408     def __on_tick_event(self, msg):
409         ticker_id = msg.tickerId
410         field_type = msg.field
411
412         # Store information from last traded price
413         if field_type == datatype.FIELD_LAST_PRICE:
414             last_price = msg.price
415             self.__add_market_data(ticker_id, dt.datetime.now(), last_price)
416             self.__trim_data_series()
417
418         # Post-bootstrap - make trading decisions
419         if self.strategy_params.is_bootstrap_completed():
420             self.__recalculate_strategy_parameters_at_interval()
421             self.__perform_trade_logic()
```

```
422             self.__update_charts()
423
424
425     def __add_market_data(self, ticker_index, timestamp, price):
426         symbol = self.symbols[ticker_index]
427         self.prices.loc[timestamp, symbol] = float(price)
428         self.prices = self.prices.fillna(method = 'ffill')  # Clear NaN values
429         self.prices.sort_index(inplace = True)
430
431
432     def __update_charts(self):
433         if len(self.prices) > 0 and len(self.strategy_params.indicators) > 0:
434             self.chart.display_chart(self.prices,
435                                      self.strategy_params.indicators)
436
437
438     def __trim_data_series(self):
439         cutoff_timestamp = dt.datetime.now() - self.moving_window_period
440         self.prices = self.prices[self.prices.index >= cutoff_timestamp]
441         self.strategy_params.trim_indicators_series(cutoff_timestamp)
442
443
444     @staticmethod
445     def __print_elapsed_time(start_time):
446         elapsed_time = time.time() - start_time
447         print("Completed in %.3f seconds." % elapsed_time)
448
449
450     def __cancel_market_data_request(self):
451         for i, symbol in enumerate(self.symbols):
452             self.conn.cancelMktData(i)
453             time.sleep(1)
454
455
456     def start(self, symbols, trade_qty):
457         print("IB PairsTrading Algo model started.")
458         self.trade_qty = trade_qty
459
460         self.conn.connect()  # Get IB connection object
461         self.__init_stocks_data(symbols)
462         self.__request_streaming_data(self.conn)
463
464         print("Bootstrapping the model...")
465         start_time = time.time()
466         self.__request_historical_data(self.conn)
467         self.__wait_for_download_completion()
468         self.strategy_params.set_bootstrap_completed()
469         self.__print_elapsed_time(start_time)
470
471         print("Calculating strategy parameters...")
472         start_time = time.time()
473         self.__calculate_strategy_params()
474         self.__print_elapsed_time(start_time)
475
476         print("Trading started.")
```

```
        try:
            self.__update_charts()

            while True:
                self.__recalculate_strategy_parameters_at_interval()
                self.__perform_trade_logic()
                self.__update_charts()
                self.__calculate_pnls()
                time.sleep(1)


        except Exception as e:
            print("Exception:", e)
            print("Cancelling...", )
            self.__cancel_market_data_request()

            print("Disconnecting...")
            self.conn.disconnect()
            time.sleep(1)

            print("Disconnected.")

```