Informatics Institute of Technology

In Collaboration With

The University of Westminster, UK

The University of Westminster, Coat of Arms

Surpassing Time Series Forecasting Limitations using Liquid Time-stochasticity Networks

An Implementation by

Mr. Ammar Raneez

W1761196 | 2019163

Supervised by

Mr. Torin Wirasingha

April 2023

This is submitted in partial fulfilment of the requirements for the

BSc (Hons) Computer Science degree at

the University of Westminster.

Contents

List of Tables

List of Figures

List of Abbreviations

AI	Artificial Intelligence.
API	Application Programming Interface.
AD	Automatic Differentiation.
ARIMA	Autoregressive Integrated Moving Average.
BPTT	Back-Propagation Through Time.
BTC	Bitcoin.
CT-GRU	Continuous-time Gated Recurrent Unit.
CT-RNN	Continuous-time Recurrent Neural Network.
DL	Deep Learning.
GPU	Graphics Processing Unit.
LSTM	Long Short-Term Memory.
LTC	Liquid Time-constant.
ML	Machine Learning.
(s)MAPE	Symmetric Mean Absolute Product Error.
MASE	Mean Absolute Scaled Error.
MSE	Mean Squared Error.
N-BEATS	Neural Basis Expansion Analysis for interpretable Time Series
NLP	Natural Language Processing.
ODE	Ordinary Differential Equations.
POC	Proof-Of-Concept.
REST	Representational State Transfer.
RMSE	Root Mean Squared Error.
RNN	Recurrent Neural Network.
SDE	Stochastic Differential Equation.
SGD	Stochastic Gradient Descent.
TS	Time Series.
UI	User Interface.

1. CHAPTER OVERVIEW

In this chapter, the author describes the core implementation of the system and the necessary decisions taken to approach that implementation. Moreover, the chosen tools, languages, and technologies are presented alongside their reasoning.

2. TECHNOLOGY SELECTION

2.1. Technology stack

The chosen technologies are depicted in the diagram below.

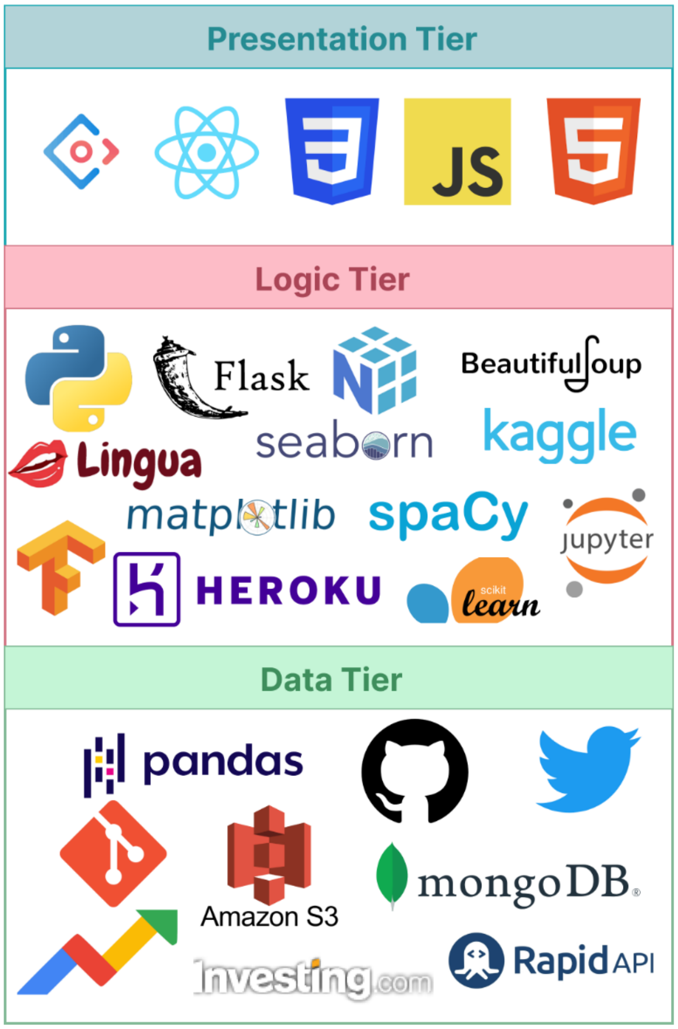


Figure 1: Tech stack (*Self-Composed*)

2.2. Data selection

As this is a data science project, the data quality necessary. The author utilized multiple sources of data that are potential contributions to the target inference; the following were required:

- BTC historical data
- BTC block reward size
- BTC tweets
- BTC Twitter volume
- BTC Google Trends

The multivariate single-horizon forecasting model combined the above data, while the univariate multi-horizon forecasting model solely used the historical data. The below table describes the sources of each respective dataset.

Table 1: Dataset sources (*Self-Composed*)

Dataset	**Source**
BTC historical data	From a third-party investing.com API.
BTC block reward size, BTC Twitter volume, BTC Google Trends	From a public dashboard that provides multiple different information about a specific cryptocurrency.
BTC tweets	Tweets from 2014-2019 were downloaded from Kaggle – the remaining dates were extracted from a Twitter tweet scraper.

Gathering the data was long and arduous as it was not as simple as downloading available datasets, and specific APIs being rate-limited. Dedicated python scripts were written to extract the data, store it into a MongoDB database, and to streamline updating data. The author will publicize these scripts and the data to facilitate future research.

2.3. Selection of programming language

Programming languages were analyzed before development. Specifically for three main aspects: the client, the data science component, and the API communicating between the model and the client.

The below table summarizes the analysis for the language chosen for the data science component, where each option was given a score within H - High, M – Medium, and L – Low.

Table 2: Selection of data science language (*Self-Composed*)

Data science Two of the most popular languages used widely for data science were analyzed to implement the core data science components.			
Aspect	**Relevance**	**Python**	**R**
Availability of libraries.	A language supporting multiple libraries is paramount, as the author would require numerous techniques to gather the necessary data and streamline the model and algorithm development.	H	M
Author familiarity and ease of implementation.	Implementing the algorithm, the mathematical intricacies, and the respective model should be as simple as possible. It is an additional benefit if the author has hands-on experience with the chosen language,	H	M
Learning curve	The difficulty of the chosen language must not be a hindrance as the goal is to utilize the tool to implement a system rather than spending time learning the language.	L	M
Community and documentation.	Community support and well-written documentation is paramount as the author will not have time to debug trivial issues.	H	M
Conclusion Based on the analysis, the author decided to use **Python**, as it was more relevant.			

To develop the user interface, not much competition is present to analyze. **JavaScript** is the stand-alone leader and is the author's choice, as it is dynamic and can handle user interactions seamlessly. Although recent technology has presented the usage of C# for frontend development, high latency issues and lack of community knowledge are a downfall.

APIs are required to set up the communication between the model and the user interface. Multiple technologies are available for API development. The author chose **Python** as their core data science component is also built using Python; therefore, utilizing the same language would reduce the time taken to learn new languages for insignificant reasons.

2.4. Selection of development framework

2.4.1 DL framework

The author chose Python for developing the core data science component. As the core algorithm and model will be DL-based, DL frameworks must be meticulously analyzed to select the most relevant framework. The two most popular frameworks, TensorFlow and PyTorch, were analyzed.

Table 3: Selection of DL framework (*Self-Composed*)

Framework	**Description**
TensorFlow	Used for production-level applications, has detailed documentation and community support, and handles large datasets. It also provides better visualization options, making it easy to debug and monitor training, which is vital as a novel algorithm is being built, and no comparison is present.
PyTorch	It is more lightweight and developer-friendly, as it provides a higher-level development. Therefore, it has a much smaller learning curve, easier to get started, and feels more intuitive as it is simpler to build models.
Conclusion The author opted to use **TensorFlow**. Although it is more complicated, the higher-level API: Keras, is now officially a part of TensorFlow. Therefore, model development has become much more straightforward. Additionally, building the algorithm requires more low-level details. (PyTorch vs. TensorFlow: 2022 Deep Learning Comparison | Built In, 2022)	

2.4.2. UI framework

As JavaScript was chosen for developing the UI, respective JavaScript frontend frameworks and libraries must be analyzed. There is an ocean of JavaScript libraries - the top four were selected for evaluation: Angular, Vue, Svelte, and React.

Table 4: Selection of UI framework (*Self-Composed*)

Framework	**Description**
Angular	Suitable for large-scale applications with dedicated submodules for particular functionalities. However, it can be less performant in comparison and unnecessarily heavy.
Vue	A tiny framework that takes little to no time to startup and is much more intuitive as the code is simple. Additionally, based on simulations, it has been identified to perform better than Angular and React. However, it has much fewer resources.
Svelte	The most lightweight and genuinely reactive. Much more performant than the rest; however, it has a small community of developers and is relatively new.
React	Customizable and promotes code reusability via functions as components. It carries a large community and is open-source while being SEO-friendly. Additionally, the React developer tools is very handy.
Conclusion Based on the analysis, the author chose **React** as the GUI built will be simple, and there is no requirement for large-scale applications, as it is not the primary focus. (Angular vs React | Angular vs Vue | React vs Vue - Know the Difference, 2021)	

2.4.3. API web framework

As python was chosen for the API development, respective Python web frameworks must be analyzed to select the more relevant one. Analysis was conducted between Django and Flask, as they are the two most popular frameworks.

Table 5: Selection of web framework (*Self-Composed*)

Framework	**Description**
Flask	A very lightweight framework that provides only the simplest of functionalities. However, it is the preferred choice for ML API development because it is light.

Django	Suitable for more larger scaled applications that provide a vast range of functionalities, it is stricter and less flexible. Therefore, is much more demanding and heavier.
Conclusion The author chose **Flask** as it provides only the necessities in exposing an ML model and since the luxury features provided by Django (ex: authentication) were not required. (Flask Vs Django: Which Python Framework to Choose?, 2021)	

2.5. Other libraries & tools

Table 6: Chosen libraries (*Self-Composed*)

Library	**Justification**
NumPy	Facilitates mathematical functions and calculations that are immensely required when building the algorithm.
Pandas	To create dataframes to perform analysis, cleaning, transformations, filtration, etc., on the datasets.
Scikit-learn	To create data splits and feature scaling.
Lingua	To detect the language of the tweets. As this project is limited to using only English tweets, they must first be identified.
SpacCy	To perform NER to extract entities that could be within the pre-defined impactful index.
Matplotlib + Seaborn	For analysis, visualizations, and dashboarding.
Beautiful Soup	For scraping the block reward size and the Twitter volume from the public dashboard.
VADER	Perform sentiment analysis on the tweets.
Redux	For API requests from the client.
Ant design	Makes creating appealing user interfaces hassle-free.
MongoDB	To store the datasets and retrieve/update whenever necessary.
AWS S3	Store the models in use.
Heroku	To host the Flask API.

2.6. Integrated Development Environment (IDE)

Table 7: Chosen IDEs (*Self-Composed*)

IDE	**Justification**
Kaggle	Consists of 32GB of RAM; therefore, all datasets can be loaded and processed at once without needing to process sections of data at a time. Additionally, it provides easy integration with existing Kaggle datasets and user-uploaded datasets.
Jupyter	For local trials, testing, and model training.
VSCode	Lightweight and extremely powerful. It consists of multiple shortcuts, extensions, and snippets that can significantly boost development productivity.

2.7. Summary of chosen tools & technologies

Table 8: Chosen tools & technologies (*Self-Composed*)

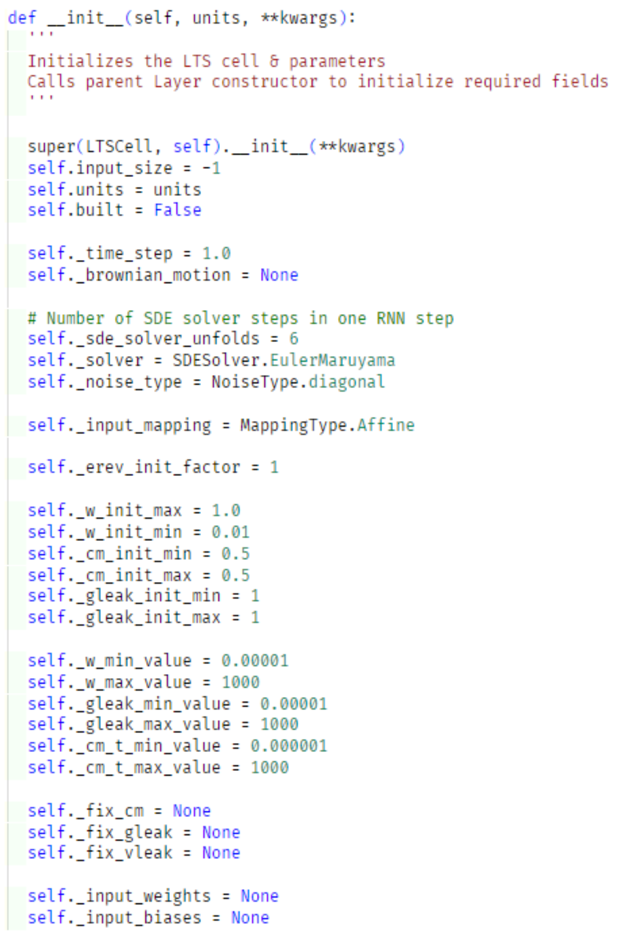
Component	**Tools**
Programming languages	Python, JavaScript
Development framework	Flask, TensorFlow
UI development framework	React
Libraries & tools	Ant design, NumPy, Pandas, Scikit-learn, Beautiful Soup, Lingua, Matplotlib, Seaborn, VADER sentiment analyzer, Redux, Ant design, MongoDB, Heroku
IDEs	Kaggle and Jupyter notebooks; VSCode.
Version control	Git + GitHub

3. IMPLEMENTATION OF CORE FUNCTIONALITIES

The novel algorithm, the scripts to fetch the required data, and the preprocessing performed can be considered the core functionalities of the project.

3.1. Algorithm implementation

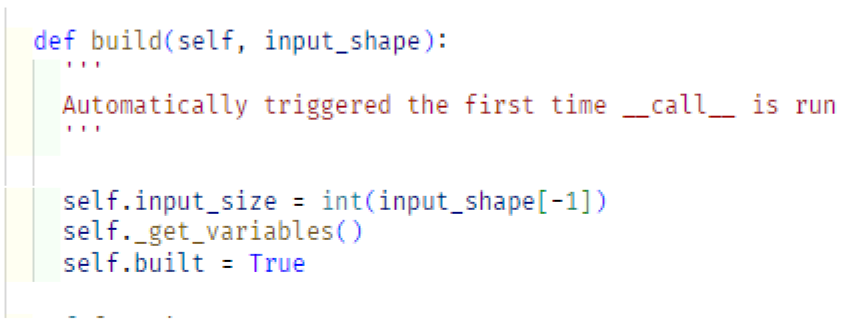
The author initially implemented the LTC architecture since there is no modern reference utilizing recommended best practices and approaches. The author then built on this architecture, replacing the underlying ODEs with SDEs.



```python
def __init__(self, units, **kwargs):
    '''
    Initializes the LTS cell & parameters
    Calls parent Layer constructor to initialize required fields
    '''

    super(LTSCell, self).__init__(**kwargs)
    self.input_size = -1
    self.units = units
    self.built = False

    self._time_step = 1.0
    self._brownian_motion = None

    # Number of SDE solver steps in one RNN step
    self._sde_solver_unfolds = 6
    self._solver = SDESolver.EulerMaruyama
    self._noise_type = NoiseType.diagonal

    self._input_mapping = MappingType.Affine

    self._erev_init_factor = 1

    self._w_init_max = 1.0
    self._w_init_min = 0.01
    self._cm_init_min = 0.5
    self._cm_init_max = 0.5
    self._gleak_init_min = 1
    self._gleak_init_max = 1

    self._w_min_value = 0.00001
    self._w_max_value = 1000
    self._gleak_min_value = 0.00001
    self._gleak_max_value = 1000
    self._cm_t_min_value = 0.000001
    self._cm_t_max_value = 1000

    self._fix_cm = None
    self._fix_gleak = None
    self._fix_vleak = None

    self._input_weights = None
    self._input_biases = None
```

Figure 2: Initialize algorithm (*Self-Composed*)

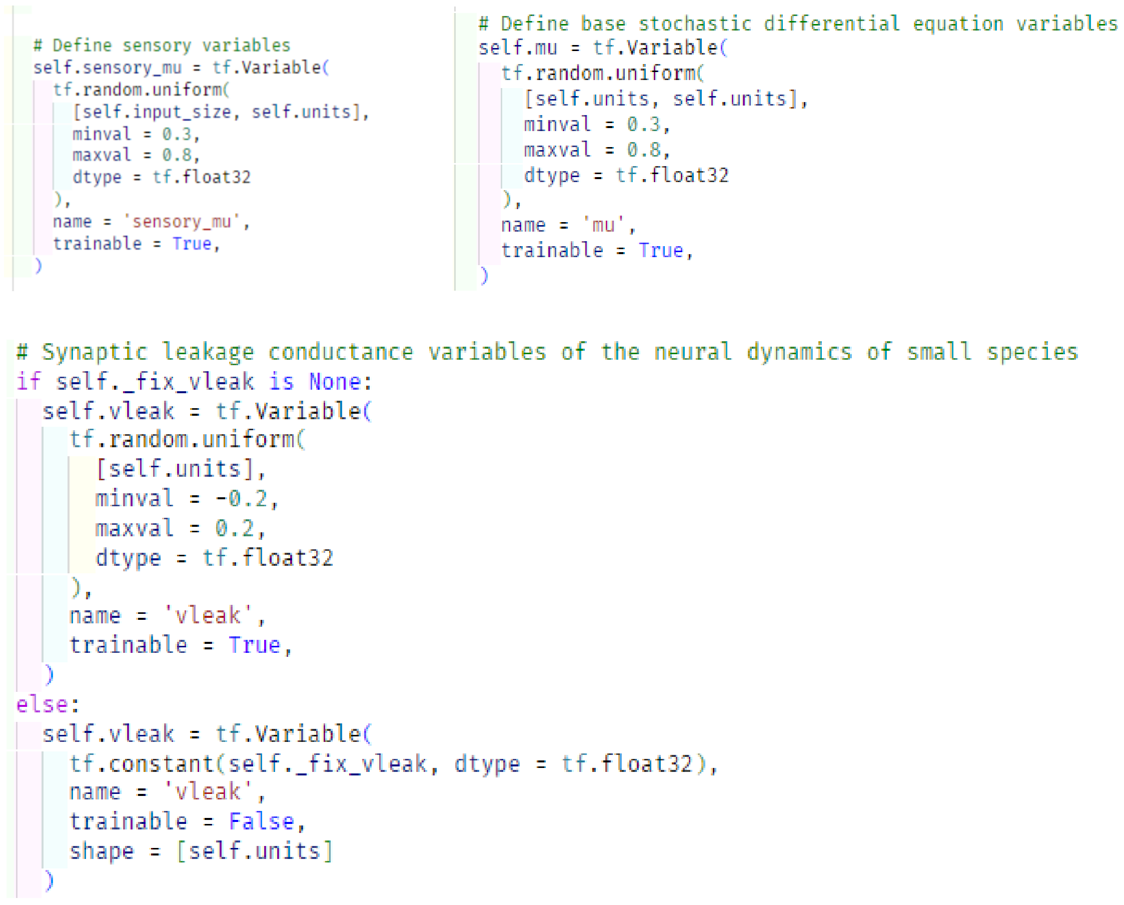
The above code snippet initializes the algorithm cell with the necessary variable maximum and minimum values. In the above method, the built model can perform input-independent initializations. By inheriting from the base Keras Layer class, the ability to be used in the higher level of the model’s layer definition is obtained (as existing LSTM and RNN cells).



```
def build(self, input_shape):
    '''
    Automatically triggered the first time __call__ is run
    '''

    self.input_size = int(input_shape[-1])
    self._get_variables()
    self.built = True
```

Figure 3: Build algorithm (*Self-Composed*)

The above snippet defines what occurs upon initialization; in other words, it "builds" the algorithm cell. A helper function is utilized here that defines the variables (sigma, mu, weights, and leakage conductance variables (Hasani et al., 2020)). The input shape is available within the above function; therefore, the model can initialize the variables used here. The below snippet demonstrates how some of these variables are initialized.



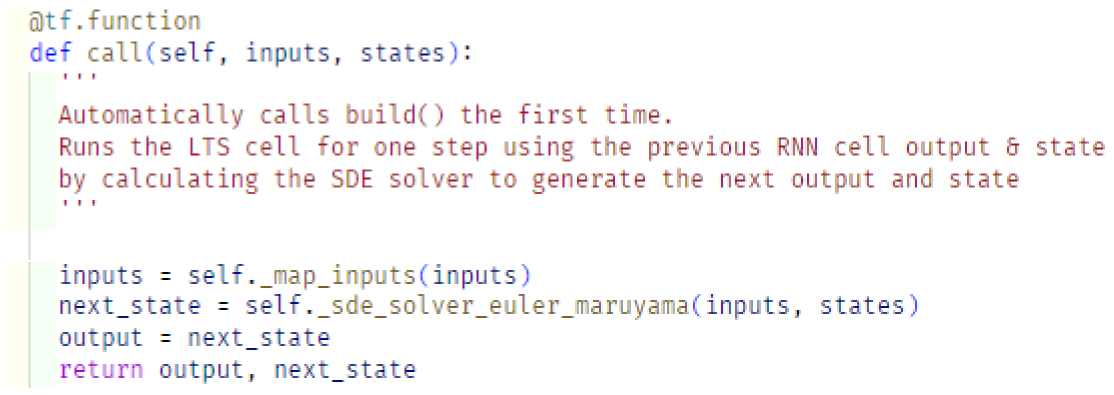
```
# Define sensory variables
self.sensory_mu = tf.Variable(
  tf.random.uniform(
    [self.input_size, self.units],
    minval = 0.3,
    maxval = 0.8,
    dtype = tf.float32
  ),
  name = 'sensory_mu',
  trainable = True,
)
```

```
# Define base stochastic differential equation variables
self.mu = tf.Variable(
  tf.random.uniform(
    [self.units, self.units],
    minval = 0.3,
    maxval = 0.8,
    dtype = tf.float32
  ),
  name = 'mu',
  trainable = True,
)
```

```
# Synaptic leakage conductance variables of the neural dynamics of small species
if self._fix_vleak is None:
  self.vleak = tf.Variable(
    tf.random.uniform(
      [self.units],
      minval = -0.2,
      maxval = 0.2,
      dtype = tf.float32
    ),
    name = 'vleak',
    trainable = True,
  )
else:
  self.vleak = tf.Variable(
    tf.constant(self._fix_vleak, dtype = tf.float32),
    name = 'vleak',
    trainable = False,
    shape = [self.units]
  )
```

Figure 4: Algorithm - sensory, stochastic and leakage variables (*Self-Composed*)

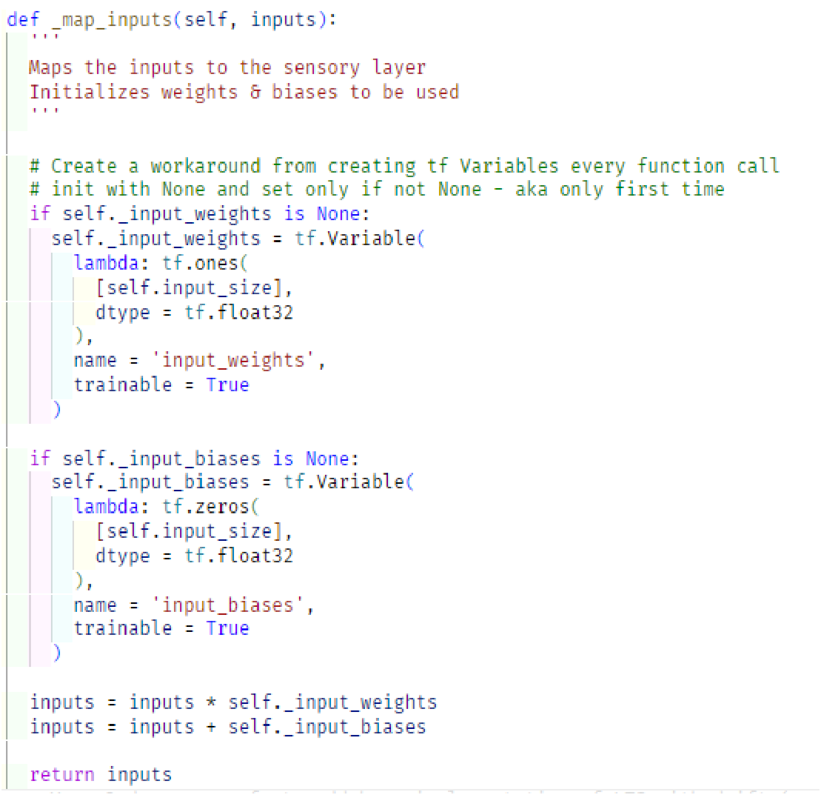
The final step is the forward computation process that will occur on each epoch, in other words, the forward propagation process.



```
@tf.function
def call(self, inputs, states):
    '''
    Automatically calls build() the first time.
    Runs the LTS cell for one step using the previous RNN cell output & state
    by calculating the SDE solver to generate the next output and state
    '''

    inputs = self._map_inputs(inputs)
    next_state = self._sde_solver_euler_maruyama(inputs, states)
    output = next_state
    return output, next_state
```

Figure 5: Algorithm - forward propagation (*Self-Composed*)

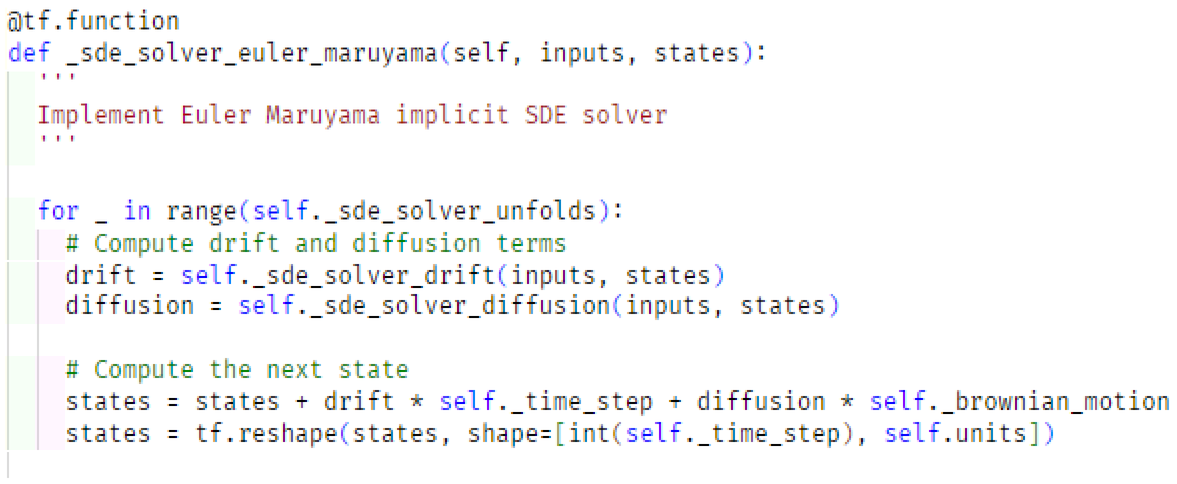
The above function is run automatically on each epoch. Initially, a helper function defines the weights and biases of the network, as demonstrated below.



```
def _map_inputs(self, inputs):
    '''
    Maps the inputs to the sensory layer
    Initializes weights & biases to be used
    '''

    # Create a workaround from creating tf Variables every function call
    # init with None and set only if not None - aka only first time
    if self._input_weights is None:
        self._input_weights = tf.Variable(
            lambda: tf.ones(
                [self.input_size],
                dtype = tf.float32
            ),
            name = 'input_weights',
            trainable = True
        )

    if self._input_biases is None:
        self._input_biases = tf.Variable(
            lambda: tf.zeros(
                [self.input_size],
                dtype = tf.float32
            ),
            name = 'input_biases',
            trainable = True
        )

    inputs = inputs * self._input_weights
    inputs = inputs + self._input_biases

    return inputs
```

Figure 6: Algorithm - define weights and biases (*Self-Composed*)

As determined in previous chapters, the optimal way of performing the forward computation of SDEs is to use the Euler-Maruyama method. The below code snippet is an implementation of the Euler-Maruyama SDE solver used by the author utilizing Brownian motion as the noise, as demonstrated by Duvenaud (2021).



```
@tf.function
def _sde_solver_euler_maruyama(self, inputs, states):
    '''
    Implement Euler Maruyama implicit SDE solver
    '''

    for _ in range(self._sde_solver_unfolds):
        # Compute drift and diffusion terms
        drift = self._sde_solver_drift(inputs, states)
        diffusion = self._sde_solver_diffusion(inputs, states)

        # Compute the next state
        states = states + drift * self._time_step + diffusion * self._brownian_motion
        states = tf.reshape(states, shape=[int(self._time_step), self.units])
```

Figure 7: Algorithm - Euler-Maruyama SDE solver (*Self-Composed*)

3.2. Data fetchers

The data fetchers are scripts that are used to extract the data to be used by the model. The scripts are placed under **APPENDIX I**.

3.3. Preprocessing

Preprocessing steps are required to prepare the data fetched from the data fetchers before being used by the model. The preprocessing scripts are also placed under **APPENDIX I**.

4. CHAPTER SUMMARY

This chapter focused on defining the technologies and tools that facilitate the software development that would demonstrate the research. Additionally, the implementation of the core features is demonstrated with accompanying code snippets.

REFERENCES

Flask Vs Django: Which Python Framework to Choose? (2021). *InterviewBit*. Available from https://www.interviewbit.com/blog/flask-vs-django/ [Accessed 12 December 2022].

Angular vs React | Angular vs Vue | React vs Vue - Know the Difference. (2021). *Radixweb*. Available from https://radixweb.com/blog/angular-vs-react-vs-vue [Accessed 12 December 2022].

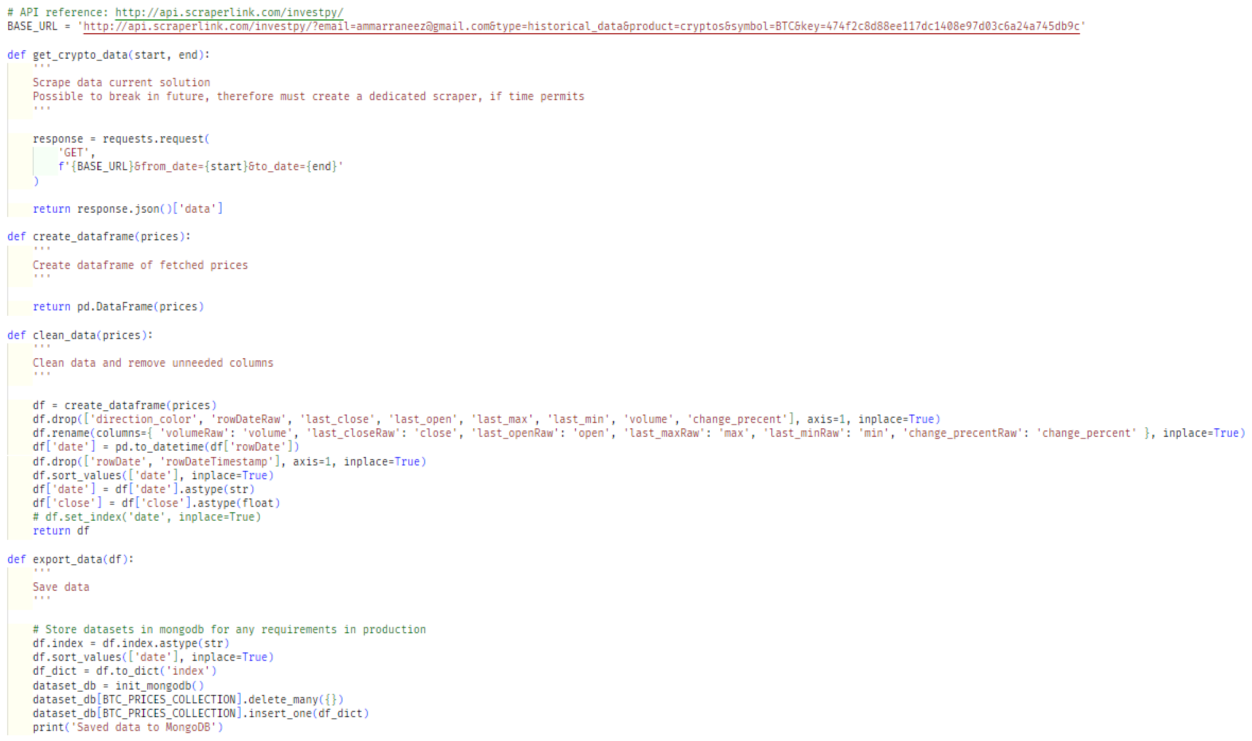
PyTorch vs. TensorFlow: 2022 Deep Learning Comparison | Built In. (2022). Available from https://builtin.com/data-science/pytorch-vs-tensorflow [Accessed 12 December 2022].

Hasani, R. et al. (2020). Liquid Time-constant Networks. Available from https://doi.org/10.48550/arXiv.2006.04439 [Accessed 25 September 2022].

Duvenaud, D (2021). Directions in ML: Latent Stochastic Differential Equations: An Unexplored Model Class. *YouTube*. Available from https://www.youtube.com/watch?v=6iEjF08xgBg. [Accessed on 30 Sep. 2022].

APPENDIX I – Implementation Snippets

Fetch historical prices

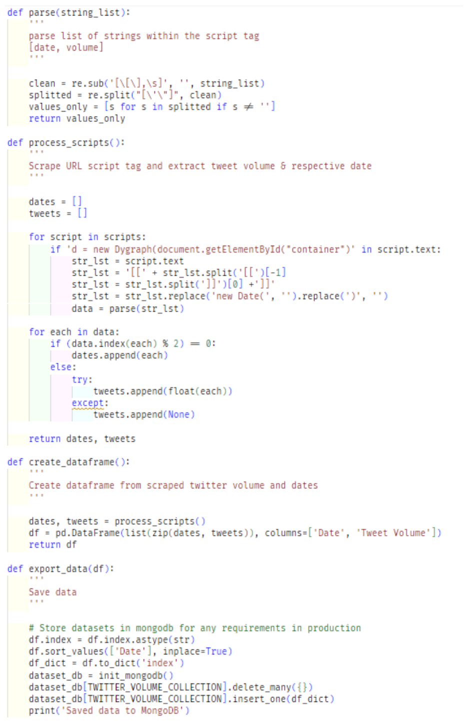


```python
# API reference: http://api.scraperlink.com/investpy/
BASE_URL = 'http://api.scraperlink.com/investpy/?email=ammarraneez@gmail.com&type=historical_data&product=cryptos&symbol=BTC&key=474f2c8d88ee117dc1408e97d03c6a24a745db9c'

def get_crypto_data(start, end):
    '''
    Scrape data current solution
    Possible to break in future, therefore must create a dedicated scraper, if time permits
    '''

    response = requests.request(
        'GET',
        f'{BASE_URL}&from_date={start}&to_date={end}'
    )

    return response.json()['data']

def create_dataframe(prices):
    '''
    Create dataframe of fetched prices
    '''

    return pd.DataFrame(prices)

def clean_data(prices):
    '''
    Clean data and remove unneeded columns
    '''

    df = create_dataframe(prices)
    df.drop(['direction_color', 'rowDateRaw', 'last_close', 'last_open', 'last_max', 'last_min', 'volume', 'change_precent'], axis=1, inplace=True)
    df.rename(columns={ 'volumeRaw': 'volume', 'last_closeRaw': 'close', 'last_openRaw': 'open', 'last_maxRaw': 'max', 'last_minRaw': 'min', 'change_precentRaw': 'change_percent' }, inplace=True)
    df['date'] = pd.to_datetime(df['rowDate'])
    df.drop(['rowDate', 'rowDateTimestamp'], axis=1, inplace=True)
    df.sort_values(['date'], inplace=True)
    df['date'] = df['date'].astype(str)
    df['close'] = df['close'].astype(float)
    # df.set_index('date', inplace=True)
    return df

def export_data(df):
    '''
    Save data
    '''

    # Store datasets in mongodb for any requirements in production
    df.index = df.index.astype(str)
    df.sort_values(['date'], inplace=True)
    df_dict = df.to_dict('index')
    dataset_db = init_mongodb()
    dataset_db[BTC_PRICES_COLLECTION].delete_many({})
    dataset_db[BTC_PRICES_COLLECTION].insert_one(df_dict)
    print('Saved data to MongoDB')
```

Figure 8: Fetch historical prices (*Self-Composed*)

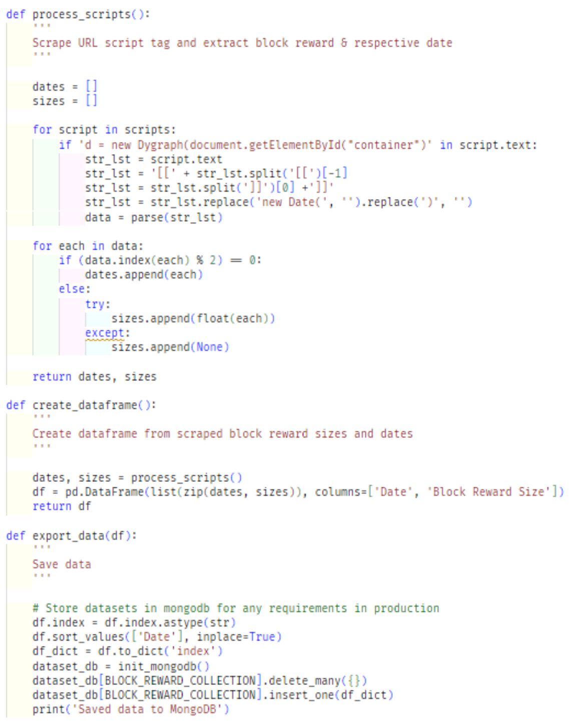
The above script describes a couple of functions that can be used to fetch the latest BTC historical prices data and create a new updated CSV file that can be later read from by the model. A third-party API was used to fetch the data, as existing APIs are all discontinued.

Fetch Twitter volume & block reward size



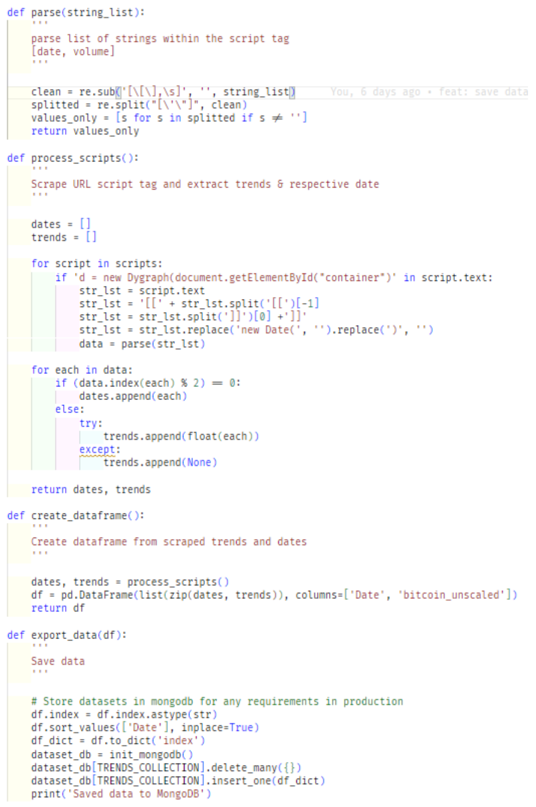
```
def parse(string_list):
    '''
    parse list of strings within the script tag
    [date, volume]
    '''

    clean = re.sub('[\[\],\s]', '', string_list)
    splitted = re.split("[\'\"]", clean)
    values_only = [s for s in splitted if s ≠ '']
    return values_only

def process_scripts():
    '''
    Scrape URL script tag and extract tweet volume & respective date
    '''

    dates = []
    tweets = []

    for script in scripts:
        if 'd = new Dygraph(document.getElementById("container")' in script.text:
            str_lst = script.text
            str_lst = '[[' + str_lst.split('[[')[-1]
            str_lst = str_lst.split(']]')[0] +']]'
            str_lst = str_lst.replace('new Date(', '').replace(')', '')
            data = parse(str_lst)

    for each in data:
        if (data.index(each) % 2) == 0:
            dates.append(each)
        else:
            try:
                tweets.append(float(each))
            except:
                tweets.append(None)

    return dates, tweets

def create_dataframe():
    '''
    Create dataframe from scraped twitter volume and dates
    '''

    dates, tweets = process_scripts()
    df = pd.DataFrame(list(zip(dates, tweets)), columns=['Date', 'Tweet Volume'])
    return df

def export_data(df):
    '''
    Save data
    '''

    # Store datasets in mongodb for any requirements in production
    df.index = df.index.astype(str)
    df.sort_values(['Date'], inplace=True)
    df_dict = df.to_dict('index')
    dataset_db = init_mongodb()
    dataset_db[TWITTER_VOLUME_COLLECTION].delete_many({})
    dataset_db[TWITTER_VOLUME_COLLECTION].insert_one(df_dict)
    print('Saved data to MongoDB')
```

Figure 9: Fetch Twitter volume (*Self-Composed*)



```
def process_scripts():
    '''
    Scrape URL script tag and extract block reward & respective date
    '''

    dates = []
    sizes = []

    for script in scripts:
        if 'd = new Dygraph(document.getElementById("container")' in script.text:
            str_lst = script.text
            str_lst = '[[' + str_lst.split('[[')[-1]
            str_lst = str_lst.split(']]')[0] +']]'
            str_lst = str_lst.replace('new Date(', '').replace(')', '')
            data = parse(str_lst)

    for each in data:
        if (data.index(each) % 2) == 0:
            dates.append(each)
        else:
            try:
                sizes.append(float(each))
            except:
                sizes.append(None)

    return dates, sizes

def create_dataframe():
    '''
    Create dataframe from scraped block reward sizes and dates
    '''

    dates, sizes = process_scripts()
    df = pd.DataFrame(list(zip(dates, sizes)), columns=['Date', 'Block Reward Size'])
    return df

def export_data(df):
    '''
    Save data
    '''

    # Store datasets in mongodb for any requirements in production
    df.index = df.index.astype(str)
    df.sort_values(['Date'], inplace=True)
    df_dict = df.to_dict('index')
    dataset_db = init_mongodb()
    dataset_db[BLOCK_REWARD_COLLECTION].delete_many({})
    dataset_db[BLOCK_REWARD_COLLECTION].insert_one(df_dict)
    print('Saved data to MongoDB')
```

Figure 10: Fetch block reward size (*Self-Composed*)

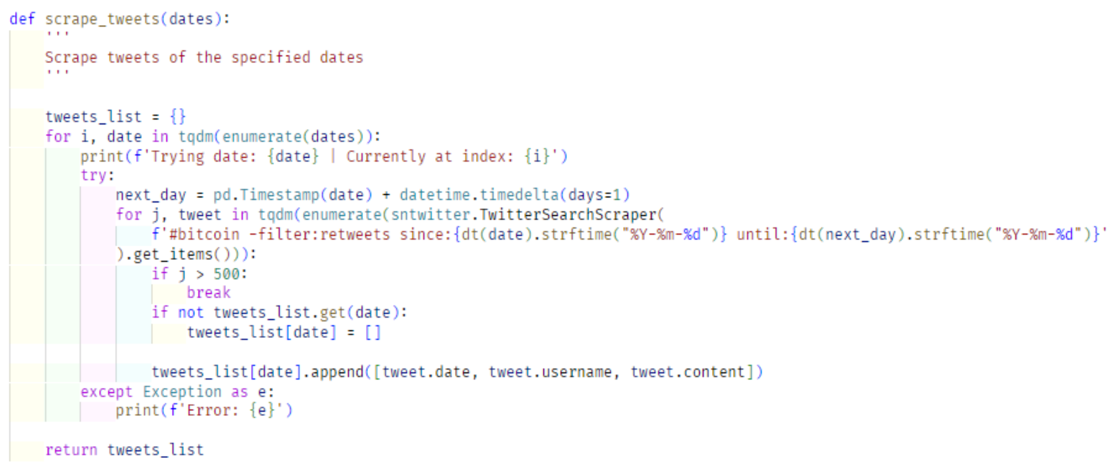


```
def parse(string_list):
    '''
    parse list of strings within the script tag
    [date, volume]
    '''

    clean = re.sub('[\[\],\s]', '', string_list)    You, 6 days ago • feat: save data
    splitted = re.split("[\'\"]", clean)
    values_only = [s for s in splitted if s ≠ '']
    return values_only

def process_scripts():
    '''
    Scrape URL script tag and extract trends & respective date
    '''

    dates = []
    trends = []

    for script in scripts:
        if 'd = new Dygraph(document.getElementById("container")' in script.text:
            str_lst = script.text
            str_lst = '[[' + str_lst.split('[[')[-1]
            str_lst = str_lst.split(']]')[0] +']]'
            str_lst = str_lst.replace('new Date(', '').replace(')', '')
            data = parse(str_lst)

    for each in data:
        if (data.index(each) % 2) == 0:
            dates.append(each)
        else:
            try:
                trends.append(float(each))
            except:
                trends.append(None)

    return dates, trends

def create_dataframe():
    '''
    Create dataframe from scraped trends and dates
    '''

    dates, trends = process_scripts()
    df = pd.DataFrame(list(zip(dates, trends)), columns=['Date', 'bitcoin_unscaled'])
    return df

def export_data(df):
    '''
    Save data
    '''

    # Store datasets in mongodb for any requirements in production
    df.index = df.index.astype(str)
    df.sort_values(['Date'], inplace=True)
    df_dict = df.to_dict('index')
    dataset_db = init_mongodb()
    dataset_db[TRENDS_COLLECTION].delete_many({})
    dataset_db[TRENDS_COLLECTION].insert_one(df_dict)
    print('Saved data to MongoDB')
```

Figure 11: Fetch google trends (*Self-Composed*)

The above scripts fetch the Twitter volume, Google Trends and block reward from a website that publicly exposes this data. Therefore, a simple website scraping tool can be used without authentication or authorization.

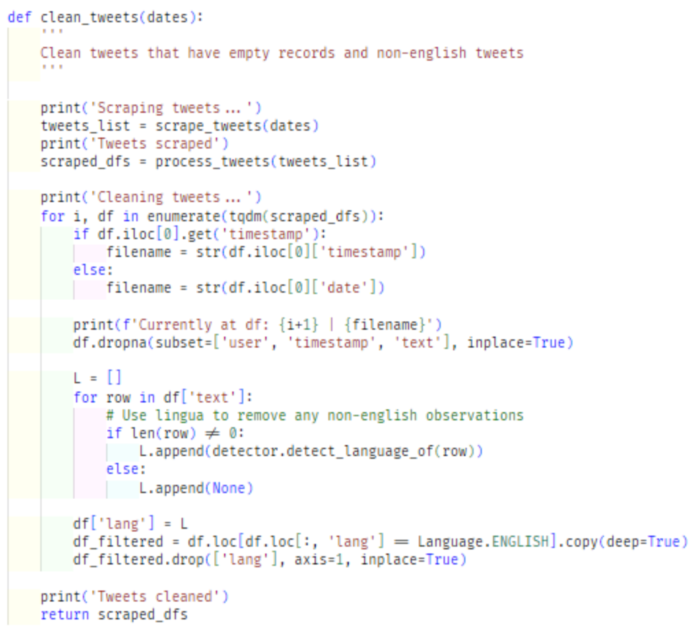
Fetch tweet data



```python
def scrape_tweets(dates):
    '''
    Scrape tweets of the specified dates
    '''

    tweets_list = {}
    for i, date in tqdm(enumerate(dates)):
        print(f'Trying date: {date} | Currently at index: {i}')
        try:
            next_day = pd.Timestamp(date) + datetime.timedelta(days=1)
            for j, tweet in tqdm(enumerate(sntwitter.TwitterSearchScraper(
                f'#bitcoin -filter:retweets since:{dt(date).strftime("%Y-%m-%d")} until:{dt(next_day).strftime("%Y-%m-%d")}'
            ).get_items())):
                if j > 500:
                    break
                if not tweets_list.get(date):
                    tweets_list[date] = []

                tweets_list[date].append([tweet.date, tweet.username, tweet.content])
        except Exception as e:
            print(f'Error: {e}')

    return tweets_list
```

Figure 12: Scrape tweets (*Self-Composed*)

Obtaining the tweet data required a more tedious process as the Twitter API had been updated only to provide tweets for the past week. However, third-party libraries offer this functionality. Tweets fetched were limited to 500 for a single day due to time, performance, and storage constraints, and the application is not the core contribution. Initially, tweets were fetched up to a specific time point; in the future, the above script could be run to scrape tweets of particular dates that are described to be from the days currently existing in the data folder up to the day at which the script is run. There is a further limitation as only '#bitcoin' is searched.



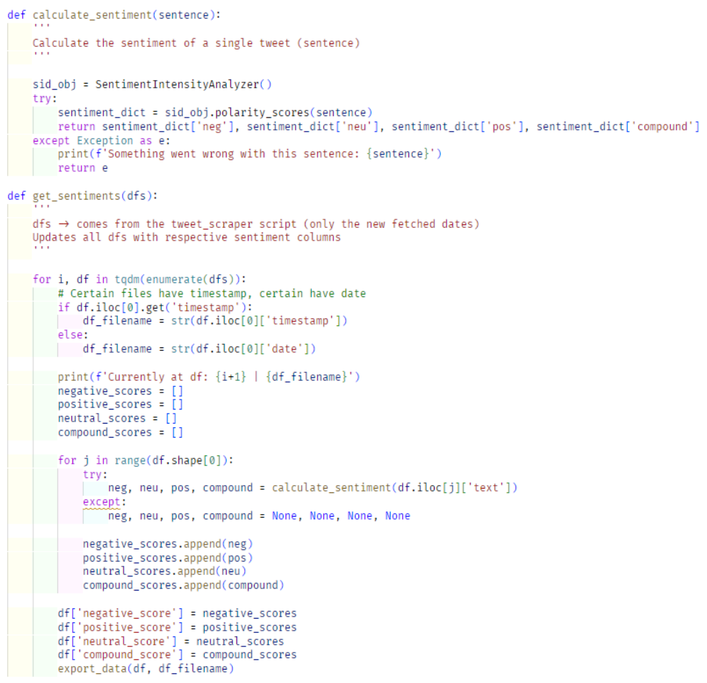
```python
def clean_tweets(dates):
    '''
    Clean tweets that have empty records and non-english tweets
    '''

    print('Scraping tweets ... ')
    tweets_list = scrape_tweets(dates)
    print('Tweets scraped')
    scraped_dfs = process_tweets(tweets_list)

    print('Cleaning tweets ... ')
    for i, df in enumerate(tqdm(scraped_dfs)):
        if df.iloc[0].get('timestamp'):
            filename = str(df.iloc[0]['timestamp'])
        else:
            filename = str(df.iloc[0]['date'])

        print(f'Currently at df: {i+1} | {filename}')
        df.dropna(subset=['user', 'timestamp', 'text'], inplace=True)

        L = []
        for row in df['text']:
            # Use lingua to remove any non-english observations
            if len(row) ≠ 0:
                L.append(detector.detect_language_of(row))
            else:
                L.append(None)

        df['lang'] = L
        df_filtered = df.loc[df.loc[:, 'lang'] == Language.ENGLISH].copy(deep=True)
        df_filtered.drop(['lang'], axis=1, inplace=True)

    print('Tweets cleaned')
    return scraped_dfs
```

Figure 13: Clean tweets (*Self-Composed*)

As this research is currently limited to only English, the tweets are filtered, and non-English tweets are removed.

Sentiment analysis

The main step of preprocessing is to perform sentiment analysis on the obtained tweet data. In this research, the VADER sentiment analyzer is used as determined in previous chapters.

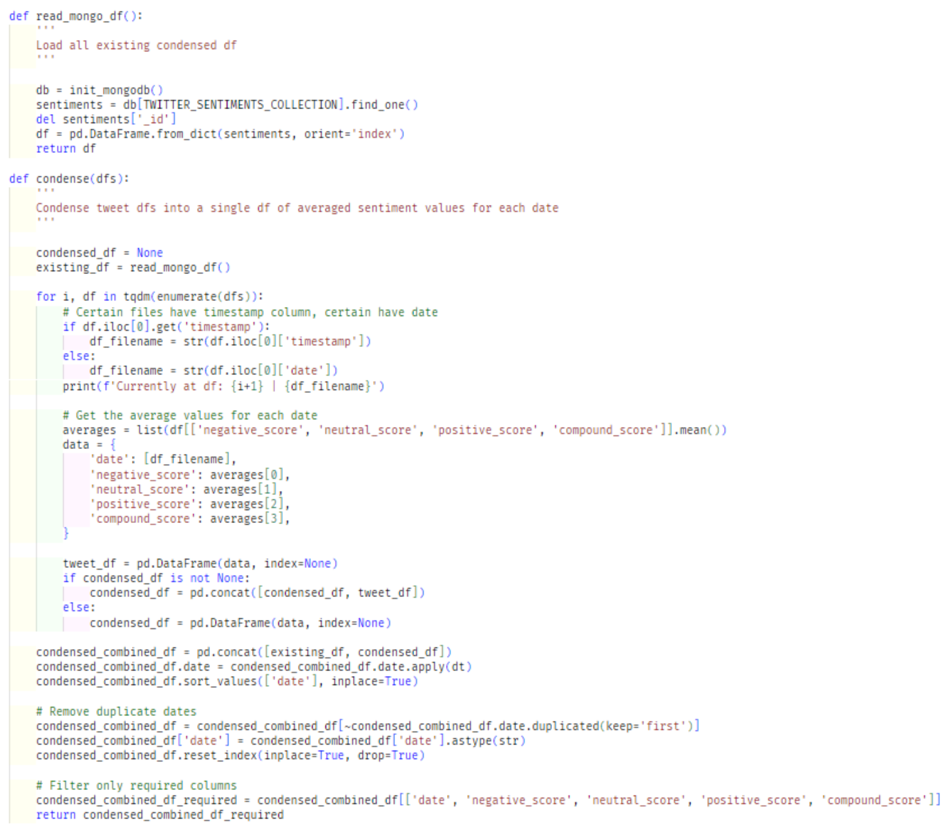


```python
def calculate_sentiment(sentence):
    '''
    Calculate the sentiment of a single tweet (sentence)
    '''

    sid_obj = SentimentIntensityAnalyzer()
    try:
        sentiment_dict = sid_obj.polarity_scores(sentence)
        return sentiment_dict['neg'], sentiment_dict['neu'], sentiment_dict['pos'], sentiment_dict['compound']
    except Exception as e:
        print(f'Something went wrong with this sentence: {sentence}')
        return e

def get_sentiments(dfs):
    '''
    dfs → comes from the tweet_scraper script (only the new fetched dates)
    Updates all dfs with respective sentiment columns
    '''

    for i, df in tqdm(enumerate(dfs)):
        # Certain files have timestamp, certain have date
        if df.iloc[0].get('timestamp'):
            df_filename = str(df.iloc[0]['timestamp'])
        else:
            df_filename = str(df.iloc[0]['date'])

        print(f'Currently at df: {i+1} | {df_filename}')
        negative_scores = []
        positive_scores = []
        neutral_scores = []
        compound_scores = []

        for j in range(df.shape[0]):
            try:
                neg, neu, pos, compound = calculate_sentiment(df.iloc[j]['text'])
            except:
                neg, neu, pos, compound = None, None, None, None

            negative_scores.append(neg)
            positive_scores.append(pos)
            neutral_scores.append(neu)
            compound_scores.append(compound)

        df['negative_score'] = negative_scores
        df['positive_score'] = positive_scores
        df['neutral_score'] = neutral_scores
        df['compound_score'] = compound_scores
        export_data(df, df_filename)
```

Figure 14: Analyze sentiments (*Self-Composed*)

The above script is used to perform sentiment analysis on the tweets and concatenates the negative, positive, neutral, and compound scores into the existing tweet dataset, which can then be condensed down to create an average score for a single day.

Tweet dataset condensation



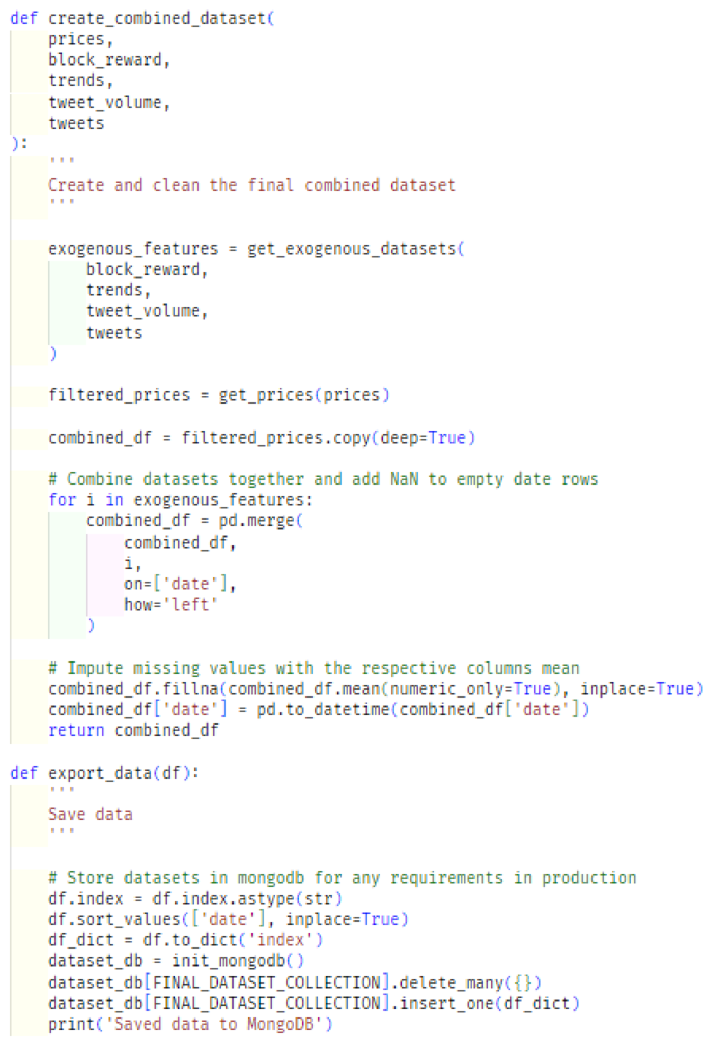
```python
def read_mongo_df():
    '''
    Load all existing condensed df
    '''

    db = init_mongodb()
    sentiments = db[TWITTER_SENTIMENTS_COLLECTION].find_one()
    del sentiments['_id']
    df = pd.DataFrame.from_dict(sentiments, orient='index')
    return df

def condense(dfs):
    '''
    Condense tweet dfs into a single df of averaged sentiment values for each date
    '''

    condensed_df = None
    existing_df = read_mongo_df()

    for i, df in tqdm(enumerate(dfs)):
        # Certain files have timestamp column, certain have date
        if df.iloc[0].get('timestamp'):
            df_filename = str(df.iloc[0]['timestamp'])
        else:
            df_filename = str(df.iloc[0]['date'])
        print(f'Currently at df: {i+1} | {df_filename}')

        # Get the average values for each date
        averages = list(df[['negative_score', 'neutral_score', 'positive_score', 'compound_score']].mean())
        data = {
            'date': [df_filename],
            'negative_score': averages[0],
            'neutral_score': averages[1],
            'positive_score': averages[2],
            'compound_score': averages[3],
        }

        tweet_df = pd.DataFrame(data, index=None)
        if condensed_df is not None:
            condensed_df = pd.concat([condensed_df, tweet_df])
        else:
            condensed_df = pd.DataFrame(data, index=None)

    condensed_combined_df = pd.concat([existing_df, condensed_df])
    condensed_combined_df.date = condensed_combined_df.date.apply(dt)
    condensed_combined_df.sort_values(['date'], inplace=True)

    # Remove duplicate dates
    condensed_combined_df = condensed_combined_df[~condensed_combined_df.date.duplicated(keep='first')]
    condensed_combined_df['date'] = condensed_combined_df['date'].astype(str)
    condensed_combined_df.reset_index(inplace=True, drop=True)

    # Filter only required columns
    condensed_combined_df_required = condensed_combined_df[['date', 'negative_score', 'neutral_score', 'positive_score', 'compound_score']]
    return condensed_combined_df_required
```

Figure 15: Combine and condense tweets (*Self-Composed*)

As the other data being used directly creates a single CSV file with a row for each date, the condensation process is unnecessary. However, as the tweet data fetched consists of a separate CSV file for each date, this data must be compressed to the same format as other datasets.

The above script condenses the tweet dataset into a single CSV file by averaging the sentiment scores for each day.

Final dataset creation



```python
def create_combined_dataset(
    prices,
    block_reward,
    trends,
    tweet_volume,
    tweets
):
    '''
    Create and clean the final combined dataset
    '''

    exogenous_features = get_exogenous_datasets(
        block_reward,
        trends,
        tweet_volume,
        tweets
    )

    filtered_prices = get_prices(prices)

    combined_df = filtered_prices.copy(deep=True)

    # Combine datasets together and add NaN to empty date rows
    for i in exogenous_features:
        combined_df = pd.merge(
            combined_df,
            i,
            on=['date'],
            how='left'
        )

    # Impute missing values with the respective columns mean
    combined_df.fillna(combined_df.mean(numeric_only=True), inplace=True)
    combined_df['date'] = pd.to_datetime(combined_df['date'])
    return combined_df

def export_data(df):
    '''
    Save data
    '''

    # Store datasets in mongodb for any requirements in production
    df.index = df.index.astype(str)
    df.sort_values(['date'], inplace=True)
    df_dict = df.to_dict('index')
    dataset_db = init_mongodb()
    dataset_db[FINAL_DATASET_COLLECTION].delete_many({})
    dataset_db[FINAL_DATASET_COLLECTION].insert_one(df_dict)
    print('Saved data to MongoDB')
```

Figure 16: Combine all datasets (*Self-Composed*)

The above script is used to create the final dataset that the model uses. It fetches all the datasets and combines them into a single data frame. Initially, a helper function removes unneeded columns from the data files, which were decided upon conducting correlation tests. The mean of their respective columns imputes missing values of each feature of specific dates. This combined dataset can be saved so the model can finally utilize it.