Real-Time Trade Bot

What Justice Can You Do?

GROUP 9 PROJECT REPORT

Members:
Shruti Agarwal, Dev Priya Goel, Trikay Nalamada,
Divyansh Mangal, Mahfoozur Rahman Khan

[October 15, 2020]

Disclaimer

The content of this report and the product made is only meant for the learning process as part of the course. This is not for use in making publication or making commercialization without the mentor's consent. My contribution won't demand any claim in the future for further progress of Mentor's development and innovation along the direction unless there is a special continuous involvement

Reference

https://github.com/brandomr/realtimecrypto

Links

GitHub Repository Link

Introduction

In this project, we have built a big data pipeline which receives streaming data from a cryptocurrency exchange (Gemini exchange), processes the data, applies predictive algorithms (LSTM and Prophet), executes trades, and stores the results for future usage.

What is Real-Time Trading?

Real-time trading deals with large volumes of data and it's analysis. Real-Time Trading involves the following:

Processing Large Volumes of Data

We need to process the data we receive to get the relevant information. Data processing includes:

- Normalizing Data Fields: It involves grouping similar fields
- Extracting Information: Filtering out the relevant information from all the data coming at us
- Data Enrichment: Adding new fields to the data to make it more meaningful

Combining Sources

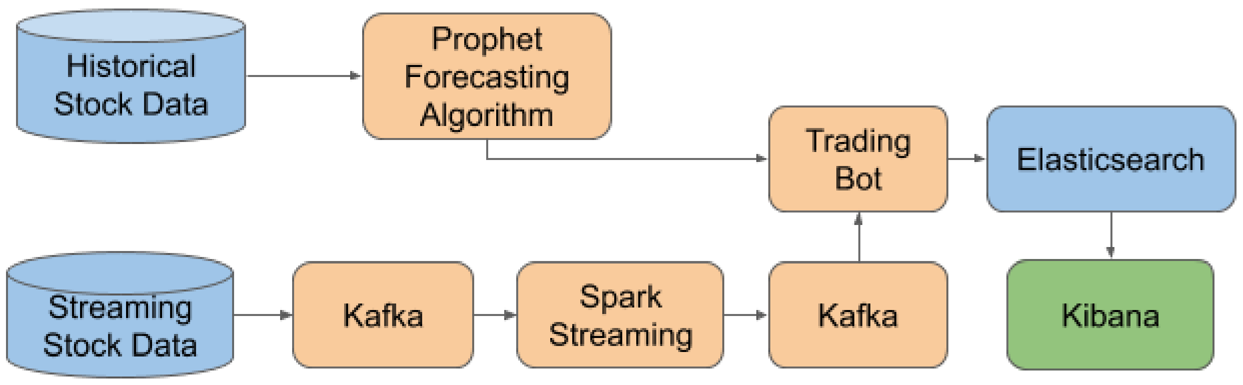
Data will be coming to us from multiple sources. We need to combine all the data to make accurate trading decisions. In our case we will be combining historical pricing data with real-time pricing data from the exchange.

Applying Machine Learning Models

The machine learning model is trained on a historic static dataset and the predictions are stored. These predictions are then used in real-time to make trading decisions.

Road Map to What We are Building

Let's discuss the architecture. This section will explain the trade bot pipeline that we are going to build and the technologies involved.



Real-Time Trade Bot Architecture

Above flow-chart shows the basic working of our trade bot. Let's explain the map in detail.

In the first section, we will take historical stock data which is then fed into a time series forecasting algorithm. Here we will be using the Prophet Forecasting Algorithm which is developed by Facebook to build a machine learning model which learns through the historical stock data extracted. The model's output which we will obtain is directly fed into our trading bot.

The second section will take streaming stock data from Gemini Exchange using Kafka as a message broker. From Kafka, the live data will be sent to Spark for calculation of market's liquidity. The market's liquidity is an important factor that governs the bid-ask spread which in turn affects the volatility of the stock. The liquidity which is obtained from the Spark streaming is fed again to Kafka which provides it to our bot to make decisions whether to make trades or not.

Our bot will directly interact with Gemini exchange, use live data to calculate market liquidity. It will use forecasts to execute trades as long as liquidity levels do not indicate massive volatility. As the bot trades, we store all the data it processes in Elasticsearch from where we will use Kibana to visualize and analyse our results.

Docker

Docker is an OS level virtualization to run applications in a container. A container is a standard unit of software that packages up code and all its dependencies so that the application runs reliably from one computing environment to another. We are building our trade bot and the various components under docker containers.
The need to orchestrate software configuration across a computing cluster and that too for a big data stack is the reason why we are using docker.

In docker, everything is an image. In our project also, we will be using pre-built images of docker containers on which several components will run. An image is basically a snapshot of containers and containers are working versions of images.

In this project, we are using docker-compose to build the containers. Since kibana runs on elasticsearch, we need to run both the kibana and elasticsearch services simultaneously. So we use docker-compose to include both the services so we can run and stop those containers simultaneously.

Pieces of the Pipeline

The code for this is in the pipeline folder in the GitHub link.

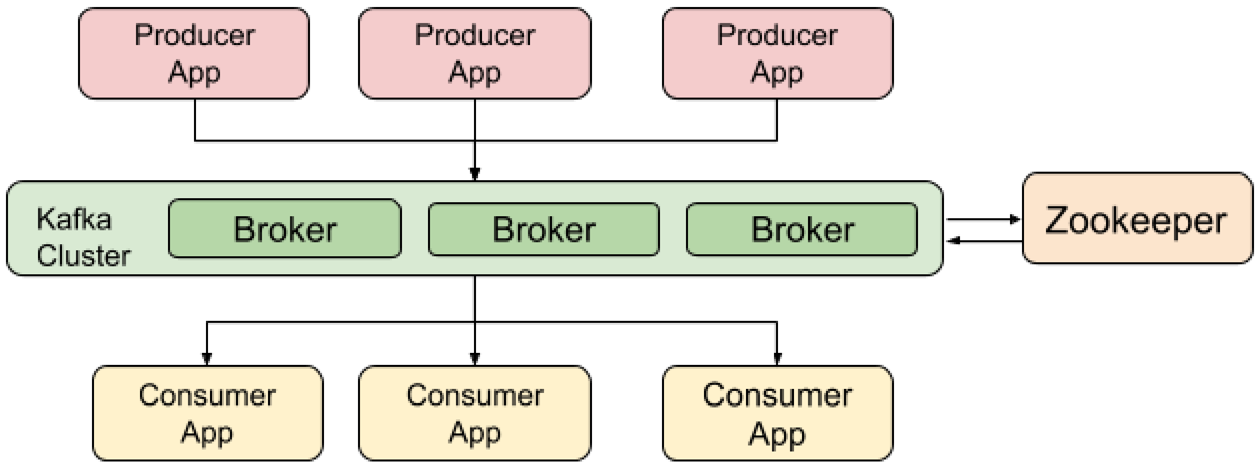
Kafka

Kafka is a distributed streaming platform. It was developed by Netflix. It has since been open sourced. Currently, it is managed by the Apache Software Foundation. It is actively used by LinkedIn, Netflix, Spotify, Uber and other such major players. The Docker container we are using for Kafka is maintained by Spotify.

As a distributed streaming platform, let us look at the streaming platform component of Kafka's functionality. For our use case, we will be dealing with a real-time price feed for Bitcoin. This is a continuous feed of data and tracks the bid/ask price on Gemini exchange. This is the raw data that needs to be processed downstream by Spark for volatility calculations. We do not wish to store this data but instead queue it and use it (Spark). Kafka, as a stream platform, makes this possible. Kafka creates a feed-like structure called "**topic**" which supports two operations namely, **production** which adds data to a topic and **consumption** which consumes data from the topic. Kafka Consumption happens in-order which meets our requirements of chronological streaming of Bitcoin Price data.

As a distributed platform, a single instance of Kafka can be linearly scaled across a cluster of servers. Kafka API abstracts this away from us hence we will not go into further detail at this point.

In Kafka, a topic is partitioned and stored with different brokers and a lot of effort goes into ensuring in-order delivery of data to the consumer.



Kafka Architecture

Kafka enables high throughputs as can be seen from the Kafka Architecture diagram above. Multiple Producer Apps can write to the same topic. Multiple Consumer Apps can consume from the same topic. A Kafka cluster is made up of a number of Kafka brokers. For managing coordination between brokers, Zookeeper, a centralized service is used. It holds relevant information like which broker holds which topic and also information about consumer groups.

Kafka topic is split into partitions which may then be split across multiple brokers or may be stored with a single broker. Kafka uses replication for resiliency to broker failure. Metadata is of paramount importance to the working of Kafka. When a Producer wants to write to a topic, the metadata provided to them by the broker gives them the relevant information about which broker holds which topic. The producer then uses a load balancing partition strategy to decide where it should append a given record.

Each record is stored at an integer offset to the partition since the beginning. These offsets are used by consumers to track their location and also enables them to pick up where they left off in case of failure.

Consumers can form groups and coordinate reading partitions to read in parallel this enables the high throughput we previously discussed. If there are fewer consumers in a consumer group than partitions of a topic then some consumers may have to read from multiple partitions. Also, in case there are more consumers in a group than the number of partitions of the topic they want to read then some consumers don’t read any data.

For the demo, we run `docker-compose up` to compose the docker-compose.yml file. This starts up the Zookeeper and Kafka containers. Once the server is up and running, for single producer and single consumer writing to and reading from a topic, you can run the Kafka.ipynb Jupyter notebook. If you want to play around with multiple producers and consumers running simultaneously, you could run producer.py and consumer.py files on different tabs in the terminal to see multiple producers writing to the same topic and multiple consumers consuming from the same topic. The producer output is “interleaved” according to the time at which they write.

Spark

Apache Spark is a free and open source cluster computing software that can quickly perform processing tasks on very large datasets and can also distribute data processing tasks across multiple computers (termed as nodes in a cluster).Spark is used to conduct massive computations by major tech companies including eBay, Amazon, Baidu, TripAdvisor, and even NASA.

Spark can run on an arbitrary number of compute nodes and automatically scale your computing operations across the cluster, which is called parallelizing the operation.
The need for such **parallelization** arises from the facts that having a large datasets create two major problems , which are stated as :

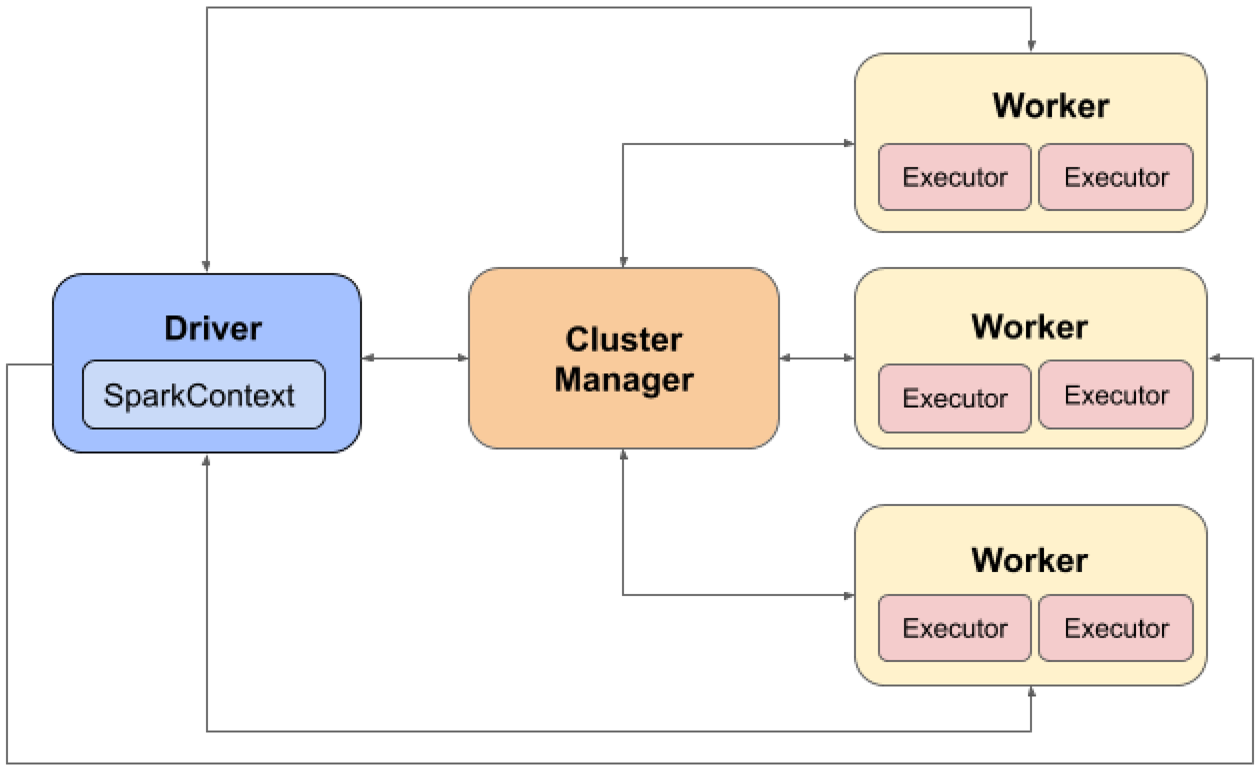
- The data may not completely fit in memory and thus would require external storage requirements.
- Even if the data fits in internal memory the processing speed of accessing large quantities of data will be slow on performing a particular operation.

At its core, Spark consists of a small number of key components.
This includes :-

- **Shark (Spark SQL) :** this component is similar to Pandas in Python in that it enables operations on dataframes
- **Spark Streaming :** this component enables Spark to operate on streaming data.
- **MLlib :** this component allows Spark to train and deploy machine learning algorithms.
- **GraphX :** this component allows for graph processing with spark.

Other components include SparkR,BlindDB etc.
Among these components the component that we are going to focus most is Spark Streaming.The streaming data from kafka will be operated using Spark Streaming.

Similar to Kafka, a Spark cluster is comprised of a set of nodes. We'll call these nodes **Workers**. Workers report to the **cluster manager**. The cluster manager is in charge of figuring out which workers are available for work so that when an application is submitted to the cluster by a driver, the workers are appropriately tasked.
Once the cluster manager has assigned workers to a job the workers then report directly back to the **driver** (also known as the Application Master). In this way, coordination of job execution itself can be done by the driver.



Spark Architecture

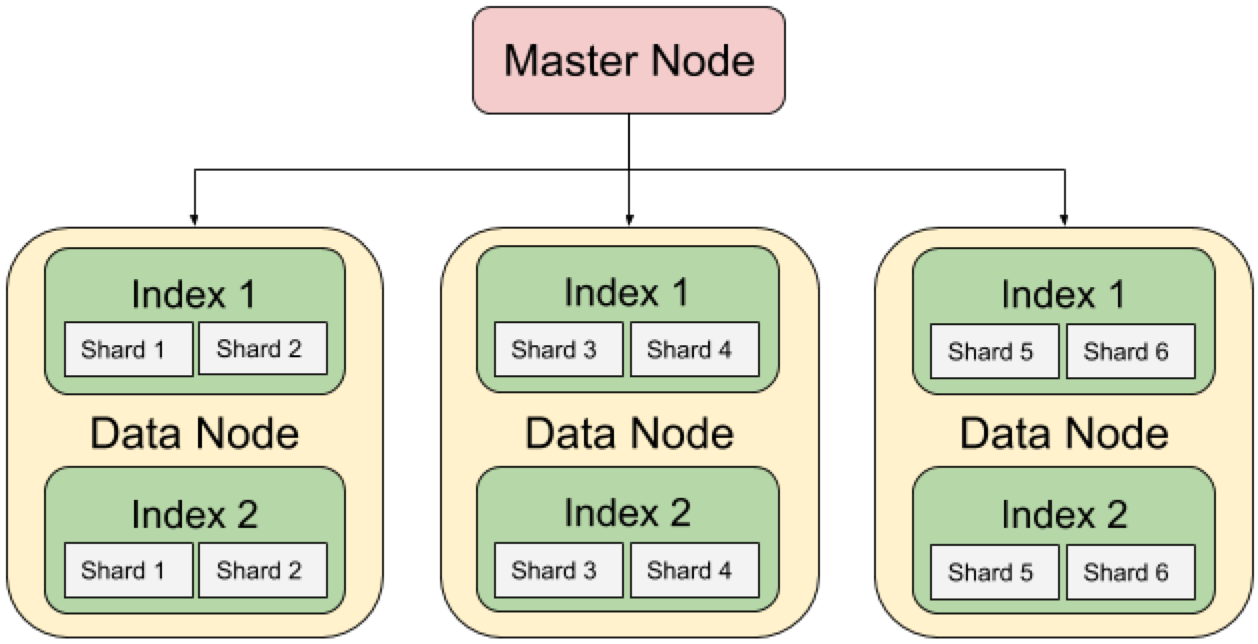
One feature of Spark is that Spark may actually not execute anything until it is time to store data as its final step. That is because Spark separates tasks into **transformations** and **actions**.
A transformation manipulates the data in place or converts data into a new RDD. An example of this may be converting a string to uppercase or filtering for lines which contain a specific keyword.
An action is a task which requires calculations to be performed on the transformation. These include counting records in an RDD, storing these records as a file or to a data store, or taking a sample of the RDD.

Faust

Faust is a stream processing library for python, which we use to process the streaming data from Kafka. We included Faust in the pipeline as a replacement of Spark.

Elasticsearch

Elasticsearch is a distributed document store and search engine. It excels at handling unstructured documents such as HTML pages as it is fundamentally non-relational (unlike the well known SQL database). It can also be used to index JSON objects like the ones which we will be generating.
Unlike SQL which is a schema based data storage, Elasticsearch is document oriented.
Just like in a SQL based relational database where we have multiple databases and tables, Elasticsearch has indexes and types.
Index is a collection of documents that have generally similar characteristics. For example, you might have an index for Tweets that your bot has collected. You might have another index for grocery price data. Each index has a document **type**. Elasticsearch supports one type per index.
Elasticsearch is a cluster with a set of data nodes managed by a master node as shown in the diagram. The **master node** is responsible for creating and deleting indices and determining the status of the data nodes. **Data nodes** do pretty much what you'd expect: they store the data!

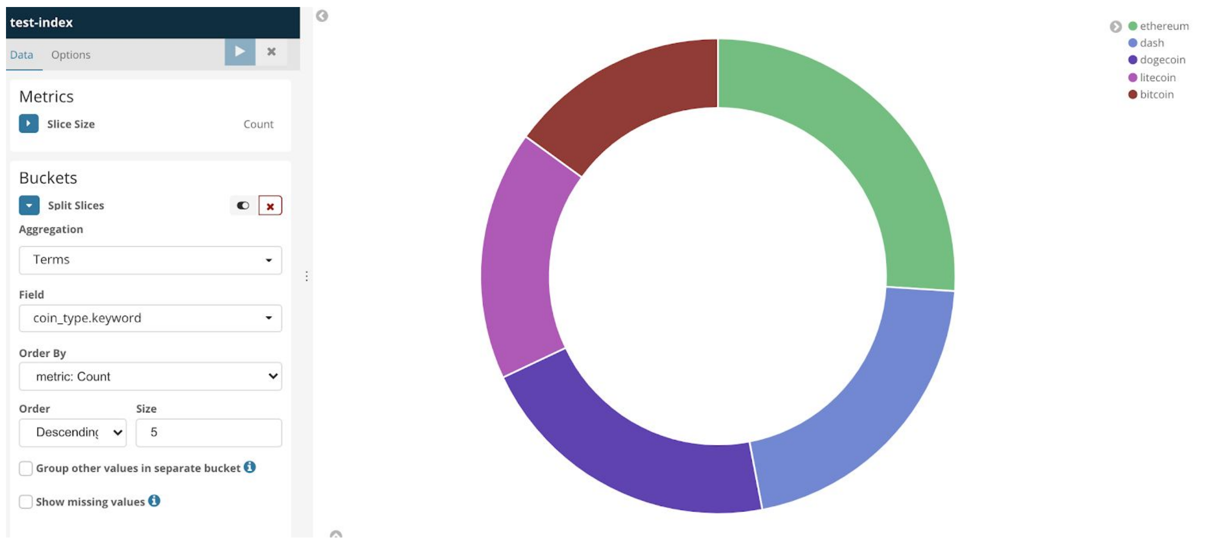


ElasticSearch Architecture

Data from any index can be stored across all available data nodes. Each data node will contain a configurable number of shards for each index. In a production environment it is likely (ahem, advisable!) that some of these shards will be replica shards. That means that if there is an issue with a data node the master node can route traffic to a data node which holds replicas of the down node's data.

Kibana

ElasticSearch and Kibana are together a part of the ELK stack. Index data input into ElasticSearch can be visualized in Kibana as graphs, charts etc. It allows us to make inferences about large amounts of data through simple visualizations. It also offers powerful and easy to use features such as histograms, line graphs etc, with built in geospatial support.
Following is an example of a pie chart created using the coin type data that we stored in ElasticSearch.


test-index

Data Options

Metrics

Slice Size Count

Buckets

Split Slices

Aggregation: Terms

Field: coin_type.keyword

Order By: metric: Count

Order: Descending

Size: 5

☐ Group other values in separate bucket

☐ Show missing values

Pie-Chart Visualisation

Price Forecasting

In this section we explore 2 major methods for forecasting out the future prices of the financial asset of interest, in this case, Bitcoin Prices. We initially collect minute to minute historic Bitcoin price data from kaggle (kaggle.com/mczielinski/bitcoin-historical-data). We use this data to fit/train a model of our choosing and then use the trained model to provide predictions of Bitcoin prices for times in the future / not previously observed.

Data

Attached below is a snapshot of the tail of the dataset.

Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
1508457360	5690.88	5690.88	5690.88	5690.88	0.168941	961.421706	5690.880000
1508457420	5698.13	5704.10	5695.63	5704.10	2.311662	13174.852877	5699.300163

The timestamps are in Unix Epoch Time, which is the number of seconds since 1 January, 1970. The first step of preprocessing involves converting this to a datetime format. The other main variable of interest is the Weighted Price. It is the value of trade prices, weighted by the volume of the trade. This is the variable we will try to forecast and use for our trading. We in fact take the log of weighted price and fit our model according to these labels, because we are more interested in predicting the change in prices, rather than the absolute value of prices themselves. Therefore our final data frame consists of two columns, one for the datetime, and another for the log of weighted prices, which we call y.

Prophet Algorithm

The first tool we use for forecasting is an open source time series package, developed by Facebook, called Prophet. Prophet takes historic data, such as BitCoin prices, Stock prices, and provides a prediction about where the data will be in the future, along with a confidence interval of the prediction.

Prophet is the algorithm due to several reasons, few being:

- It is flexible enough for a wide range of business time series tasks.
- It provides a sophisticated regression model which extracts complex features such as monthly, weekly and daily seasonality components.
- The above features allow us to make forecasts about the price of an asset on specific days of the week, and also on specific weeks of a year.
- It also has functionality to handle holidays and other major events, in the price prediction.

Under the hood, prophet takes the given form

$y(t) = g(t) + s(t) + h(t) + e(t)$

$g(t)$ models the trend of the model. In other words it is responsible for capturing the non-periodic components which lead to an increase of decrease of prices over a time period. It does this by fitting a logistic growth model.

$s(t)$ is responsible for the seasonality or repetitive behaviour of the data. It captures the periodic behaviour in the data by using a partial Fourier sum.

$h(t)$ captures holidays or special events which can cause sudden increases or decreases in the prices. Finally e(t) is just the error term of the model.

Finally we use prophet for trading as follows:

1) Use the algorithm to fit a model on the data
2) Use the model to forecast minute to minute bitcoin prices of future dates.
3) Calculate delta(t) of a particular time, which is the difference in the predicted prices of time step t+1 and t
    a) If delta(t) > 0, it indicates that the price will be rising over the next step, and hence we buy
    b) If delta(t) < 0, it indicates the opposite and hence we will sell

LSTM Algorithm

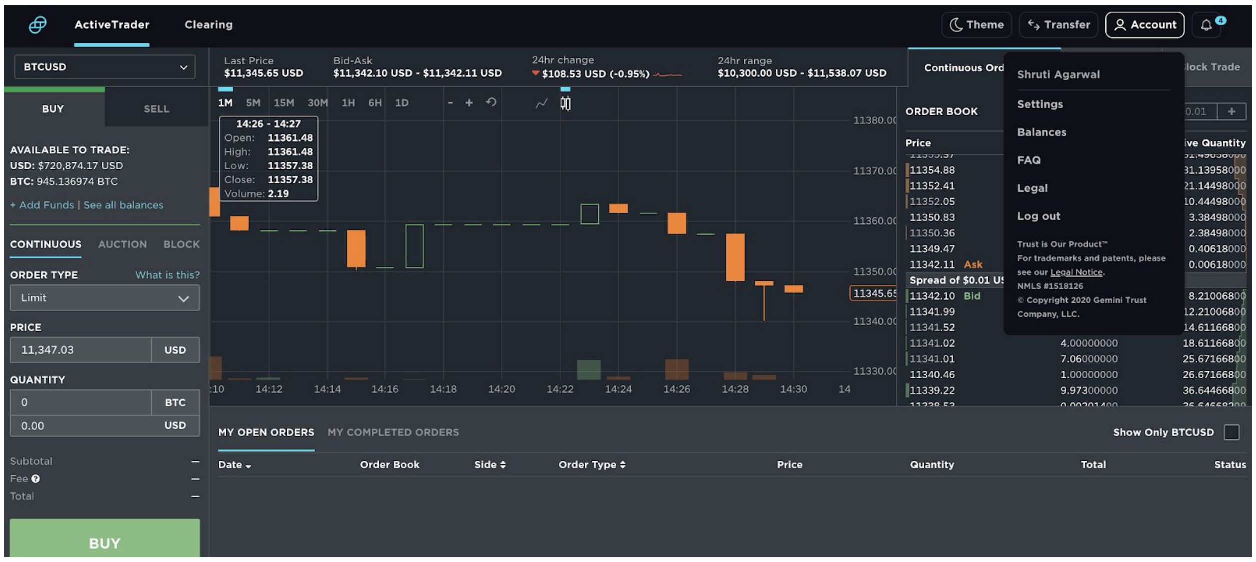
Long short-term memory(LSTM) is a special kind of recurrent neural network. An LSTM module has a cell state and three gates which provides them with the power to selectively learn, unlearn or retain information from each of the units. The cell state in LSTM helps the information to flow through the units without being altered by allowing only a few linear interactions.
In this project, we can use LSTM as an alternative to Prophet. Both of them are capable of learning long term dependencies in data. The LSTM prediction is based on a set of last values, we are therefore less prone to variance due to seasonality and already consider the current trend. In contrast to that, the prophet model does a good job modeling as an additive system and finding out and displaying seasonalities.

Putting it all Together

The code for this is in the tradebot folder in the GitHub link. We have worked with both Jupyter notebooks and Colab. Using colab, the need for docker has been eliminated and collaboration was much easier. However we still worked with Docker and Jupyter notebook for the Kibana visualization part.

Putting all the Pipeline components together involves:
1. Streaming live bid/ask data from Gemini
2. Piping the bid/ask data to Kafka
3. Processing data with Spark Streaming (which we have replace with **Faust** due to integration issues)
4. Piping the processed data back to Kafka
5. Applying the Prophet (LSTM, too) forecasting model and executing trades on Gemini
6. Piping the results into Elasticsearch
7. Visualizing the pipeline in Kibana

We had to overcome a myriad of challenges in putting it all together.
Firstly, since we will not be making actual trades, we made a Gemini SandBox account which creates a pseudo trading account with 1,00,000.00 USD balance, 1,000.00 BTC balance, 20,000.00 ETH balance, 20,000.00 BCH balance, 20,000.00 LTC balance and 20,000.00 ZEC to make trade and score them.



Last Price $11,345.65 USD
Bid-Ask $11,342.10 USD - $11,342.11 USD
24hr change $108.53 USD (-0.95%)
24hr range $10,300.00 USD - $11,538.07 USD

AVAILABLE TO TRADE:
USD: $720,874.17 USD
BTC: 945.136974 BTC

ORDER TYPE: Limit
PRICE: 11,347.03 USD
QUANTITY: 0 BTC, 0.00 USD

14:26 - 14:27
Open: 11361.48
High: 11361.48
Low: 11357.38
Close: 11357.38
Volume: 2.19

Date	Order Book	Side	Order Type	Price	Quantity	Total	Status

Gemini Sandbox Account

As we trade only with BTC and USD, only those values get modified as we trade.

In the next stage when Spark tried to consume from Kafka, there were some logistic issues. To solve these, we replaced Spark with Faust.

Gemini to Kafka

From the coding perspective, we first create a Gemini WebSocket which streams live trading data from Gemini and writes it to the Kafka topic ‘gemini-feed’.

```
def on_message(ws, message):
    message = json.loads(message)
    if message['type'] == 'update':
        for i in message['events']:
            if 'side' in i:
                payload = {'side': i['side'], 'price': i['price'], 'remaining':
i['remaining']}
                sent = producer.send(topic='gemini-feed', 'utf-8'),
value=bytes(json.dumps(payload), 'utf-8'))
        producer.flush()
```

```
ws = websocket.WebSocketApp("wss://api.gemini.com/v1/marketdata/BTCUSD",
                            on_message=on_message)
```

The stream looks like this:

Streaming output truncated to the last 5000 lines.

```
{"side": "bid", "price": "11412.68", "remaining": "0.54571124"}
{"side": "ask", "price": "11421.64", "remaining": "4"}
{"side": "bid", "price": "11380.97", "remaining": "5.1518"}
{"side": "bid", "price": "11397.86", "remaining": "0"}
{"side": "ask", "price": "11424.37", "remaining": "0"}
{"side": "ask", "price": "11421.14", "remaining": "4.9846"}
{"side": "bid", "price": "11328.67", "remaining": "22.67"}
{"side": "bid", "price": "11414.37", "remaining": "2"}
```

Faust Stream Processing

The ‘gemini-feed’ topic data is consumed by faust which performs volatility calculations on the streaming data and sends the output to ‘faust.out’ topic in Kafka.
Faust enables us to define an agent which will process our streams. Following is the snipped for the agent we developed for this project:

```
app = faust.App('gemini', broker='kafka://localhost:9092')
topic = app.topic('gemini-feed', value_type=bytes)
@app.agent(topic)
async def gemini_feed_process(stream):
      async for messages in stream.take(1e6, within=60):
      parsed_pv = list(map(lambda x: price_volume(x), messages)) #map object:
convert to list using list()
      grouped_sorted = list(accumulate(sorted(parsed_pv)))
      for rec in grouped_sorted:
            handler(rec)
      bid_to_ask_ratio = ("price_volume",
grouped_sorted[1][1]/grouped_sorted[0][1])
      handler(bid_to_ask_ratio)
```

1. Here stream.take(, within=60) processes the data of 60 seconds in each iteration.
2. The processing involves following tasks:
    a. Group the data by bid and ask type and take the sum of price volume for each group which will result in 2 tuples: (bid, sum_of_bid_price_volume) and (ask, sum_of_ask_price_volume). We then send these tuples to ‘faust.out’ topic,

with type as bid/ask and values as their sum_price_volume and timestamp (current time.)

   b. We also calculate the ratio of sum_of_bid_price_volume and sum_of_ask_price_volume and store it in kafka as well labelling the type as price_volume.

3. For every 60 seconds the faust.out will be populated with 3 entries:

After that we run the faust worker by calling this module from terminal:

```
$ faust -A gemini -l info worker --web-port=6066
```

Our 'faust.out' stream looks like this:

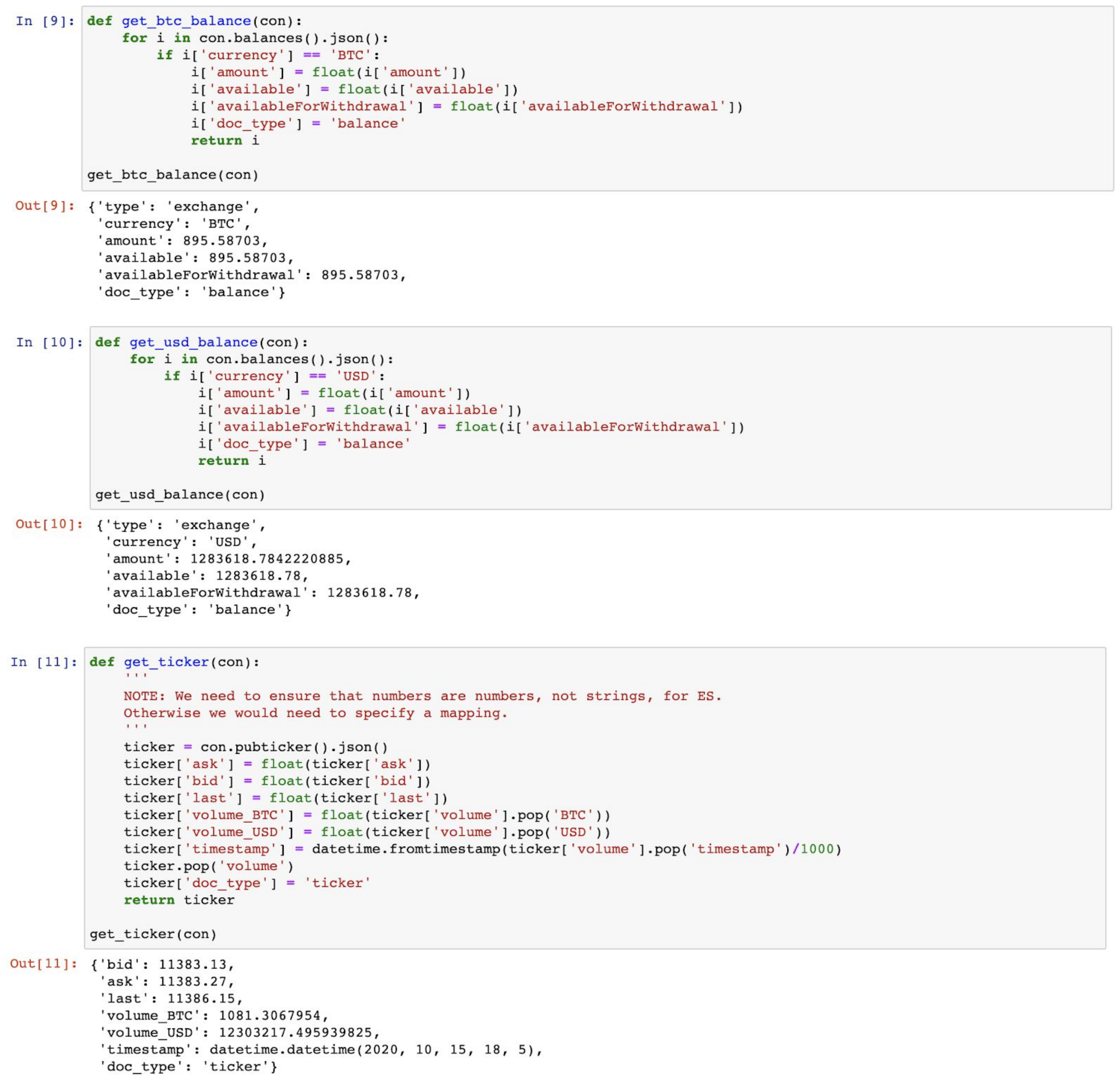
```
{"type": "ask", "value": 83751098.75963986, "timestamp": 1602770086}
{"type": "bid", "value": 36606864.31078205, "timestamp": 1602770086}
{"type": "price_volume", "value": 0.43709115286763395, "timestamp": 1602770086}
{"type": "ask", "value": 6938006.9343151, "timestamp": 1602770146}
{"type": "bid", "value": 11105785.22598726, "timestamp": 1602770146}
{"type": "price_volume", "value": 1.60071693947991, "timestamp": 1602770146}
{"type": "ask", "value": 9853402.307537861, "timestamp": 1602770206}
{"type": "bid", "value": 7883640.705184912, "timestamp": 1602770206}
{"type": "price_volume", "value": 0.8000932529826698, "timestamp": 1602770206}
{"type": "ask", "value": 10834873.597073978, "timestamp": 1602770266}
{"type": "bid", "value": 11022729.357473651, "timestamp": 1602770266}
{"type": "price_volume", "value": 1.0173380666342433, "timestamp": 1602770266}
```

Gemini Trading

Gemini trading involves importing our forecast model for making buy/sell decisions. We then create a consumer app for Kafka topic 'faust.out' to get volatility data. This is followed by initializing a connection to Elasticsearch which will be used for keeping the track of our trades and Kibana visualization. We then connect to Gemini Sandbox to use various functionalities which let us make pseudo-trades easily. We also specify a threshold range for market liquidity in which we will make trades. We also set the percentage of our Gemini account balance that we put to risk when we trade.
Next we have defined a few functions that help in smooth interaction with Gemini.

We have defined functions to get our account balance(both Bitcoin and US Dollars), to get the latest price and volume data for the exchange, to create an order and to format an order so that we can ship it to the exchange in the expected format.

Following are snippets of the code and their outputs showing functionalities of Gemini API:



```
In [9]: def get_btc_balance(con):
            for i in con.balances().json():
                if i['currency'] == 'BTC':
                    i['amount'] = float(i['amount'])
                    i['available'] = float(i['available'])
                    i['availableForWithdrawal'] = float(i['availableForWithdrawal'])
                    i['doc_type'] = 'balance'
                    return i

        get_btc_balance(con)

Out[9]: {'type': 'exchange',
         'currency': 'BTC',
         'amount': 895.58703,
         'available': 895.58703,
         'availableForWithdrawal': 895.58703,
         'doc_type': 'balance'}
```

```
In [10]: def get_usd_balance(con):
             for i in con.balances().json():
                 if i['currency'] == 'USD':
                     i['amount'] = float(i['amount'])
                     i['available'] = float(i['available'])
                     i['availableForWithdrawal'] = float(i['availableForWithdrawal'])
                     i['doc_type'] = 'balance'
                     return i

         get_usd_balance(con)

Out[10]: {'type': 'exchange',
          'currency': 'USD',
          'amount': 1283618.7842220885,
          'available': 1283618.78,
          'availableForWithdrawal': 1283618.78,
          'doc_type': 'balance'}
```

```
In [11]: def get_ticker(con):
             '''
             NOTE: We need to ensure that numbers are numbers, not strings, for ES.
             Otherwise we would need to specify a mapping.
             '''
             ticker = con.pubticker().json()
             ticker['ask'] = float(ticker['ask'])
             ticker['bid'] = float(ticker['bid'])
             ticker['last'] = float(ticker['last'])
             ticker['volume_BTC'] = float(ticker['volume'].pop('BTC'))
             ticker['volume_USD'] = float(ticker['volume'].pop('USD'))
             ticker['timestamp'] = datetime.fromtimestamp(ticker['volume'].pop('timestamp')/1000)
             ticker.pop('volume')
             ticker['doc_type'] = 'ticker'
             return ticker

         get_ticker(con)

Out[11]: {'bid': 11383.13,
          'ask': 11383.27,
          'last': 11386.15,
          'volume_BTC': 1081.3067954,
          'volume_USD': 12303217.495939825,
          'timestamp': datetime.datetime(2020, 10, 15, 18, 5),
          'doc_type': 'ticker'}
```

Now, let us talk about the actual trading. We consume data which is the output of faust with volatility calculations. We then see if our model predicts this is the time (timestamp included in output of faust) to buy or sell using `lookup_side(fcst)` function. Also, we see if the previous forecast was buy or sell to decide if we want to trade.

Selling: Check if BitCoin balance is greater than zero. If yes, sell. At the same point of time, score your last trade using the current price.

Buying: If we have already executed the trade, do nothing. If not, check liquidity calculated by faust. If it does not fall within the threshold, do nothing. If it does, execute trade.

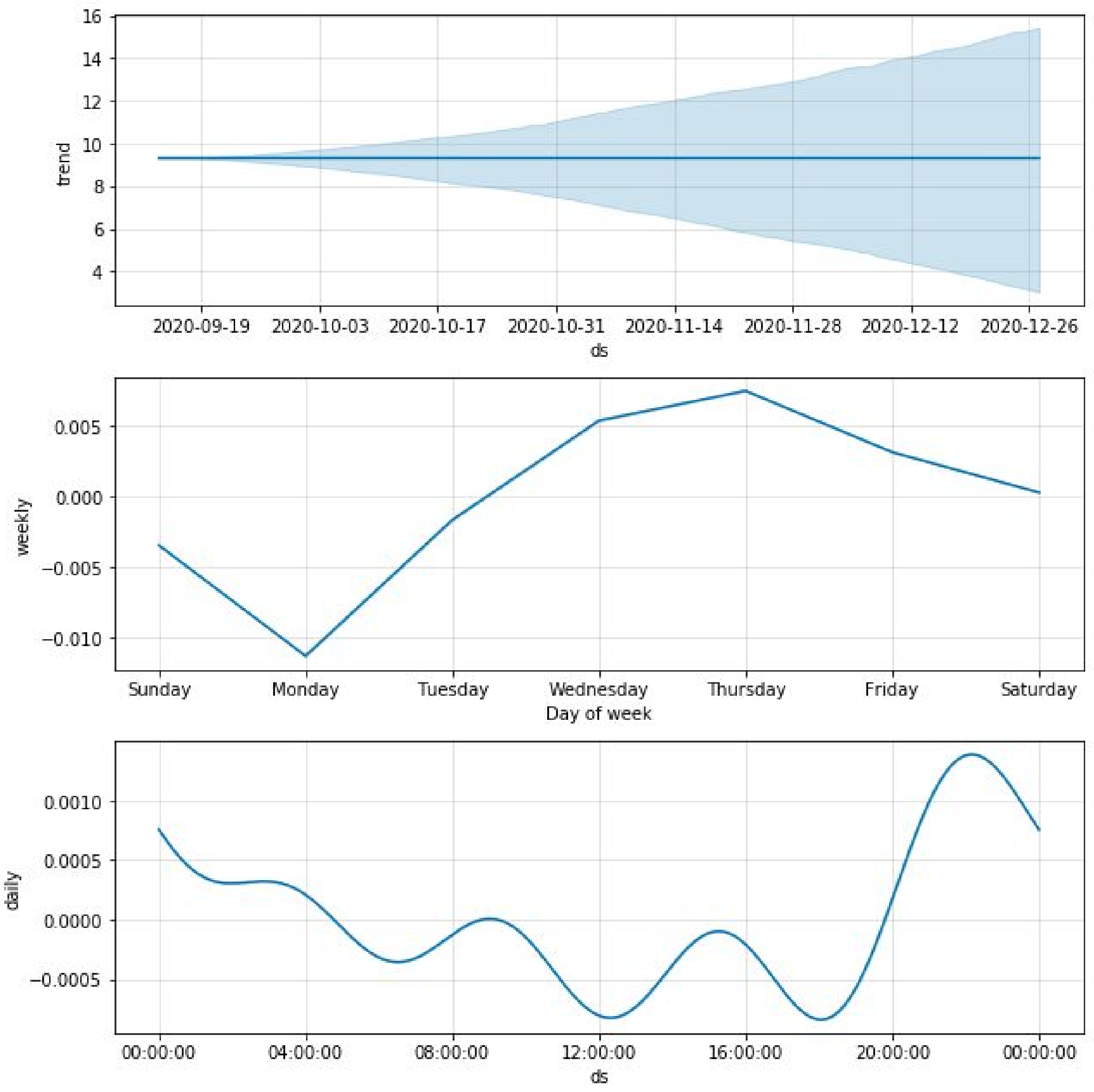
Before running the code for this, we must have the forecasts from LSTM and Prophet with us. They can be generated by using the code from the /tradebot/forecast folder in the GitHub link provided.
A trade order looks like this:

```
{
    "avg_execution_price": 11408.59,
    "doc_type": "order",
    "exchange": "gemini",
    "executed_amount": 0.395168,
    "id": "659834884",
    "is_cancelled": true,
    "is_hidden": false,
    "is_live": false,
    "options": [
        "immediate-or-cancel"
    ],
    "order_id": "659834884",
    "original_amount": 56.0432,
    "price": 11408.59,
    "reason": "ImmediateOrCancelWouldPost",
    "remaining_amount": 55.648032,
    "side": "buy",
    "symbol": "btcusd",
    "timestampms": 1602770477395,
    "type": "exchange limit",
    "was_forced": false
}
```

Challenges In The Forecasting Models

The data which we obtained from kaggle had minute-by-minute prices entries spanning from the year 2012 to 2020. This equates to more than 42L data points. Using such large amounts of data to fit our prophet model was simply infeasible due to the large time required. The approximate time required to fit the model was in the order of several hours, and hence was simply out of our computational scope. In order to simplify this process, we use only data from the year 2020. This gives us around 4L data points and helps us fit our model in under an hour.
On top of this, given data contained a large number of Nan values. Some amounts of preprocessing was required to drop the corresponding Nan rows in the data. This was not an issue in Prophet as it automatically had functionality to avoid this, but for LSTM, these Nan inputs were causing loss to blow up and hence no training occurs.

The results provided by the Prophet algorithm are shown in the below diagrams:
The Diagrams show the overall trend predicted by the Prophet algorithm as well as the Daily and Weekly predictions.



We have used the following description for our LSTM Model.

```
s = 200
model = models.Sequential()
model.add( layers.LSTM(input_shape=(1,s), units=50,
activation='relu', 4 4 return_sequences=False) )
```

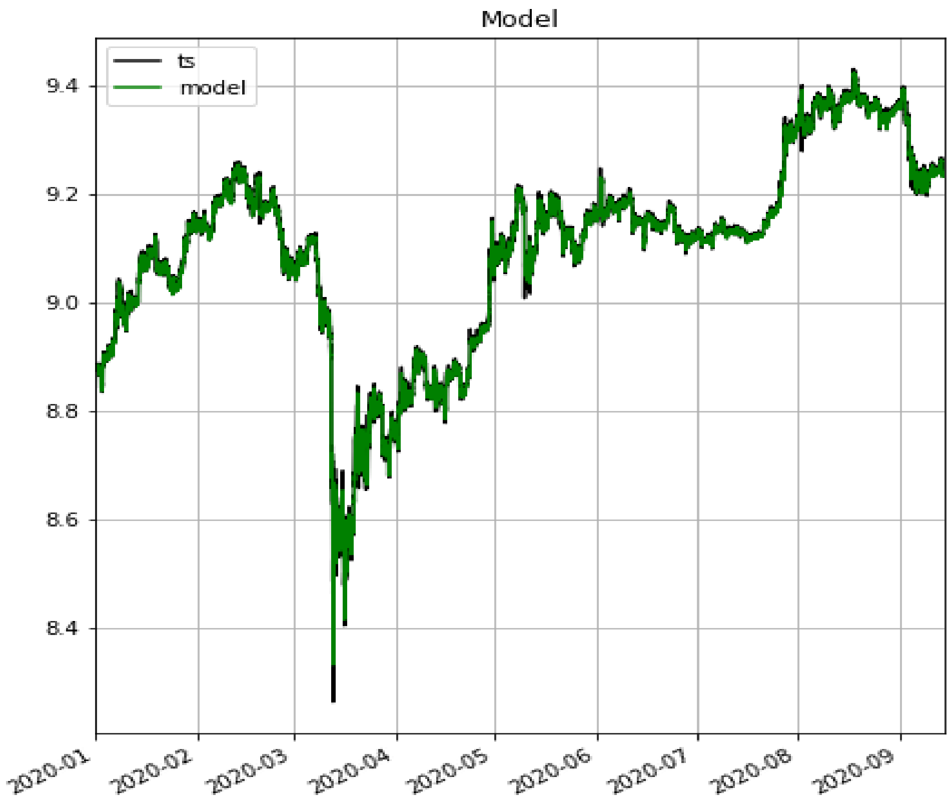
```
model.add( layers.Dense(1) )
model.compile(optimizer='adam', loss='mean_absolute_error')
model.summary()
```

Output for the above line of codes

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 50)                50200
_________________________________________________________________
dense (Dense)                (None, 1)                 51
=================================================================
Total params: 50,251
Trainable params: 50,251
Non-trainable params: 0
```

Training of LSTM came with its own set of difficulties. In the model definition above we have used a lookback of 200 and a fully connected layer of size 50x1. The model parameters for such a basic model are already more than 50k. The interpretation of the lookback is that it takes in the last 200 datapoint values as input and aims to predict the next one. Since we have minute to minute data, this amounts to taking in the last 200 minutes data, or the last 3 hours data and predicting the next one. This is clearly not what we need as our model should learn long range dependencies including weekly and yearly cycles. That means our model should have a lookback consisting of at least a year, which is equivalent to 365*24*60 minutes = 525,600. Such a lookback would give us a model complexity of **105,130,251** params. This is an astronomical figure and is just not feasible given our computational scope.

Despite the above difficulties, we provide a notebook in which we train the basic LSTM model as defined above and plot the fitted model vs true values graph :


Model
ts
model
9.4
9.2
9.0
8.8
8.6
8.4
2020-01
2020-02
2020-03
2020-04
2020-05
2020-06
2020-07
2020-08
2020-09