CS6003 - BIG DATA ANALYTICS

[6th Semester, JAN’23 - MAY’23]

CRYPTO INSIGHT

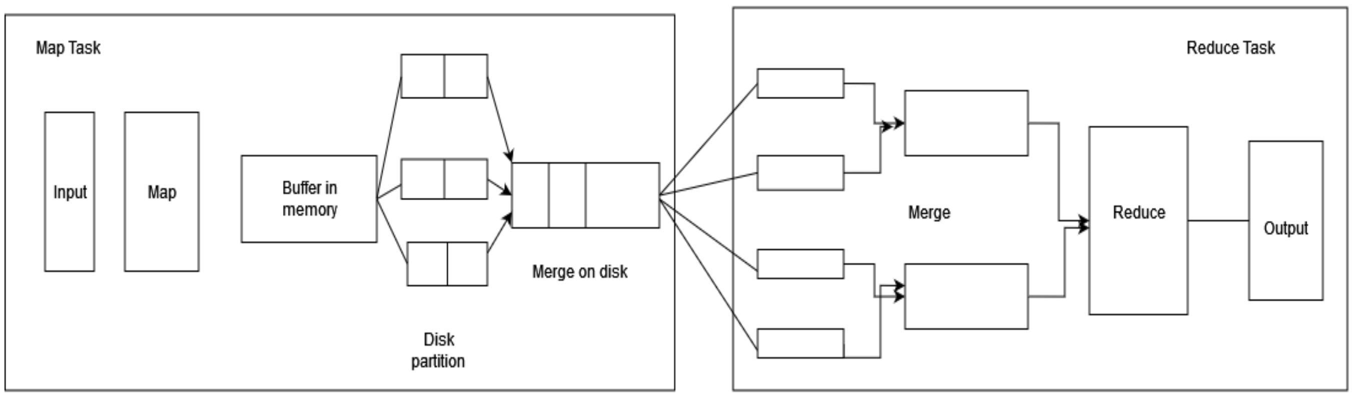
Insights and Prediction for the crypto market using Data Analysis

Team Members:

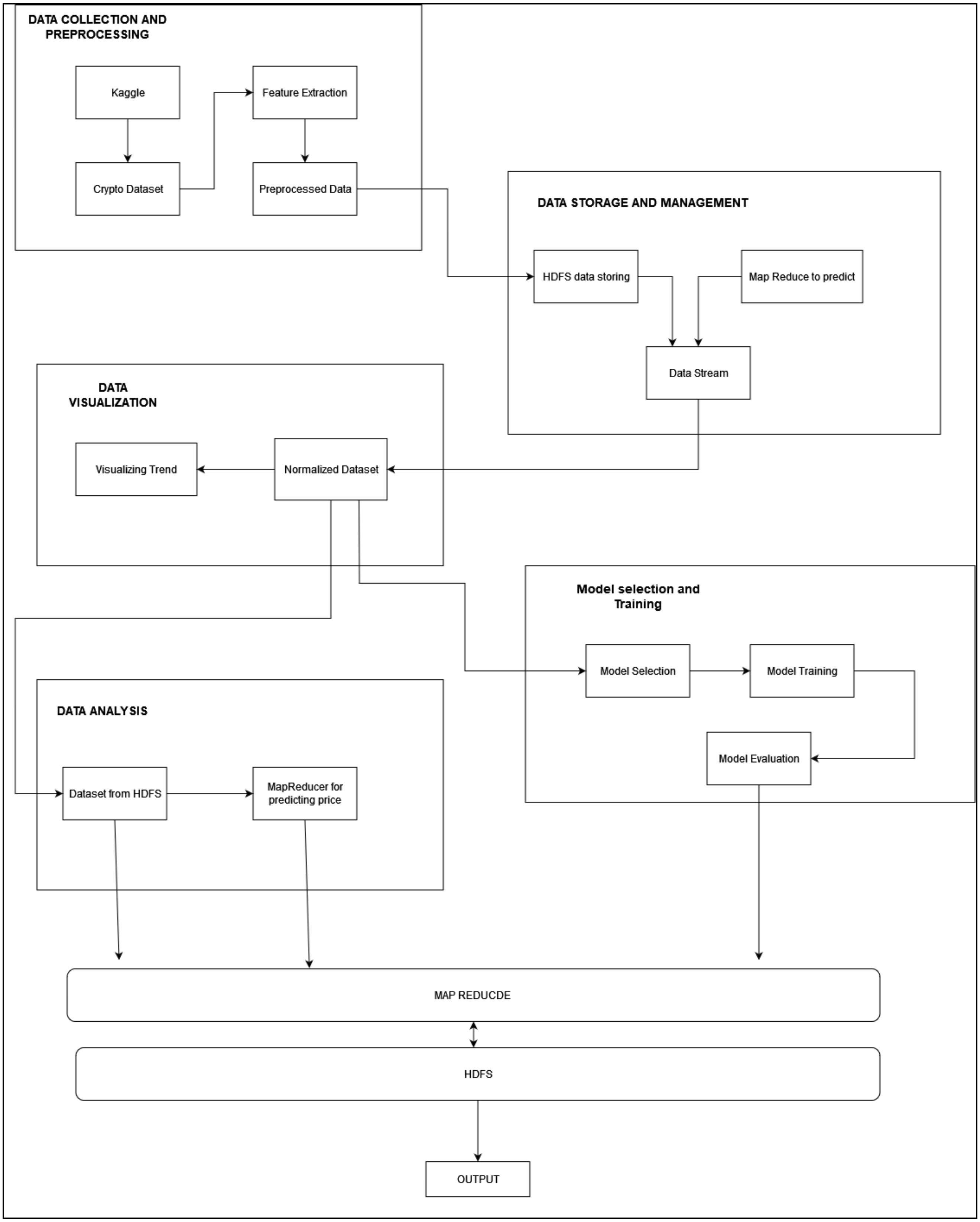
Praveen Kumar R - 2020103036
Sujith Vishal V - 2020103572

Architecture Diagram:

MapReduce is a programming model and processing framework for processing large datasets in a distributed environment. A MapReduce program is composed of two main functions: the Map function and the Reduce function.



Map Reduce FrameWork



Architecture Diagram

Dataset:

Dataset: G-Research Crypto Forecasting dataset
Columns/ features of the dataset are:

- **timestamp** - A timestamp for the minute covered by the row.
- **Asset_ID** - An ID code for the crypto asset.
- **Count** - The number of trades that took place this minute.
- **Open** - The USD price at the beginning of the minute.
- **High** - The highest USD price during the minute.
- **Low** - The lowest USD price during the minute.
- **Close** - The USD price at the end of the minute.
- **Volume** - The number of crypto asset units traded during the minute.
- **VWAP** - The volume weighted average price for the minute.
- **Target** - 15 minute residualized returns. See the 'Prediction and Evaluation' section of this notebook for details of how the target is calculated.

List of Modules:

- Data Collection & Preprocessing
- Data Analysis
- Model Training & Evaluation
- Hyperparameter Tuning
- Results

Data Collection & Preprocessing:

The dataset with **2,42,36,806 rows** is loaded which has the following features: timestamp, Assest_ID, Count, open, High, Low, Close, Volume, VMAP, Target. It has the data of crypto currency prices starting from **2018-01-01** T 00:01:00 until **2021-09-21** T 00:00:00

Dataset Features

```
In [2]: import pandas as pd
        import numpy as np
        import os
        import time

        import matplotlib.pyplot as plt

        from datetime import datetime
```

```
In [5]: start = time.time()

        crypto_df = pd.read_csv("./input/train.csv")

        end = time.time()
        print(end - start)

        25.552823781967163
```

```
In [6]: crypto_df.shape
Out[6]: (24236806, 10)
```

```
In [7]: crypto_df.head(10)
Out[7]:
```

	timestamp	Asset_ID	Count	Open	High	Low	Close	Volume	VWAP	Target
0	1514764860	2	40.0	2376.580000	2399.5000	2357.1400	2374.590000	19.233005	2373.116392	-0.004218
1	1514764860	0	5.0	8.530000	8.5300	8.5300	8.530000	78.380000	8.530000	-0.014399
2	1514764860	1	229.0	13835.194000	14013.8000	13666.1100	13850.176000	31.550062	13827.062093	-0.014643
3	1514764860	5	32.0	7.659600	7.6596	7.6567	7.657600	6626.713370	7.657713	-0.013922
4	1514764860	7	5.0	25.920000	25.9200	25.8740	25.877000	121.087310	25.891363	-0.008264
5	1514764860	6	173.0	738.302500	746.0000	732.5100	738.507500	335.987856	738.839291	-0.004809
6	1514764860	9	167.0	225.330000	227.7800	222.9800	225.206667	411.896642	225.197944	-0.009791
7	1514764860	11	7.0	329.090000	329.8800	329.0900	329.460000	6.635710	329.454118	NaN
8	1514764920	2	53.0	2374.553333	2400.9000	2354.2000	2372.286667	24.050259	2371.434498	-0.004079

```
In [8]: start = crypto_df.iloc[0].timestamp.astype('datetime64[s]')
        end = crypto_df.iloc[-1].timestamp.astype('datetime64[s]')

        print(f'Data from {start} until {end}')

        Data from 2018-01-01T00:01:00 until 2021-09-21T00:00:00
```

```
In [11]: asset_details_df = pd.read_csv("./input/asset_details.csv")
```

```
In [12]: asset_details_df
Out[12]:
```

	Asset_ID	Weight	Asset_Name
0	2	2.397895	Bitcoin Cash
1	0	4.304065	Binance Coin
2	1	6.779922	Bitcoin
3	5	1.386294	EOS.IO
4	7	2.079442	Ethereum Classic
5	6	5.894403	Ethereum
6	9	2.397895	Litecoin
7	11	1.609438	Monero
8	13	1.791759	TRON
9	12	2.079442	Stellar
10	3	4.406719	Cardano
11	8	1.098612	IOTA
12	10	1.098612	Maker
13	4	3.555348	Dogecoin

The training data is visualized with a candle stick graph to know about the consistency of the data given. The candlestick plot is given below:

Trading Data visualization

In [13]:
```
btc_mini_df = crypto_df[crypto_df.Asset_ID == 1].iloc[-60:]
```

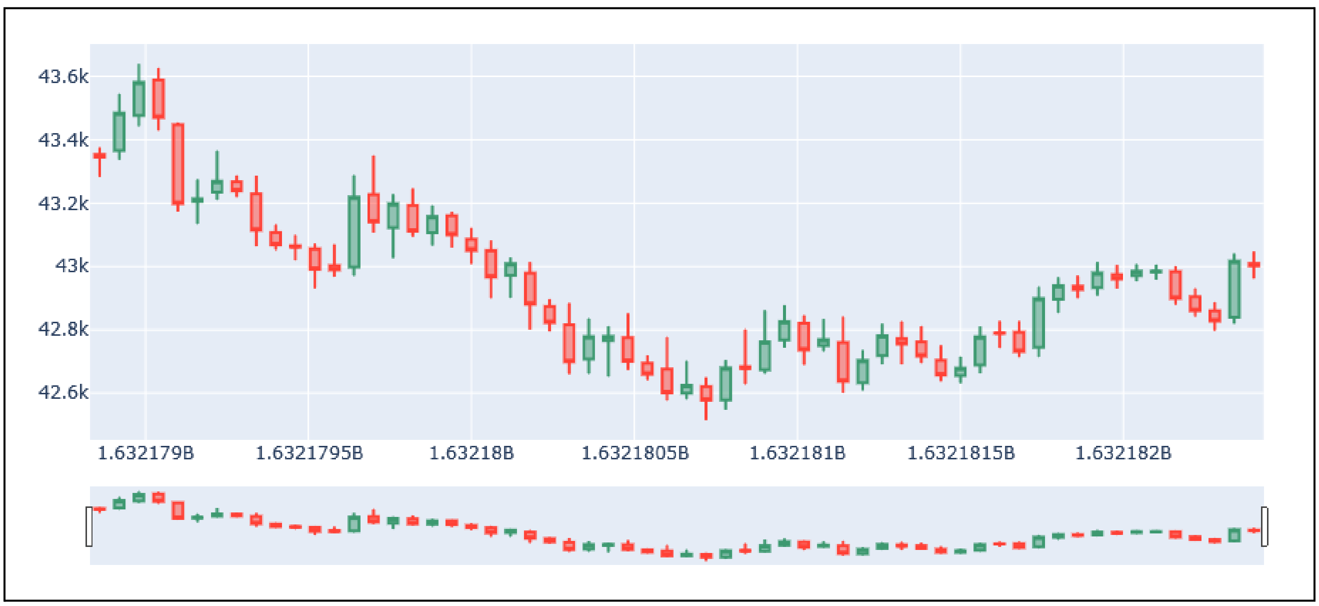
In [15]:
```
btc_mini_df.head(10)
```

Out[15]:

	timestamp	Asset_ID	Count	Open	High	Low	Close	Volume	VWAP	Target
24235969	1632178860	1	1952.0	43353.120000	43376.00	43283.10	43344.558571	65.051627	43329.733310	-0.000184
24235983	1632178920	1	4369.0	43365.748750	43546.61	43335.64	43484.613750	145.414597	43446.758145	0.000255
24235997	1632178980	1	4638.0	43477.087500	43640.00	43441.56	43580.823750	180.392877	43544.364733	0.001151
24236011	1632179040	1	3211.0	43588.102500	43627.00	43428.80	43470.795000	149.275363	43535.659814	0.002428
24236025	1632179100	1	5038.0	43447.602857	43455.00	43172.90	43198.788571	138.454840	43344.478199	0.001363
24236039	1632179160	1	4671.0	43205.556250	43276.57	43133.08	43213.690000	199.959730	43209.370264	0.000565
24236053	1632179220	1	2791.0	43234.952857	43366.00	43209.90	43269.547143	101.917324	43295.323457	-0.000912
24236067	1632179280	1	1844.0	43267.830000	43288.47	43218.81	43239.261250	66.517441	43253.365759	-0.000915
24236081	1632179340	1	3832.0	43229.501250	43287.50	43062.90	43114.236141	100.369113	43188.795414	-0.002730
24236095	1632179400	1	3207.0	43106.669255	43133.15	43050.90	43067.630000	122.076485	43083.913981	-0.002652

In [16]:
```
btc_mini_df = btc_mini_df.set_index("timestamp")
```

In [17]:
```
import plotly.graph_objects as go

fig = go.Figure(data=[go.Candlestick(x=btc_mini_df.index,
                                     open=btc_mini_df['Open'],
                                     high=btc_mini_df['High'],
                                     low=btc_mini_df['Low'],
                                     close=btc_mini_df['Close'])])
fig.show()
```



The data is then preprocessed which includes removing null values, replacing null values, finding NA values in the dataset.

Data Preprocessing

```
In [18]: btc_df = crypto_df[crypto_df.Asset_ID == 1].set_index('timestamp')

         btc_df.info(show_counts =True)

<class 'pandas.core.frame.DataFrame'>
Int64Index: 1956282 entries, 1514764860 to 1632182400
Data columns (total 9 columns):
 #   Column    Non-Null Count    Dtype
---  ------    --------------    -----
 0   Asset_ID  1956282 non-null  int64
 1   Count     1956282 non-null  float64
 2   Open      1956282 non-null  float64
 3   High      1956282 non-null  float64
 4   Low       1956282 non-null  float64
 5   Close     1956282 non-null  float64
 6   Volume    1956282 non-null  float64
 7   VWAP      1956282 non-null  float64
 8   Target    1955978 non-null  float64
dtypes: float64(8), int64(1)
memory usage: 149.3 MB

In [19]: btc_df.isna().sum()

Out[19]: Asset_ID      0
         Count         0
         Open          0
         High          0
         Low           0
         Close         0
         Volume        0
         VWAP          0
         Target      304
         dtype: int64
```

The dataset should have one row per minute per asset. Since the data is extracted for a single asset, we expect consecutive rows to have a difference of 60 seconds between their index values. The gap above 60 seconds is filled using a simple imputation method: fill in the missing data with the value from the most recent available minute.

```
In [20]: # look at the time lag between consecutive entries in dataset which should be 60 seconds

         (btc_df.index[1:]-btc_df.index[:-1]).value_counts().head()

Out[20]: 60     1956136
         120         78
         180         12
         240         11
         420          9
         Name: timestamp, dtype: int64

In [21]: # to fill the gaps more than 60 seconds, fill in the missing data with the value from the most recent available minute

         btc_df = btc_df.reindex(range(btc_df.index[0],btc_df.index[-1]+60,60), method='pad')

In [22]: (btc_df.index[1:]-btc_df.index[:-1]).value_counts().head()

         # now all the data has gap of 60 seconds (1 min)

Out[22]: 60    1956959
         Name: timestamp, dtype: int64

In [23]: btc_df['datetime'] = btc_df.apply(lambda r: np.float64(r.name).astype('datetime64[s]'), axis=1)

         btc_df.set_index('datetime', inplace=True);

         btc_df.head(10)

Out[23]:
```

datetime	Asset_ID	Count	Open	High	Low	Close	Volume	VWAP	Target
2018-01-01 00:01:00	1	229.0	13835.194	14013.8	13666.11	13850.176	31.550062	13827.062093	-0.014643
2018-01-01 00:02:00	1	235.0	13835.036	14052.3	13680.00	13828.102	31.046432	13840.362591	-0.015037
2018-01-01 00:03:00	1	528.0	13823.900	14000.4	13601.00	13801.314	55.061820	13806.068014	-0.010309
2018-01-01 00:04:00	1	435.0	13802.512	13999.0	13576.28	13768.040	38.780529	13783.598101	-0.008999
2018-01-01 00:05:00	1	742.0	13766.000	13955.9	13554.44	13724.914	108.501637	13735.586842	-0.008079
2018-01-01 00:06:00	1	554.0	13717.714	14000.7	13520.00	13717.112	70.805776	13706.952030	-0.004422
2018-01-01 00:07:00	1	546.0	13720.922	14001.4	13501.01	13670.940	70.762103	13683.843336	-0.008873

Feature Engineering:

From the feature timestamp 4 features: date, month, year, hour are extracted inorder to provide as input for supervised learning using feature engineering

Creating dataset for supervised learning

Transform the timestamp into day, hour, min, thus creating three new features

In [25]:
```
btc_mini_df = btc_df[-7201:].copy(deep=True)
```

In [26]:
```
btc_mini_df.head(5)
```

Out[26]:

datetime	Asset_ID	Count	Open	High	Low	Close	Volume	VWAP	Target
2021-09-16 00:00:00	1	2466.0	48142.545443	48188.12	48086.22	48101.315714	100.182817	48132.845372	0.001079
2021-09-16 00:01:00	1	1757.0	48101.928395	48164.10	48080.00	48135.600000	64.948787	48112.135086	0.001212
2021-09-16 00:02:00	1	2331.0	48135.697143	48211.23	48069.62	48186.595714	101.655798	48125.096256	0.001464
2021-09-16 00:03:00	1	2231.0	48185.676423	48222.63	48119.94	48163.827500	53.992481	48174.667732	0.001517
2021-09-16 00:04:00	1	2241.0	48164.208136	48182.36	48067.83	48106.416629	65.479223	48122.721662	0.001424

In [27]:
```
btc_mini_df['time'] = btc_mini_df.apply(lambda r:r.name, axis=1)

# parse timestamp into year, month, day and hour
btc_mini_df['year'] = [btc_mini_df.iloc[i].time.year for i in range(len(btc_mini_df))]
btc_mini_df['month'] = [btc_mini_df.iloc[i].time.month for i in range(len(btc_mini_df))]
btc_mini_df['day'] = [btc_mini_df.iloc[i].time.day for i in range(len(btc_mini_df))]
btc_mini_df['hour'] = [btc_mini_df.iloc[i].time.hour for i in range(len(btc_mini_df))]
```

In [28]:
```
btc_mini_df.head(5)
```

Out[28]:

datetime	Asset_ID	Count	Open	High	Low	Close	Volume	VWAP	Target	time	year	month	day	hour
2021-09-16 00:00:00	1	2466.0	48142.545443	48188.12	48086.22	48101.315714	100.182817	48132.845372	0.001079	2021-09-16 00:00:00	2021	9	16	0
2021-09-16 00:01:00	1	1757.0	48101.928395	48164.10	48080.00	48135.600000	64.948787	48112.135086	0.001212	2021-09-16 00:01:00	2021	9	16	0
2021-09-16 00:02:00	1	2331.0	48135.697143	48211.23	48069.62	48186.595714	101.655798	48125.096256	0.001464	2021-09-16 00:02:00	2021	9	16	0

Now, the dataset is made to reflect the change between two consecutive intervals and using this previous data, future predictions are done for the next 60 seconds using supervised learning. This is then repeated for three consecutive intervals to get the mean ratio.

```
In [29]: # average data per hour
         tmp = btc_mini_df.groupby(['year', 'month', 'day', 'hour']).mean()

         # restore the multilevel index created by groupby into the year, month, day, hour columns that we created earlier
         tmp.reset_index(inplace=True)

In [30]: cols = ['year', 'month', 'day', 'hour', 'Close']

         tmp[cols].head(5)

Out[30]:
```

	year	month	day	hour	Close
0	2021	9	16	0	48011.855849
1	2021	9	16	1	47989.773176
2	2021	9	16	2	47961.590146
3	2021	9	16	3	47622.839055
4	2021	9	16	4	48175.404142

The idea is to predict the value at the current timestamp based on the value from the previous timestamp(s).

```
In [31]: # We use pandas' dataframe.shift() function, which shifts values vertically or horizontally, fills in with NaN values and leaves

         tmp = btc_mini_df['Close'] # extract only the Close price

         lag_df = pd.concat([tmp.shift(1, axis = 0), tmp], axis=1) # downward shift by 1 step

         # the original price series becomes the time t value,
         # while the downward shifted series is time t+1
         lag_df.columns = ['Close(t)', 'Close(t+1)']

         lag_df.head()

Out[31]:
```

datetime	Close(t)	Close(t+1)
2021-09-16 00:00:00	NaN	48101.315714
2021-09-16 00:01:00	48101.315714	48135.600000

The mean of the close feature for consecutive intervals is found which is now used for prediction. The most common aggregate value is the lag window (which is the moving average or rolling mean) which slides through the values.

```
         lag_df = pd.concat([tmp.shift(3), tmp.shift(2), tmp.shift(1), tmp], axis=1)

         lag_df.columns = ['Close(t-2)', 'Close(t-1)', 'Close(t)', 'Close(t+1)'] # rename columns for easier read

         lag_df.head()

Out[32]:
```

datetime	Close(t-2)	Close(t-1)	Close(t)	Close(t+1)
2021-09-16 00:00:00	NaN	NaN	NaN	48101.315714
2021-09-16 00:01:00	NaN	NaN	48101.315714	48135.600000
2021-09-16 00:02:00	NaN	48101.315714	48135.600000	48186.595714
2021-09-16 00:03:00	48101.315714	48135.600000	48186.595714	48163.827500
2021-09-16 00:04:00	48135.600000	48186.595714	48163.827500	48106.416629

```
In [33]: # Now we compute summary statistics of these values and use them as features for prediction
         # The most common aggregate value is the mean of the lag window (also called moving average or rolling mean)

         tmp = btc_mini_df['Close'] # extract only the Close price

         lag_df = tmp.shift(1) # downward shift by 1

         window = lag_df.rolling(window=2) # rolling window size of 2
         means = window.mean() # compute the means for the rolling windows

         new_df = pd.concat([means, tmp], axis=1) # concatenate the two series vertically

         new_df.columns = ['mean(t-1,t)', 't+1'] # rename columns for easier reading

         new_df.head()

Out[33]:
```

datetime	mean(t-1,t)	t+1
2021-09-16 00:00:00	NaN	48101.315714
2021-09-16 00:01:00	NaN	48135.600000
2021-09-16 00:02:00	48118.457857	48186.595714

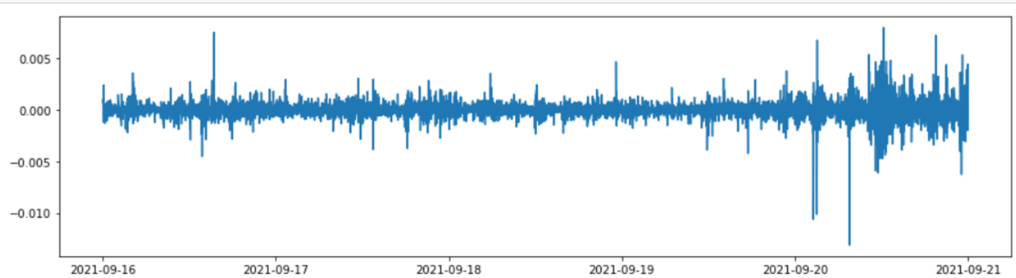
By using the lag window which slides, we found the min, max, mean in the values to which the window slides in the dataset.

```
In [34]: window = tmp.expanding()

         dataframe = pd.concat([window.min(), window.mean(), window.max(), tmp.shift(-1)], axis=1)

         dataframe.columns = ['min', 'mean', 'max', 't+1']

         print(dataframe.head(5))

                                   min          mean           max           t+1
         datetime
         2021-09-16 00:00:00  48101.315714  48101.315714  48101.315714  48135.600000
         2021-09-16 00:01:00  48101.315714  48118.457857  48135.600000  48186.595714
         2021-09-16 00:02:00  48101.315714  48141.170476  48186.595714  48163.827500
         2021-09-16 00:03:00  48101.315714  48146.834732  48186.595714  48106.416629
         2021-09-16 00:04:00  48101.315714  48138.751112  48186.595714  48075.588750

In [35]:
         # printing initial dataset to verify that tha dataframe above is correct
         tmp.head(5)

Out[35]: datetime
         2021-09-16 00:00:00    48101.315714
         2021-09-16 00:01:00    48135.600000
         2021-09-16 00:02:00    48186.595714
         2021-09-16 00:03:00    48163.827500
         2021-09-16 00:04:00    48106.416629
         Name: Close, dtype: float64
```

The ratio between the current price and price at the previous time point which is computed as log is plotted against the time and the volatility of the data(market) is visualized.

A typical feature for assets price time series is the ratio between the current price and the price at the previous time point. This is usually computed as a log of the ratio in time series modelling, because that facilitates some apperations (additions, etc). In this case, they are called log returns

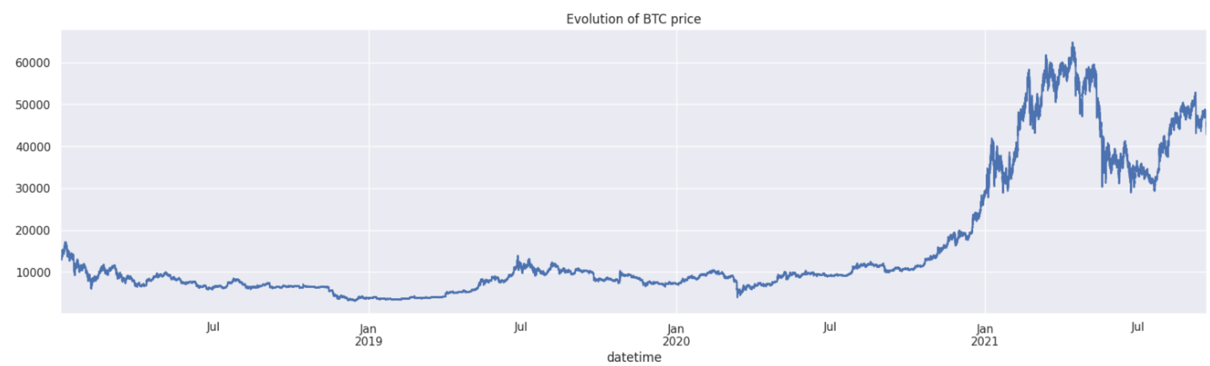
```
In [36]: # helper function to compute the log returns
         def log_returns(series, periods = 1):
             return np.log(series).diff(periods = periods)

In [37]: f = plt.figure(figsize = (15,4))

         lret_btc = log_returns(btc_mini_df.Close,1)[1:]

         plt.plot(lret_btc)

         plt.show()
```



Data Analysis and Visualization:

Data analysis and visualization play a crucial role in machine learning, enabling researchers and practitioners to extract valuable insights and make informed decisions. By leveraging various statistical techniques and visualization tools, data analysts can uncover patterns, trends, and correlations within complex datasets.Data analysis and visualization in machine learning facilitate the exploration of large datasets and help identify relevant features that contribute to the predictive models' performance. Techniques such as dimensionality reduction and feature selection aid in extracting the most informative aspects from the data, streamlining the model training process

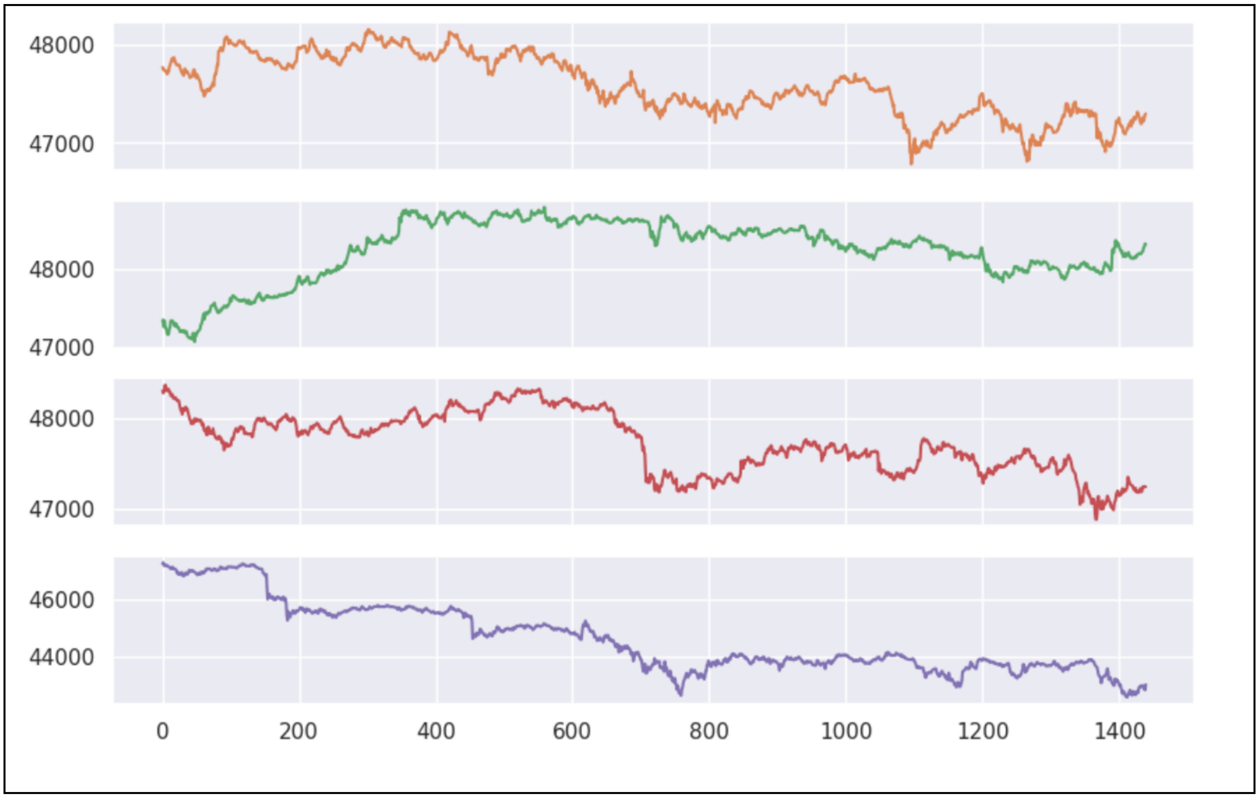
```
#btc_df.plot(x='datetime', y='Close', figsize=(8,5))
btc_df.Close.plot(figsize=(20,5))
plt.title('Evolution of BTC price')
plt.show()
```



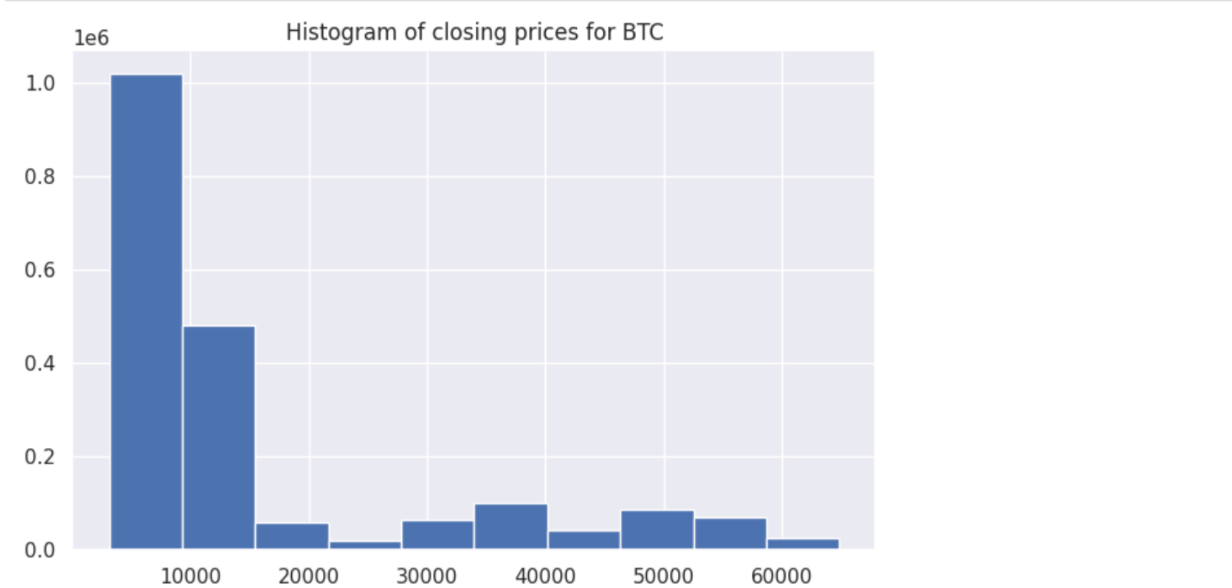
The code plots the evolution of BTC price using btc_df.Close.plot(figsize=(20,5)). The figsize parameter is set to (20, 5) to control the size of the plot.The title of the plot is set as 'Evolution of BTC price' using plt.title().

```
groups = btc_mini_df.groupby('day')

days = pd.DataFrame()

for name, group in groups:
    if name == 21: # skip the last day, which seems to be incomplete
        continue
    days[name] = group.Close.values

days.plot(subplots=True, legend=False, figsize=(10,8), title='BTC price evolution throughout the day\n2021-09-16 to 2021-09-20');
```

The code groups the btc_mini_df dataframe by the 'day' column, creating a groups object.An empty DataFrame called days is initialized.A loop iterates over each group in groups. If the name of the group is 21 (representing the last day), it is skipped. For each group, the 'Close' values are extracted and assigned to the corresponding column in the days DataFrame using the group's name as the column label.



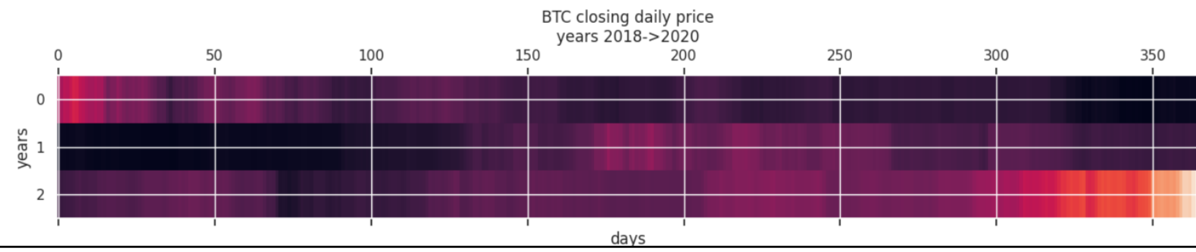

```
btc_df.Close.hist(figsize=(8,5))
plt.title('Histogram of closing prices for BTC')
plt.show()
```



This code generates a histogram of the closing prices for BTC using btc_df.Close.hist(). The figsize parameter is set to (8, 5) to control the size of the plot.The title of the histogram is set as 'Histogram of closing prices for BTC' using plt.title().

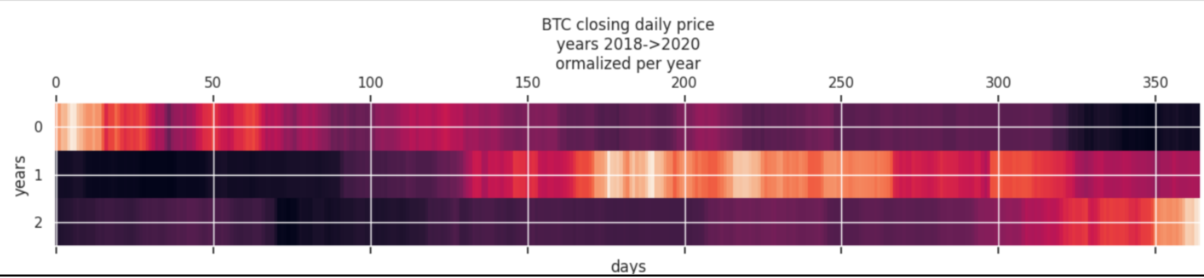
5.4 Heat map plots

```
yrs = ['Close', 'Close2019', 'Close2020']

plt.matshow(years[yrs].dropna().T, interpolation=None, aspect='auto')

plt.title('BTC closing daily price\nyears 2018->2020')
plt.ylabel('years')
plt.xlabel('days')
plt.show()
```



Using plt.matshow, a matrix plot is created with the selected columns from the years dataframe. Missing values are dropped, and the matrix is transposed for better visualization. The parameters interpolation=None and aspect='auto' control the appearance of the plot. The title of the plot is set as 'BTC closing daily price, years 2018->2020'. The y-axis represents the years, and the x-axis represents the days. Finally, plt.show() is used to display the plot, providing a visual representation of the BTC closing daily price for the specified years, highlighting any patterns or trends over time.

```
norm_years=(years-years.min())/(years.max()-years.min())

plt.matshow(norm_years[yrs].dropna().T, interpolation=None, aspect='auto')

plt.title('BTC closing daily price\nyears 2018->2020\normalized per year')
plt.ylabel('years')
plt.xlabel('days')

plt.show()
```

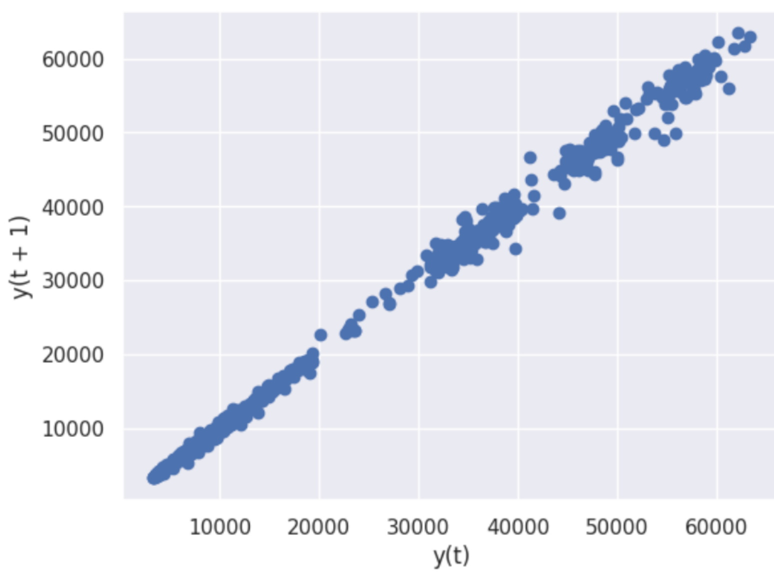


This code aims to visualize the BTC closing daily price for the years 2018 to 2020, normalized per year. First, the variable norm_years is created by normalizing the years using min-max scaling. This ensures that the years are scaled between 0 and 1, making them suitable for visualization. Then, plt.matshow is used to plot the matrix of the normalized years. The input norm_years[yrs].dropna().T selects the relevant years, drops any missing values, and transposes the matrix for better visualization.The y-axis represents the years, and the x-axis represents the days. Finally, plt.show() is used to display the plot.

Lag Plots

- Time series data implies a relationship between the value at a time t+1 and values at previous points in time.
- The step size we take to go back in time is called lag (lag of 1, lag 2 etc).
- Pandas provides the lag plot method. Let's examine the plot first.

```
from pandas.plotting import lag_plot

lag_plot(btc_df.groupby(pd.Grouper(freq='D')).Close.mean())

plt.show()
```

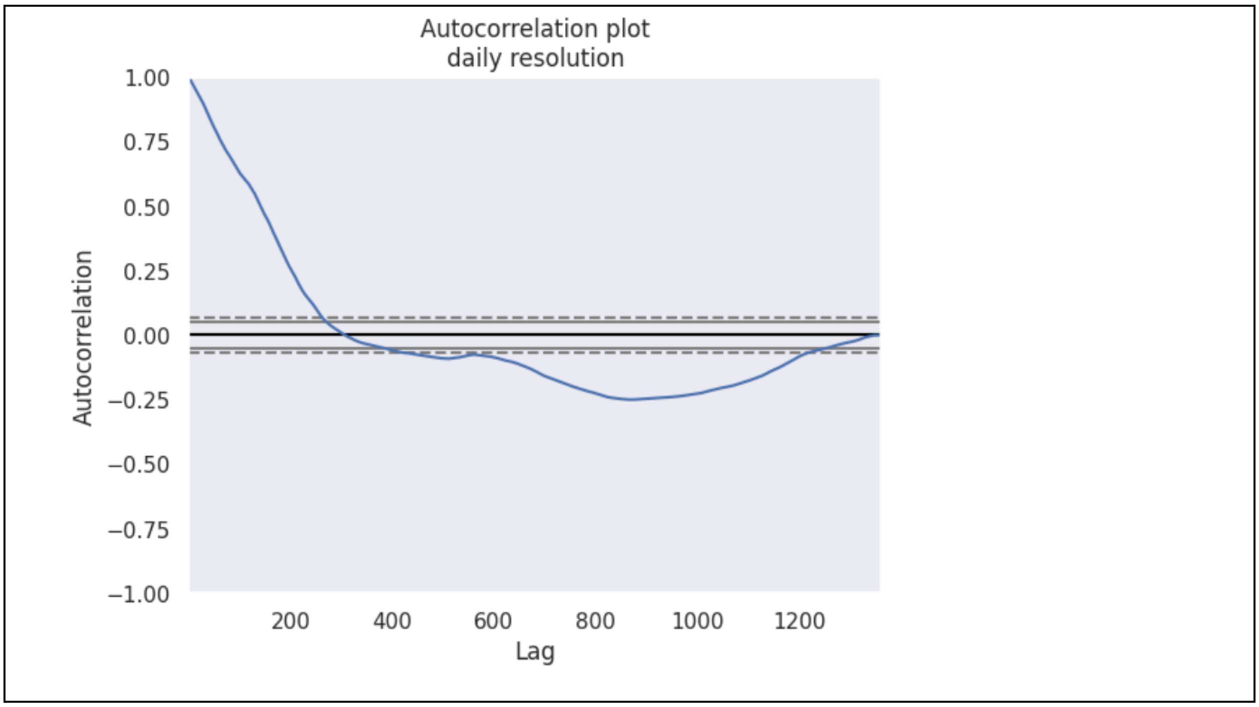


The lag plot visualizes the relationship between each data point and its lagged counterpart. It helps to identify any autocorrelation or patterns in the data.The resulting plot is displayed using plt.show(). It provides insights into the temporal dependency or lack thereof in the BTC closing prices, indicating if previous values have an impact on the current values.

Autocorrelation plots

For a lag=1, we can compute the correlation between the current time step value and the previous time step value. If we have n time steps in our data, we'll have n-1 correlation values. These values can be anywhere in the interval [-1,1].

- -1 (strongest negative correlation)
- 0 (no relationship at all)
- 1 (strongest positive correlation)



White noise

A time series is white noise if the variables are independent and identically distributed with a mean of zero.

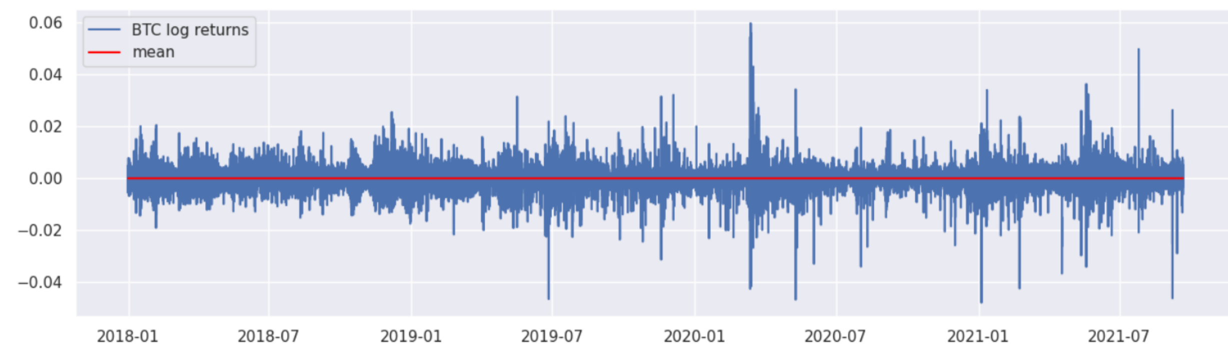
White noise is important in time series forecasting for two reasons:

1. Prediction: If a time series is a white noise, then it's by definition random and cannot be predicted.

2. Diagnosis: The errors of a time series model should be white noise. What does this mean? That the error contains no information, as all the information from the time series was harnessed by the model itself. And the opposite? If the errors are not white noise, the model can be improved further.

```
lret_btc = log_returns(btc_df.Close,1)[1:]
```

```
f = plt.figure(figsize = (15,4))

plt.plot(lret_btc)
plt.plot(lret_btc.index, [lret_btc.mean()] * len(lret_btc), color='red')
plt.legend(['BTC log returns', 'mean'])

plt.show()
```



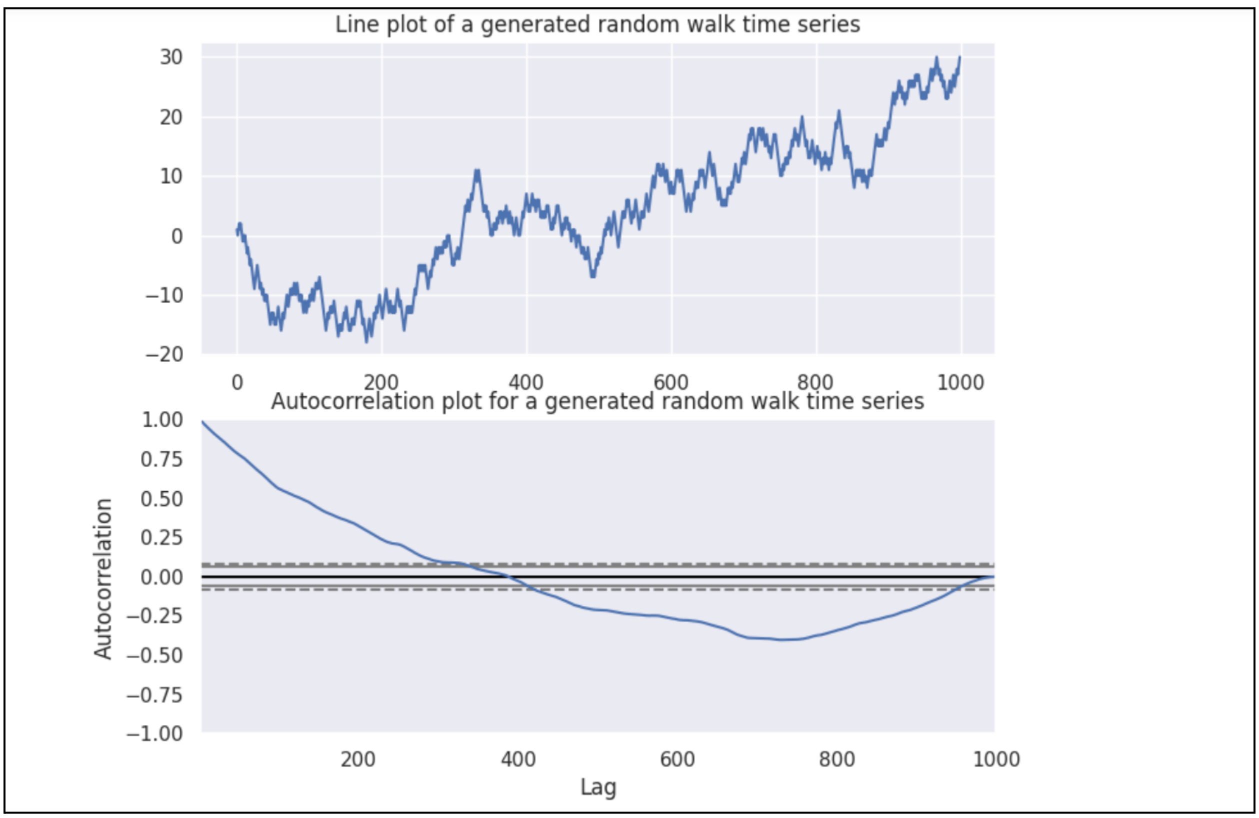
The code creates a figure with a size of 15 inches by 4 inches. It plots the log returns of BTC (lret_btc) using plt.plot. A horizontal line representing the mean of the log returns is also plotted using [lret_btc.mean()] * len(lret_btc). The legend is added to the plot, labeling the BTC log returns as well as the mean line.

Random walk

- First, let's see a perfect random walk dataset and then we'll look at our own data again.
- A time series is constructed through a *random walk* process as follows:
  y(t) = X(t-1) + rnd_step,
  where rnd_step is randomly selected from {-1, 1} and os is x(0).

```python
from random import seed
from random import random

seed(101)

values = [-1 if random() < 0.5 else 1] # x(0)

for i in range(1, 1000):
    rnd_step = -1 if random() < 0.5 else 1
    y_t = values[i-1] + rnd_step
    values.append(y_t)

plt.figure(figsize=(8,7))

# linear plot
plt.subplot(211)
plt.plot(values)
plt.title('Line plot of a generated random walk time series')

# correlogram
plt.subplot(212)
autocorrelation_plot(values)
plt.title('Autocorrelation plot for a generated random walk time series')

plt.show()
```

This code generates a random walk time series of length 1000. It starts with an initial value of -1 or 1 and then adds a random step of -1 or 1 at each time step. The time series is plotted using a line plot in the first subplot of a figure with size 8x7. The line plot visualizes the random walk pattern. In the second subplot, an autocorrelation plot is generated using the autocorrelation_plot function.



It shows the correlation between the time series and itself at different lags. The figure generated by the code above displays both the line plot and the autocorrelation plot, providing insights into the random nature and autocorrelation properties of the generated random walk time series.

Time series decomposition

A time series is conceptualized as having these types of components:

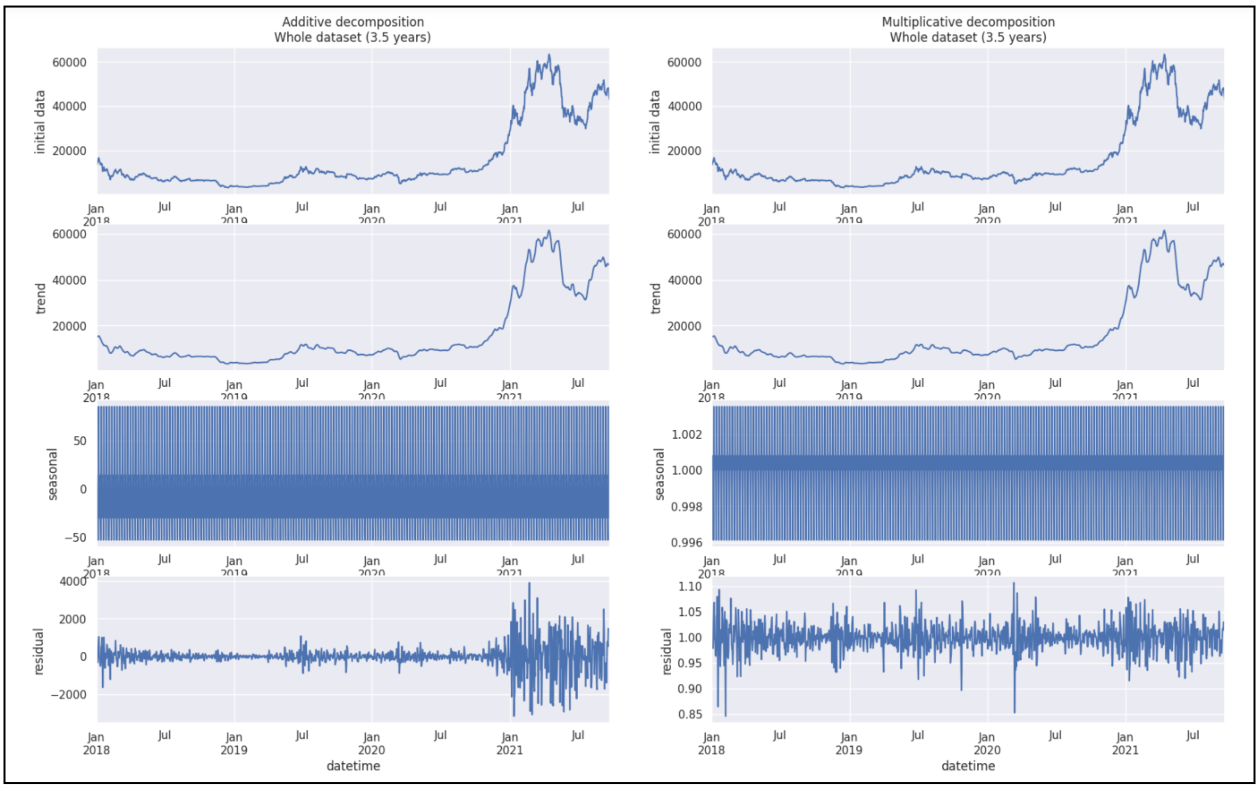
1. systematic components
    - level = overall average value
    - trend = temporary upward or downward movement
    - seasonality = a short-term cycle that repeats itself
2. non-systematic components
    - random noise

The 4 components are thought to combine in two possible ways into a time series:

- additive
  Close(t) = level + trend + seasonality + noise
- multiplicative
  *Close(t) = level * trend * seasonality * noise*

The below code performs seasonal decomposition of the given Bitcoin data (btc_days_df) using both additive and multiplicative models. In the first part, the data is divided into a smaller time window of length 160. The additive decomposition is applied using seasonal_decompose and plotted in four subplots: initial data, trend, seasonal, and residual. In the second part, the same steps are repeated with the multiplicative decomposition model.The resulting figure has a size of 20 inches by 12 inches. Each decomposition is visualized in its corresponding subplot, allowing for comparisons between the additive and multiplicative approaches.

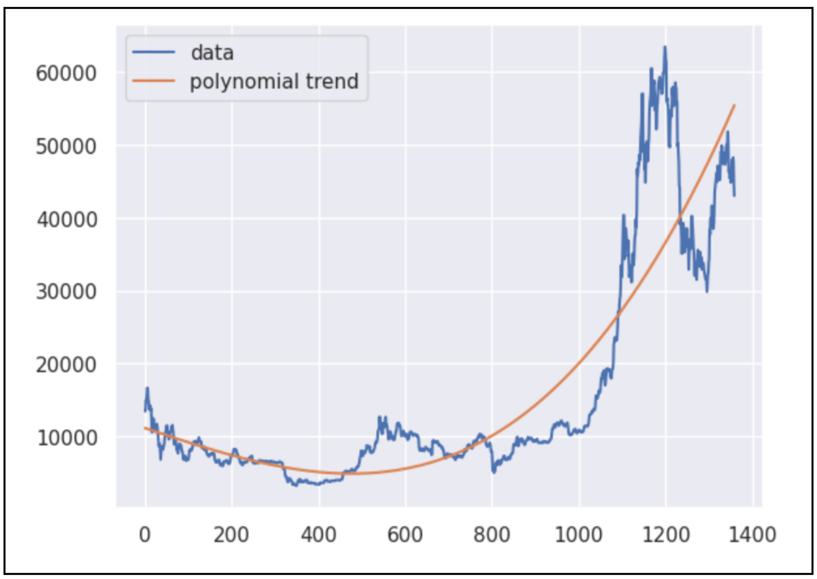
```python
from statsmodels.tsa.seasonal import seasonal_decompose

##whole data
data = btc_days_df
decomp = seasonal_decompose(data, model='additive')

plt.figure(figsize=(20,12))

plt.subplot(421)
data.plot()
plt.ylabel('initial data')
plt.title('Additive decomposition\nWhole dataset (3.5 years)')

plt.subplot(423)
decomp.trend.plot()
plt.ylabel('trend')

plt.subplot(425)
decomp.seasonal.plot()
plt.ylabel('seasonal')

plt.subplot(427)
decomp.resid.plot()
plt.ylabel('residual')

##small window
data = btc_days_df
decomp = seasonal_decompose(data, model='multiplicative')

plt.subplot(422)
data.plot()
plt.ylabel('initial data')
plt.title('Multiplicative decomposition\nWhole dataset (3.5 years)')
```



Removing Trends:
Need to look into the trend of the dataset:

- can inform us which modeling algorithm we can use
- we could remove the identified trend and simplify prediction (remove information)
- trend information can become an extra feature for training (add information)

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from scipy.optimize import curve_fit

data = btc_days_df

# create our independent variable; our X values are the timesteps,
# but we can make it just as well an array from 0 to ..., since the
# actual value does not matter
X = [i for i in range(0, len(data))]
X = np.reshape(X, (len(X), -1))

y = data.values # dependent variable is the closing price

pf = PolynomialFeatures(degree=3) # a cubic polynomial model
Xp = pf.fit_transform(X)          # transform X into a quadratic form (each row i will be: x[i]^0 x[i]^1 x[i]^2 x[i]^3)

# fit the quadratic model through ordinary least squares Linear Regression
md2 = LinearRegression()
md2.fit(Xp, y)

trendp = md2.predict(Xp)

plt.plot(X, y)
plt.plot(X, trendp)
plt.legend(['data', 'polynomial trend'])
plt.show()
```



This code performs polynomial regression by fitting a cubic polynomial model to the closing prices of the Bitcoin data. It visualizes the original data and the fitted polynomial trend, allowing for an assessment of how well the trend captures the overall pattern in the data.

Removing Seasonality:

Seasonality is a short-term pattern that repeats itself at a fixed frequency. A one-time cycle is just that, a cycle, not seasonality.

Its effect:

- may obscure the pattern in our data (and we remove it)
- can be picked up by our modeling algorithm (we can use it as extra feature)

  Both are valid approaches.

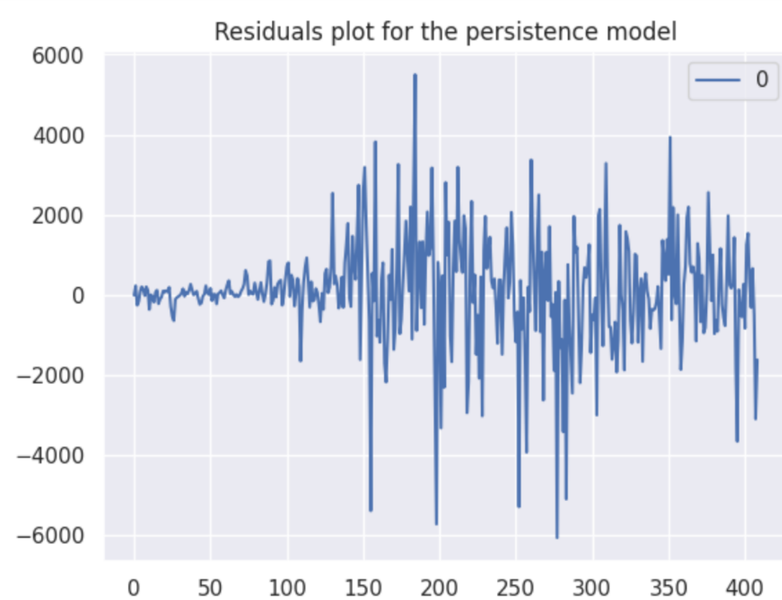
```
groups = btc_days_df.groupby(pd.Grouper(freq='A'))
days_per_year = []
for name, group in groups:
    days_per_year.append(len(group.values))

plt.figure(figsize=(12,20))

start = 0
years = [2018, 2019, 2020, 2021]
count = len(days_per_year)

legend = ['Detrended series', 'Fitted weekly seasonality', 'Fitted monthly seasonality', 'Fitted trimester seasonality']

for i, days in enumerate(days_per_year):
    series = detrpoly[start:start+days]

    plt.subplot(count*100 + 10 + i+1)
    plt.title(f'Year {years[i]}')

    plt.plot(series)

    intervals = [7, 30, 90]
    cols = ['red', 'green', 'magenta']

    for cnt, inter in enumerate(intervals):
        X = [i%inter for i in range(0, len(series))] # let's try to model a weekly seasonality as a sinusoid
        y = series

        degree = 4

        coef = np.polyfit(X, y, degree)

        # create curve
        curve = list()
```

The code begins by grouping a given dataframe (btc_days_df) by year using pd.Grouper(freq='A'). It then calculates the number of days per year and stores them in the days_per_year list. A new figure is created with a size of 12 inches by 20 inches, preparing for the visualization of the results. Subsequently, a loop iterates over the selected years, creating subplots for each year with appropriate titles. The detrended series for each year is extracted and plotted.

Model Training and Evaluation:

In this module, a model is trained with the dataset chosen and the model is evaluated with the testing data.

```
# calculate residuals
residuals = [y_test[i]-preds[i] for i in range(len(preds))]
residuals = pd.DataFrame(residuals)

residuals.plot()
plt.title('Residuals plot for the persistence model')
plt.show()
```

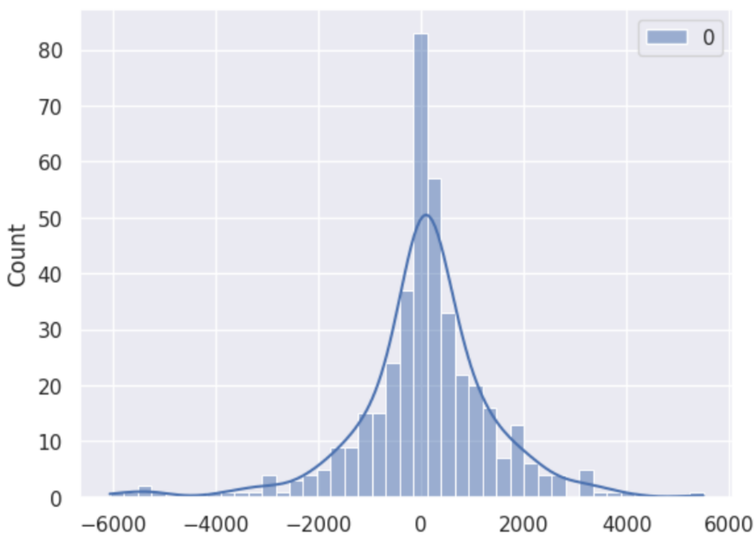


The code calculates the residuals between the actual and predicted values, creates a line plot of the residuals, and adds a title to the plot. The resulting plot provides insights into the patterns or deviations in the model's predictions compared to the actual values.

Residuals distribution:

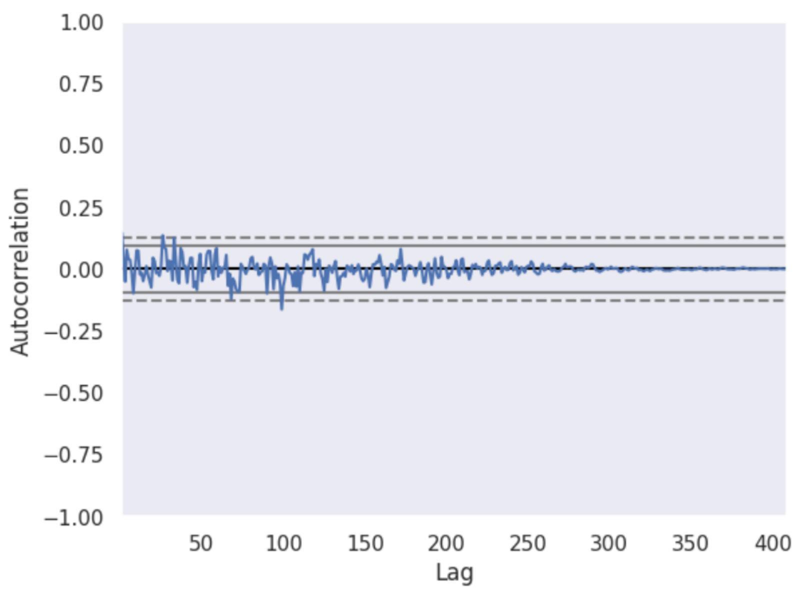
The distribution should be Gaussian, otherwise it means there's something wrong with our model.

```
import seaborn as sns

sns.histplot(data = residuals, kde=True)

plt.show()
```



The histplot function from Seaborn is used to generate the plot, with the data parameter specifying the input data and kde=True enabling the kernel density estimate. Finally, plt.show() from Matplotlib is called to display the histogram plot with the KDE.

```
# Autocorrelation plot of residuals
# We should find no autocorrelation. If there is some, we should improve our model and capture it.
autocorrelation_plot(residuals)
plt.show()
```



This code creates an autocorrelation plot to analyze the correlation between a series of residuals. The resulting plot helps in identifying any significant lagged relationships or patterns in the residual data.

LSTM MODEL:

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture that excels in handling sequential data by capturing long-range dependencies and overcoming the vanishing gradient problem. It is widely used in various domains such as natural language processing, time series analysis, and speech recognition.LSTM models can effectively handle input sequences of varying lengths, making them flexible for processing data with different temporal patterns. By considering both past and current information, LSTMs can capture the context and contextually adapt their predictions, leading to improved performance in tasks requiring temporal understanding

```python
def normalise_zero_base(df):
    """ Normalise dataframe column-wise to reflect changes with respect to first entry. """
    return df / df.iloc[0] - 1

def normalise_min_max(df):
    """ Normalise dataframe column-wise min/max. """
    return (df - df.min()) / (data.max() - df.min())
```

```python
def extract_window_data(df, window_len=10, zero_base=True):
    """ Convert dataframe to overlapping sequences/windows of len `window_data`.

        :param window_len: Size of window
        :param zero_base: If True, the data in each window is normalised to reflect changes
            with respect to the first entry in the window (which is then always 0)
    """
    window_data = []
    for idx in range(len(df) - window_len):
        tmp = df[idx: (idx + window_len)].copy()
        if zero_base:
            tmp = normalise_zero_base(tmp)
        window_data.append(tmp.values)
    return np.array(window_data)
```

The extract_window_data function converts a dataframe into overlapping sequences or windows of a specified length. It takes parameters for the dataframe, window length, and whether to normalize the data with respect to the first entry. A list called window_data is initialized to store the resulting window sequences. The code iterates through the dataframe, extracts each window, and optionally normalizes it. The function returns the window_data list converted to a numpy array.

```python
def build_lstm_model(input_data, output_size, neurons=20, activ_func='linear',
                     dropout=0.25, loss='mae', optimizer='adam'):
    model = Sequential()

    model.add(LSTM(neurons, input_shape=(input_data.shape[1], input_data.shape[2])))
    model.add(Dropout(dropout))
    model.add(Dense(units=output_size))
    model.add(Activation(activ_func))

    model.compile(loss=loss, optimizer=optimizer)
    return model
```

The build_lstm_model function constructs and returns an LSTM model. The model is defined as a sequential stack of layers using the Keras library. It adds an LSTM layer with a specified number of neurons and input shape based on the input data dimensions. A dropout layer is added to prevent overfitting by randomly setting input units to 0 during training. A dense layer is added to produce the desired output size. An activation function is applied to introduce non-linearity to the model's output. The model is compiled with a specified loss function and optimizer before being returned.

```
model = build_lstm_model(
    X_train, output_size=1, neurons=lstm_neurons, dropout=dropout, loss=loss,
    optimizer=optimizer)
history = model.fit(
    X_train, y_train, epochs=epochs, batch_size=batch_size, verbose=1, shuffle=True)
```

```
14063/14063 [==============================] - 62s 4ms/step - loss: 32.2641
Epoch 62/70
14063/14063 [==============================] - 63s 4ms/step - loss: 31.6802
Epoch 63/70
14063/14063 [==============================] - 62s 4ms/step - loss: 30.6356
Epoch 64/70
14063/14063 [==============================] - 63s 4ms/step - loss: 29.9255
Epoch 65/70
14063/14063 [==============================] - 62s 4ms/step - loss: 29.0628
Epoch 66/70
14063/14063 [==============================] - 63s 4ms/step - loss: 28.3972
Epoch 67/70
14063/14063 [==============================] - 62s 4ms/step - loss: 28.3484
Epoch 68/70
14063/14063 [==============================] - 63s 4ms/step - loss: 30.3535
Epoch 69/70
14063/14063 [==============================] - 62s 4ms/step - loss: 27.8341
Epoch 70/70
14063/14063 [==============================] - 62s 4ms/step - loss: 27.3837
```

```
model.save("lstm_crypto.h5")
```

This code builds and trains an LSTM model using the provided training data. The model's architecture is defined, and the training process updates the model's parameters to minimize the specified loss function. The training history is stored for further analysis and evaluation.

Test set Prediction & error

```
targets = test[target_col][window_len:]
preds = model.predict(X_test).squeeze()
```

```
3125/3125 [==============================] - 11s 3ms/step
```

```
mean_absolute_error(preds, y_test)
```

```
26.074582964929455
```

The code selects the actual target values, generates predictions using a trained model, and calculates the MAE between the predicted values and the actual targets as a measure of prediction accuracy.

Predicting Stream Data

```
endpoint = 'https://min-api.cryptocompare.com/data/histoday'
res = requests.get(endpoint + '?fsym=BTC&tsym=USD&limit=2000')
hist = pd.DataFrame(json.loads(res.content)['Data'])
hist = hist.set_index('time')
hist.index = pd.to_datetime(hist.index, unit='s')
```

```
hist.head()
print(hist.shape)
```

```
(2001, 8)
```

```
hist.drop(['conversionType', 'conversionSymbol'],axis=1,  inplace=True)
hist
```

time	high	low	open	volumefrom	volumeto	close
2017-11-23	8266.55	8012.35	8234.50	68010.70	5.554651e+08	8013.41
2017-11-24	8332.94	7900.17	8013.38	72994.63	5.957104e+08	8200.80
2017-11-25	8761.98	8153.70	8203.45	84670.41	7.184837e+08	8754.69
2017-11-26	9474.62	8746.56	8754.62	85891.98	7.825000e+08	9318.42
2017-11-27	9733.61	9316.84	9318.42	106902.79	1.025176e+09	9733.20

The code fetches historical daily price data for Bitcoin in USD from the CryptoCompare API. The data is retrieved using an API endpoint and specified parameters. The response is converted into a Pandas DataFrame called hist with timestamps as the index. Unnecessary columns are dropped from the DataFrame. The resulting DataFrame contains the historical price data for Bitcoin in USD.

```
model = keras.models.load_model('/content/drive/MyDrive/NM-AI/lstm-crypto1.h5')
train, test, X_train, X_test, y_train, y_test = prepare_data(hist, target_col, window_len=window_len, zero_base=zero_base, test_s
```

```
targets = test[target_col][window_len:]
preds = model.predict(X_test).squeeze()
```

```
60/60 [==============================] - 1s 4ms/step
```

```
mean_absolute_error(preds, y_test)
```

```
0.02644908540510029
```

```
preds = test[target_col].values[:-window_len] * (preds + 1)
preds = pd.Series(index=targets.index, data=preds)

line_plot(targets, preds, 'actual', 'prediction', lw=3)
```

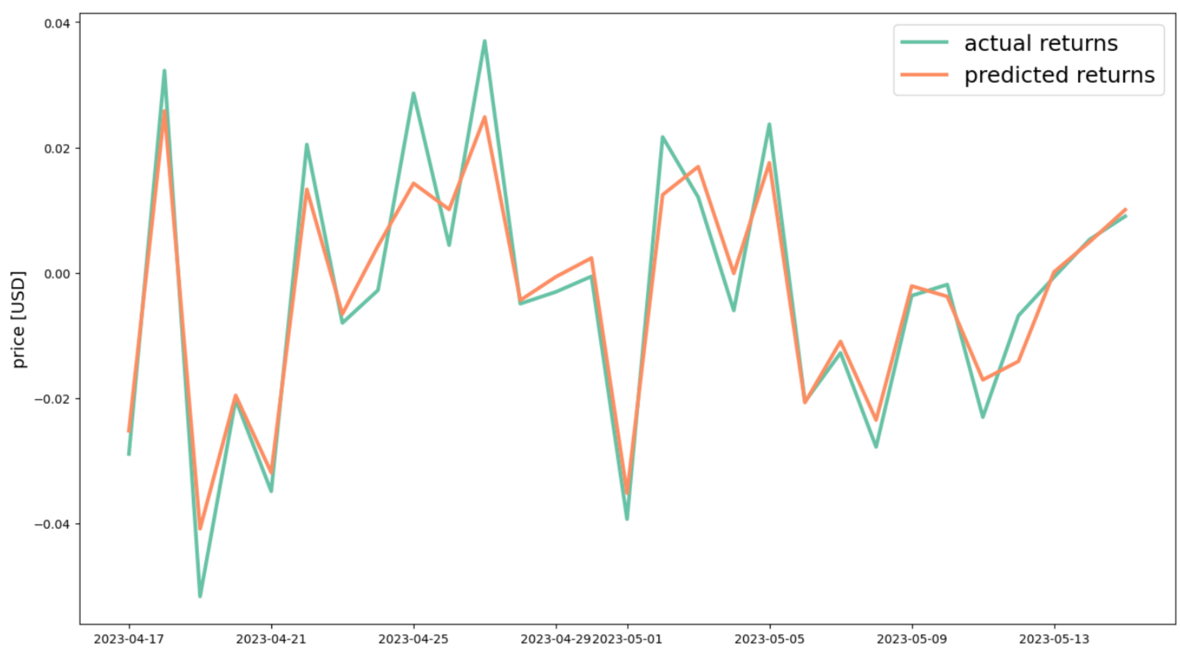
The code loads a pre-trained LSTM model for cryptocurrency price prediction. It prepares the data by splitting it into training and testing sets. Using the loaded model, it predicts the target values for the test data. The actual target values and predicted values are stored in targets and preds variables, respectively.

Results:

Plot for the results predicted by the model in real time for a stream of crypto coin data as input and the return value(closing value) of that coin given as output is shown below.

This code is used to plot a line chart comparing the actual returns and the predicted returns. The actual_returns and predicted_returns are selected based on the last n_points values, with a shift applied to the predicted returns to align them with the subsequent actual returns for comparison. The plot will have the y-axis labeled as "actual returns" and "predicted returns", and the line width of the plot will be set to 3.

```
line_plot(actual_returns[-n_points:][:-1], predicted_returns[-n_points:].shift(-1),
          'actual returns', 'predicted returns', lw=3)
```



References:

[1] Peng, Z. (2019, January). Stocks analysis and prediction using big data analytics. In *2019 international conference on intelligent transportation, big data & smart City (ICITBS)* (pp. 309-312). IEEE.

[2] Dataset: [G-Research Crypto Forecasting | Kaggle]