Predicting the Value of Ether over Time

Stephanie Marker

December 22, 2017

1 Definition

1.1 Project Overview

Ethereum is a decentralized digital cryptocurrency platform invented by Vitalik Buterin in 2013. It uses public key cryptography and a consensus algorithm called "proof of work" (POW) to prevent denial of service attacks on the system. [1] Ether, Ethereum's currency, does not go through a bank, rather, Ether are managed in peer to peer transactions. Therefore, Ether transactions are much cheaper. Transactions are tracked on the blockchain, a distributed ledger. As **Figure 1** indicates, every block in the chain contains a pointer from the previous block, a timestamp, a nonce, and a copy of valid transaction records within that block.

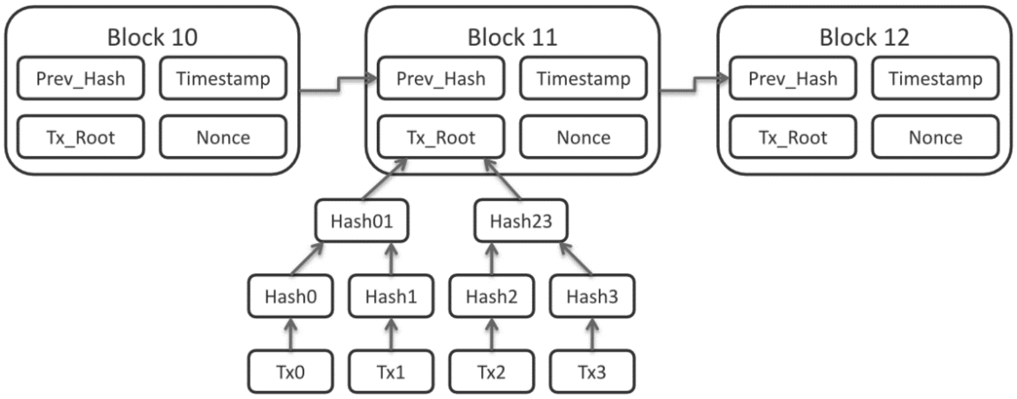


Figure 1: Blockchain [2]

When a person wants to send Ether to another user, that person will request to have their transaction be validated by the miners and added to the chain. [2] Each miner maintains their own copy of the blockchain, so that they can verify each other's work. Miners race to validate these requests. The miner with the longest chain and a majority approval from the other miners working on their own copies of the chain receives a monetary reward in Ether for the transactions that the miner verified. Only when the miners agree, does the transaction get added to the latest block in their copy of the chain.

Because blockchain is peer to peer and relies on proof of work, it deters people from attempting to alter the history of transactions. This is because a crook would have to fool the other miners that their copy of the chain with an altered beginning transaction is valid. Remember, the miner who has the longest verified chain has their chain accepted as the source of truth for all the other miners. Therefore, a crook would have to do expensive calculations from the early block to the current block faster than all the other miners compute the current block as shown in **Figure 2**.

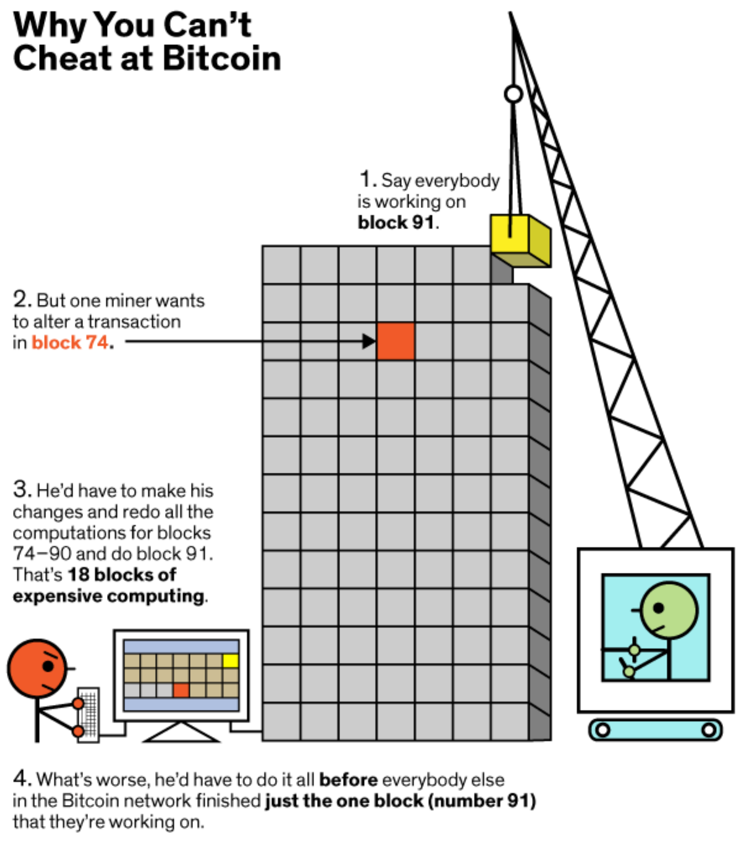

Why You Can't Cheat at Bitcoin
1. Say everybody is working on block 91.
2. But one miner wants to alter a transaction in block 74.
3. He'd have to make his changes and redo all the computations for blocks 74–90 and do block 91. That's 18 blocks of expensive computing.
4. What's worse, he'd have to do it all before everybody else in the Bitcoin network finished just the one block (number 91) that they're working on.

Figure 2: Blockchain Security [3]

1.2 Problem Statement

The goal of this project is to predict the daily closing prices of Ether a week into the future. This can be solved with supervised learning since at least a year of data exists with several labels like address, block size, and ether price. The approach to solving this problem is described below:

Approach:

1. Extract Ether data from Etherscan.
2. Filter any Ethereum data prior to March 2017.
3. Create time-indexed timeseries DataFrames of Ether data.
4. Make price predictions with different models.
    a. Univariate ARIMA
        i. Preprocess the data before constructing an ARIMA model.
            - Format the data as timeseries data, i.e. index the data in time order. Remove any NaN values.
            - Perform an augmented Dickey Fuller Test to see if the timeseries data is stationary. If it is not, perform a first time-difference on the data to make the data stationary.
        ii. Find the correct parameters to apply the ARIMA model.

            - Determine whether the process is AR or MA using a total correlation chart.
            - Find the degree of the AR (or MA) series.
            - Verify these parameters with grid search
        iii. Build a univariate ARIMA model.
        iv. Predict with the ARIMA model.
    b. LSTM RNN
        i. Preprocess the data before constructing a LSTM RNN model.
            - Perform feature selection with recursive feature extraction
            - Given the selected features, format the data as time series
            - Take first time-difference to make the data stationary
            - Apply MinMaxScaler on the data to scale the data
        ii. Find the correct parameters to apply to the LSTM RNN model.
            - Determine the optimal number of epochs, hidden layer nodes, input layer nodes, and output nodes
        iii. Build a LSTM RNN model
        iv. Predict with the model
        v. Given outputs from the model, invert the scaling and differencing
5. Verify the accuracy of each model by computing the mean squared error.

For the 7-day prediction, I expect that the predictions for the most recent date will be more accurate than days farther into the future.

1.3 Metrics

The accuracy of prediction can be measured by its mean squared error (MSE), the average squared distance between actual output and prediction:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

The goal will be to minimize the MSE. In addition, to punish very large predictions off the actual values, I will also use the root mean squared error (RMSE). [4] The RMSE will tell how close or far off the data is from the line of best fit.

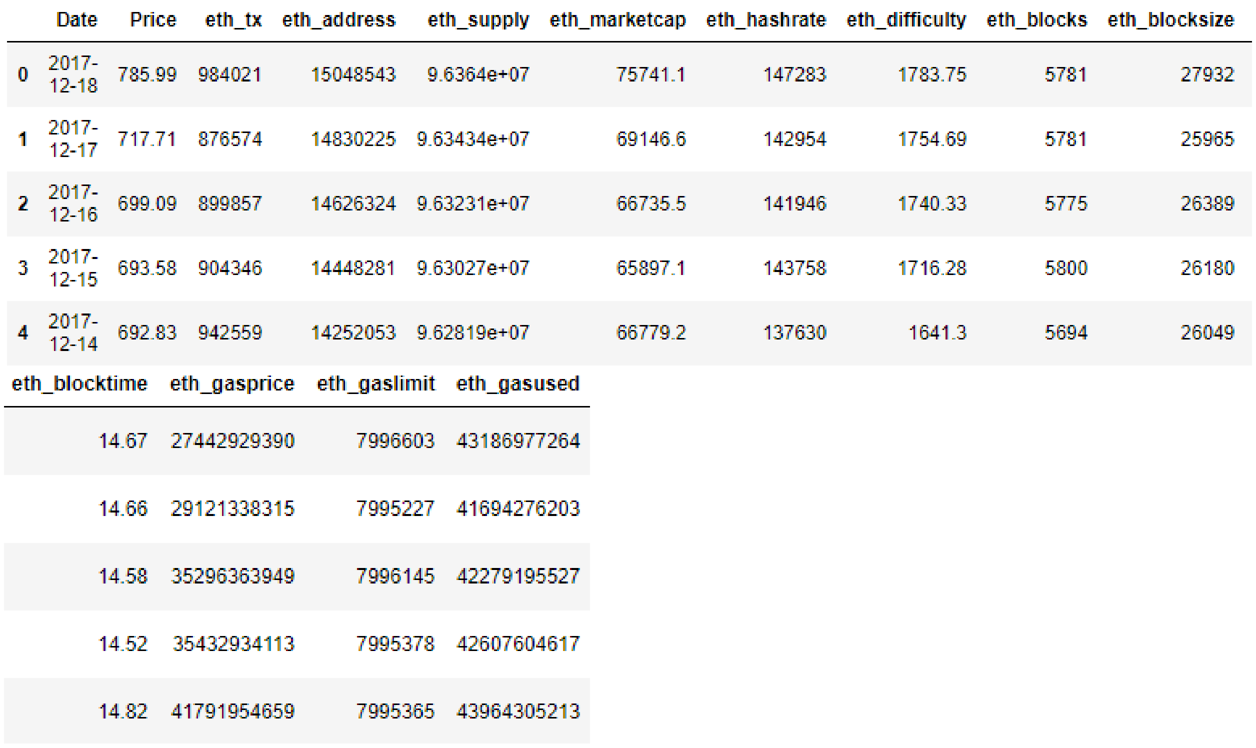
$$MSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

Forecasting, climatology, and regression analysis commonly use RMSE to assess results, so RMSE fits well with forecasting the price of Ether.

2 Analysis

2.1 Data Exploration and Visualizations

Input data used in this project is found in **/data**. First, I will discuss **ETH_DataSet.csv**. The 5 most recent entries of Ether data are shown below in **Figure 3**. Features I will explain in more detail include ether transaction count and ether gas price. Other features included in **Figure 3** will be considered in Ether price prediction, but will need to be investigated in more detail about the magnitude of influence they have on Ether price.



	Date	Price	eth_tx	eth_address	eth_supply	eth_marketcap	eth_hashrate	eth_difficulty	eth_blocks	eth_blocksize	eth_blocktime	eth_gasprice	eth_gaslimit	eth_gasused
0	2017-12-18	785.99	984021	15048543	9.6364e+07	75741.1	147283	1783.75	5781	27932	14.67	27442929390	7996603	43186977264
1	2017-12-17	717.71	876574	14830225	9.63434e+07	69146.6	142954	1754.69	5781	25965	14.66	29121338315	7995227	41694276203
2	2017-12-16	699.09	899857	14626324	9.63231e+07	66735.5	141946	1740.33	5775	26389	14.58	35296363949	7996145	42279195527
3	2017-12-15	693.58	904346	14448281	9.63027e+07	65897.1	143758	1716.28	5800	26180	14.52	35432934113	7995378	42607604617
4	2017-12-14	692.83	942559	14252053	9.62819e+07	66779.2	137630	1641.3	5694	26049	14.82	41791954659	7995365	43964305213

Figure 3: Ether Data

First, I will discuss Ether transactions. The mean of transactions is ~290,000 and the standard deviation is ~189,000. The coefficient of variation (CV) is < 1, so there's not much variation in ether transactions.

I predict that the more people that make Ether transactions, the more valuable Ether is, because increased use validates its acceptance in the community. Therefore, as Ether transactions increase, I would expect Ether price to also increase. This correlation of Ether price and Ether transaction count is shown in **Figure 4**. As the number of transactions (in red) of Ether increased, so did the price (in blue). In addition, the largest number of transactions, 984,021 occurred on Monday, December 18, 2017.

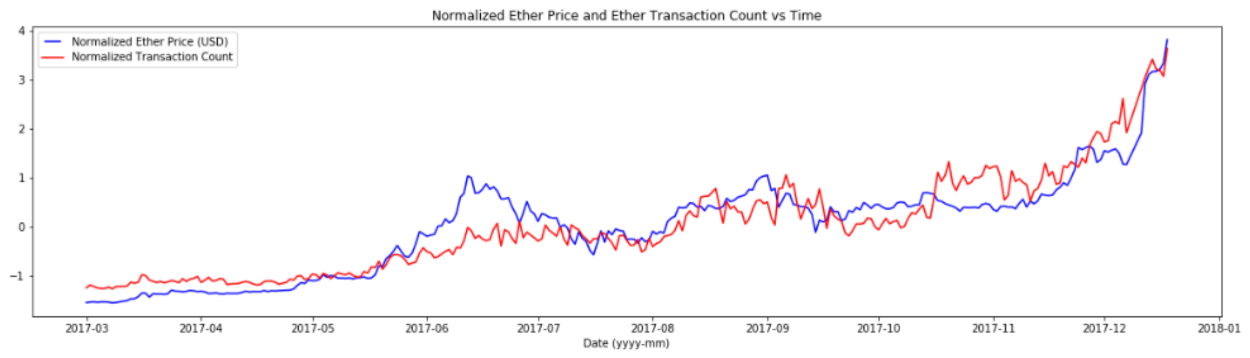


Figure 4: Normalized Ether Price and Ether Transaction Count over Time

Next, I will study Ether gas price. The mean of Ether gas price is 2.41E10 and the standard deviation is 7.3E9. The coefficient of variation is < 1, so there is not much variation of Ether gas price.

I predict that the higher the Ether gas price, the lower Ether price is, because less people would make transactions at a high Ether gas price. There is a small inverse correlation between Ether gas price and Ether price shown in **Figure 5**. For example, looking closely at the month of November, there were a few dips in Ether gas price and at each dip, there was a small increase in Ether price. Similarly, looking at the dates where normalized gas price spiked (late June, early September, and early December), there was a corresponding dip in Ether price. The length of time that Ether price decreased, and the magnitude of the Ether price decrease corresponded with how long Ether gas price stayed high. For example, in late June and early September, Ether gas price remained relatively high for half a month and Ether price also decreased for around half a month to a month. In early December, there was a large but brief Ether gas price spike and a small Ether price dip.

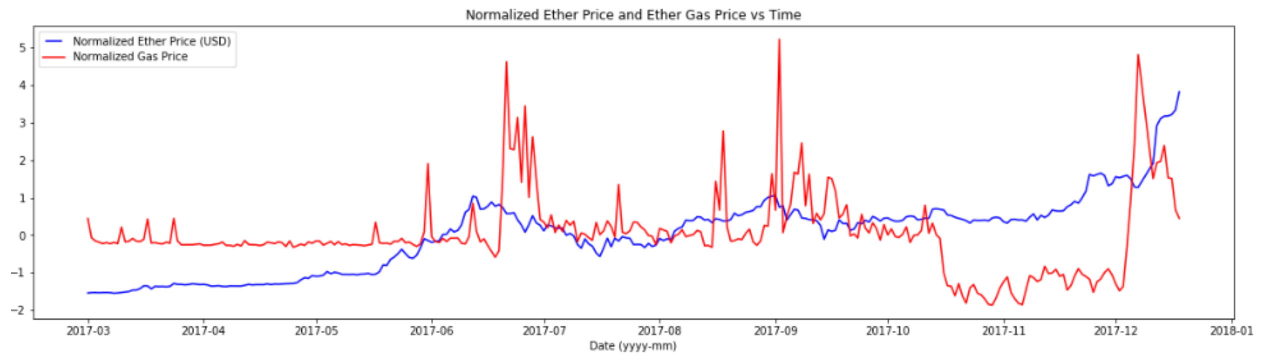


Figure 5: Normalized Ether Price and Ether Gas Price over Time

2.2 Algorithms and Techniques

2.2.1 Univariate ARIMA

I will first use univariate ARIMA as a baseline predictor, where I will model Ether price over time. ARIMA is commonly used to analyze and forecast time series data. This model is suitable since Ether price data are not independent, since the Ether price observation order matters. Changing the order of the data would change the meaning of the data.

Univariate ARIMA requires stationarity of data, i.e. the mean, variance, autocorrelation, etc. are all constant over time. **Figure 6** shows an example of stationary timeseries data achieved after taking a first time-difference.

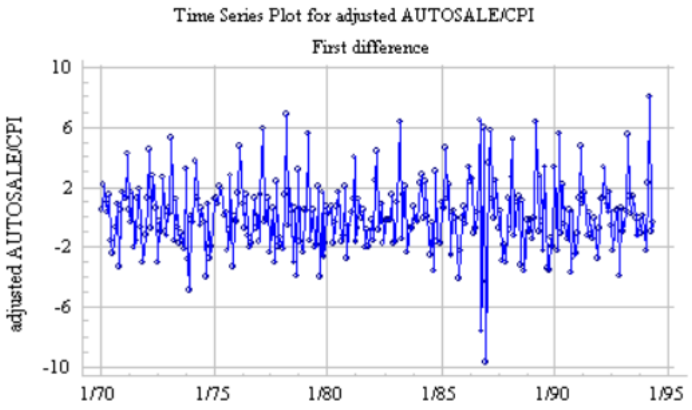


Figure 6: Example of Stationary Time Series Data [5]

ARIMA has 3 hyperparameters, p, d, and q. The p parameter will allow the model to include the effect of past values on the model. The d parameter will include the amount of differencing the timeseries needed to become stationary. The q parameter will set the error of the model as a linear combination of errors at previous dates. [6]

Data fed into the ARIMA model will be formatted as timeseries data, i.e. the data will be indexed by date and contain only Ether price as the values. The data may need to be time-differenced, if it is not stationary.

Though ARIMA is one of the standard tools for time series data, it has disadvantages. One disadvantage specified in a paper on ***ARIMA and Random Forest time series models for prediction of avian influenza H5N1 outbreaks*** states that "ARIMA assumes linear relationships between independent and dependent variables. Real-world relationships are often non-linear and therefore more complex than the assumptions built into the model". Consequently, ARIMA often performs poorly where data follows a more complex structure. [7]

One example of an external event is China's ICO ban around Labor Day, which influenced a 30% crypto market drop caused by fear of a missing a large player in the crypto market. [8] In addition, CryptoKitties, one of the first popular Ethereum-based decentralized apps, had an increase in use in early December, which "accounted for nearly 14 percent of the entire Ethereum network's transaction volume, which is higher than the transaction volume of all other cryptocurrencies in the market combined, including bitcoin". CryptoKitties showed the value of blockchain, thus contributing to a large Ether price surge. [9]

Though Ether price has many external dependent factors like China's ICO ban and the popular CryptoKitties app, it is still useful to get a general idea of the trend in price and use these findings in comparison with other more complex multivariate models, like LSTM RNNs.

2.2.1 Long Short-Term Memory (LSTM) Recurrent Neural Networks

I will then use a LSTM recurrent neural network to model Ether price over time. LSTM RNNs can also analyze and forecast time series data like ARIMA, but also allow one to model multivariate problems like predicting the price of Ether given several features such as the number of Ether transactions, block size, gas price, etc.

LSTM RNNs require stationarity of data, so time differencing may need to be done. It is also important to normalize the input variables before modeling a LSTM RNN, so that large variations in feature values do not get hidden or outweigh other features at differing scales. In addition, feature normalization can make training faster and lower the chances of being stuck in local optima.

Several hyperparameters can be tuned to improve a LSTM model such as batch size, the number of epochs, and the number of hidden layer neurons. An epoch is one forward and backward pass of all the training points. Batch size determines the amount of training samples for the model to consider before updating the network's weights. Choose a batch size such that it can divide the number of inputs. For the number of hidden layer nodes, it's recommended to use between the amount of input nodes the amount of output nodes (https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw):

$$N_h = \frac{N_s}{\left(\alpha * (N_i + N_o)\right)}$$

$N_i = number\ of\ input\ layer\ neurons$
$N_o = number\ of\ output\ layer\ neurons$
$N_s = number\ of\ samples\ in\ the\ training\ set$
$\alpha\ \ = arbitrary\ scaling\ factor, usually\ between\ 2\ and\ 10$

Data fed into the LSTM RNN model will be formatted as time series data, which contains the date and several features to consider during training. The data may need to be time-differenced if it is not stationary and scaled.

Advantages of LSTM RNNs are that they contain a 'memory cell' unlike other standard RNNs that will let them store information in memory for a long time, thus allowing them to learn long-term time dependencies. [10]

Drawbacks of a LSTM RNNs are that it can be slow to train and that you must be very careful with choosing hyperparameters, so that the model is not over or underfitted. It is easy to overfit with LSTMs. Dropout can counter overfitting, which excludes LSTM units from the activation and weight updates during training. (https://machinelearningmastery.com/use-dropout-lstm-networks-time-series-forecasting/) In addition, having a 'memory cell' adds more weights to nodes which need to be trained, thus adding dimensions to the problem. Having more dimensionality, may require the model to be trained on more data to achieve optimal results.

2.3 Benchmark

There is another example of someone using a multidimensional LSTM network to predict cryptocurrency prices. This person tried daily 1-step-ahead predictions and 50-step-ahead predictions of bitcoin price. He found that the 1-step-ahead predictions did well. For the 50-step-ahead predictions, the model did not do as well, but still was able to follow the trends in price. [11]

3 Methodology

3.1 Data Preprocessing

Even though Ethereum has been around for a few years, Ether price data is taken from March 1, 2017 to present, since that is around the time when the price of Ether started take off and vary significantly.

3.1.2 Univariate ARIMA

For the univariate ARIMA model, no feature selection was needed. Only Ether price feature and date were needed.

3.1.3 LSTM RNN

Since not much input data exist, feature selection was needed to reduce dimensionality. Recursive feature elimination was used to determine ranked features in order of importance. This process selects features recursively by looking at smaller subsets of features. At each step, the least important features are discarded until it reaches the chosen number of features to select is determined. After feature selection, the top 6 features were price, number of unique addresses, supply, market cap, hash rate, and block size. Ranks of features are shown in **Figure 7**. A low rank means that the feature contributes strongly to the price of Ether, so low numbered ranks are best.

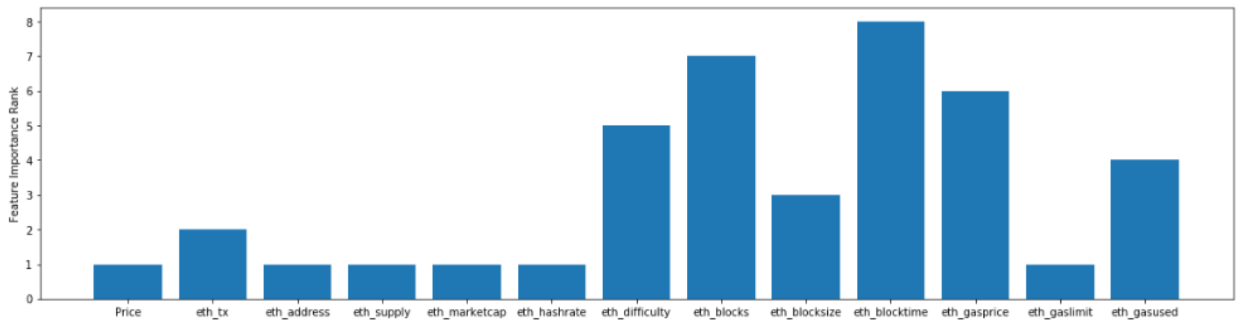


Figure 7: Ranked Features after Feature Selection

Price over time is the most important feature, followed by number of unique addresses. Unique address count is predictive, because the more unique addresses there are, the higher the price of Ether as more people can make Ether transactions. There is no cap on the supply of Ether, so its supply increasing as a function of time. It would make sense that Ether price should go down with increasing supply, however, less and less Ether is issued over time, making it more desirable. The supply could even go down when the rate of issuance of Ether is lower than the transaction fees. Ether market cap is the total dollar value of Ether, this is closely related to Ether price. Ether hash rate is important to miners, and it is related to the amount of time one can run the hash function per second. Having a high hash rate would indicate a high price of Ether as Ethereum currently uses Proof-of-Work. If Ethereum moves away from Proof-of-Work (more computational power yields quick results) to Proof-of-Stake (more money down yields quick results), then this feature will not be a defining feature anymore. Lastly Ether block size refers to the amount of space that is available within a block to store transactions. To summarize, refer to **Figure 8** for graphs of the top 6 features values over time.

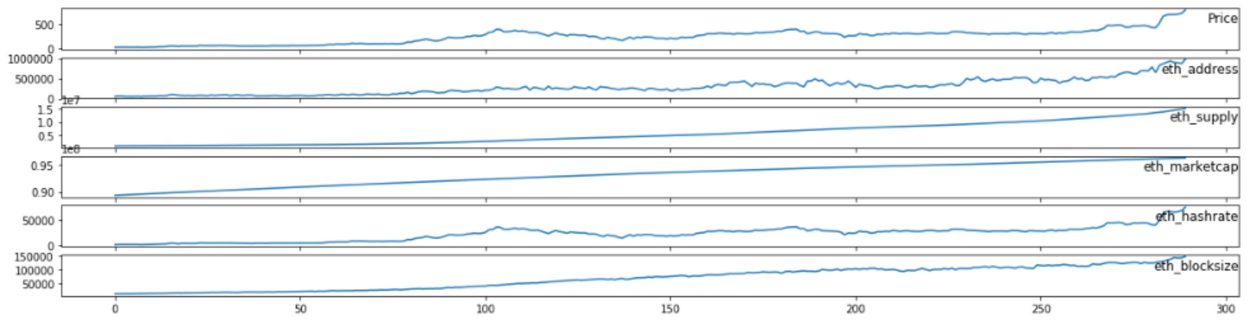


Figure 8: Top 6 Features

3.2 Implementation

3.2.1 Univariate ARIMA

1. Format the Ether data as time series with date as the index.

```
# Format time series data for ARIMA model with Date as the index
def create_arima_ts(df):
    return df[['Date', 'Price']].set_index('Date')

ts = create_arima_ts(df)
print(ts.head())
```

```
             Price
Date
2017-12-18  785.99
2017-12-17  717.71
2017-12-16  699.09
2017-12-15  693.58
2017-12-14  692.83
```

2. Check for stationarity of the data with the Augmented Dickey Fuller (ADF) test. The below function calculates and prints the ADF results.

```
def print_adf(ts):
    COLUMNS = ts.columns
    for column in COLUMNS:
        augmented_dickey_fuller = adfuller(ts[column])
        print('\nADF Results for column:', column)
        print('ADF Statistic: %f' % augmented_dickey_fuller[0])
        print('p-value: %f' % augmented_dickey_fuller[1])
        print('Critical Values:')
        for key, value in augmented_dickey_fuller[4].items():
            print('\t%s: %.3f' % (key, value))
```

3. The original time series data was not stationary, so first time-differences of the data were taken.

```python
def time_difference(ts):
    differenced_ts = ts.diff()
    differenced_ts.dropna(inplace=True)
    return differenced_ts
```

The data look stationary after the first time-difference, which is shown in **Figure 9**.

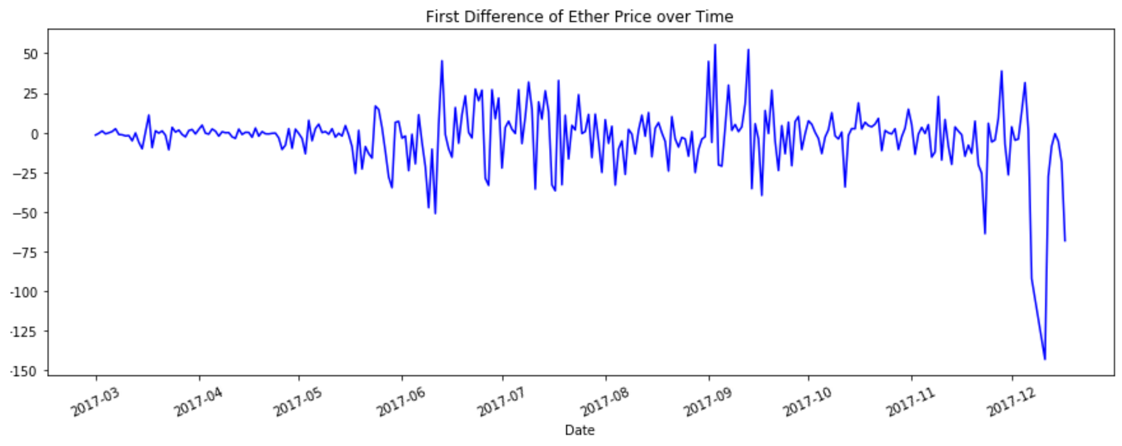


Figure 9: First time-differenced Ether data for univariate ARIMA

4. Find the correct parameters to be used in the ARIMA model by plotting the autocorrelation function (ACF) and partial autocorrelation (PACF) plots, shown in **Figure 10**, and determining when the lag is negative to get the values of the ARIMA hyperparameters. Since the lags are negative at 2, the lag value and residual error lag values for ARIMA are both 2.

```python
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
fig = plot_acf(x=ts_first_differences['Price'], lags=5)
fig = plot_pacf(x=ts_first_differences['Price'], lags=5)
```

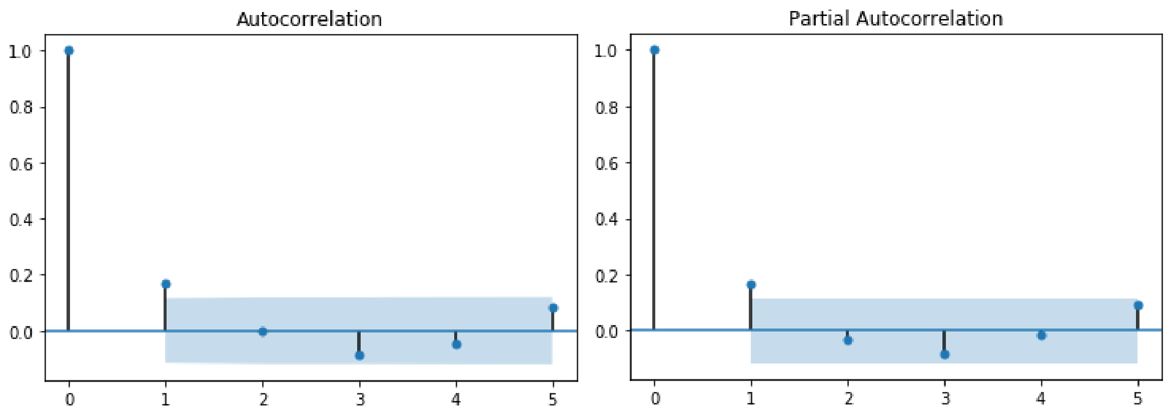
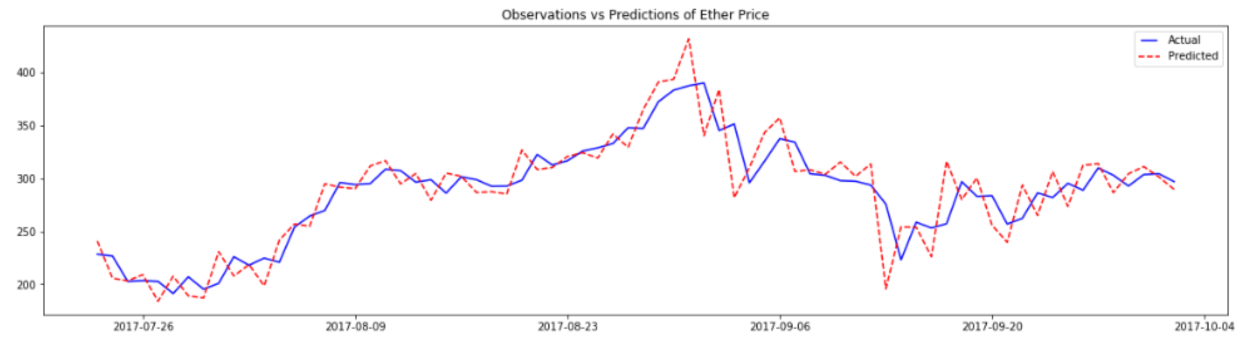


Figure 10: ACF and PACF plots to determine ARIMA hyperparameters

5. Perform Grid Search to verify the optimal hyperparameters for ARIMA. The optimal hyperparameters were found to be (p, d, q) = (2, 2, 0).

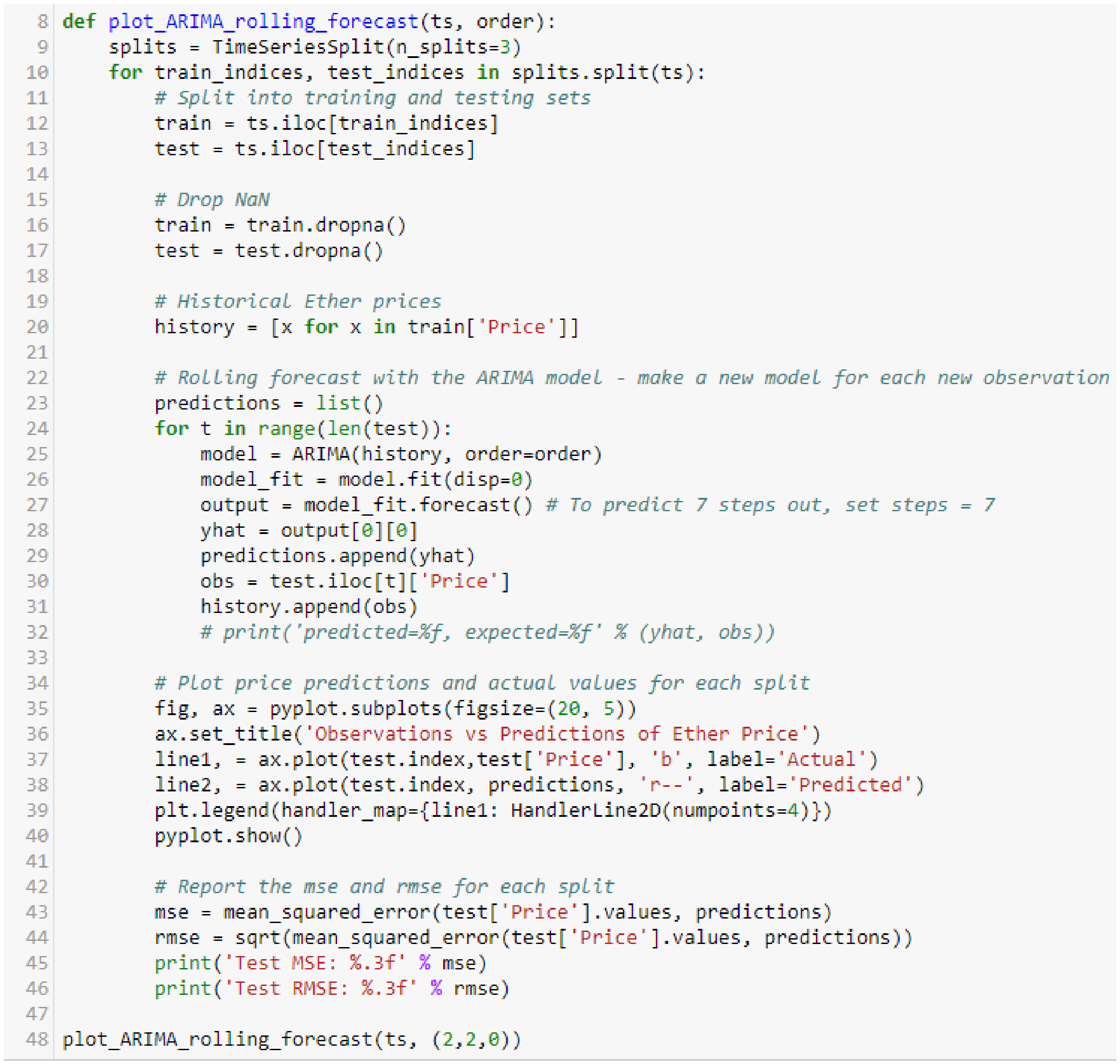
```python
def find_optimal_ARIMA_parameters(ts):
    p_values = [0, 1, 2]
    d_values = range(0, 3)
    q_values = range(0, 3)

    best_rmse = 10000 # some large number
    best_order = (0,0,0)
    for p in p_values:
            for d in d_values:
                for q in q_values:
                    order = (p,d,q)
                    try:
                        ARIMA_actual, ARIMA_predicted, model_fit = ARIMA_seven_day_forecast(ts_reversed, order)
                        prediction_summary(ARIMA_actual, ARIMA_predicted)
                        rmse = sqrt(mean_squared_error(ARIMA_actual, ARIMA_predicted))
                        if rmse < best_rmse:
                            best_rmse = rmse
                            best_order = order
                    except:
                        print("non stationary")
    return best_rmse, best_order
```

6. Perform a rolling forecast with cross validation using univariate ARIMA to see if the model can generalize well to new data. **Figure 11** shows one run of rolling forecast on a split.



```
Test MSE: 541.923
Test RMSE: 23.279
```

Figure 11: Univariate ARIMA rolling forecast



```python
def plot_ARIMA_rolling_forecast(ts, order):
    splits = TimeSeriesSplit(n_splits=3)
    for train_indices, test_indices in splits.split(ts):
        # Split into training and testing sets
        train = ts.iloc[train_indices]
        test = ts.iloc[test_indices]

        # Drop NaN
        train = train.dropna()
        test = test.dropna()

        # Historical Ether prices
        history = [x for x in train['Price']]

        # Rolling forecast with the ARIMA model - make a new model for each new observation
        predictions = list()
        for t in range(len(test)):
            model = ARIMA(history, order=order)
            model_fit = model.fit(disp=0)
            output = model_fit.forecast() # To predict 7 steps out, set steps = 7
            yhat = output[0][0]
            predictions.append(yhat)
            obs = test.iloc[t]['Price']
            history.append(obs)
            # print('predicted=%f, expected=%f' % (yhat, obs))

        # Plot price predictions and actual values for each split
        fig, ax = pyplot.subplots(figsize=(20, 5))
        ax.set_title('Observations vs Predictions of Ether Price')
        line1, = ax.plot(test.index,test['Price'], 'b', label='Actual')
        line2, = ax.plot(test.index, predictions, 'r--', label='Predicted')
        plt.legend(handler_map={line1: HandlerLine2D(numpoints=4)})
        pyplot.show()

        # Report the mse and rmse for each split
        mse = mean_squared_error(test['Price'].values, predictions)
        rmse = sqrt(mean_squared_error(test['Price'].values, predictions))
        print('Test MSE: %.3f' % mse)
        print('Test RMSE: %.3f' % rmse)

plot_ARIMA_rolling_forecast(ts, (2,2,0))
```

7. Perform a 7-Day forecast of Ether price with univariate ARIMA. The model does not generalize well to new data. This poor generalization is shown in **Figure 12**, since the actual and predicted values are not very close to each other.

```python
def ARIMA_seven_day_forecast(ts, order):
    # Get split index
    train_index = len(ts) - 7 # subtract last 7 days to predict

    # Split into Training and Testing Data
    train = ts[0:train_index]
    test = ts[train_index:len(ts)]

    # Fit ARIMA Model
    model = ARIMA(train['Price'], order=order)
    try:
        model_fit = model.fit(disp=0)
        forecast = model_fit.forecast(steps=7) # To predict 7 steps out, set steps = 7
        ARIMA_actual = test['Price'].values
        ARIMA_predicted = forecast[0]

        # Plot actual vs predicted values
        DATES = ts.index

        plt.figure(figsize=(15, 10))
        plt.plot(DATES[train_index:], ARIMA_actual, label='actual')
        plt.plot(DATES[train_index:], ARIMA_predicted, label='predicted')
        plt.legend(['actual', 'predicted'], loc='upper right')
        plt.show()
        return ARIMA_actual, ARIMA_predicted, model_fit
    except:
        print("non stationary")
```

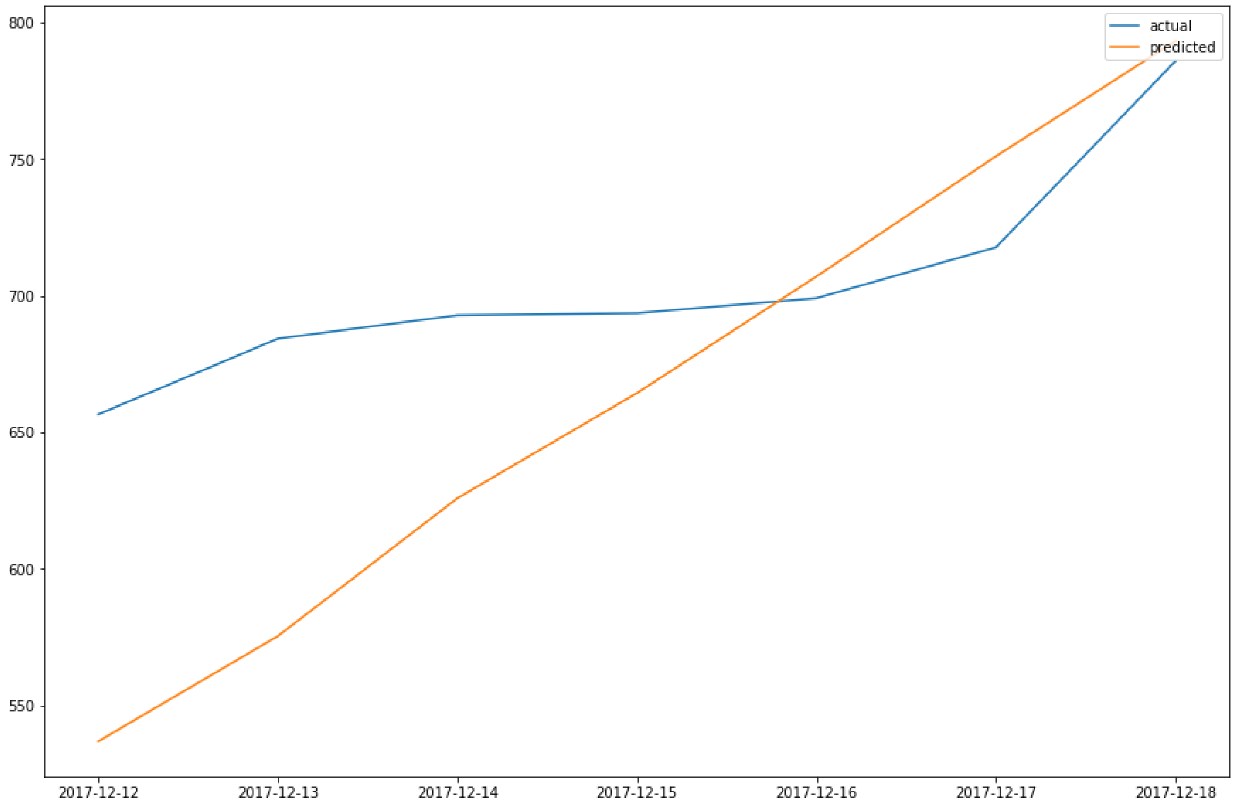


Figure 12: Univariate ARIMA 7-Day Forecast (RMSE = 68.45)

3.2.2 LSTM RNN

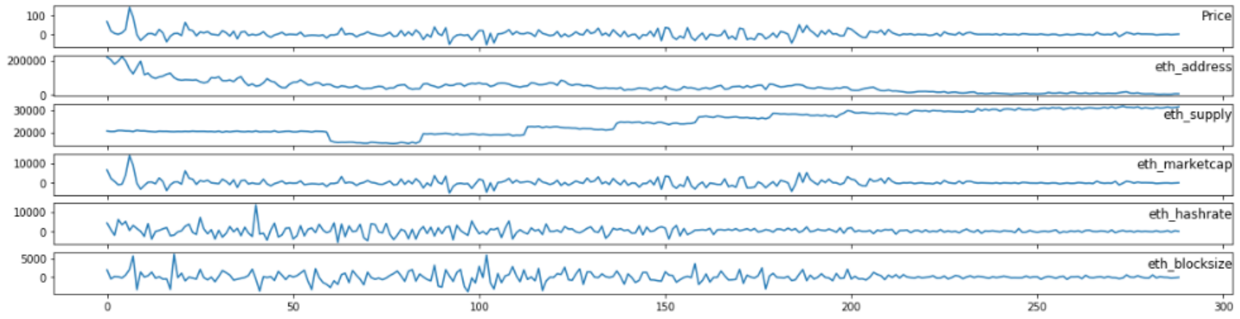
1. Using the features chosen as most prominent features by feature selection, create time series data that only contains those chosen features.

```python
def create_lstm_ts(df, selected_features):
    return df[['Date', *selected_features]].set_index('Date').iloc[::-1]
```

2. **run_lstm** creates a LSTM RNN model and runs predictions.

   Steps in **run_lstm**:

   a. Make the time series stationary. **Figure 13** shows the top features after time differencing.



   b. Split data into training and testing sets
   c. Apply min-max scaling on each set
   d. Split each set into inputs and output formats that the LSTM model accepts
   e. Create the LSTM model
   f. Make a prediction with the model
   g. Invert the min-max scaling on the predictions
   h. Invert the time differencing on the predictions
   i. Return the root mean squared error and plot the results

Below is the code that will create and run predictions on the LSTM RNN model:

```python
from keras.models import Sequential
from keras.layers import Dropout, Dense, LSTM
from sklearn.preprocessing import MinMaxScaler
from numpy import concatenate
pd.options.mode.chained_assignment = None

def difference_ts(ts, interval=1):
    stationary_ts = ts.copy(deep=True)
    COLUMNS = ts.columns
    for column in COLUMNS:
        diff = list()
        for i in range(interval, len(ts)):
            stationary_ts.loc[:, column][i] = ts.loc[:, column][i] - ts.loc[:, column][i - interval]
    return stationary_ts

def invert_time_difference(raw, yhat, interval):
    return yhat + raw[-interval]

def apply_minmax_scaling(train, test, features):
    # min max scaling
    SCALER = MinMaxScaler(feature_range=(-1, 1))
    TRAIN_SCALED = SCALER.fit_transform(train)
    TEST_SCALED = SCALER.fit_transform(test)

    # convert to DataFrame
    TRAIN_SCALED = pd.DataFrame(TRAIN_SCALED)
    TRAIN_SCALED.columns = features
    TEST_SCALED = pd.DataFrame(TEST_SCALED)
    TEST_SCALED.columns = features

    return TRAIN_SCALED, TEST_SCALED, SCALER

def invert_minmax_scaling(scaled_X_2d, scaled_y, scaler):
    # concat y and 2d X rows with no price, to a ts with all selected features
    scaled_Xy = concatenate((scaled_y, scaled_X_2d.loc[:, scaled_X_2d.columns != 'Price']), axis=1)
    inverted = scaler.inverse_transform(scaled_Xy)
    return inverted[:,0]

def train_test_split(ts, test_set_size):
    TRAIN = ts[:len(ts) - test_set_size]
    TEST = ts[-test_set_size:]
    return TRAIN, TEST

def input_output_split(train, test, feature):
    train_X, train_y = train, train[feature]
    test_X, test_y = test, test[feature]
    return train_X, train_y, test_X, test_y
```

```python
def reshape_as_3d(ts):
    return ts.values.reshape((ts.shape[0], 1, ts.shape[1]))

def reshape_as_2d(ts, features):
    ts = pd.DataFrame(ts.reshape((ts.shape[0], ts.shape[2])))
    ts.columns = features
    return ts
```

```python
def run_lstm(ts, epochs, batch_size, alpha):
    RAW = ts.copy(deep=True)

    FEATURES = ts.columns
    TEST_SET_SIZE = 7

    # make ts stationary
    STATIONARY = difference_ts(ts)

    # Split time series data into training and testing sets
    STATIONARY_TRAIN, STATIONARY_TEST = train_test_split(STATIONARY, TEST_SET_SIZE)

    # Apply min max scaling to stationary data
    SCALED_STATIONARY_TRAIN, SCALED_STATIONARY_TEST, SCALER = apply_minmax_scaling(STATIONARY_TRAIN, STATIONARY_TEST, FEATURES)

    # split into input and outputs to feed into lstm
    train_X, train_y, test_X, test_y = input_output_split(SCALED_STATIONARY_TRAIN, SCALED_STATIONARY_TEST, 'Price')

    # copy test x and test y before converting test x and test y to 3d for lstm
    test_X_copy = test_X.copy(deep=True)
    test_y_copy = test_y.copy(deep=True)

    # reshape input to be 3D [samples, timesteps, features] for lstm model
    train_X = reshape_as_3d(train_X)
    test_X = reshape_as_3d(test_X)

    fit_lstm_model(train_X, train_y, test_X, test_y, RAW, SCALER, FEATURES, epochs, batch_size, alpha)


EPOCHS = [300, 400, 500, 600, 700, 800, 900, 1000]
BATCH_SIZE = 128
ALPHA = 2
run_lstm(LSTM_TS, EPOCHS, BATCH_SIZE, ALPHA)
```

```python
from keras.models import Sequential
from keras.layers import Dropout, Dense, LSTM
from sklearn.preprocessing import MinMaxScaler
from numpy import concatenate
import matplotlib.pyplot as plt

def fit_lstm_model(train_X, train_y, test_X, test_y, RAW, SCALER, FEATURES, epochs, batch_size, alpha):
    TEST_SET_SIZE = len(test_y)
    TRAIN_SET_SIZE = len(ts) - TEST_SET_SIZE

    ALPHA = alpha # scaling factor for hidden layer neurons
    NUM_INPUT_LAYER_NEURONS = len(RAW.columns) # number of features
    NUM_OUTPUT_LAYER_NEURONS = 1 # features to predict
    HIDDEN_LAYER_NUM_NEURONS = int(TRAIN_SET_SIZE/(ALPHA * (NUM_INPUT_LAYER_NEURONS + NUM_OUTPUT_LAYER_NEURONS)))
    BATCH_SIZE = batch_size
    EPOCH_LIST = epochs

    # fit network and determine the optimal number of epochs according to lowest rmse
    lowest_rmse = 2^63 - 1
    best_epoch_num = EPOCH_LIST[0]

    print("Running LSTM on a list of epochs: ", EPOCH_LIST)
    for index, EPOCH in enumerate(EPOCH_LIST):
        # create the LSTM network
        model = Sequential()
        model.add(LSTM(HIDDEN_LAYER_NUM_NEURONS, input_shape=(train_X.shape[1], train_X.shape[2]), dropout=0.2))
        model.add(Dense(1))
        model.compile(loss='mae', optimizer='adam')
        history = model.fit(train_X, train_y, epochs=EPOCH, batch_size=BATCH_SIZE,
                            validation_data=(test_X, test_y), verbose=0, shuffle=False)

        print("\nNum Epochs: " + str(EPOCH))

        # plot history
        plt.figure(figsize=(20, 5))
        plt.plot(history.history['loss'], label='train')
        plt.plot(history.history['val_loss'], label='test')
        plt.legend(['train', 'validation'], loc='upper right')
        plt.show()

        # make a prediction
        model_output = model.predict(test_X)
    # reshape back to 2d
    test_X_2d = reshape_as_2d(test_X, FEATURES)
    test_y = pd.DataFrame(test_y)
    test_y.columns = ['Price']

    # invert scaling on forecast
    predictions = invert_minmax_scaling(test_X_2d, model_output, SCALER)

    # invert differencing on forecast
    inverted = list()
    for i in range(len(predictions)):
        value = invert_time_difference(RAW['Price'], predictions[i], len(predictions) - i + 1 )
        inverted.append(value)

    # calculate RMSE
    rmse = sqrt(mean_squared_error(RAW[-TEST_SET_SIZE:]['Price'], inverted))
    if rmse < lowest_rmse:
        lowest_rmse = rmse
        best_epoch_num = EPOCH

    plt.figure(figsize=(20, 5))

    LSTM_actual = RAW[-TEST_SET_SIZE:]['Price']
    LSTM_predicted = inverted

    # Plot actual vs predicted values
    plt.plot(LSTM_TS.index[-TEST_SET_SIZE:], LSTM_actual, label='actual')
    plt.plot(LSTM_TS.index[-TEST_SET_SIZE:], LSTM_predicted, label='predicted')
    plt.legend(['actual', 'predicted'], loc='upper right')
    plt.show()

    prediction_summary(LSTM_actual, LSTM_predicted)

print("\nBest Epoch Number is " + str(best_epoch_num))
print("Best RMSE " + str(lowest_rmse))
```

The most difficult part of LSTM RNN prediction was inversing the time differencing and tuning the model parameters. Inversing time differencing was more difficult since there wasn't a built-in python function for this operation. Initially, I kept using the wrong indices for the time differences, which skewed my results. Also, tuning model parameters was difficult, since it is very easy to overfit a LSTM model.

3.3 Refinement

3.3.1 ARIMA Model

Adjusting the hyperparameters for ARIMA improved results. Looking at the autocorrelation plot, partial autocorrelation plot, and the amount of time differencing needed to make the data stationary yielded (2,1,2) as the best hyperparameters. However, running ARIMA with these hyperparameters resulted in an error: "The computed initial AR coefficients are not stationary You should induce stationarity, choose a different model order, or you can pass your own start_params". Running grid search yielded optimal hyperparameters (2,2,0).

Figure 14 shows the results of ARIMA performed on an optimal order of (2,2,0). Its root mean squared error is 68.44.

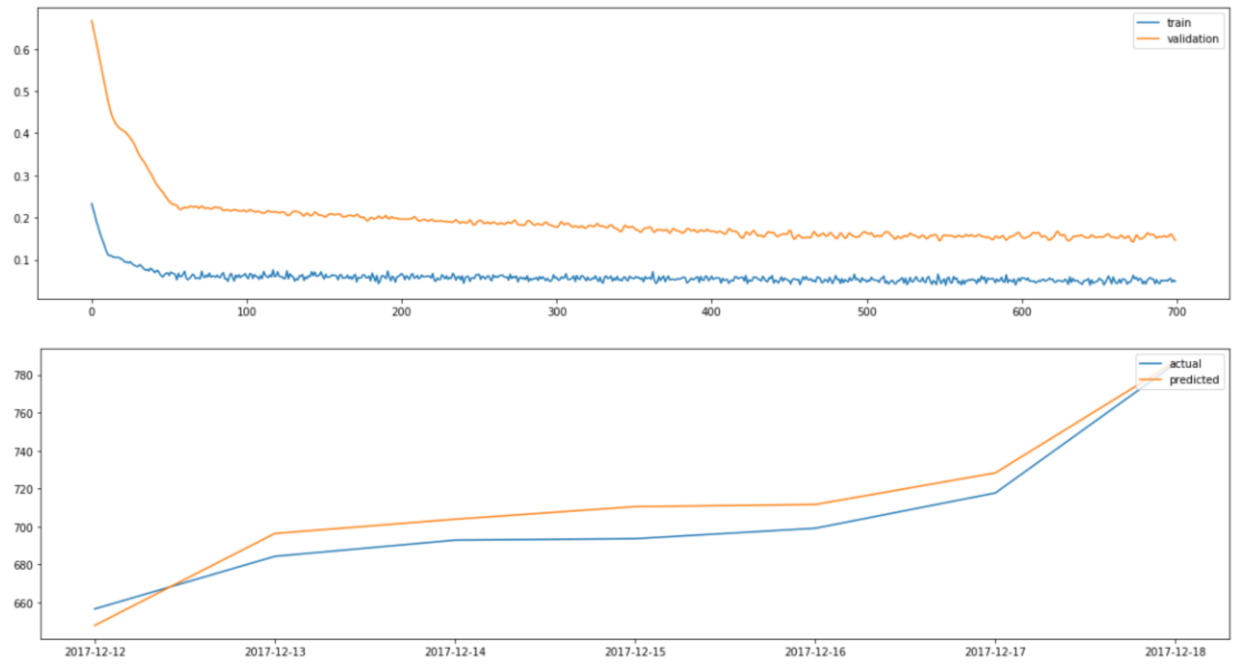
Actual	656.52	684.27	692.83	693.58	699.09	717.71	785.99
Predicted	536.77	575.30	625.83	664.32	707.04	751.09	793.27
Differences	119.75	108.98	67.00	29.26	7.95	33.38	7.28

Figure 14: ARIMA with order = (2,2,0)

3.3.2 LSTM RNN Model

Adjusting the number of epochs for the LSTM model improved the results. Predictions were run on epochs of size 300, 400, 500, 600, 700, 800, 900, and 1000. The best epoch number was 700 as it had the lowest root mean squared error of 11.33. **Figure 15** shows the LSTM model's predictions with an epoch size of 700.

```
Num Epochs: 700
```



```
actual: [656.52, 684.27, 692.83, 693.58, 699.09, 717.71, 785.99]
predicted: [647.83, 696.34, 703.87, 710.5, 711.62, 728.32, 787.01]
differences: [8.69, 12.08, 11.04, 16.92, 12.53, 10.62, 1.02]
test RMSE: 11.339
```

Figure 15: LSTM model prediction with 700 epochs

The table below shows the model performing on varying number of epochs and 0.2 dropout, with 700 epochs performing the best:

Epochs	300	400	500	600	**700**	800	900	1000
RMSE	13.84	13.07	11.96	12.55	**11.34**	11.98	12.01	13.47

4 Results

4.1 Model Evaluation and Validation

The LSTM RNN model is chosen as the final model. This model was chosen because of its ability to learn long term dependencies in the data. The model is relatively robust, since it does well predicting farther into the future than the univariate ARIMA model. The best ARIMA model had a root mean squared error (RMSE) of 68.44 for a 7-day forecast (**Figure 16**). The best LSTM RNN model with 0.4 dropout had a RMSE of 16.88 and 500 epochs for a 7-day forecast (**Figure 17**).

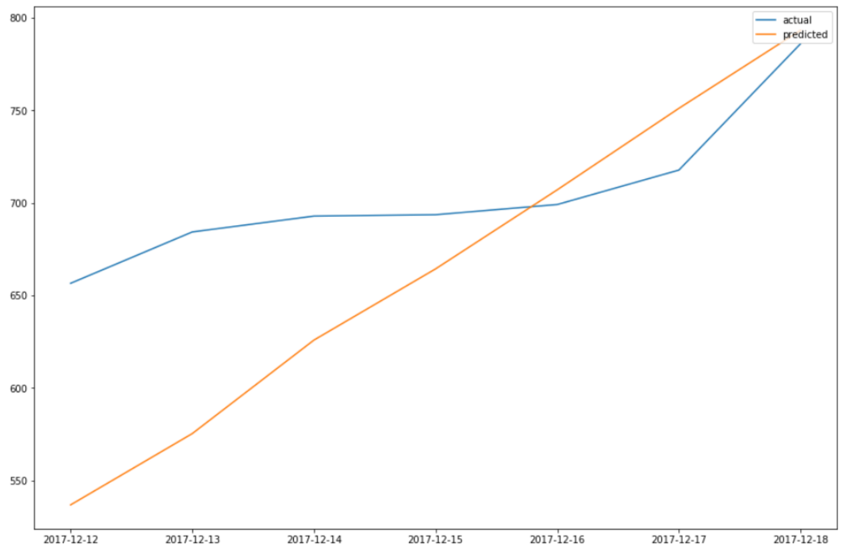


Figure 16: ARIMA 7-Day Forecast

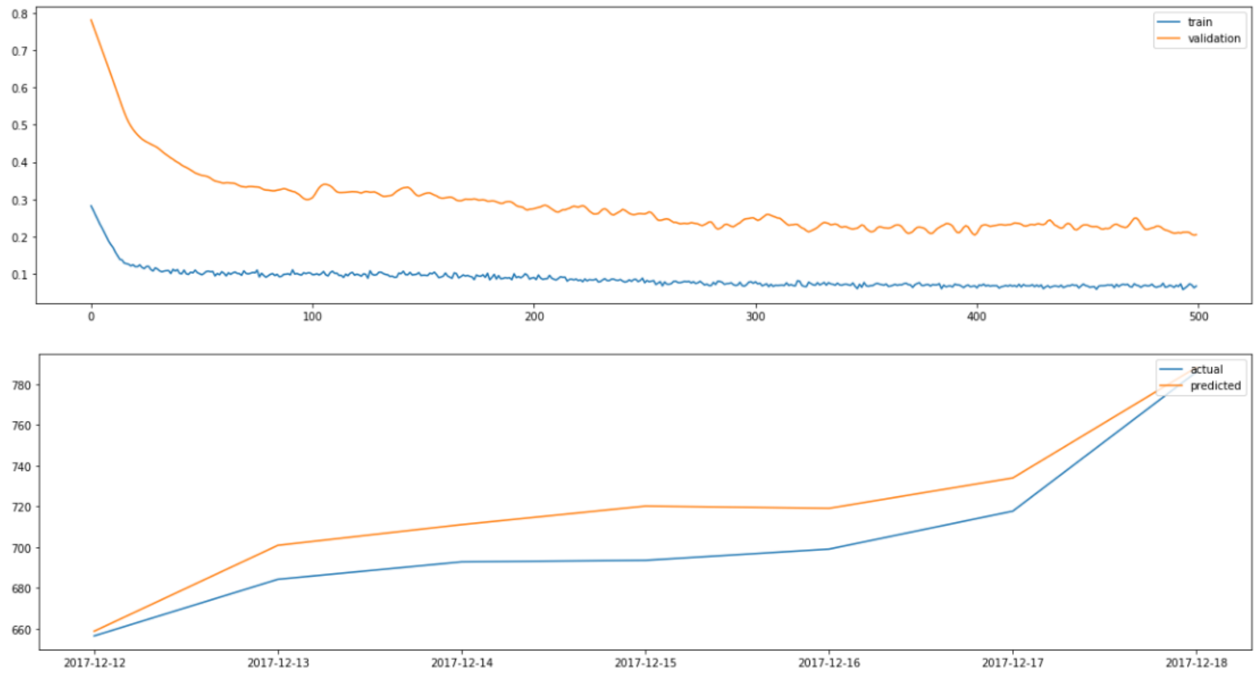


Figure 17: LSTM RNN 7-Day Forecast

In addition, varying the number of input data points, still allows the model to make reasonable predictions. For example, splitting the data into 80% train and 20% test, yields a root mean squared error of 13.45. Results of this split is in **Figure 18**. The actual and predicted values for this model are close to each other, so this model could be overfitting the data. Reducing the number of epochs could alleviate this problem.

```
Num Epochs: 200
```

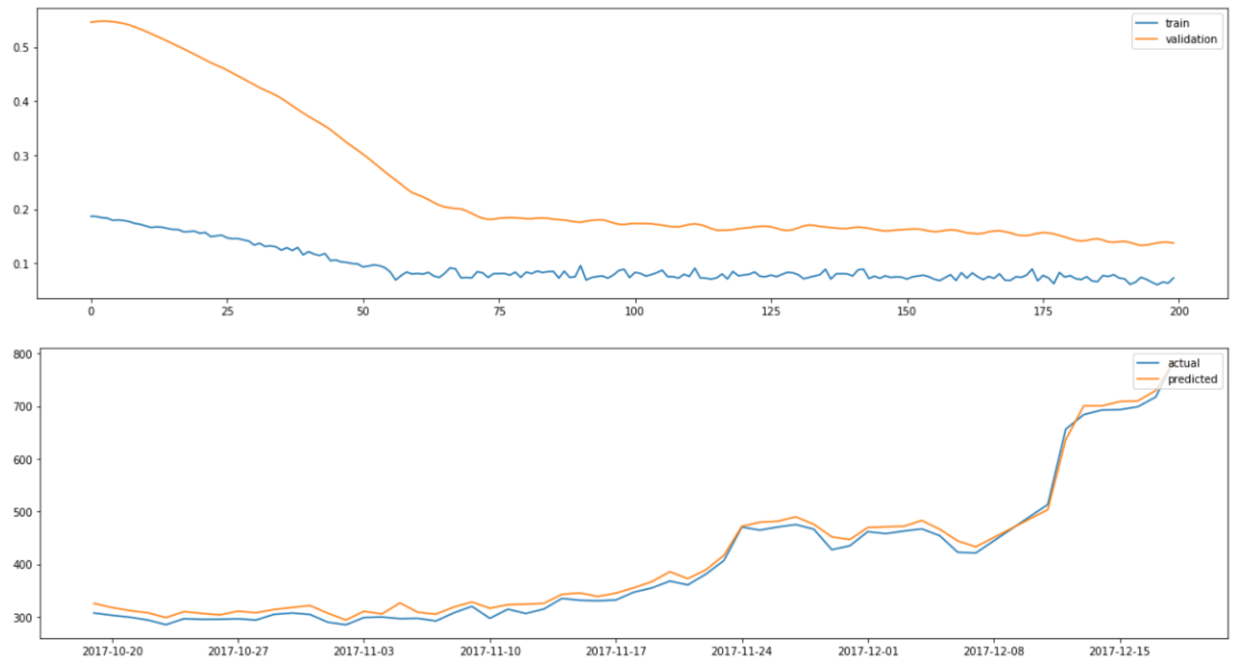


Figure 18: LSTM Model Prediction with 80% train 20% test, 200 epochs, dropout of 0.4

The results found from this model can be more trusted if they are not predictions far into the future. Careful choices of model parameters are also essential, otherwise this model will overfit and not generalize well to new data.

4.2 Justification

The benchmark model predicted the price of bitcoin with a window n-steps ahead. This is shown in **Figure 19**.

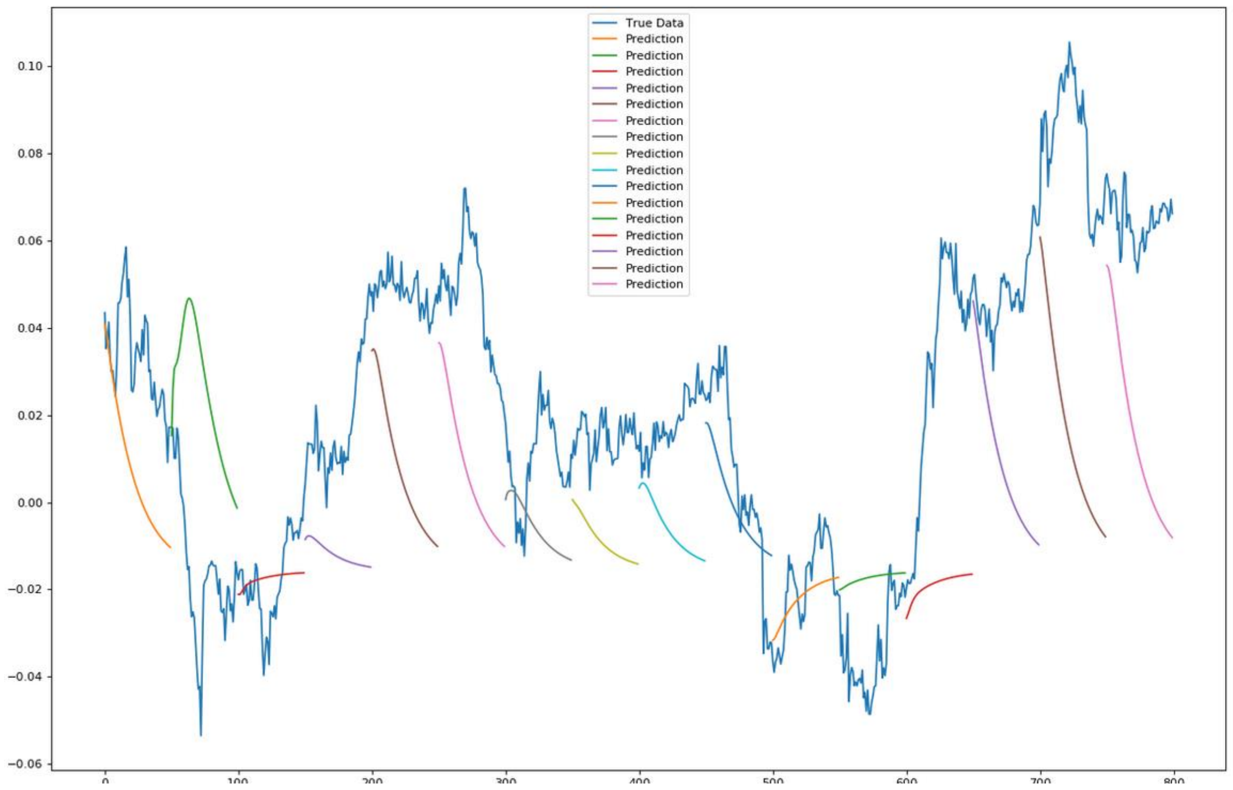


Figure 19: Benchmark model predicting 50 time-steps from different points

Comparing the benchmark model with my model that predicted 50 timesteps into the future with 80 epochs is shown in **Figure 20**. The RMSE is 19.40.

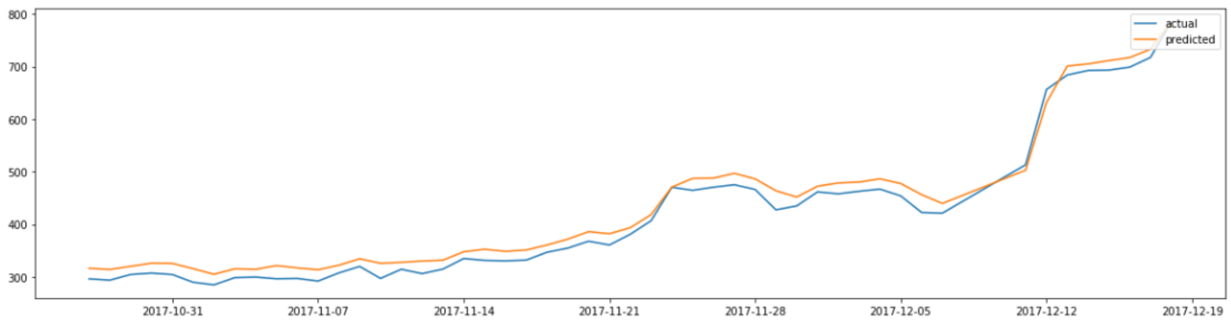


Figure 20: LSTM RNN model predicting 50 time-steps from the last training point

I don’t have a RMSE for the benchmark to compare with my LSTM model, but looking **Figure 19** and **Figure 20**, it looks like my model predicts better 50 timesteps into the future. This could be since the benchmark model only used Open, Close, Volume (BTC) and Volume (Currency) as features, whereas I had used price, eth_marketcap, eth_hashrate, and eth_blocksize as features. I believe that my model predicts better than the benchmark model through looking at the difference in actual and predicted values on each graph.

5 Conclusion

5.1 Free-Form Visualization

Figure 15 shows an autocorrelation plot of the Ether time series data. Autocorrelation determines the amount of the relationship between a current observation and previous observations. There is not an obvious autocorrelation trend in the plot. It seems that earlier lags have larger autocorrelation. This makes sense, because the first few months of Ether price data showed generally low Ether price data with low variation in price. Thus, there is a relationship between early Ether price timeseries data points. However, in later months, Ether price data was a lot more random, so the autocorrelations for later timeseries data is closer to 0 than the autocorrelations in earlier months which were much larger in magnitude. The first lag has a large negative autocorrelation and around the 20th lag there is a large positive autocorrelation.

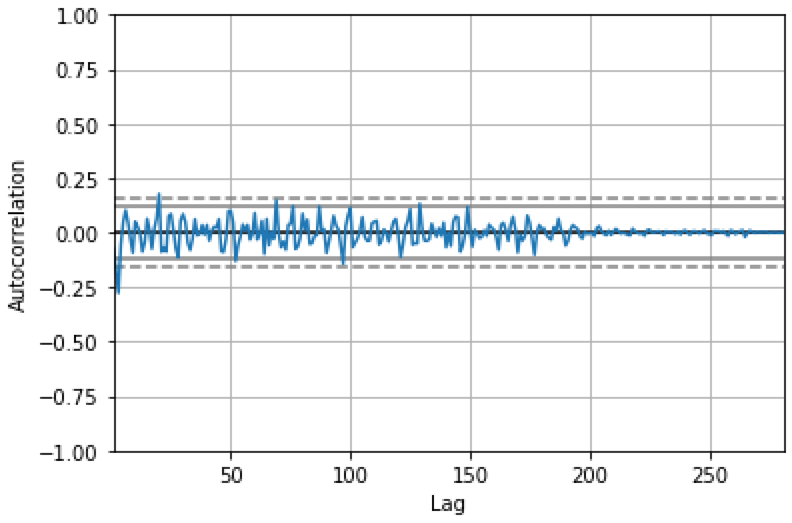


Figure 15: Autocorrelation Plot of Ether Time Series (March 1, 2017 – December 18, 2017)

5.2 Reflection

To summarize the process, Ether data from March 2017 to December was extracted from Etherscan. Then, time-indexed timeseries DataFrames of Ether data were created. Price predictions were made from univariate ARIMA and LSTM RNN models. For univariate ARIMA, the data was made stationary and hyperparameters were selected. Then, the ARIMA model was built and used in prediction. The univariate ARIMA model did not do very well at generalizing to new data, since it only had price as the sole feature and not many data points. To improve prediction, a LSTM RNN model was used. Feature selection was applied to reduce dimensionality of the problem. This was necessary, because there were not many data points. A time series DataFrame was constructed only with these selected features. Then, the time series data was differenced to make it stationary. After that, scaling was applied so that one feature did not overpower the others during model creation. When creating a LSTM RNN model, it was important to select a good number of epochs, batch size, number of hidden layer neurons, etc. so that the model would not be over or under trained. After creating the model, predictions were made. These predictions needed to be unscaled and have time differencing removed, so that they were scaled the same as the original data. Both ARIMA and LSTM RNN models outputted their respective RMSE's. The most difficult part of the project was training a LSTM RNN. The part that took the longest was inverting the time differencing on the data. At first, I had differenced the data twice to make the time series stationary, but found it hard to invert the data twice. I ended up time differencing once and inverting once, but removed two features that were not stationary after one time-difference.

5.3 Improvement

To improve predictions, I need a lot more data. 290 data points are not enough to make a very accurate prediction, especially considering I would like to train with multiple different features. In addition, I can improve on the model by tuning LSTM RNN model parameters. For instance, I can find an optimal batch size, alpha (used in determining the number of hidden layer neurons), and dropout percentage. I also want to make my LSTM RNN model more generalizable, as I think it may be a little overtrained.

6 References

[1] https://github.com/ethereum/wiki/wiki/White-Paper
[2] https://en.wikipedia.org/wiki/Blockchain
[3] https://spectrum.ieee.org/image/MjY0NzUzNw.png
[4] https://stats.stackexchange.com/questions/292059/the-unit-of-root-mean-square-error-rmse
[5] https://people.duke.edu/~rnau/411diff.htm
[6] https://www.digitalocean.com/community/tutorials/a-guide-to-time-series-forecasting-with-arima-in-python-3
[7] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4152592/
[8] https://bitcoinmagazine.com/articles/ether-price-analysis-chinas-ico-ban-may-lead-further-pull-backs/
[9] https://www.cryptocoinsnews.com/ethereum-price-achieves-new-time-high-518-cryptokitties-effect/
[10] https://stats.stackexchange.com/questions/222584/difference-between-feedback-rnn-and-lstm-gru
[11] http://www.jakob-aungiers.com/articles/a/Multidimensional-LSTM-Networks-to-Predict-Bitcoin-Price