The new playground of statistical arbitrage: the fall and rise of cryptocurrencies

Preface: If you are an individual day trader, please read on because this blog post could provide you with the quickest and easiest way to profiting from day trading! If you are the manager of an institution investor, please read on because this blog post could provide you with the fastest and most accurate way to obtaining real-time market data for decision making! If you are an cryptocurrency broker, please read on, because this blog post could provide you with a perfect example of trading bot where you could plugin your fabulous business!

Bitcoin has left a permanent imprint in the financial world since its dramatic fall in price near the beginning of 2018. Many investors have since then cursed cryptocurrencies and some are still licking their wounds. However, recently there seems to be a strong rebound and revived interest from mainstream investors on cryptocurrencies and market is recovering in an amazing pace. Whenever there are volatilities, there are opportunities for free lunches. Thanks to their nature of openness and decentralization, professional trading on cryptocurrencies is available to everyone because it is orders of magnitude easier to gain access to high quality market data on cryptocurrencies compared to traditional exchange traded assets. If you are a fan of statistical arbitrage and have some interest in trading bots, let us take a joint venture (no pun intended :)) on it! Python will be used throughout this article, but the ideas and logics could be easily transferred to other programming languages. The major libraries that we are going to leverage are Python's requests library and websocket-client library.

There is a golden rule in data science: garbage-in-garbage-out. It is of vital importance to obtain high quality data if you are going to bet serious money on them. At the moment of writing almost every cryptocurrency exchange on the planet provides their market data free of charge. Everyone could write programs to collect those data and store them, although that is a ton of data and the focus of this blog post is about trading rather than data-warehousing. So I will pick one data provider, i.e. amberdata.io to get our precious high-quality market data to start building our lovely robot.

To use their data, we need to sign up to get an api key. Let us start by looking at some of the basic information about our market data:

```
import requests
from log import logger
from constants import Constants
if __name__ == '__main__':
    logger.info('main start')
    url = Constants.AMBERDATA_BASE_MARKET + '/exchanges'
    headers = { 'x-api-key': Constants.AMBERDATA_API_KEY }
    response = requests.get(url, headers=headers)
    payload = response.json()['payload']
    exchange_count_pair = [(k, len(v)) for (k,v) in payload.items()]
    exchange_count_pair.sort(key=lambda x: x[1], reverse=True)
    logger.info(exchange_count_pair[:10])
    pair_count_exchange_dict = {}
    for k, v in payload.items():
        for pair in list(v.keys()):
            if pair not in pair_count_exchange_dict:
                pair_count_exchange_dict[pair] = 0
            pair_count_exchange_dict[pair] += 1
    pair_count_exchange = [(k, v) for (k,v) in pair_count_exchange_dict.items()]
    pair_count_exchange.sort(key=lambda x: x[1], reverse=True)
    logger.info(pair_count_exchange[:10])
    logger.info('main end')
```

This gives us some basic ideas about which exchanges have more supported pairs as well as which pairs have wider supports by exchanges.

[('binance', 502), ('huobi', 440), ('bitfinex', 349), ('zb', 217), ('bithumb', 82), ('kraken', 70), ('gdax', 38), ('bitstamp', 15), ('gemini', 15), ('bitmex', 8)]

[('eth_btc', 7), ('ltc_btc', 7), ('xrp_btc', 6), ('eth_usd', 6), ('ltc_usd', 6), ('etc_btc', 5), ('xlm_btc', 5), ('zrx_btc', 5), ('btc_usd', 5), ('xrp_usd', 5)]

Great. We could see that binance and huobi have a lot of pairs available for trading: significantly more than the other exchanges. Meanwhile the leading most widely supported pairs do not differ a lot from

each other. Let us pick Binance/Huobi and eth_btc for gold digging.

We will utilize 1 minute open-high-low-close price for as the starting point for our algorithm building. Let us examine on binance and huobi how much data are available:

```
import requests
from log import logger
from constants import Constants
from datetime import datetime
if __name__ == '__main__':
    logger.info('main start')
    url = Constants.AMBERDATA_BASE_MARKET + '/ohlcv/information?exchange=binance,huobi'
    headers = { 'x-api-key': Constants.AMBERDATA_API_KEY }
    response = requests.get(url, headers=headers)
    payload = response.json()['payload']
    for exchange in ['binance', 'huobi']:
        logger.info('{} eth_btc startDate is {}'.format(exchange, datetime.utcfromtimestamp(payload[exchange]['eth_btc']
['startDate']/1000.0).isoformat()))
    logger.info('main end')
```

We got:
binance eth_btc startDate is 2017-07-14T00:00:00
huobi eth_btc startDate is 2019-04-24T00:00:00
Good, for 1 minute open-high-low-close price, we have about at least 50K data points overlapping between binance and huobi for eth_btc, good enough to reveal statistical significance. The statistical metric that we are aiming at is the standard deviation between the close prices on the two exchanges.

Now let us retrieve historical 1 minute open-high-low-close price data from 2019-04-24 to 2019-05-24.

```
import requests
import statistics
from log import logger
from urllib.parse import urlencode
from constants import Constants
def join(data_a, data_b, key_index, value_index):
    data_b_dict = { x[key_index]: x[value_index] for x in data_b }
    data_joined = []
    for x in data_a:
        if x[key_index] in data_b_dict:
            data_joined.append([x[key_index], x[value_index], data_b_dict[x[key_index]]])
    return data_joined

if __name__ == '__main__':
    logger.info('main start')
    url = Constants.AMBERDATA_BASE_MARKET + '/ohlcv/eth_btc/historical?'
    filters = {}
    filters['exchange'] = 'binance,huobi'
    filters['timeInterval'] = 'minutes'
    filters['startDate'] = 1556064000 #'2019-04-24'
    filters['endDate'] = 1558656000 #'2019-05-24'
    url += urlencode(filters)
    logger.info('url = {}'.format(url))
    headers = { 'x-api-key': Constants.AMBERDATA_API_KEY }
    response = requests.get(url, headers=headers)
    data = response.json()['payload']['data']
    data_joined = join(data['binance'], data['huobi'], 0, 4)
    diff_price_stdev = statistics.stdev([x[1] - x[2] for x in data_joined])
    logger.info('diff_price_stdev = {}'.format(diff_price_stdev))
    logger.info('main end')
```

We got the standard deviation of prices between binance and huoi for eth_btc. Now for simplicity we make an important assumption that the price difference for the same asset between two exchanges follows a normal distribution: this means that most of the time the difference will stay between -2*sigma and +2 sigma (use 2 here since classical pair arbitrage usually uses 2), but occasionally it might walk outside that range. As time elapses, it will eventually be pulled back into that range. The classical pair arbitrage will long X eth (Note: in the eth_btc trading pair, not eth_usd) on binance and short X eth on huobi when the price difference is below -2*sigma, and when that difference is pulled back to near 0 (e.g -0.1*sigma), we exit those two positions, netting a profit. Similarly, the classical pair arbitrage will short X eth on binance and long X eth on huobi when the

price difference is above 2*sigma, and when that difference is pulled back to near 0 (e.g 0.1*sigma), we exit those two positions, also netting a profit. The beauty of this classical strategy is that it is market neutral, independent of whether the actual prices are going up or down. The profits only depend on the fact that the market does fluctuate.

Now we have our trading strategies, let us setup the trading bot. This time we need to use websocket connections to get realtime best-bid-offers for etc_btc on binance and huobi to capture entry and exit points for our positions.

```
import websocket
from log import logger
import ssl
import json
from constants import Constants

prices = {}
sigma = 3.862523415579312e-05
def on_open(ws):
    logger.info('websocket {} was connected'.format(ws.url))
    ws.send(json.dumps({
        'jsonrpc': '2.0',
        'id': 1,
        'method': 'subscribe',
        'params': ['market:bbos', {'pair': 'eth_btc', 'exchange': 'binance'}]
    }))
    ws.send(json.dumps({
        'jsonrpc': '2.0',
        'id': 2,
        'method': 'subscribe',
        'params': ['market:bbos', {'pair': 'eth_btc', 'exchange': 'huobi'}]
    }))

def place_order_if_condition_met(prices):
    if prices['binance_bid'] and prices['huobi_ask']:
        diff = abs(prices['binance_bid'] - prices['huobi_ask'])
        if diff > 2 * sigma:
            logger.info('here we need to take some positions')
        elif diff < 2 * sigma:
            logger.info('here we need to exit some positions')
    if prices['binance_ask'] and prices['huobi_bid']:
        diff = abs(prices['binance_ask'] - prices['huobi_bid'])
        if diff > 2 * sigma:
            logger.info('here we need to take some positions')
        elif diff < 2 * sigma:
            logger.info('here we need to exit some positions')

def on_message(ws, message):
    logger.info('message = {}'.format(message))
    json_message = json.loads(message)
    if json_message.get['params'] and json_message.get['params'].get('result'):
        result = json_message.get['params'].get('result')
        if result['exchange'] == 'binance':
            if result['isBid']:
                prices['binance_bid'] = result['price']
                place_order_if_condition_met(prices)
            else:
                prices['binance_ask'] = result['price']
                place_order_if_condition_met(prices)
        elif result['exchange'] == 'huobi':
            if result['isBid']:
                prices['huobi_bid'] = result['price']
                place_order_if_condition_met(prices)
            else:
                prices['huobi_ask'] = result['price']
                place_order_if_condition_met(prices)
if __name__ == '__main__':
    logger.info('main start')
    ws = websocket.WebSocketApp(Constants.AMBERDATA_WEBSOCKET_BASE)
    ws.header = {'x-api-key': Constants.AMBERDATA_API_KEY}
    ws.on_open = on_open
    ws.on_message = on_message
    ws.run_forever(sslopt={'cert_reqs': ssl.CERT_NONE})
```

For those of you who are conservative traders and think that websocket might be somewhat dangerous when trading real money because a tiny bug could very, very quickly trigger some disaster, we could use a periodic polling logic on the ticker endpoint to replace the websocket continuous feed:

```
import requests
from concurrent.futures import ThreadPoolExecutor,as_completed
```

```
from log import logger

import time
import json
from constants import Constants

sigma = 3.862523415579312e-05

def place_order_if_condition_met(prices):
    if prices['binance_bid'] and prices['huobi_ask']:
        diff = abs(prices['binance_bid'] - prices['huobi_ask'])
        if diff > 2 * sigma:
            logger.info('here we need to take some positions')
        elif diff < 2 * sigma:
            logger.info('here we need to exit some positions')
    if prices['binance_ask'] and prices['huobi_bid']:
        diff = abs(prices['binance_ask'] - prices['huobi_bid'])
        if diff > 2 * sigma:
            logger.info('here we need to take some positions')
        elif diff < 2 * sigma:
            logger.info('here we need to exit some positions')

if __name__ == '__main__':
    logger.info('main start')
    while True:
        url = Constants.AMBERDATA_BASE_MARKET + '/tickers/eth_btc/latest?exchange=binance,huobi'
        headers = { 'x-api-key': Constants.AMBERDATA_API_KEY }
        response = requests.get(url, headers=headers)
        payload = response.json()['payload']
        prices = {}
        prices['binance_bid'] = payload['binance'].get('bid')
        prices['binance_ask'] = payload['binance'].get('ask')
        prices['huobi_bid'] = payload['huobi'].get('bid')
        prices['huobi_ask'] = payload['huobi'].get('ask')
        place_order_if_condition_met(prices)
        time.sleep(60)
    logger.info('main end')
```

Before we put our trading bot into a real battle field with large sums of dollars, there are several things deserving further investigation and scrutinization: a. In our code, we have largely omitted error handling for simplicity. For safety, it is important to add timeout logics into requests for external calls, surround those logics with try-except, and handle those errors accordingly. b. There are a few places where we've chosen some threshold values such as 2*sigma, 0.1*sigma, etc., these values are adjustable and the choices would affect the confidence level of such strategies. c. A trading bot handling real money would need to have components for stop-loss logics, portfolio rebalance, order bookkeeping, etc. etc. d. Cryptocurrencies usually (e.g. for proof-of-work protocol) have some latency for transaction confirmation. If you know the wallet's address, it would be a better idea to check the transaction status by yourself rather than solely depending on the exchange's words, thanks to the decentralized nature of cryptocurrencies.