

Final Project Report

Team Member Contributions:

Byeonghwa Jung – problem statement, dataset description, EDA of the dataset about Bitcoin and Ethereum prices

Jiyeon Cheon – EDA of the dataset regarding average gas fees

Eldor Fozilov – data preprocessing and model building / evaluation

Problem Statement

Interest in cryptocurrency is hot these days. Cryptocurrency is a combination of crypto, which means encryption, and currency, which is a virtual asset that can be safely transmitted through public key encryption in a distributed ledger and easily prove ownership using hash functions. In general, cryptocurrency operates based on blockchain, and Bitcoin and Ethereum are representatives.

In the case of Ethereum, you have to pay a kind of fee when exchanging coins or signing a smart contract. The fee at this time is called the ‘Gas Fee’. The cost of this gas fluctuates from time to time, as the network becomes congested, the cost of gas increases and vice versa. By introducing these gas costs, Ethereum plays a role in preventing network overload due to transactions. As many people make transactions in the Ethereum network, gas costs have become more and more important. The amount can be determined by the user who generates the transaction, such as paying a high gas bill for those who want their transaction to be executed quickly and paying a low gas bill for those who do not. In general, many people want fast transmission speed, so the burden of paying for gas is bound to increase. With this in mind, we tried to predict the changing gas cost through time-series analysis so that the gas cost could be more affordable.

Dataset

We collected the Ethereum gas data from April 26, 2022, to December 11, 2022. The reason why we use the data for the last six months is that the previous data showed quite different dynamics from our data, such as having a much higher mean and a larger variance.

The dataset contained both the highest and lowest prices as well as the open price and closing price of Ethereum gas fee by time. The average gas fee data by hour was extracted through the preprocess, which will be described later.

	open	close	low	high	avgGas	timestamp
0	51.4000	50.4000	30.2900	63.0500	80868.519046	2022-04-26T13:34:21.641Z
1	74.9800	49.4400	36.8900	119.4100	85449.256791	2022-04-26T14:00:09.549Z
2	59.3500	72.7700	42.2300	121.5400	82192.413722	2022-04-26T15:00:08.630Z
3	82.8200	70.2500	36.0000	98.4700	82451.786800	2022-04-26T16:01:05.379Z
4	68.5000	59.6100	27.3600	84.8300	84236.446559	2022-04-26T17:00:58.575Z
...
5479	14.3790	14.0238	12.0000	18.1021	166736.391980	2022-12-11T05:00:28.462Z
5480	14.8214	14.6782	12.0000	17.8660	157472.907955	2022-12-11T06:00:33.482Z
5481	14.6852	14.7600	12.5715	20.0906	115399.969702	2022-12-11T07:00:37.563Z
5482	13.7311	13.6790	12.2000	16.6647	119084.124717	2022-12-11T08:00:42.075Z
5483	14.2724	14.1978	12.1000	18.4967	151872.965195	2022-12-11T09:00:46.209Z

5484 rows × 6 columns

We additionally collected bitcoin and Ethereum price data to identify exogenous factors affecting Ethereum gas fee. The data included BTC and ETH prices from September 12, 2022, to December 11, 2022. Therefore, for the analysis of the exogenous factors, we further created one dataset that made the existing gas fee data equal to the time scale of the BTC and ETH price data.

UNIX Timestamp	Timestamp	Bitcoin Price (USD)	Ethereum Price (USD)	open	close	low	high	avgGas	timestamp
0	1.660000e+12	22195.69118	1750.655946	5.9900	9.6800	4.2500	16.9200	92232.945200	2022-09-12T10:00:01.499Z
1	1.660000e+12	22180.53379	1746.531098	52.9700	5.5400	4.1300	56.3300	88143.438267	2022-09-12T11:00:03.234Z
2	1.660000e+12	22292.74291	1751.864674	20.0000	50.9900	10.6600	56.1600	82982.461422	2022-09-12T12:00:04.915Z
3	1.660000e+12	22359.17007	1747.678093	18.2600	26.9600	13.3200	37.1300	84544.989353	2022-09-12T13:00:06.988Z
4	1.660000e+12	22391.44726	1752.387514	30.8300	16.9700	12.7300	91.7600	89251.971497	2022-09-12T14:00:10.398Z
...
2150	1.670000e+12	17177.26840	1273.466338	14.3790	14.0238	12.0000	18.1021	166736.391980	2022-12-11T05:00:28.462Z
2151	1.670000e+12	17182.10721	1274.061867	14.8214	14.6782	12.0000	17.8660	157472.907955	2022-12-11T06:00:33.482Z
2152	1.670000e+12	17188.39360	1275.846801	14.6852	14.7600	12.5715	20.0906	115399.969702	2022-12-11T07:00:37.563Z
2153	1.670000e+12	17187.42631	1275.330135	13.7311	13.6790	12.2000	16.6647	119084.124717	2022-12-11T08:00:42.075Z
2154	1.670000e+12	17174.48064	1276.717978	14.2724	14.1978	12.1000	18.4967	151872.965195	2022-12-11T09:00:46.209Z

2155 rows × 4 columns

2158 rows × 6 columns

Data Preprocessing

For Average Gas Fee data, we first generated average values. Since our aim is to predict average gas fee, we generated average gas fee value with the lowest and highest fee values.

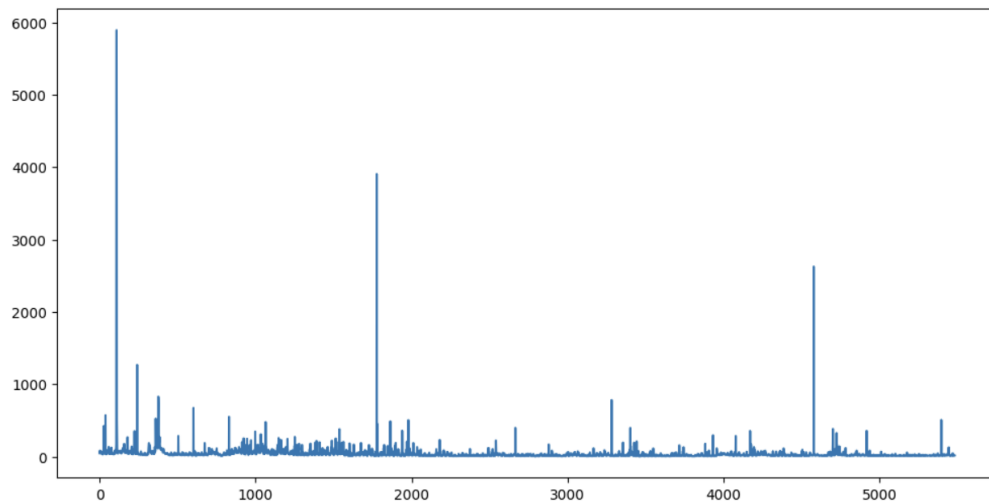
```
gas_average = gas_dataset.apply(lambda x: round((x['low'] + x['high']) / 2, 4), axis = 1)
```

```
gas_average.describe()
```

```
count    5484.000000
mean      38.881064
std      140.416634
min        3.520000
25%      13.560925
50%      21.802500
75%      39.437500
max     5892.560000
dtype: float64
```

Then we processed outliers and null values. There were noticeably high values after plotting the data of gas_average.

```
gas_average.plot(figsize = (12,6));
```

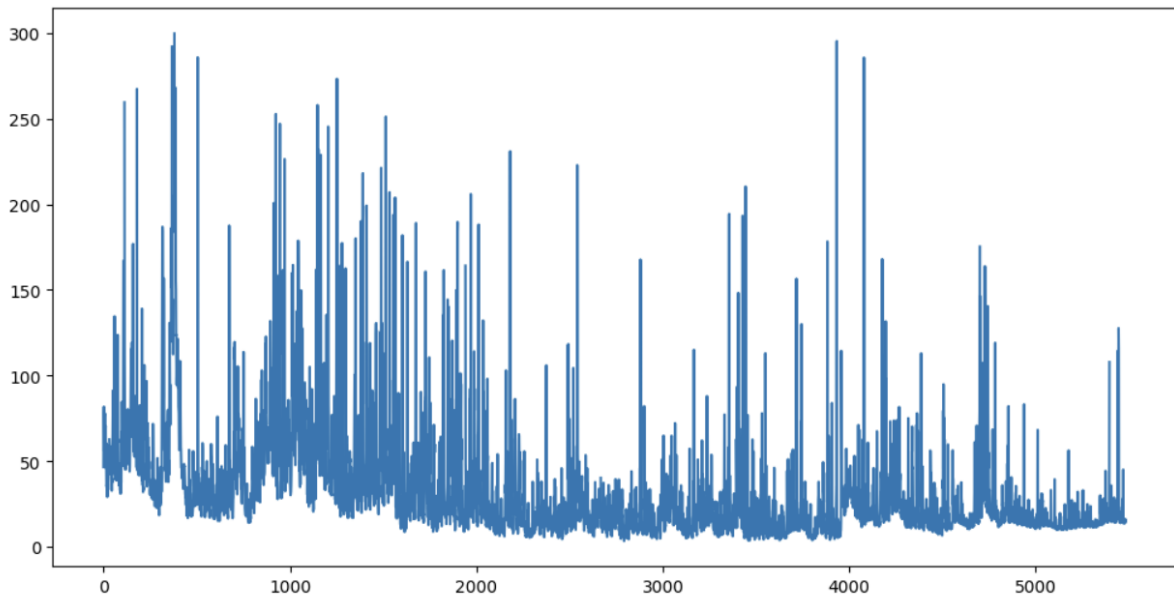


So, we deleted them with some range-at first, less than 1000 and then less than 300.

```
gas_average = gas_average.apply(lambda x: x if x < 1000 else np.nan)
```

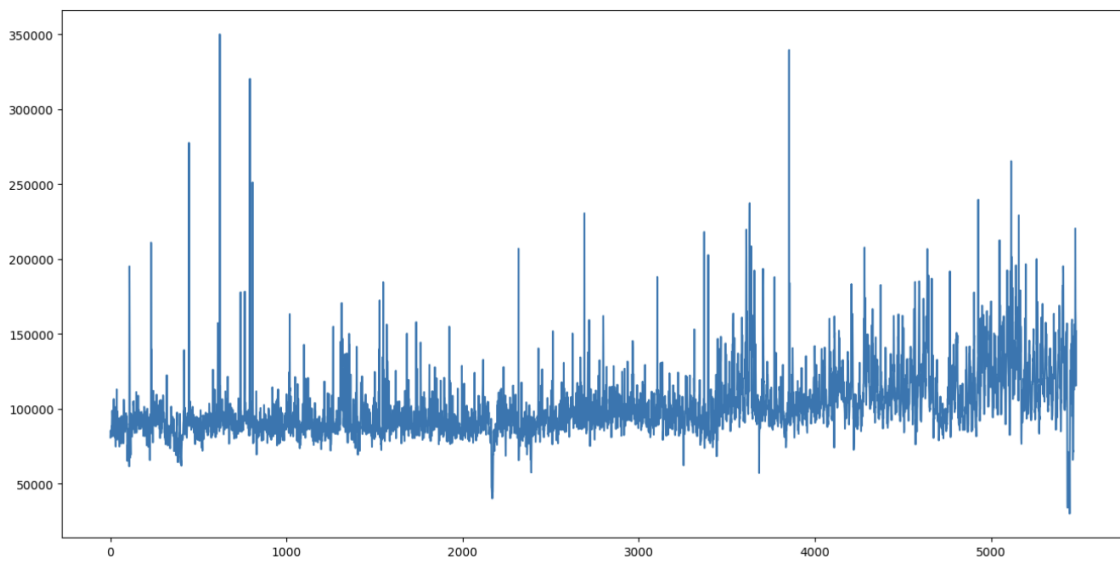
```
gas_average = gas_average.apply(lambda x: x if x < 300 else np.nan)
```

```
gas_average.plot(figsize = (12,6));
```



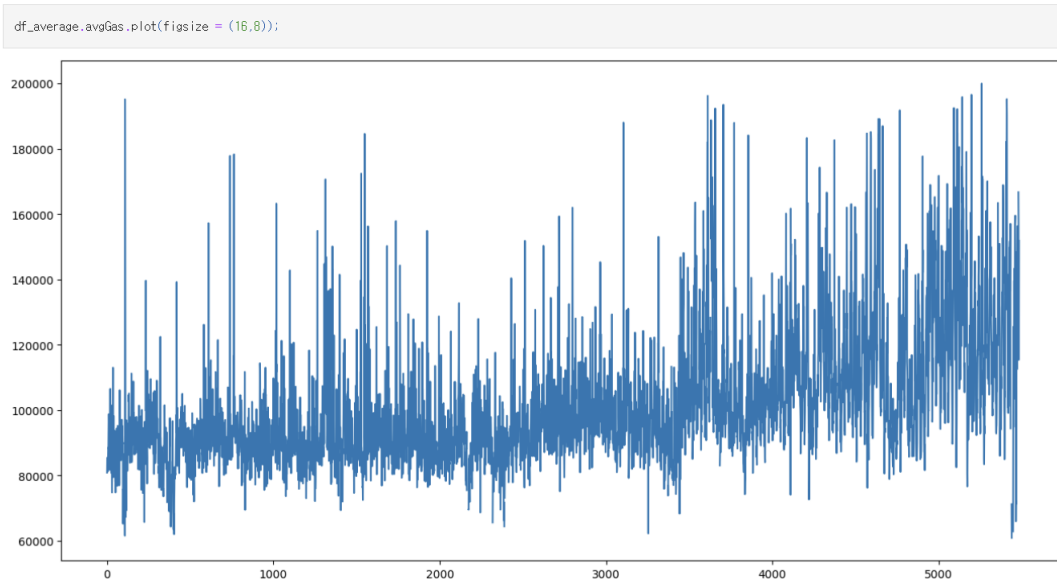
There were also noticeably high values after plotting the data of avgGas.

```
df_average.avgGas.plot(figsize = (16,8));
```



So, we deleted them with some range-less than 200000 and greater than 60000.

```
df_average['avgGas'] = df_average.avgGas.apply(lambda x: x if x < 200000 and x > 60000 else np.nan)
```



We also checked null values to avoid error.

```
df_average['average_gas_fee'] = df_average['average_gas_fee'].fillna(method = 'ffill')
df_average['avgGas'] = df_average['avgGas'].fillna(method = 'ffill')
```

```
sum(df_average['average_gas_fee'].isnull() == True)
```

0

Then we transformed the timestamp. Since only the information of hour part was needed, we extracted it.

```
# timestamp transformation
def fun(x):
    x = x.replace('T', ' ')[:-11]
    return pd.to_datetime(x, format = '%Y-%m-%d %H')

df_average['timestamp'] = df_average['timestamp'].apply(fun)
```

```
check = []
for index in range(len(df_average) - 1):
    var1 = int(str(df_average['timestamp'][index])[-8:-6]) # extract hour
    var2 = int(str(df_average['timestamp'][index+1])[-8:-6])
    if (var2 - var1) % 24 != 1:
        check.extend([index, index + 1])
```

And there were some empty values, so we added those with the average of their before and after values. Also, we deleted some data that exceeded the intended time range.

```
df_average.loc[2568,5] = [31.2000,96767.948429, pd.to_datetime('2022-08-11 20:00:00')]
df_average.loc[3999,5] = [42.8250, 126796.040075, pd.to_datetime('2022-10-10 12:00:00')]
df_average.loc[4893,5] = [31.8526, 89847.465683, pd.to_datetime('2022-11-16 19:00:00')]
```

```
df_average = df_average.sort_index().reset_index(drop=True)
```

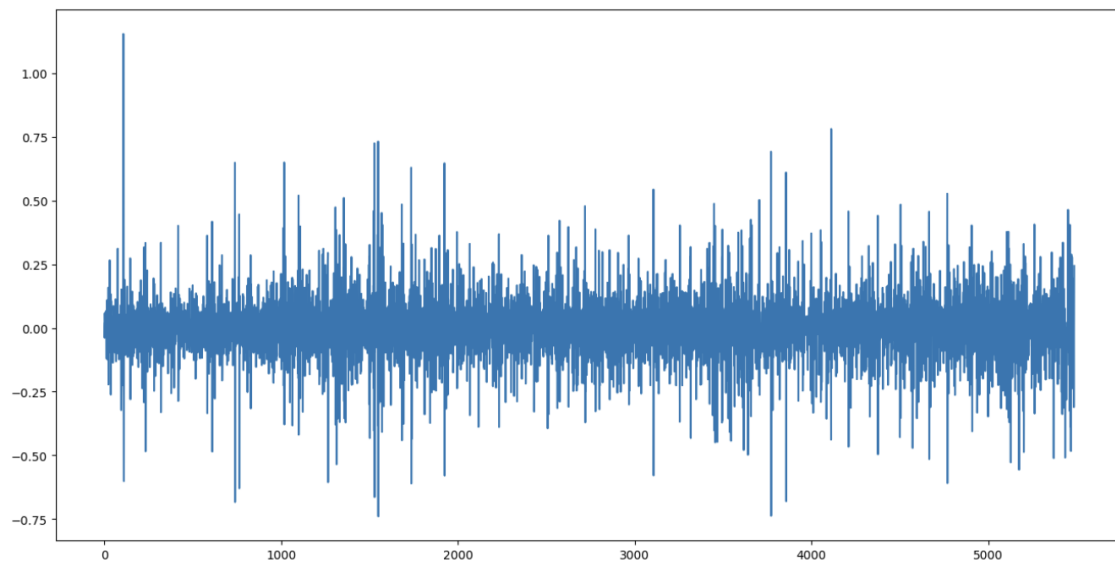
```
df_average.iloc[-24*90:] # data of the last 90 days (we only need data that start from hour 10
# and beyond, so next we remove samples 3324 and 3325)
```

	average_gas_fee	avgGas	timestamp
3327	10.5850	92232.945200	2022-09-12 10:00:00
3328	30.2300	88143.438267	2022-09-12 11:00:00
3329	33.4100	82982.461422	2022-09-12 12:00:00
3330	25.2250	84544.989353	2022-09-12 13:00:00
3331	52.2450	89251.971497	2022-09-12 14:00:00
...
5482	15.0511	166736.391980	2022-12-11 05:00:00
5483	14.9330	157472.907955	2022-12-11 06:00:00
5484	16.3310	115399.969702	2022-12-11 07:00:00
5485	14.4323	119084.124717	2022-12-11 08:00:00
5486	15.2983	151872.965195	2022-12-11 09:00:00

2160 rows × 3 columns

And since avgGas data has a changing trend, we took the first difference to make it stationary.

```
np.log(df_average.avgGas).diff(periods = 1).plot(figsize = (16,8));
```



Next, for Cryptocurrency Price data, we first processed an unnecessary column, 'UNIX timestamp' and transformed the timestamp by extracting the hour part.

```
crypto_price_df.drop(len(crypto_price_df) - 1, axis = 0, inplace = True)
crypto_price_df.drop('UNIX Timestamp', axis = 1, inplace = True)

crypto_price_df
```

	Timestamp	Bitcoin Price (USD)	Ethereum Price (USD)
0	2022-09-12 10:01	22195.69118	1750.655946
1	2022-09-12 11:03	22180.53379	1746.531098
2	2022-09-12 12:04	22292.74291	1751.864674
3	2022-09-12 13:01	22359.17007	1747.678093
4	2022-09-12 14:03	22391.44726	1752.387514
...
2149	2022-12-11 5:01	17177.70161	1270.787620
2150	2022-12-11 6:00	17177.26840	1273.466338
2151	2022-12-11 7:01	17182.10721	1274.061867
2152	2022-12-11 8:01	17188.39360	1275.846801
2153	2022-12-11 9:00	17187.42631	1275.330135

2154 rows x 3 columns

```
def fun2(x):
    x = x[:-3] # we don't need :mm part
    x = x.split(' ')
    if len(x[1]) == 2: # example: 10
        x = ' '.join(x)
    else:
        x = ' 0'.join(x)
    return pd.to_datetime(x, format = '%Y-%m-%d %H')

crypto_price_df['Timestamp'] = crypto_price_df['Timestamp'].apply(fun2)
```

There was some duplicate data, so we checked and deleted it.

```
def check_timestamps(df):
    check = []
    for index in range(len(df) - 1):
        var1 = int(str(df.loc[index, 'Timestamp'])[-8:-6]) # extract hour
        var2 = int(str(df.loc[index+1, 'Timestamp'])[-8:-6])
        if (var2 - var1) % 24 != 1:
            check.extend([index, index + 1])
    return check
```

```
crypto_price_df.iloc[check,:] # the data from 2022-10-07 19 p.m till 2022-10-08 7 a.m does not exist
# and the data for 2022-10-04 17:00 does not exist
```

	Timestamp	Bitcoin Price (USD)	Ethereum Price (USD)
410	2022-09-29 12:00:00	19498.02941	1339.494261
411	2022-09-29 12:00:00	19489.07544	1337.941709
411	2022-09-29 12:00:00	19489.07544	1337.941709
412	2022-09-29 12:00:00	19449.56404	1335.337360
536	2022-10-04 16:00:00	20086.85899	1353.505192
537	2022-10-04 18:00:00	19989.09172	1345.408492
550	2022-10-05 07:00:00	20228.98971	1357.485653
551	2022-10-05 07:00:00	20228.98971	1357.485653
611	2022-10-07 19:00:00	19457.30895	1324.894738
612	2022-10-08 07:00:00	19504.71322	1330.834875
1052	2022-10-26 15:00:00	20854.14207	1574.977942
1053	2022-10-26 15:00:00	20854.14207	1577.200400
1523	2022-11-15 05:00:00	16787.27903	1257.537171
1524	2022-11-15 05:00:00	16776.02538	1257.537171
1889	2022-11-30 10:00:00	16914.98974	1273.760018
1890	2022-11-30 10:00:00	16920.37796	1271.975367

```
crypto_price_df.drop([411, 412, 551, 1053, 1524, 1890], axis = 0, inplace = True)
```

And for empty values, we added a new row.

```
crypto_price_df.drop([411, 412, 551, 1053, 1524, 1890], axis = 0, inplace = True)
```

```
crypto_price_df.loc[536:537]
```

	Timestamp	Bitcoin Price (USD)	Ethereum Price (USD)
536	2022-10-04 16:00:00	20086.85899	1353.505192
537	2022-10-04 18:00:00	19989.09172	1345.408492

```
crypto_price_df.loc[536.5] = [pd.to_datetime('2022-10-04 17:00:00'), 20086.85899, 1353.505192] # add a new row (now the index became a float number)
```

Then, we generated train data. We combined average gas fee data and cryptocurrency price data and took the log difference of all values.

```
df_90days = pd.concat([df_average_90days, crypto_price_df], axis = 1, join = 'inner')
```

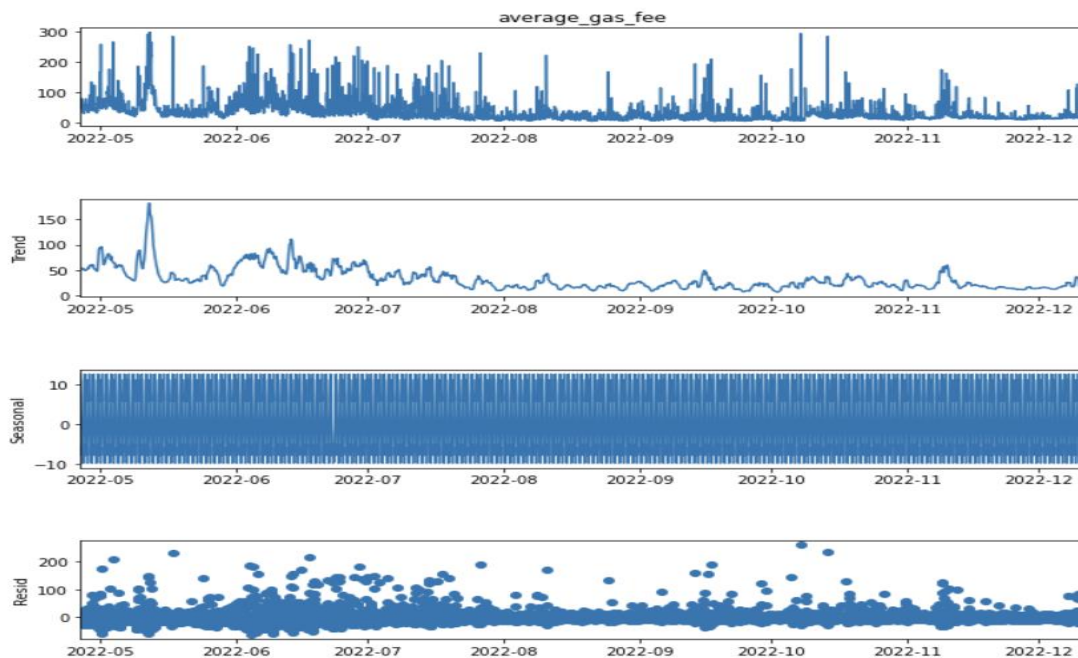


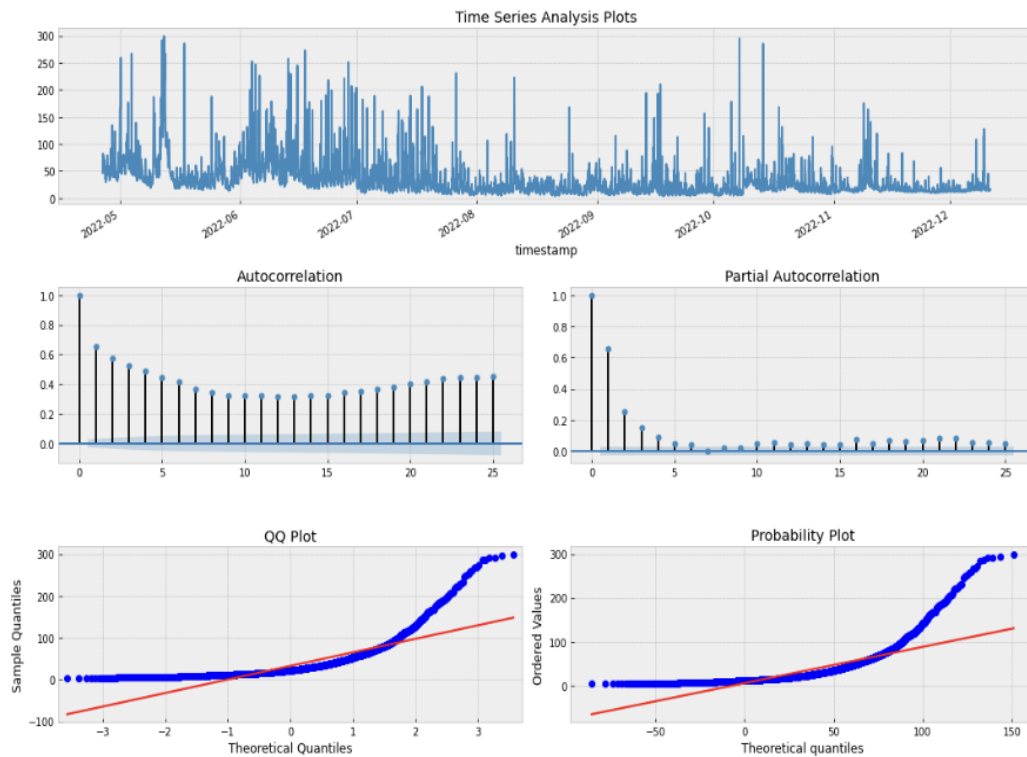
```
df_90days['average_gas_fee'] = np.log(df_90days['average_gas_fee'])
for col in df_90days.columns[1:]:
    df_90days.loc[:, col] = np.log(df_90days[col]).diff(periods = 1)
df_90days.drop(pd.to_datetime('2022-09-12 10:00:00'), axis = 0, inplace = True)
df_90days.columns = ['log average_gas_fee', 'log diff avgGas', 'log return (Bitcoin)', 'log return (Ethereum)']
df_90days.head()
```

	log average_gas_fee	log diff avgGas	log return (Bitcoin)	log return (Ethereum)
timestamp				
2022-09-12 11:00:00	3.408835	-0.045352	-0.000683	-0.002359
2022-09-12 12:00:00	3.508855	-0.060336	0.005046	0.003049
2022-09-12 13:00:00	3.227836	0.018655	0.002975	-0.002393
2022-09-12 14:00:00	3.955944	0.054180	0.001443	0.002691
2022-09-12 15:00:00	4.347565	-0.087952	-0.004134	-0.009437

EDA

1) EDA Of the Average Gas Fee Data





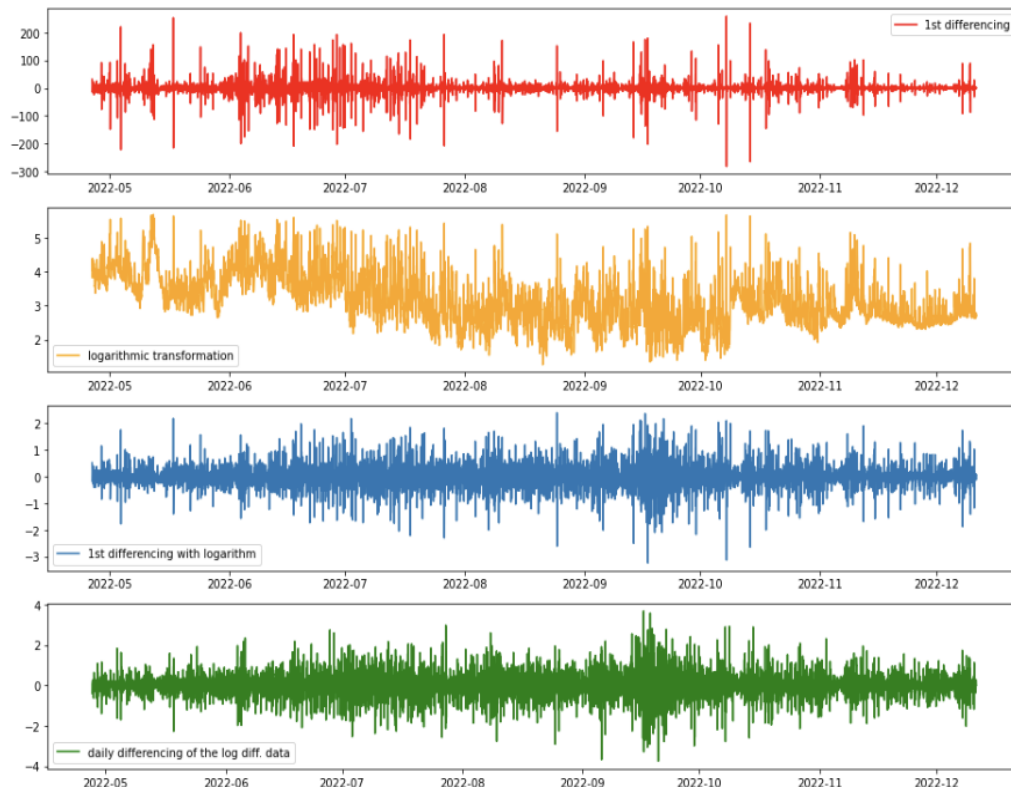
According to the results of data decomposition, first trend graph, it seems that our data has a decreasing trend. And to the seasonal part, our data has approximately 24-hour interval seasonality. It seems that the relatively high absolute value and volatility of the gas fee from May to July makes a relatively large residual. All values of ACF plots are positive, and far from the range. And in the QQ plot and probability plot, it is hard to say that our data follows the red line. Also, the right tail part is far away from the line. As a result, we can say that our data is non-stationary.

To reduce decreasing trend and keep variation width, we used logarithm transformation. And to consider that our data has a daily seasonality, daily differencing data was also compared. The results were as follows.

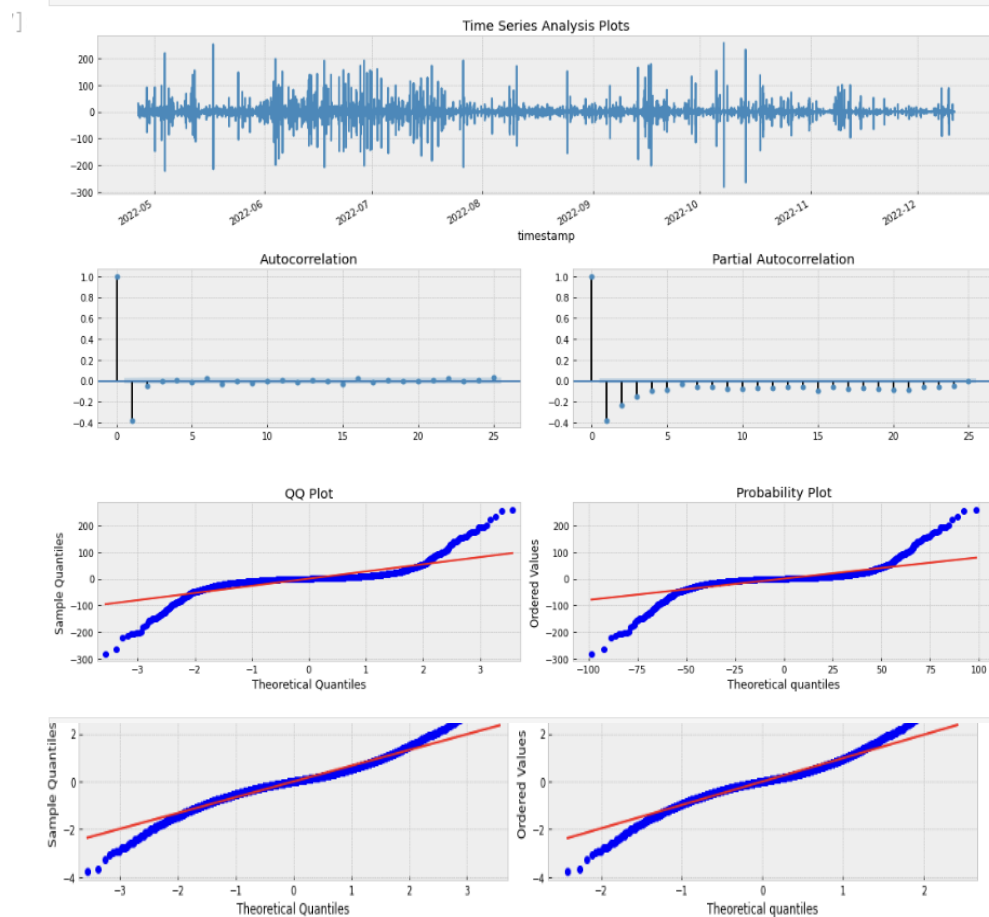
```
# ordinary first differencing
diff_df = df['average_gas_fee'].diff(periods=1).iloc[1:]

# use logarithm w/ differencing to reduce trend and keep variation width of our data
df_log = np.log(df['average_gas_fee'])
diff_df_log = df_log.diff(periods=1).iloc[1:]

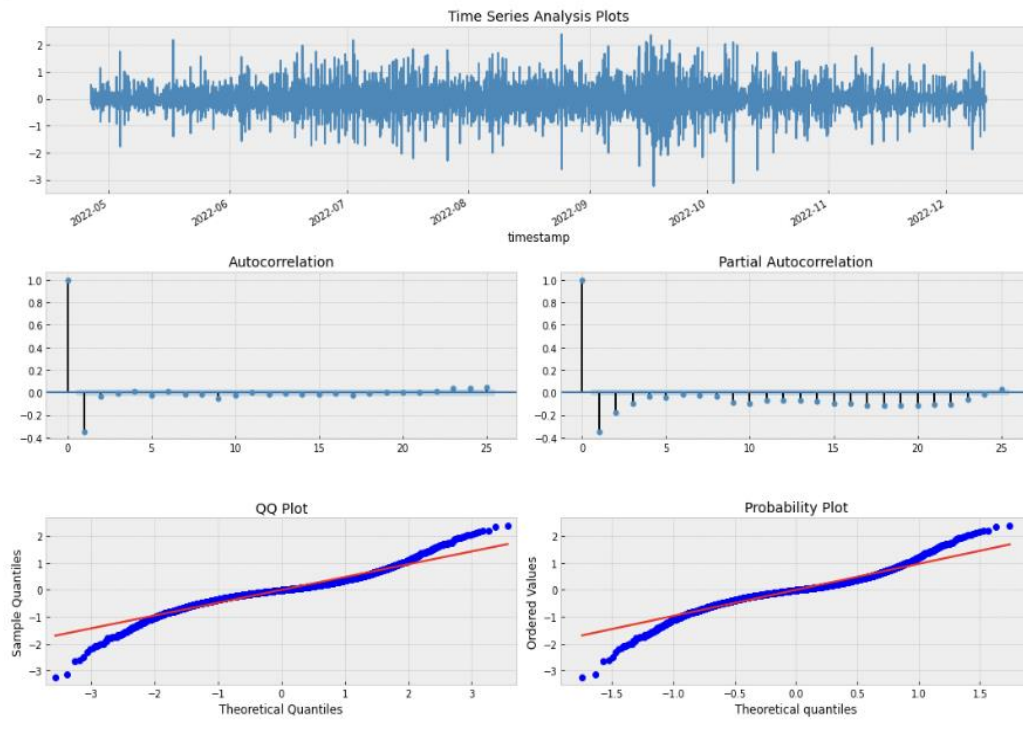
# daily differencing of log differenced data
diff_24_df = diff_df_log.diff(periods=24).iloc[24:]
```



```
tsplot(diff_df, lags=25)
```



```
tsplot(diff_df_log, lags=25) # It seems diff_df_log is more stationary according to its ACF
```



The ACF plots of the first and second differencing data both show 'cut off' after lag 1, but the PACF plots of both do not die out and stay significant. In the case of the daily differencing, the ACF plot seems to gradually decrease and converge after lag 1. But after lag 23 it does not die out and stays significant. Also, for the PACF plot, the value did not die out and pop up again after lag 23. It seems that the results reflect the 24 hours seasonality of our original data.

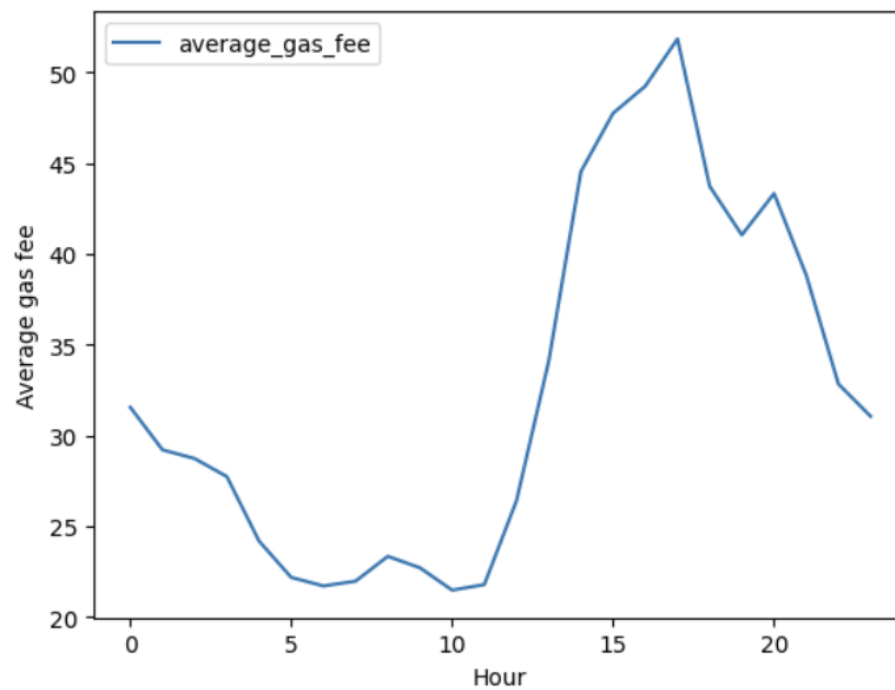
According to the simple plots, it seems that more differencing makes our data closer to white noise. Also, the QQ Plot and Probability Plot both become closer to the red line as the order increases, and the right tail also becomes closer to the line. According to the result of the ADF test, p values of all data were 0, so we could conclude that they are all stationary.

2) Average Gas Fee Changes Over Time and Day

Additionally, we looked at the changes of gas fee over time and day.

average_gas_fee			
timestamp			
0	17.178306	12	19.360908
1	16.989632	13	23.039381
2	16.138377	14	27.868837
3	16.748358	15	31.041447
4	15.913900	16	33.871074
5	14.499366	17	37.317678
6	14.795867	18	31.431753
7	14.939093	19	29.141096
8	16.988091	20	31.421217
9	14.390502	21	24.362093
10	15.089272	22	22.005593
11	15.765592	23	18.575867

```
1: df.groupby(df.index.hour)[['average_gas_fee']].mean().plot()
plt.xlabel('Hour')
plt.ylabel('Average gas fee')
plt.show()
# Average gas fee is the lowest at 10:00AM and highest at 17:00PM
```



```
[ ] df.groupby(df.index.hour)[['average_gas_fee']].mean()[12].mean()
# Average gas fee at morning time (00:00 AM – 11:00 AM)
```


```
average_gas_fee    15.786363
dtype: float64
```

```
[ ] df.groupby(df.index.hour)[['average_gas_fee']].mean()[12:].mean()
# Average gas fee at afternoon time (12:00 PM – 23:00 PM)
```

```
average_gas_fee    27.453079
dtype: float64
```

It was the lowest at 10 a.m. and the highest at 17 p.m. And it was higher in the afternoon time between 12 p.m. and 23 p.m. than in the morning time between 0 a.m. and 11 a.m.

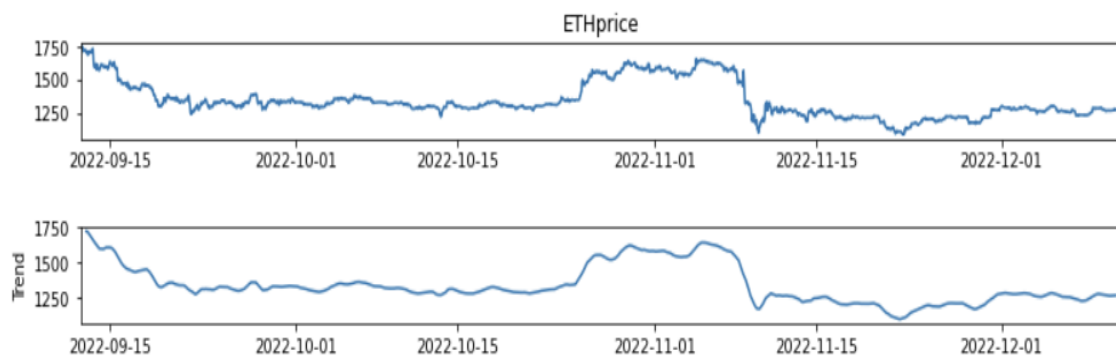
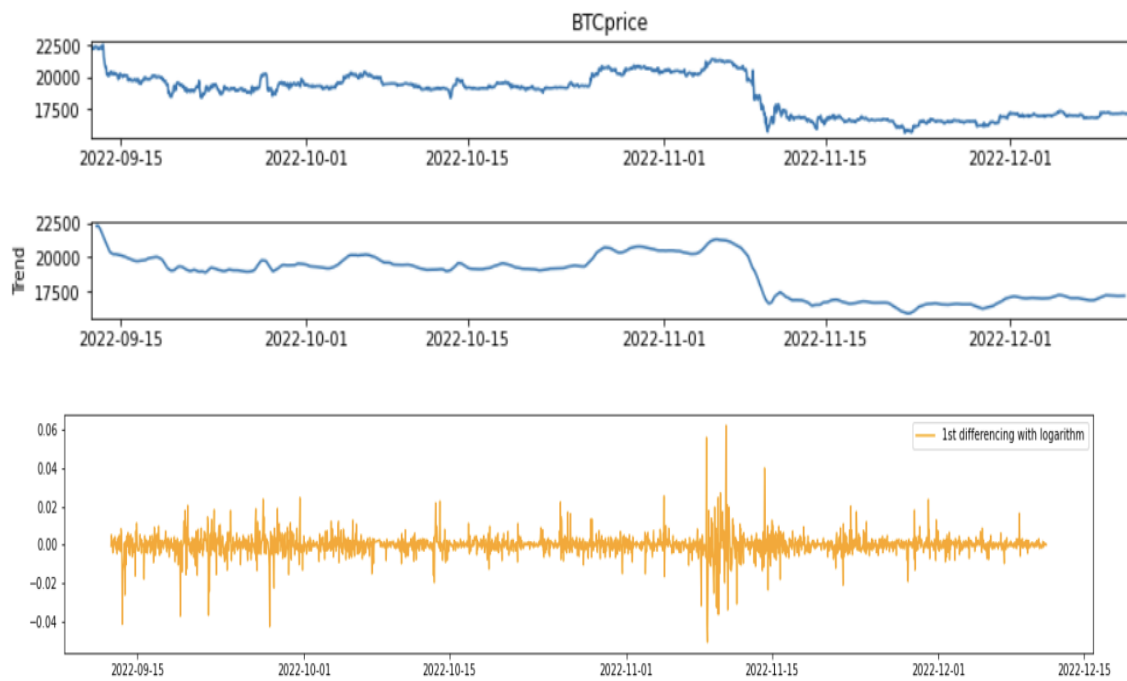
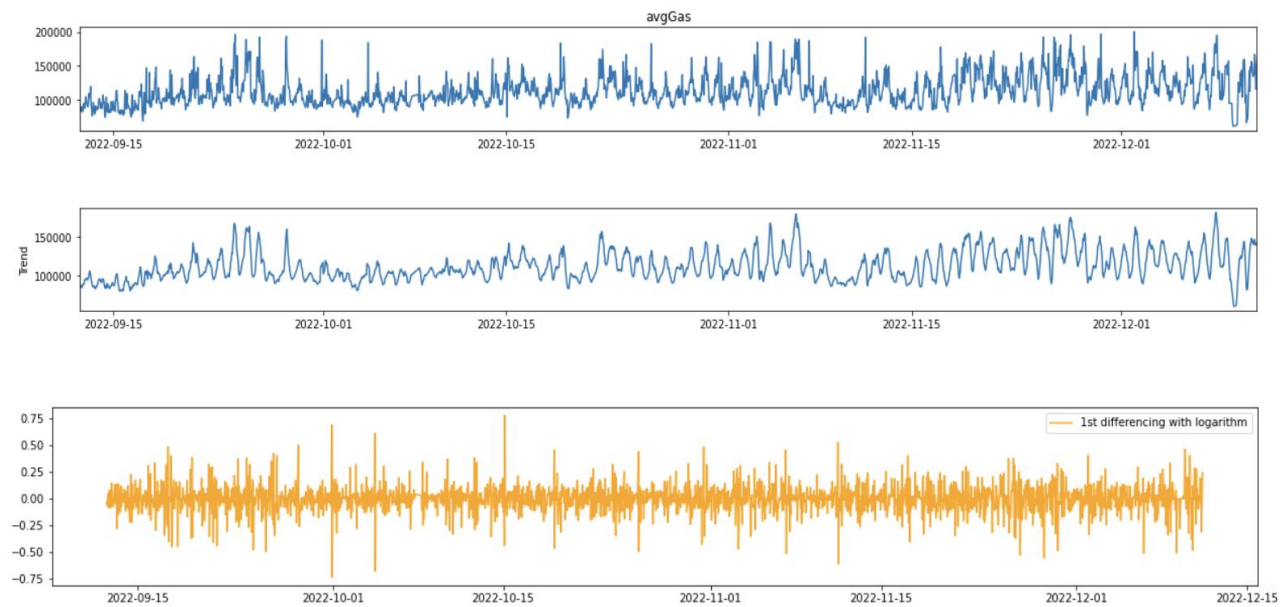
```
] df.groupby('day').mean().sort_values(by=["average_gas_fee"], ascending=False)
```

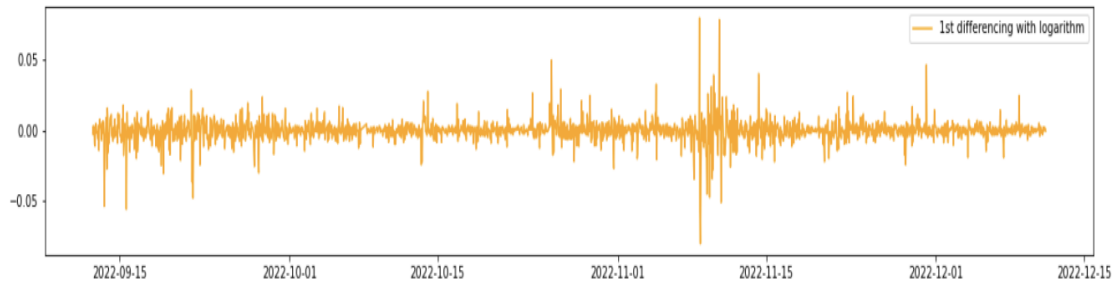
	average_gas_fee	avgGas 
day		
Thursday	39.297018	98466.591227
Wednesday	37.694451	99444.776575
Tuesday	35.181941	98568.640661
Friday	33.764786	98959.241079
Monday	33.198799	99529.275558
Saturday	24.920797	105772.546678
Sunday	23.960628	107357.437352

And it was the lowest on Sunday, and the highest on Thursday. Also, it was higher on the weekday than on the weekend.

3) Correlation With Exogenous Variables

We also looked at the correlation with external variables that would affect the formation of Ethereum gas fee. The data we checked are avgGas, Bitcoin price and Ethereum price for 90 days.





According to the trend plots of all data, it is hard to say that there is an overall trend, but there was a sharp drop in BTC and ETH price around November 10th. And the plots below are the results of logarithm with first differencing. It seems all data became stationary and also they passed the ADF test.

```
[ ] df.corr().style.background_gradient()
```

	average_gas_fee	avgGas	BTCprice	ETHprice
average_gas_fee	1.000000	-0.305356	0.059265	0.026005
avgGas	-0.305356	1.000000	-0.223470	-0.155227
BTCprice	0.059265	-0.223470	1.000000	0.862430
ETHprice	0.026005	-0.155227	0.862430	1.000000

And according to the correlation chart, there is a strong positive correlation between Ethereum price and Bitcoin price. However, both prices have very low correlation with average gas fee data. And there is a small negative correlation between avgGas data and average gas fee data.

Modelling and Evaluation

During the EDA, we saw that ACF cuts off after lag 1, which might suggest trying model MA(1), but PACF does not die out, so it is hard to decide the exact orders. So, we decided to a vanilla model IMA(0,1,1) and set it as a benchmark.

For the benchmark model, we got the following results:

```

SARIMAX Results
=====
Dep. Variable:    average_gas_fee    No. Observations:    5212
Model:           ARIMA(0, 1, 1)      Log Likelihood       -3150.890
Date:            Thu, 15 Dec 2022    AIC                  6305.780
Time:            07:45:04            BIC                  6318.898
Sample:          0                    HQIC                 6310.368
                  - 5212
Covariance Type:    opg
=====
              coef    std err          z      P>|z|      [0.025    0.975]
-----
ma.L1         -0.4518     0.009    -50.716     0.000     -0.469    -0.434
sigma2         0.1962     0.002     79.571     0.000      0.191     0.201
=====

```

So, our benchmark AIC is approximately 6305.

In our EDA, we also saw that daily differencing created very significant ACF and PACF at the lag 24, and thus we decided to build a seasonal ARIMA model with frequency 24, while experimenting with the orders of p, q, P and Q.

We first decided to extend the benchmark model by building SRIMA(0,1,1)(0,1,1)[24]. We got the following results:

```

SARIMAX Results
=====
Dep. Variable:    average_gas_fee    No. Observations:    5212
Model:           ARIMA(0, 1, 1)x(0, 1, 1, 24)  Log Likelihood       -2815.271
Date:            Thu, 15 Dec 2022    AIC                  5636.541
Time:            07:45:20            BIC                  5656.203
Sample:          0                    HQIC                 5643.419
                  - 5212
Covariance Type:    opg
=====
              coef    std err          z      P>|z|      [0.025    0.975]
-----
ma.L1         -0.6782     0.008    -83.007     0.000     -0.694    -0.662
ma.S.L24      -0.9346     0.004   -208.372     0.000     -0.943    -0.926
sigma2         0.1717     0.002     82.587     0.000      0.168     0.176
=====

```

The AIC in this case was much better than that of the benchmark model.

After that, we added an AR term with lag 1 (to the seasonal part also) to see how it will affect the AIC. We got the following results:

```

SARIMAX Results
=====
Dep. Variable:          average_gas_fee    No. Observations:      5212
Model:                 ARIMA(1, 1, 1)x(1, 1, 1, 24)    Log Likelihood        -2749.677
Date:                  Thu, 15 Dec 2022    AIC                   5509.355
Time:                  07:46:08            BIC                   5542.124
Sample:                0                   HQIC                5520.818
                        - 5212
Covariance Type:       opg
=====
              coef    std err          z      P>|z|      [0.025      0.975]
-----
ar.L1          0.2643     0.013     20.152     0.000     0.239     0.290
ma.L1         -0.8573     0.008    -102.300     0.000    -0.874    -0.841
ar.S.L24       0.0039     0.013     0.309     0.757    -0.021     0.029
ma.S.L24      -0.9366     0.005   -190.915     0.000    -0.946    -0.927
sigma2         0.1674     0.002     82.904     0.000     0.163     0.171
=====

```

We got lower AIC but the AR term in the seasonal part was statistically insignificant, so we decided to remove that term from the above model. As a result, we got even lower AIC of 5507.423.

We also implemented **auto_arima** to see the suggested optimal orders and compare its results with that of our SARIMA model. You can see the results below.

```

Performing stepwise search to minimize aic
ARIMA(1,1,0)(0,0,0)[0] intercept : AIC=6517.455, Time=1.02 sec
ARIMA(0,1,0)(0,0,0)[0] intercept : AIC=7183.392, Time=0.42 sec
ARIMA(0,1,1)(0,0,0)[0] intercept : AIC=6307.773, Time=0.97 sec
ARIMA(0,1,0)(0,0,0)[0]          : AIC=7181.393, Time=0.18 sec
ARIMA(1,1,1)(0,0,0)[0] intercept : AIC=6291.158, Time=2.35 sec
ARIMA(2,1,1)(0,0,0)[0] intercept : AIC=6010.476, Time=7.11 sec
ARIMA(2,1,0)(0,0,0)[0] intercept : AIC=6366.310, Time=0.93 sec
ARIMA(3,1,1)(0,0,0)[0] intercept : AIC=5987.110, Time=9.76 sec
ARIMA(3,1,0)(0,0,0)[0] intercept : AIC=6318.495, Time=1.13 sec
ARIMA(4,1,1)(0,0,0)[0] intercept : AIC=5988.288, Time=11.03 sec
ARIMA(3,1,2)(0,0,0)[0] intercept : AIC=6013.151, Time=10.81 sec
ARIMA(2,1,2)(0,0,0)[0] intercept : AIC=5998.006, Time=10.80 sec
ARIMA(4,1,0)(0,0,0)[0] intercept : AIC=6313.078, Time=1.31 sec
ARIMA(4,1,2)(0,0,0)[0] intercept : AIC=5988.622, Time=13.94 sec
ARIMA(3,1,1)(0,0,0)[0]          : AIC=5985.764, Time=2.42 sec
ARIMA(2,1,1)(0,0,0)[0]          : AIC=6008.875, Time=1.88 sec
ARIMA(3,1,0)(0,0,0)[0]          : AIC=6316.501, Time=0.51 sec
ARIMA(4,1,1)(0,0,0)[0]          : AIC=5986.508, Time=3.04 sec
ARIMA(3,1,2)(0,0,0)[0]          : AIC=5987.217, Time=5.30 sec
ARIMA(2,1,0)(0,0,0)[0]          : AIC=6364.315, Time=0.35 sec
ARIMA(2,1,2)(0,0,0)[0]          : AIC=5988.476, Time=3.66 sec
ARIMA(4,1,0)(0,0,0)[0]          : AIC=6311.084, Time=0.61 sec
ARIMA(4,1,2)(0,0,0)[0]          : AIC=5987.262, Time=3.44 sec

Best model: ARIMA(3,1,1)(0,0,0)[0]

```

The optimal model according to **auto_arima** was ARIMA(3,1,1), which had AIC of 5987. However, our last SRIMA model had a much better result. It should also be noted that we tried

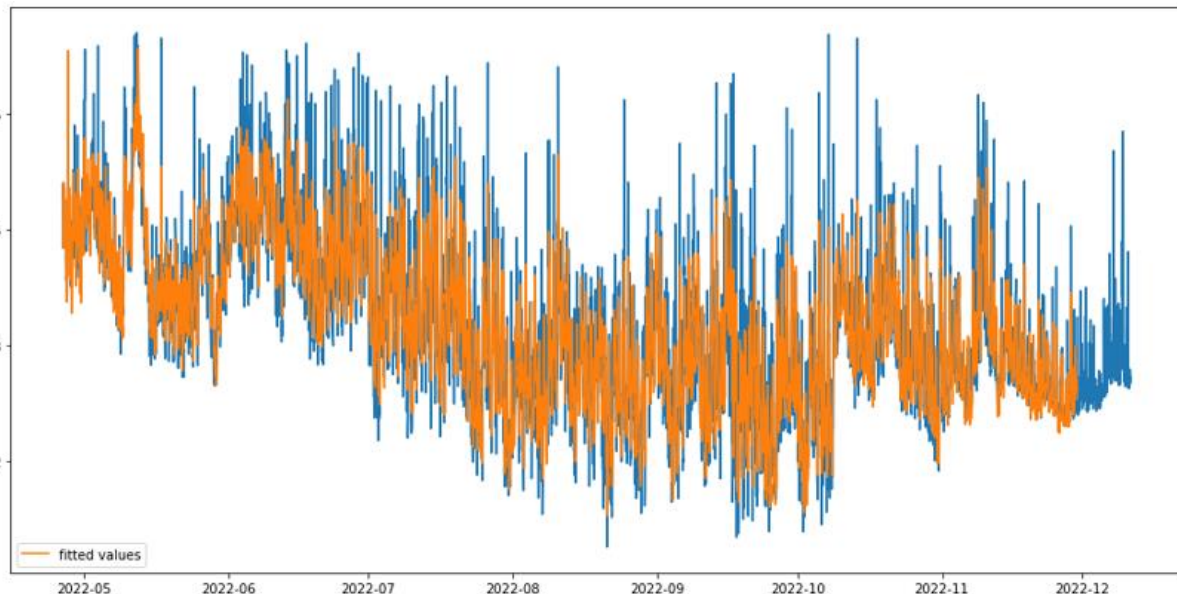
to run **seasonal auto_arima**, but, because of high-computational cost ,Google Collab crashed after a few iterations, and thus we could not get the final results.

Instead of discarding the results of **auto_arima**, we decided to change the p order of our earlier model according to the best model suggested by **auto_arima**. We got the following results for the model SARIMA(3,1,1)(0,1,1)[24]:

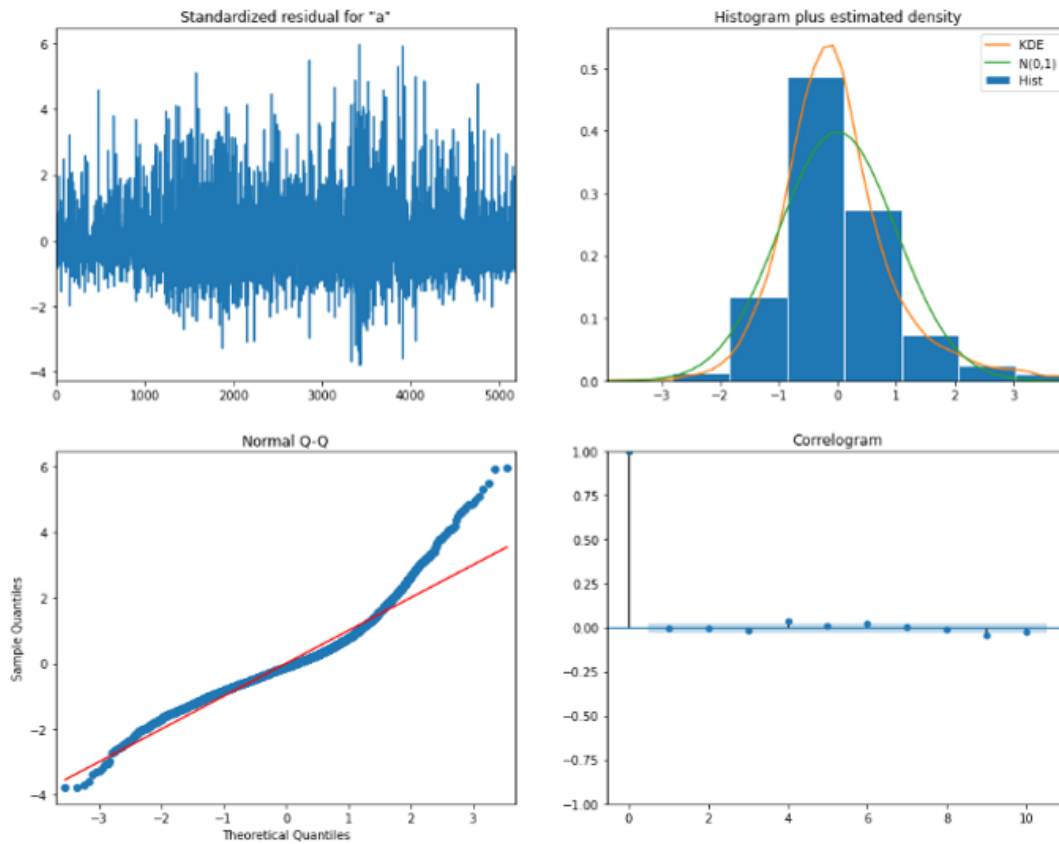
```
SARIMAX Results
=====
Dep. Variable:          average_gas_fee      No. Observations:      5212
Model:                 ARIMA(3, 1, 1)x(0, 1, 1, 24)  Log Likelihood        -2726.611
Date:                  Thu, 15 Dec 2022           AIC                   5465.222
Time:                  07:55:23                   BIC                   5504.546
Sample:                0                           HQIC                  5478.979
                    - 5212
Covariance Type:       opg
=====
              coef    std err          z      P>|z|      [0.025    0.975]
-----
ar.L1         0.3068     0.013    24.392     0.000     0.282     0.331
ar.L2         0.0963     0.014     6.997     0.000     0.069     0.123
ar.L3         0.0579     0.013     4.380     0.000     0.032     0.084
ma.L1        -0.9180     0.008   -111.457     0.000    -0.934    -0.902
ma.S.L24     -0.9434     0.004   -218.431     0.000    -0.952    -0.935
sigma2        0.1658     0.002    82.553     0.000     0.162     0.170
=====
```

As you can see, after including two more lags in the AR term in our above model, the AIC decreased from 5507 to 5465. So, we decided to stick with this model for further analysis.

Model Performance on the Training Set

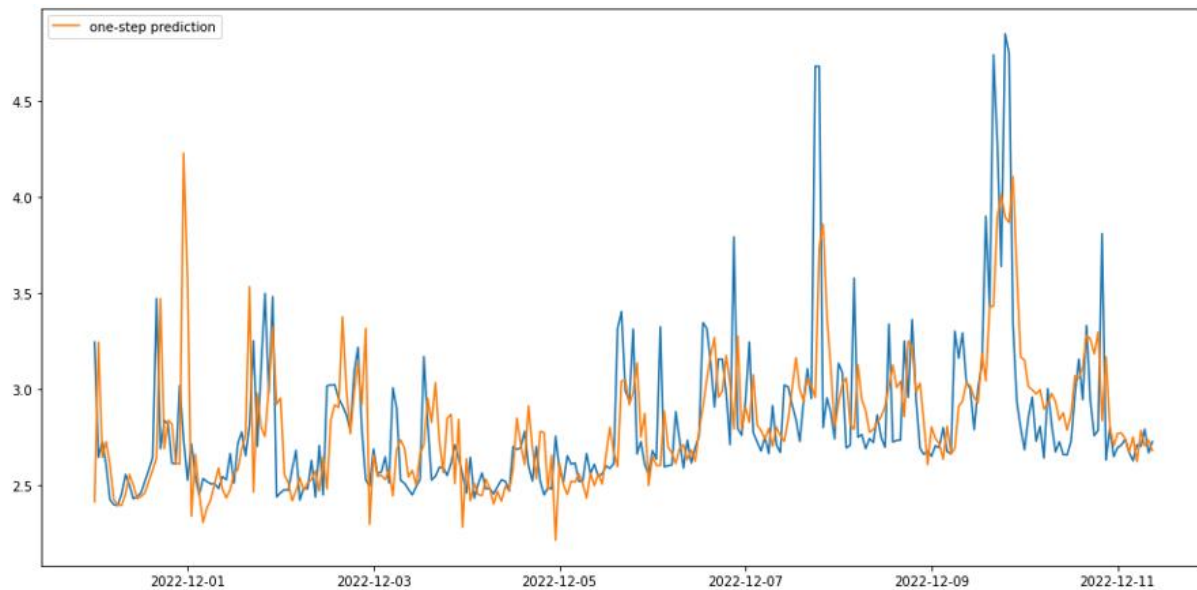


Our train set is from April 26 to Nov 29, while the test set is from Nov 30 to Dec 11. The R2 score for the train set was 0.72. Below, you can see plots analyzing residuals.



We can see the distribution of the residuals is tilted to the left slightly, but the overall shape still resembles normal distribution. Also, in the Q-Q plot, values on the upper part are not aligned with the theoretical values, however based on looking at the two other graphs, we can say that the residuals are very similar to white noise series.

Model Performance on the Test Set



The R2 score for the test was 0.22, much lower than the R2 score for training set.

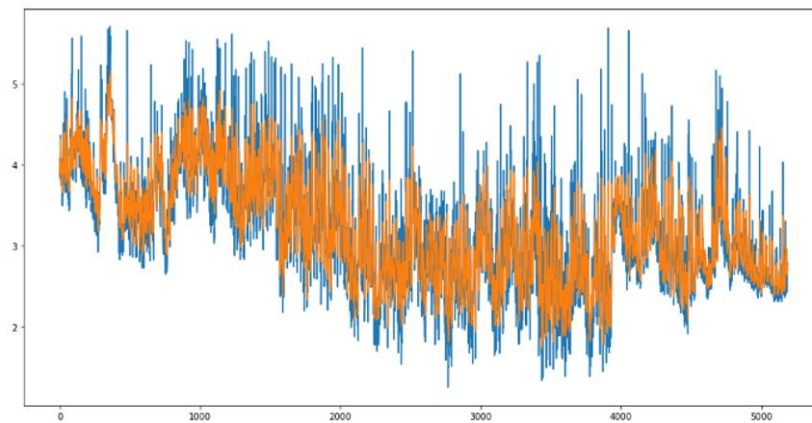
We know that SARIMA models cannot remember long-term information and probably because of that, they were having low test performance. Thus, we decided to build an LSTM model because it can remember long time information and thus probably will have better performance on the test set.

The following hyperparameters and other specifications were chosen for LSTM:

- 1) Sequence length – 24 (information from the last 24 hours)
- 2) Input size – 1 (only time series data is considered at this point)
- 3) Hidden layer size – 2
- 4) Number of layers – 1
- 5) Output – the fee for the next hour
- 6) Loss function – MSE
- 7) Number of epochs – 2000

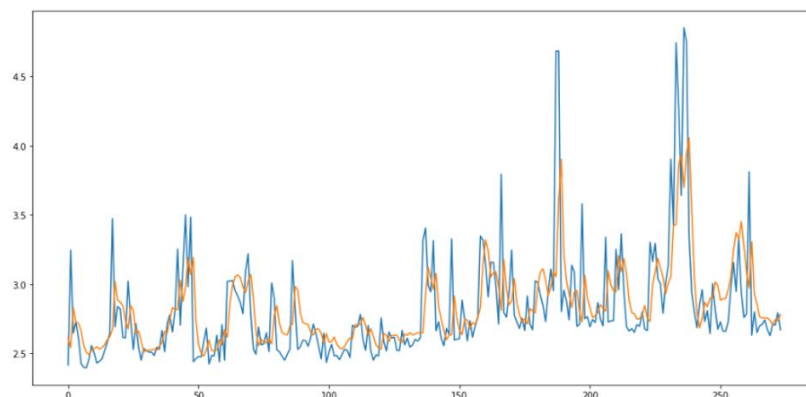
The R2 score for the train set was 0.71, which is a bit lower than that of the SARIMA model.

Model Performance on the Train Set



However, the R2 score for the test set was 0.39, which is twice as better as that of the SARIMA model. You can see the graph below.

Model Performance on the Test Set



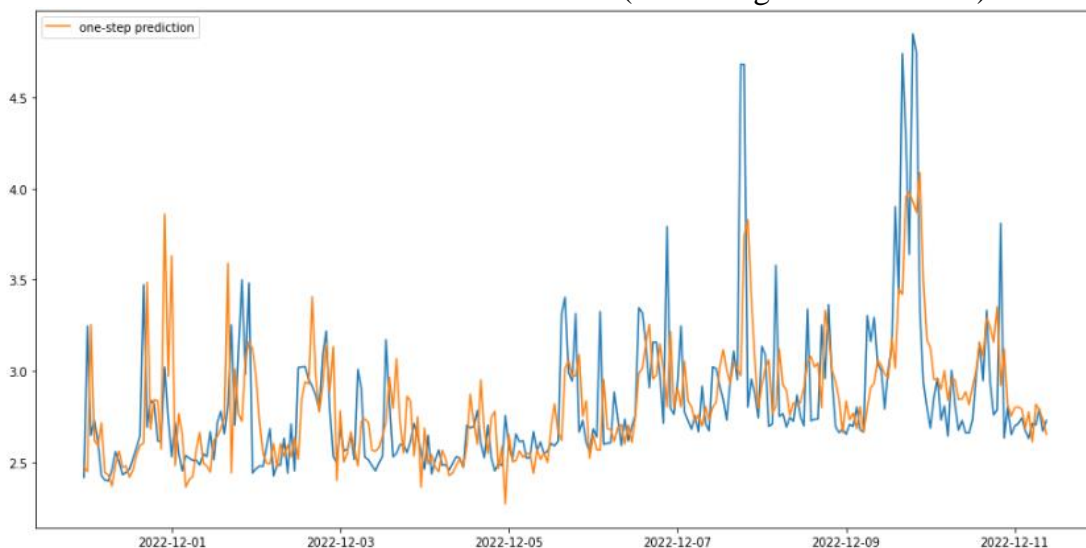
For the last 3-month data, we also had three exogenous variables, Bitcoin price, Ethereum price and average Gas value, which is the average computation work required to execute a transaction in the Ethereum blockchain network.

To understand the relationship between the gas fees and those exogenous variables, we built SRIMAX model (but first we transferred the X component so that it became stationary). The results are shown below.

SARIMAX Results						
Dep. Variable:	log average_gas_fee	No. Observations:	1873			
Model:	ARIMA(3, 1, 1)x(0, 1, 1, 24)	Log Likelihood	-1152.442			
Date:	Fri, 23 Dec 2022	AIC	2322.884			
Time:	08:02:26	BIC	2372.580			
Sample:	0	HQIC	2341.204			
	- 1873					
Covariance Type:	opg					
	coef	std err	z	P> z	[0.025	0.975]
log diff avgGas	-0.1644	0.062	-2.660	0.008	-0.285	-0.043
log return (Bitcoin)	0.1861	2.513	0.074	0.941	-4.739	5.112
log return (Ethereum)	-2.8552	1.906	-1.498	0.134	-6.591	0.881
ar.L1	0.3499	0.018	19.187	0.000	0.314	0.386
ar.L2	0.1028	0.022	4.626	0.000	0.059	0.146
ar.L3	0.0960	0.020	4.730	0.000	0.056	0.136
ma.L1	-0.9479	0.010	-92.686	0.000	-0.968	-0.928
ma.S.L24	-0.9110	0.012	-78.061	0.000	-0.934	-0.888
sigma2	0.1990	0.004	52.163	0.000	0.192	0.207
Ljung-Box (L1) (Q):	0.09	Jarque-Bera (JB):	1826.66			
Prob(Q):	0.76	Prob(JB):	0.00			
Heteroskedasticity (H):	0.29	Skew:	1.20			
Prob(H) (two-sided):	0.00	Kurtosis:	7.24			

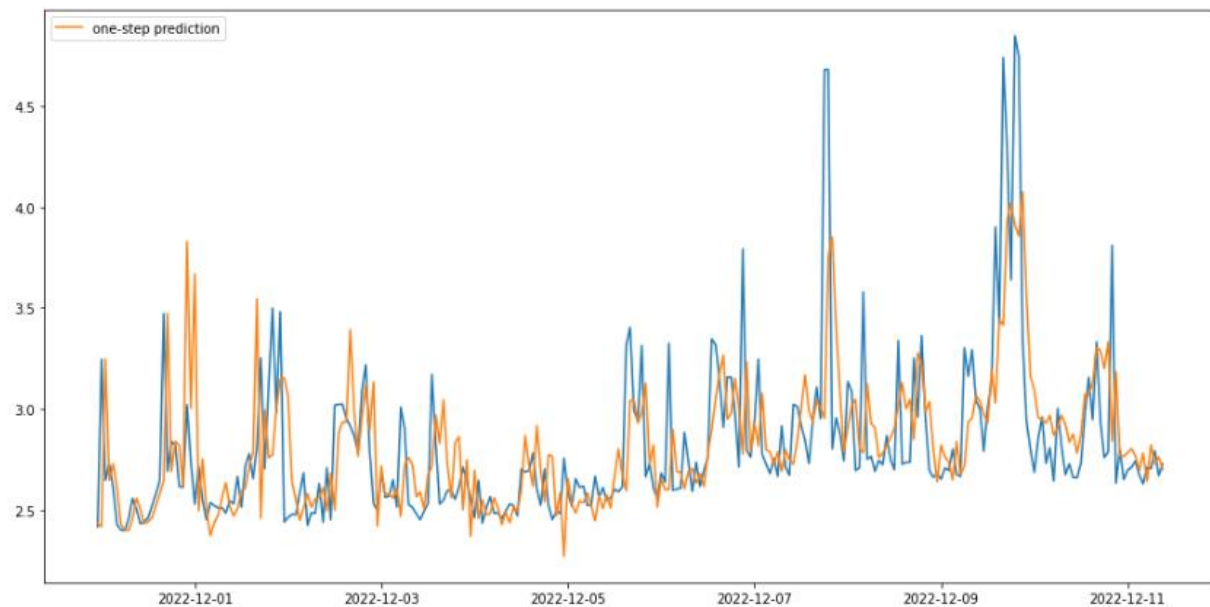
We can see that log returns of Bitcoin and Ethereum are not important variables since their coefficients have p-values higher than 0.05. However, average Gas variable was statistically important as its coefficient has p-value less than 0.05. It had a negative coefficient, meaning that for transactions that require a lot of computation work to execute, one can pay lower fees. This is not surprising as the total transaction fee for any transaction on the Ethereum network is calculated by multiplying estimated gas amount by the gas fee.

Model Performance on the Test Set (with Exogenous variables)



The R2 score for the test set, when we included exogenous variables, was 0.28.

Model Performance on the Test Set (without Exogenous variables)



In this case, the R2 score for the test set was 0.27, not much different that the model with exogenous variables.

Conclusion

1. We discovered that average transaction fees on the Ethereum network are lower before 12 p.m. (morning time) and they become much higher in the afternoon (so do your transactions in the morning)
2. We also discovered that the transaction fees on average are higher during weekdays, while they are much lower during weekends (so do your transactions in weekends)
3. Nor the price of Ethereum or Bitcoin affects the transaction fees on the Ethereum blockchain network.
4. Although SARIMA and LSTM had almost the same training performance, LSTM model showed much better test results than SARIMA.
5. It is hard to precisely predict how transaction fees will behave in the future even using Deep Learning models such as LSTM.