Predicting the Value of Ether over Time

Stephanie Marker

January 4, 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 (Buterin et al. 2017). 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.

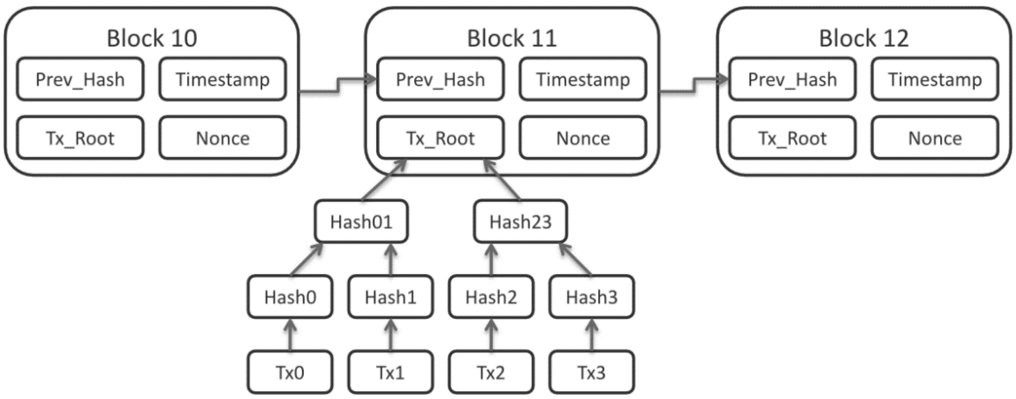


Figure 1: Blockchain (Wikipedia 2017)

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 (Wikipedia 2017). 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**.

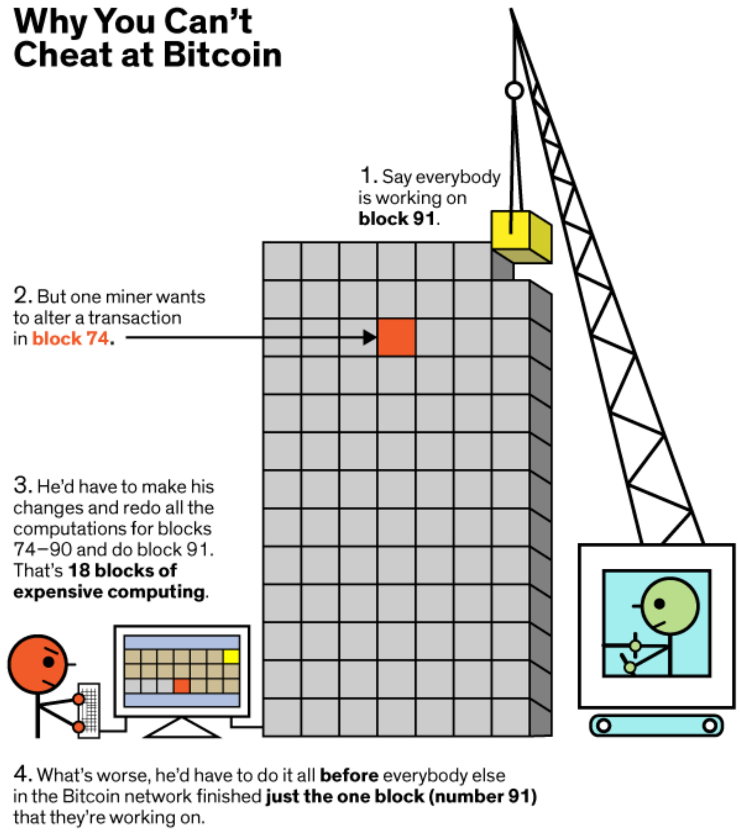


Figure 2: Blockchain Security (Peck 2015)

Though many other popular cryptocurrencies like Bitcoin and Litecoin also use blockchain, What makes Ethereum stand apart is its concept of a smart contract. Smart contracts hold account information on the blockchain. They contain Turing-complete code that can do almost anything. For example, a smart contract can be written so that it can interact with other contracts, store data, or send Ether to others (Ethereum Project). The powerful nature of smart contracts is shown in the success of CryptoKitties. CryptoKitties is a decentralized app built with Ethereum smart contracts that allows users to buy, sell, and breed kitties on the Ethereum blockchain. The success of this app contributed to a large Ether price surge (Crypto Coin News 2017). Given the large success of CryptoKitties and other external factors that influence Ether price, it would be interesting to see if Ether price can be forecasted.

To forecast Ether price, data were aggregated from two sources Etherscan and Github. The dataset used in this project is taken as CSVs from Etherscan's charts pages and as JSON output using Github's GraphQL API (Etherscan, Github). The data will be used to forecast the price of Ether a week into the future with supervised learning. Univariate ARIMA and LSTM RNN models will be used to predict Ether price and their performance will be compared. The model with lower root mean squared errors over multiple splits of training and testing data will be a better predictor of Ether price, since that model would generalize well and have a better fit.

These two models were chosen because of their success in several studies. For instance, ARIMA models are used for timeseries forecasting problems such as stock price prediction. For example, there was a study that built ARIMA models to predict the Nokia and Zenith Bank stock prices. The conclusion was that the ARIMA model had potential to satisfactorily predict short-term stock prices (Ariyo et al. 2014). In addition, LSTM RNNs did well predicting price movement of stocks in a study using data from the Brazilian stock exchange. They found that the model had high accuracy but could have had lower variance to be a more reliable model. They found that the LSTM RNN models were better at detecting high variation in price for stocks than low variation (Nelson et al. 2017).

To further investigate Ether price forecasting, I would have liked to use a CNN-sliding window model, since unlike LSTMs, this model does not use previous history in the learning process. Having previous Ether price history from beyond a time window in learning would affect prediction. Prior information could muddle prediction and not allow the model to accurately fit to the highly dynamic price changes. LSTM, RNN, and CNN-sliding window models were compared in a study on Infosys, TCS, and Cipla stock price prediction. They found that the CNN model performed the best, since it used most recent information for prediction. The sudden changes in price made LSTM and RNN models perform worse, since these models held past information and tried to use past trends to predict current price changes. CNNs used the current time window for prediction and did not depend on prior information for predictions (Selvin et al. 2017).

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. The amount of differencing will determine the I parameter of the ARIMA model.
        ii. Find the correct parameters to apply the ARIMA model.
            - Determine whether the process is AR or MA using ACF and PACF charts
            - Find the degree of the AR (or MA) series.
            - Verify these parameters with grid search

Figure 3 summarizes the approach in step 4a (i and ii):

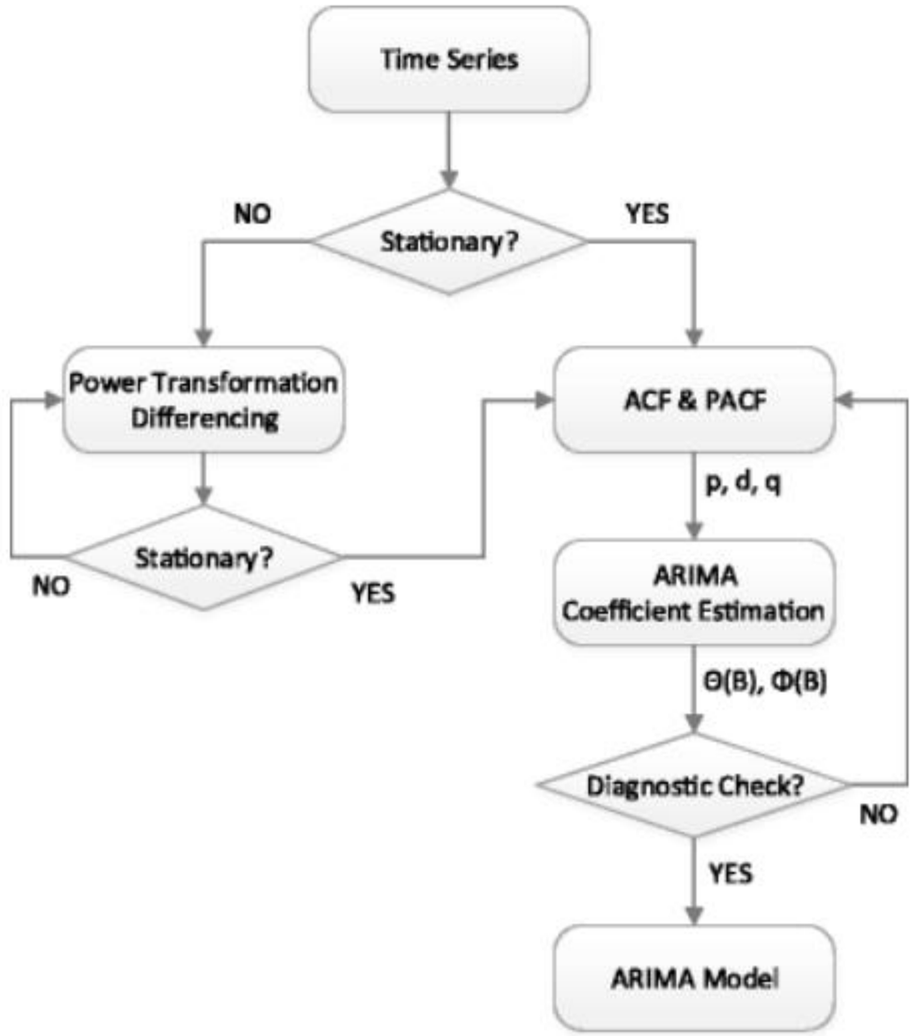


Figure 3: Finding ARIMA Model Coefficients (Demir and Ergen 2016)

iii. Fit a univariate ARIMA model.
iv. Check the model's residuals and make sure they are independent. Also, check that the mean and variance over time is constant.
v. 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. Fit 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 their root mean squared errors.

This approach is a good fit for forecasting Ether price because the ether data is timeseries so ARIMA and LSTM models will be suitable for this type of data. In addition, the both my ARIMA and LSTM RNN sections of my approach follows the Box-Jenkins method commonly used for timeseries models. **Figure 4** delineates the Box-Jenkins method (Makridakis and Hibon 1997).

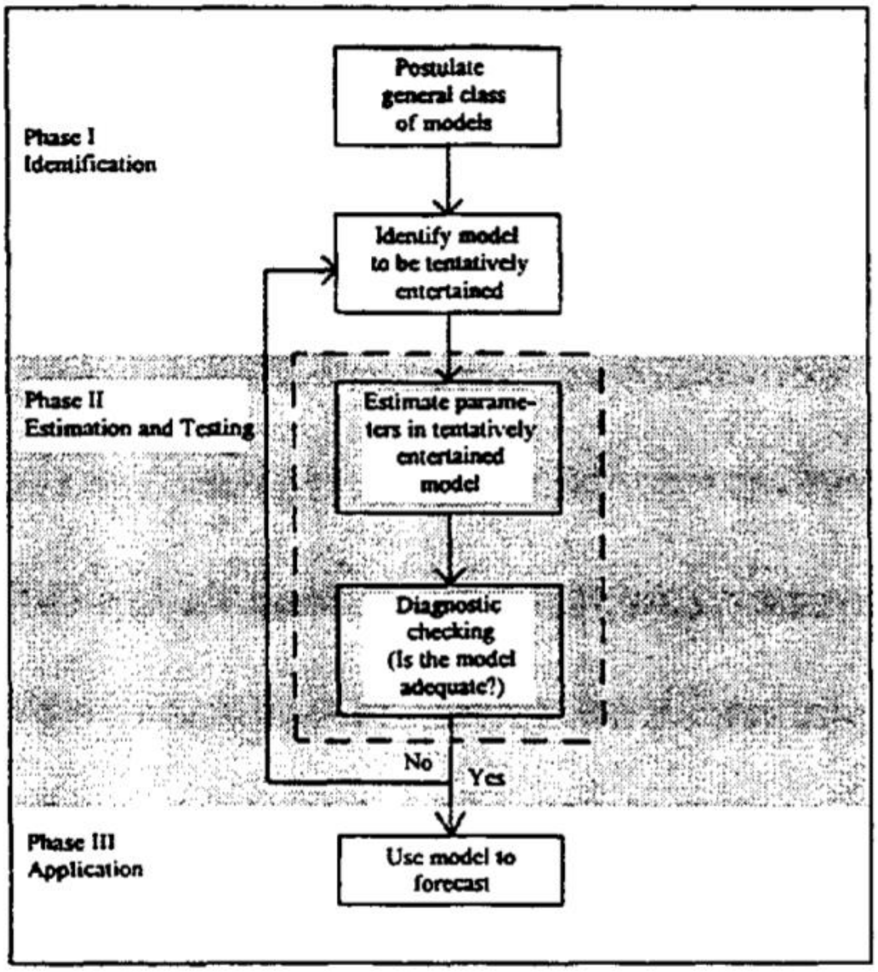


Figure 4: Box-Jenkins Method

Steps in my ARIMA section of the approach that follow this process are highlighted below:

1. **Model Identification**
   a. Check for stationarity of data and do time differencing if necessary until the data are stationary
   b. Find the degree of the AR (or MA) series
2. **Parameter Estimation and Testing**

    a. Find coefficients that best fit the ARIMA model using the maximum likelihood estimation by fitting an ARIMA model
    b. Test the model by making sure that the sure residuals are independent of each other and that the mean and variance is constant over time
3. **Application**
    a. Forecast with the ARIMA model

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) (Villegas 2017). 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**. This dataset contains 290 data points and 14 features. The 5 most recent entries of Ether data are shown below in **Figure 5**. Features I will explain in more detail include ether transaction count and ether gas price. Other features included in **Figure 5** 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 5: 5 Most Recent Ether Data Points

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 6**. 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.

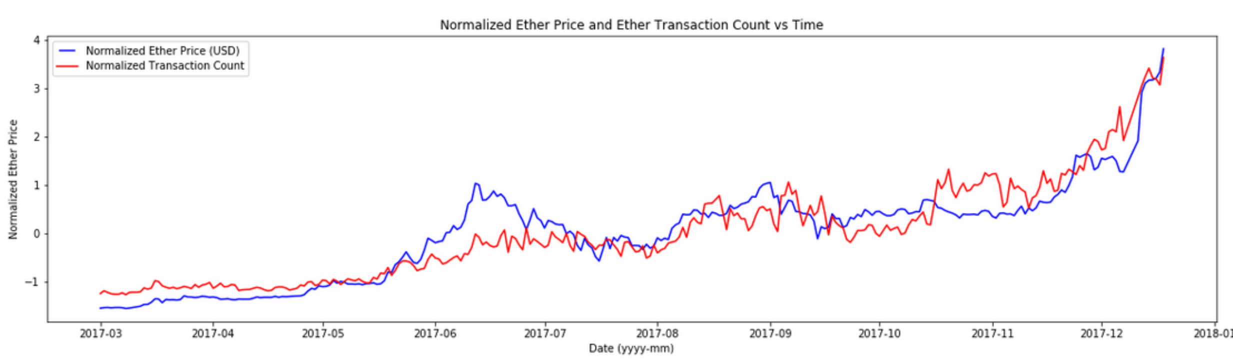


Figure 6: Normalized Ether Price and Ether Transaction Count over Time

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 7**. 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.

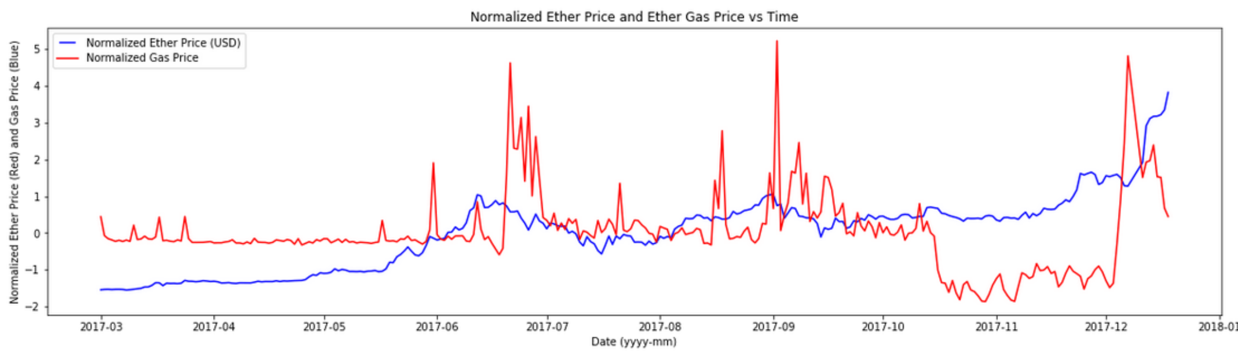


Figure 7: Normalized Ether Price and Ether Gas Price over Time

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. ARIMA models are common for "short term forecasting requiring at least 40 historical data points. It works best when your data exhibits a stable or consistent pattern over time with a minimum amount of outliers" (Morrison 2017). The Ether dataset for this problem contains 290 historical data points. It does not display a consistent pattern over time, so this model might not give very accurate results for this problem, but it still will be good to use as a baseline predictor as predictions will be made over a short period of time.

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. For data to be stationary the mean, variance, autocorrelation, etc. should all be constant over time. **Figure 8** shows an example of stationary timeseries data achieved after taking a first time-difference.

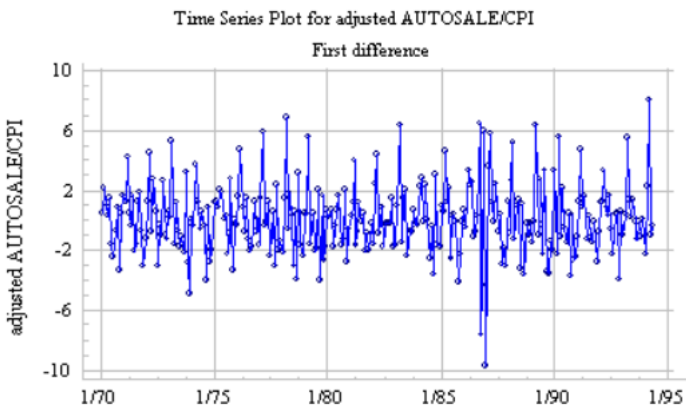


Figure 8: Example of Stationary Time Series Data (Nau 2017)

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 (Vincent 2017).

To find the values of p and q, one can look at the autocorrelation function (ACF) and partial autocorrelation function (PACF) plots.

The x-axis of both ACF and PACF plots corresponds to the lag, which is the time span between the timeseries observations.

The y-axis of the ACF plot corresponds to the autocorrelation values. The autocorrelation values are a measure of the similarity between observations as a function of the time lag between them. These values comprise of both direct and indirect correlations.

The y-axis of the PACF plot corresponds to the partial autocorrelation values. Brownlee points out that the partial autocorrelation values are the "summary of the relationship between an observation in a time series with observations at prior time steps with the relationships of intervening observations removed" (Brownlee 2017).

For example, suppose one has data on movie ticket sales, income, and people's ages and wants to see if there is a relationship between movie ticket sales and local people's income. If this person did not account for age when calculating a correlation coefficient between ticket sales and income, this would give a misleading result, because income might be numerically related to age, which then might be numerically related to movie ticket sales. Therefore, a measured correlation between movie ticket sales and income might be influenced by other correlations like age. Partial autocorrelation would avoid this issue, because it would remove intervening observations (Wikipedia 2017).

Given the ACF and PACF values, one can find what parameters p and q to use for the ARIMA model. Looking at the ACF and PACF plots and identifying where these values drop off can help one identify whether the model will be an AR or MR process.

If the ACF plot has a gradual decrease and PACF plot drops off sharply, then the model should be an AR process. **Figure 9** is an example of an AR process. Specifically, it is an AR(2) process, since the PACF chart cuts off on the 2nd lag.

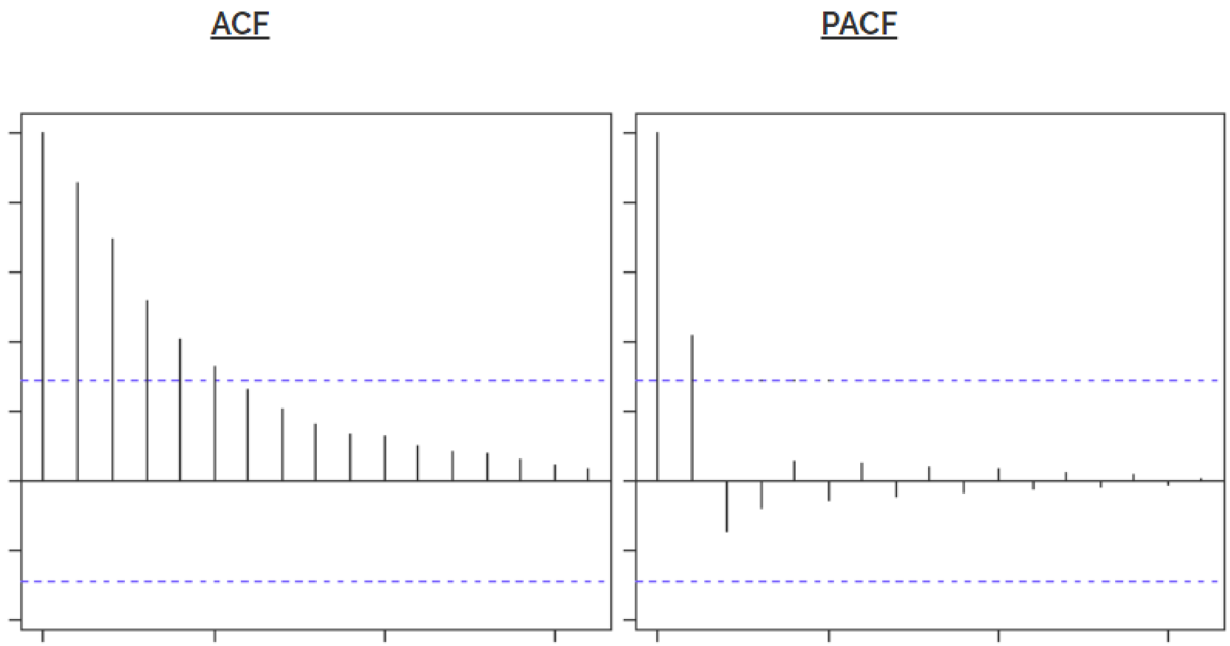


Figure 9: Example of an AR Process

If the ACF plot drops off sharply and the PACF plot has a gradual decrease, then the model should be a MA process. **Figure 10** shows an example of a MA process. In this case, it is a MA(2) process, since it cuts off on the 2nd lag on the ACF chart (Srivastava 2015).

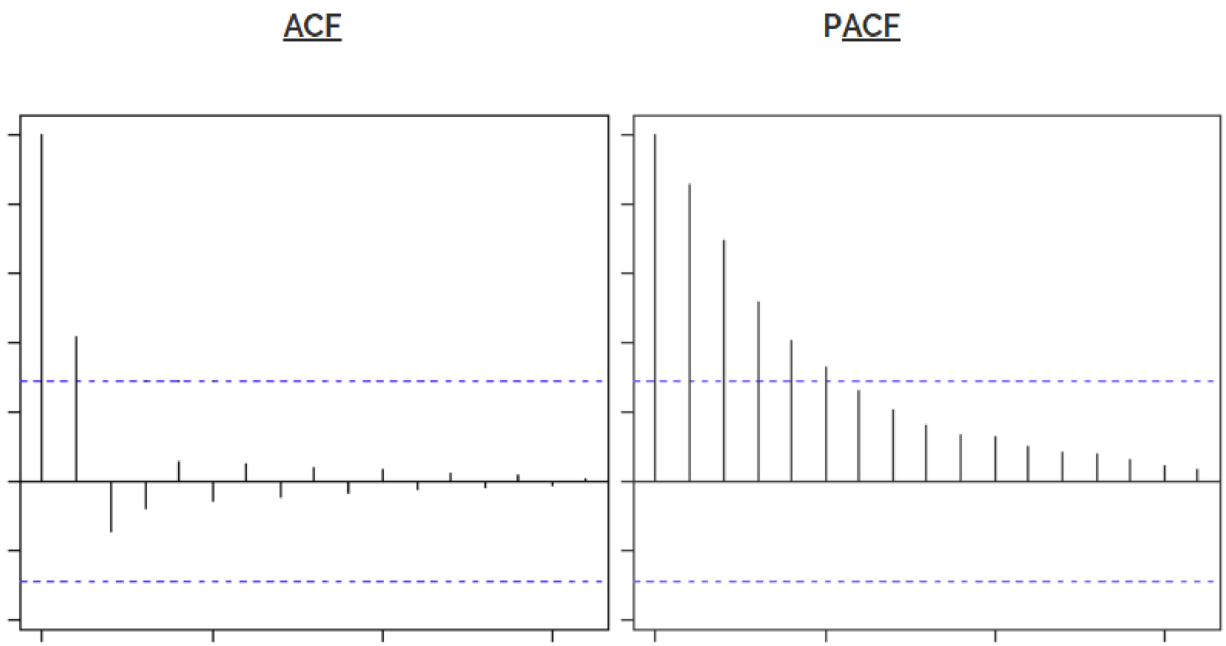


Figure 10: Example of a MA Process

ARIMA tries to describe the changes in a stationary timeseries as a function of autoregressive (AR), moving average (MA) parameters, and differencing (I). Given these parameters, one can write the ARIMA forecasting function.

The ARIMA forecasting function over different difference values is given below:

$Let:$
$y = the\ d^{th}\ difference\ of\ Y$
$Y_t = the\ timeseries$
$Y_{t-1} = the\ timeseries\ lagged\ one\ period$

$d = 0: \quad y_t = Y_t$
$d = 1: \quad y_t = Y_t - Y_{t-1}$
$d = 2: \quad y_t = (Y_t - Y_{t-1}) - (Y_{t-1} - Y_{t-2}) = Y_t - 2Y_{t-1} + Y_{t-2}$

In the following equations, the moving average parameters are negative by from Box and Jenkins's convention. The general forecasting equation with the moving average parameters is defined by:

$Let:$
$c = some\ constant\ (not\ included\ if\ the\ mean\ of\ Y\ is\ 0)$
$e = the\ error\ term\ of\ the\ model$
$a = the\ autoregressive\ parameter$
$m = the\ moving\ average\ parameter, negative\ by\ convention\ from\ Box\ and\ Jenkins$

$$d = n: \quad y_t = c + a_1 Y_{t-1} + \cdots + a_n Y_{t-n} - m_1 e_1 - \cdots - m_n e_{t-n}$$

Table 1 shows some examples of forecasting functions for ARIMA(p,d,q) models with different AR, MA, and I parameter values (Nau 2017):

Type of Model	**ARIMA(p,q,r)**	**Forecasting Function**
AR(1) model	ARIMA(1,0,0)	$Y_t = c + a * Y_{t-1}$
MA(1) model	ARIMA(0,0,1)	$Y_t = c - m * e_{t-1}$
Random walk	ARIMA(0,1,0)	$Y_t = c + Y_{t-1}$
Differenced AR(1)	ARIMA(1,1,0)	$Y_t - Y_{t-1} = c + a(Y_{t-1} - Y_{t-2})$ $Y_t = c + Y_{t-1} + a(Y_{t-1} - Y_{t-2})$

Table 1: Examples of ARIMA(p,d,q) Forecasting Functions

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. (Kane et al. 2014)

One example of an external event that would introduce complexity 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 (Bitcoin Magazine 2017). In addition, CryptoKitties, one of the first popular Ethereum-based decentralized apps where users can buy and breed new, rare cats. CryptoKitties 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 (Crypto Coin News 2017).

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 (RNN)

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 (Doug 2010):

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

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

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.

To understand how a LSTM RNN model works, first one should understand a RNN. The following information and figures are largely drawn from an excellent blog post from Christopher Olah on LSTM RNNs.

Unlike traditional neural networks, RNNs contain loops which allows information to persist. One can think of a RNN as many copies of the same network, each passing information to the next copy of the network. Visually, this is shown in **Figure 11**.

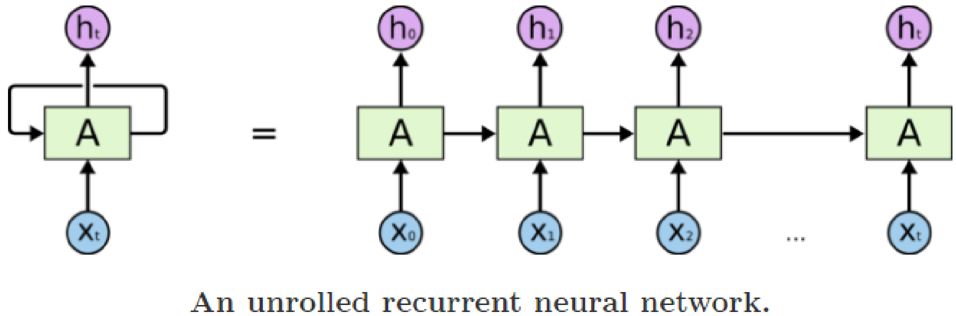


Figure 11: A Regular RNN

In **Figure 11**, A represents a piece of the neural network, x_t denotes some input at time t, and an output value at time t is denoted h_t. The RNN can be thought of like a chain of events, or snapshots in time. Each unit in the chain is called a cell. A regular RNN will only contain one layer in depicted in **Figure 12**.

RNNs can perform well when trying to predict information that only relies on short-term past events. However, when the neural network needs information from early points in the chain, far from the most recent information, the RNN will be unable to learn how to logically connect the information and make a valid prediction.

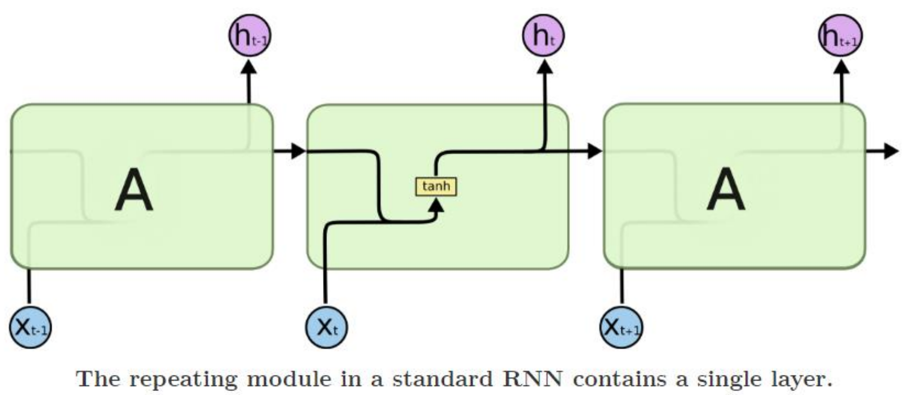


Figure 12: A RNN contains a Single Layer

LSTM RNNs can learn long term dependencies. Like regular RNNs, LSTMs are arranged in a chain, but each neural network copy in the chain contains 4 layers. **Figure 13** shows a diagram of a LSTM RNN. In **Figure 13**, a line represents a transfer of a vector from the output of one node to the input of other nodes.

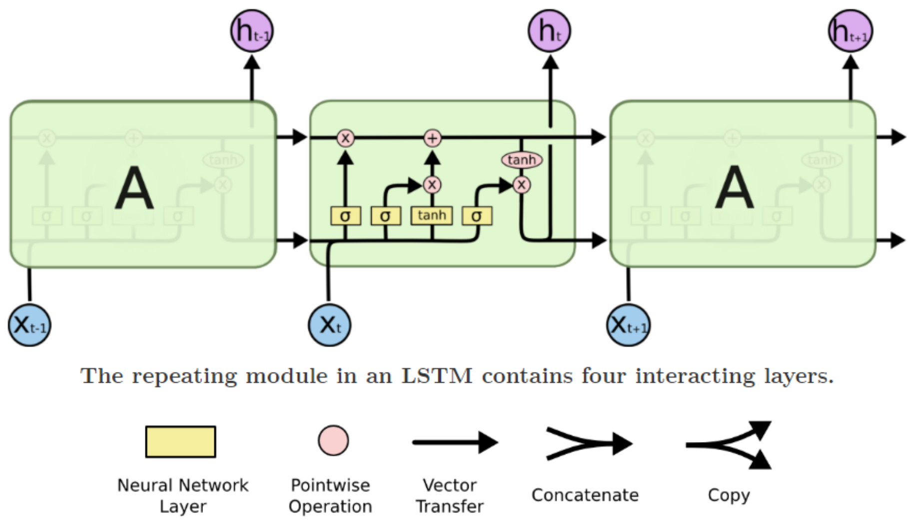


Figure 13: LSTM RNN

The dark line in **Figure 13** shows how the LSTM RNN keeps state. The green unit in **Figure 14**, the cell, holds the cell state at time t, which is denoted C_t. The previous cell state, C_{t-1}, is passed to the current cell. The pink circles denote logic gates that help determine whether information from C_{t-1} will be passed to C_t. In addition to the logic gates, there's a sigmoid layer which would output numbers between 0 and 1. This number will be multiplied by C_{t-1} and determine how much of the previous state will remain.

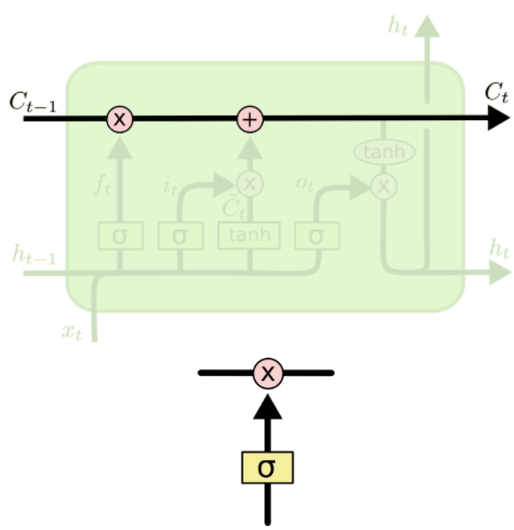


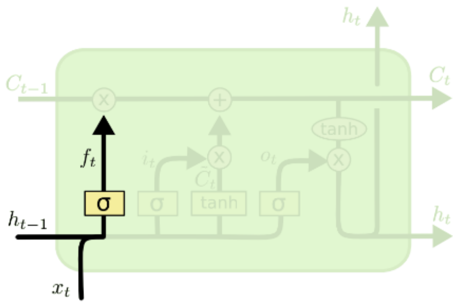
Figure 14: A LSTM cell maintains state C_t. The amount of the state that persists is controlled by a sigmoid and multiplication logic gate.

To tie the entire process together, the four steps in a regular LSTM RNN will be described next.

Steps in a LSTM RNN:

Let:
$C_t = cell\ state\ at\ time\ t$
$h_{t-1} = neural\ network\ output\ at\ time\ t - 1$
$x_t = neural\ network\ input\ at\ time$ t
$f_t = forget\ gate\ output\ at\ time\ t$
$\sigma = sigmoid\ layer\ output\ (value\ between\ 0\ and\ 1)$
$W = neural\ network\ layer\ weights$
$b = neural\ network\ layer\ biases$

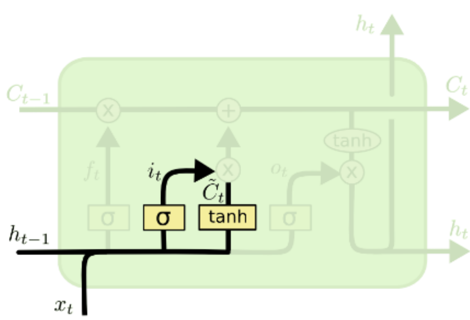
Step 1: The sigmoid layer "forget gate" will determine the cell state information to discard



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

The previous output $h_{t-1}$and current input x_t are concatenated to yield $W_f \cdot [h_{t-1}, x_t] + b_f$ for the forget layer. Then, the forget layer, σ, will output a number between 0 and 1, where 0 means completely forget the state and 1 means completely remember the state. These values are multiplied to give the forget gate output, f_t.

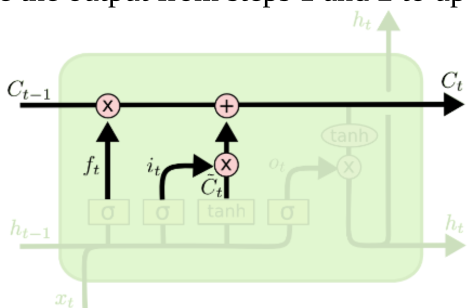
Step 2: Determine what new information will be stored in the cell state
2a: The sigmoid layer "input gate layer" will determine what values to update
2b: A $tanh$ layer creates a vector of new values, $\tilde{C}_t$, that could be added to the state



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

The previous output $h_{t-1}$and current input x_t are concatenated to yield $W_i \cdot [h_{t-1}, x_t] + b_i$ for the input layer. Then, the input layer, σ, decides the values that will be provided as input through the sigmoid function. These two values are multiplied to give the input gate output, i_t. The new values that can be added to the state, $\tilde{C}_t$ are found through the tanh layer which will make the values fall between −1 and 1.

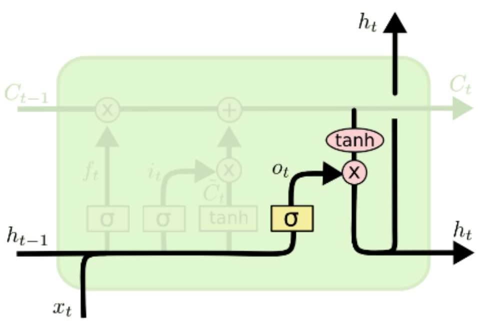
Step 3: Combine the output from steps 1 and 2 to update to the state



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

To update the cell state, multiply the old state by the output of the forget gate and then add it to the new values to be added but scaled by the output of the input gate: $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$.

Step 4: Determine what to output based on our cell state and a filter provided by another sigmoid layer.



$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

The sigmoid layer will determine what values of the cell state to output: $o_t = W_o \cdot [h_{t-1}, x_t] + b_o$. The cell state, C_t then will be put through a $tanh$ layer. Lastly, these two values will be multiplied together to yield h_t, the neural network output at time t (Olah, 2015).

Now that the process of how LSTM RNNs hold state information is delineated, it is evident how LSTMs are able to store information in memory for a long time, thus allowing them to learn long-term time dependencies (user20160, Stack Exchange 2016). This is one of the advantages of LSTM RNNs over regular RNNs when it comes to relying on past information.

Drawbacks of a LSTM RNNs are that they can be slow to train and that one 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 (Brownlee 2017). 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

The benchmark for this study will be a univariate ARIMA model. There's a total of 290 Ether timeseries data points. This model will be trained on 283 data points, with the 7 most recent dates in the test set that will be predicted to provide a 7-day Ether price forecast. The performance of this model can be compared through its root mean squared error. A lower root mean squared error and ability of the model to generalize well to new data is desirable.

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 15**. A low rank means that the feature contributes strongly to the price of Ether, so low numbered ranks are best.

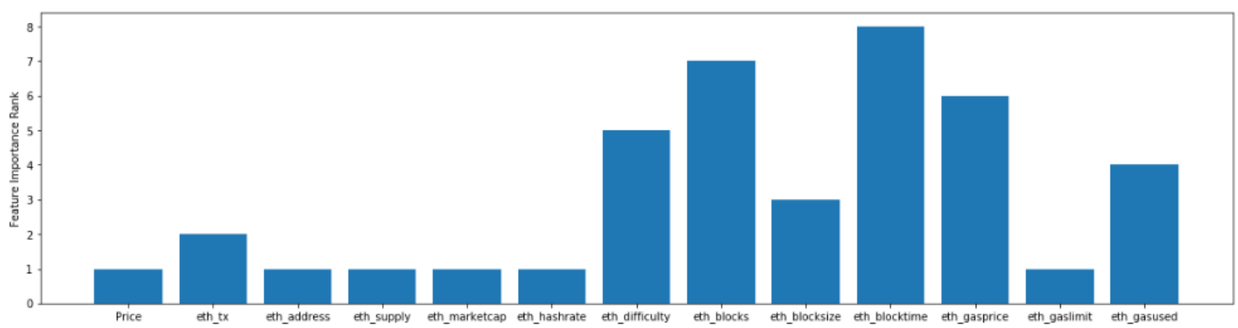


Figure 15: Ranked Features after Feature Selection

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 16** for graphs of the top 6 features values over time.

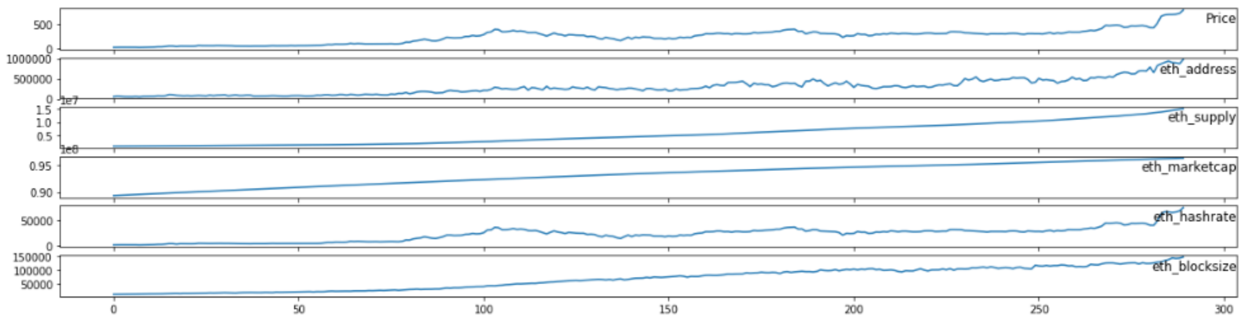


Figure 16: Top 6 Features

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 17**.

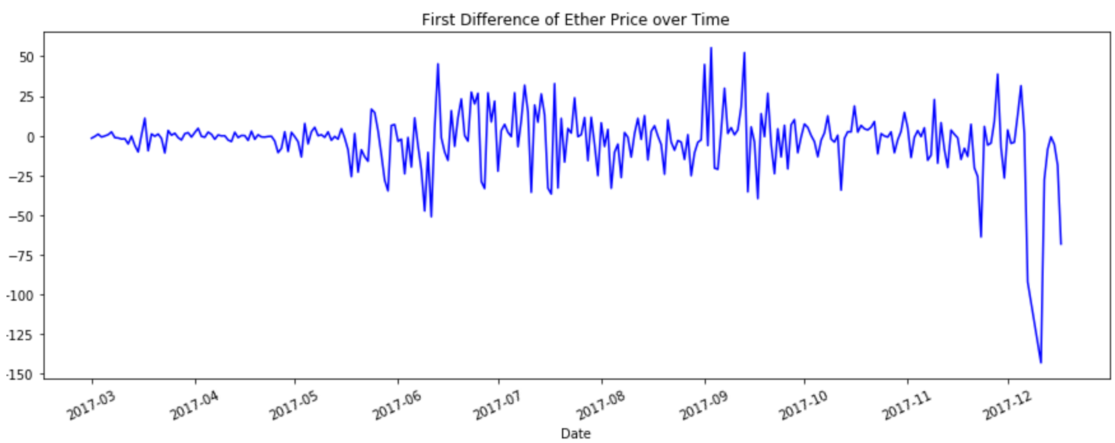


Figure 17: First time-differenced Ether Data for Univariate ARIMA

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 18**, 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)
```

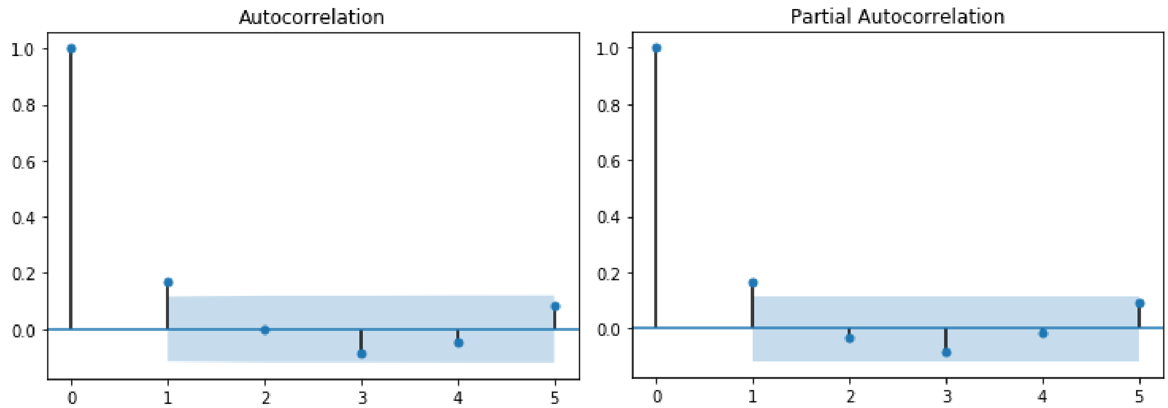


Figure 18: ACF and PACF Plots to Determine ARIMA Hyperparameters

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

```
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. Look at the residual plot and make sure that there is not much bias in the prediction. **Figure 19** shows the residual plot. The mean is around 0, which means there is not much bias in the prediction.

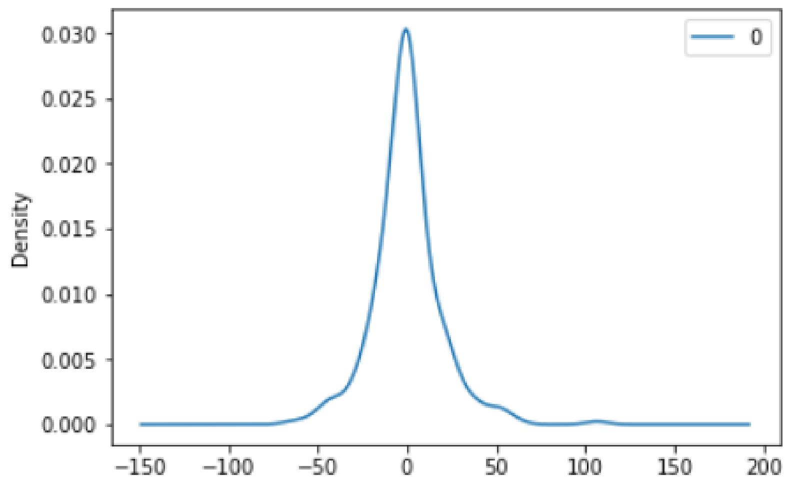
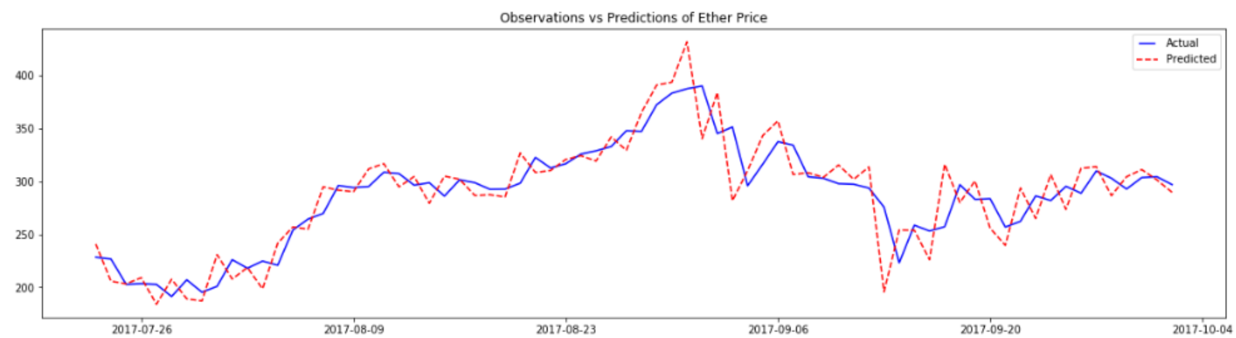


Figure 19: Residual Plot

7. Perform a rolling forecast with cross validation using univariate ARIMA to see if the model can generalize well to new data. **Figure 20** shows one run of rolling forecast on a split.




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

Figure 20: Univariate ARIMA Rolling Forecast

```
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))
```

8. 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 21**, 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")
```

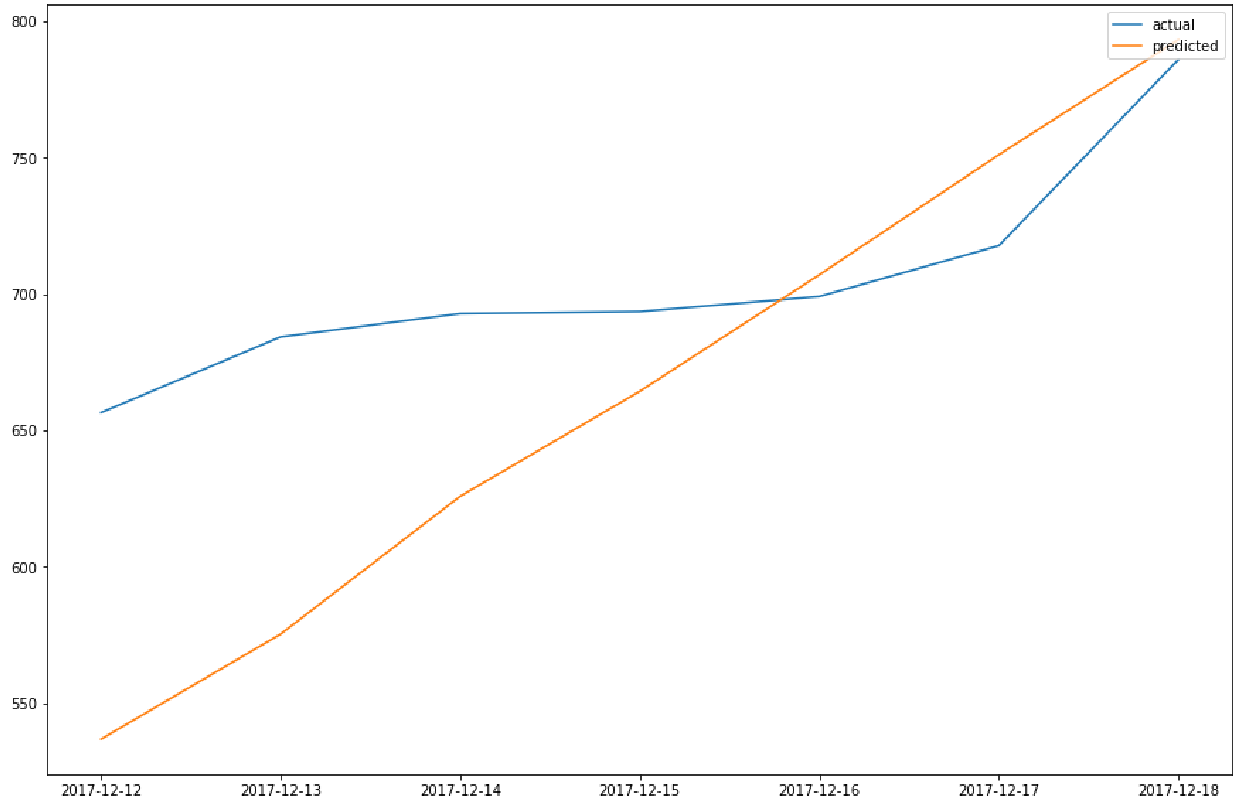


Figure 21: Univariate ARIMA 7-Day Forecast (RMSE = 68.45)

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.

```
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**:

   c. Make the time series stationary. **Figure 22** shows the top features after time differencing. Ether address and ether supply are not stationary after taking a time difference, so these two features will be dropped. The other four features look stationary.

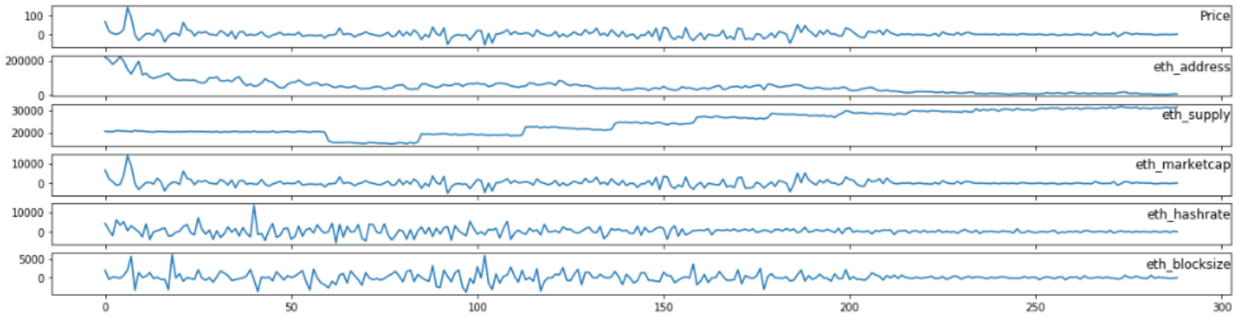


Figure 22: Top 6 Features After Time Differencing

d. Split data into training and testing sets
e. Apply min-max scaling on each set
f. Split each set into inputs and output formats that the LSTM model accepts
g. Create the LSTM model
h. Make a prediction with the model
i. Invert the min-max scaling on the predictions
j. Invert the time differencing on the predictions
k. 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:

```
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
```

```
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, test_set_size, epochs, batch_size, alpha, dropout):
    RAW = ts.copy(deep=True)

    FEATURES = ts.columns
    TEST_SET_SIZE = test_set_size

    # 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, FEATURE

    # 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, dropout)

TEST_SET_SIZE = 7
EPOCHS = [3500, 4000, 4500, 5000, 5500]
BATCH_SIZE = 32
ALPHA = 2
DROPOUT_PERCENTAGE = 0.5
run_lstm(LSTM_TS, TEST_SET_SIZE, EPOCHS, BATCH_SIZE, ALPHA, DROPOUT_PERCENTAGE)
```

```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, dropout):
    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()

        # Create a LSTM hidden layer that also specifies the input layer via the input_shape
        model.add(LSTM(HIDDEN_LAYER_NUM_NEURONS, input_shape=(train_X.shape[1], train_X.shape[2])))
        model.add(Dropout(dropout))
        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='training')
        plt.plot(history.history['val_loss'], label='testing')
        plt.legend(['training error', 'validation error'], 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).

Table 2 shows the results of the univariate ARIMA model on an optimal order of (2,2,0). Its root mean squared error is 68.44.

Days into the Future	1	2	3	4	5	6	7
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

Table 2: Comparison of ARIMA Ether Price 7-Day Forecast with order = (2,2,0) and Actual Prices

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 3,500, 4,000, 4,500, 5,000, and 5,500. The best epoch number was 5,000 as it had the lowest root mean squared error of 3.19. **Figure 23** shows the LSTM model's training and error values over 5,000 epochs. **Figure 24** shows the model's 7-day price forecast with an epoch size of 5,000.

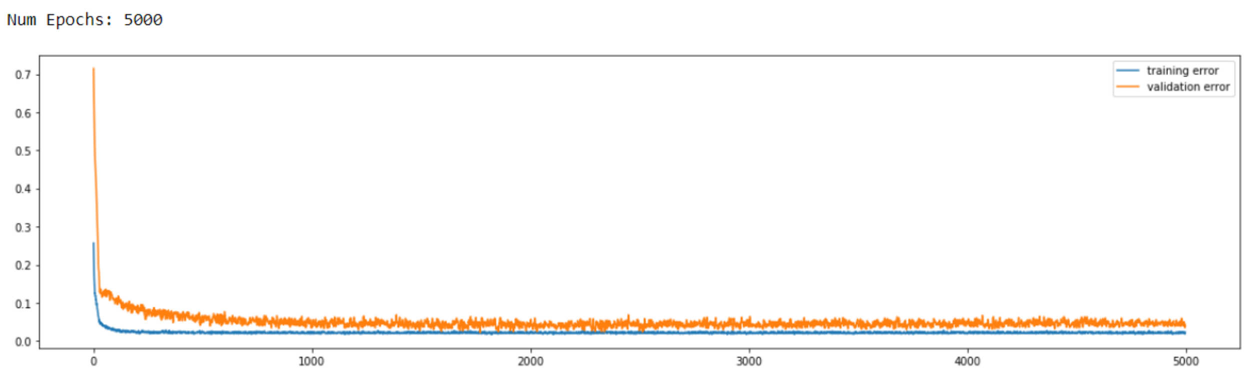


Figure 23: LSTM RNN Training and Validation Error over Epochs

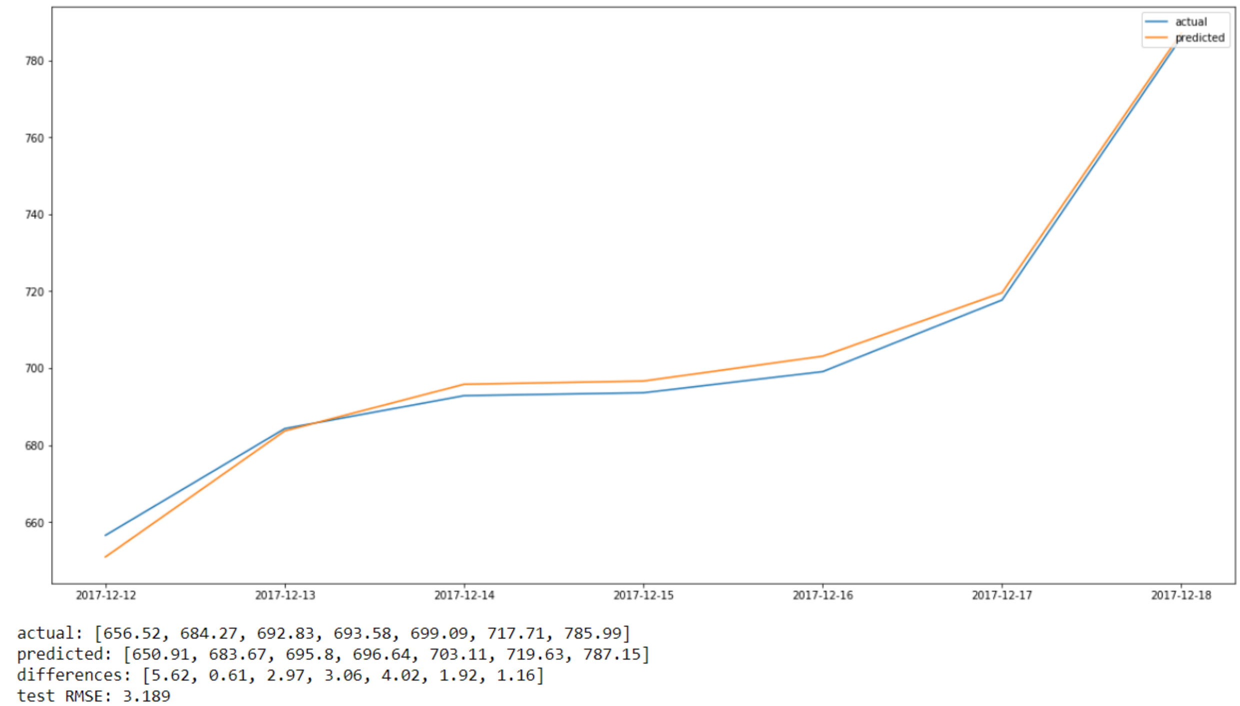


Figure 24: LSTM RNN Model 7-day Ether Price Forecast with 5,000 Epochs

Table 3 shows the model performing on varying number of epochs, 0.5 dropout and a batch size of 32, with 5,000 epochs performing the best:

Number of Epochs	3,500	4,000	4,500	5,000	5,500
RMSE	4.74	4.18	4.32	3.19	3.55

Table 3: LSTM RNN Model RMSE over a Varying Number of Epochs

4 Results

4.1 Model Evaluation and Validation

The LSTM RNN model is chosen as the final model. This model was originally chosen because of its ability to learn long term dependencies in the data. After more reflection, I do not think that a LSTM RNN would be the best choice of model for this problem due to the wild nature of Ether price changes. Because of this high volatility, I believe a CNN sliding-window model would perform better than a LSTM RNN model, because a CNN sliding-window model would use the most recent information in price prediction. Due to sudden changes in price, earlier price changes probably do not contribute as much value to prediction as the most recent price changes. However, I would not be able to use a CNN sliding-window model unless I had more fine-grained timeseries data points. Daily prices are not enough, since there would not be enough data to make valid predictions.

Though I believe a LSTM RNN may not be the most optimal choice of model, it still performs much better than my univariate ARIMA benchmark model. The LSTM RNN model performed better than univariate ARIMA, because it had access to more feature values, was able to generalize well with the use of dropout in the hidden layer, and was able to learn better through being exposed to the training data set many times through epochs.

In my LSTM RNN model, I used a dropout of 0.5 for the hidden layer in training. This 0.5 dropout probability was chosen because it was suggested in *Improving Neural Networks by Preventing Co-Adaption of Feature Detectors*. Hinton and his colleagues explained that the reason why their study chose a 0.5 dropout “so that a hidden unit cannot rely on other hidden units being present” (Hinton et al. 2012). Dropout makes the model more robust since it reduces overfitting, and enables the model to be trained for many epochs rather than utilizing an early-stopping method.

In addition, 5,000 epochs were chosen as optimal for training, because this yielded the lowest RMSE.

When determining the best batch size, I had tried batch sizes in powers of 2: 8, 16, 32, 64, 128. Smaller batch sizes took a lot longer to train than the larger batch sizes. Batch sizes of 8 and 16, took too long to reasonably train. Larger batch sizes of 64 and 128 were too large and would most likely not generalize well because there were only 290 Ether timeseries data points. I decided to use a small batch size of 32 because it would allow the model to generalize better than the larger batch sizes. This small batch size supported what Keskar and his colleagues stated in their study on how batch sizes influenced the generalizability of a model. They wrote that “it has been observed in practice that when using a larger batch there is a significant degradation in the quality of the model, as measured by its ability to generalize.” Moreover, this inability to generalize well is because large batches “tend to converge to sharp minima” (Keskar et al. 2017). Smaller batch sizes are slower to train and converge, but they tend to yield better results since they allow the model to generalize better.

This model is relatively robust, since it did 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 25**). The best LSTM RNN model had a RMSE of 3.19 and for a 7-day forecast (**Figure 26**).

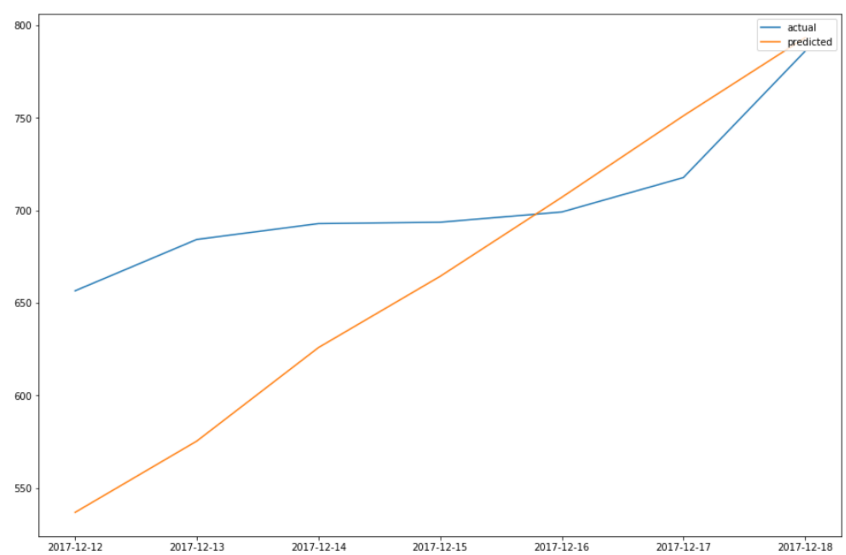
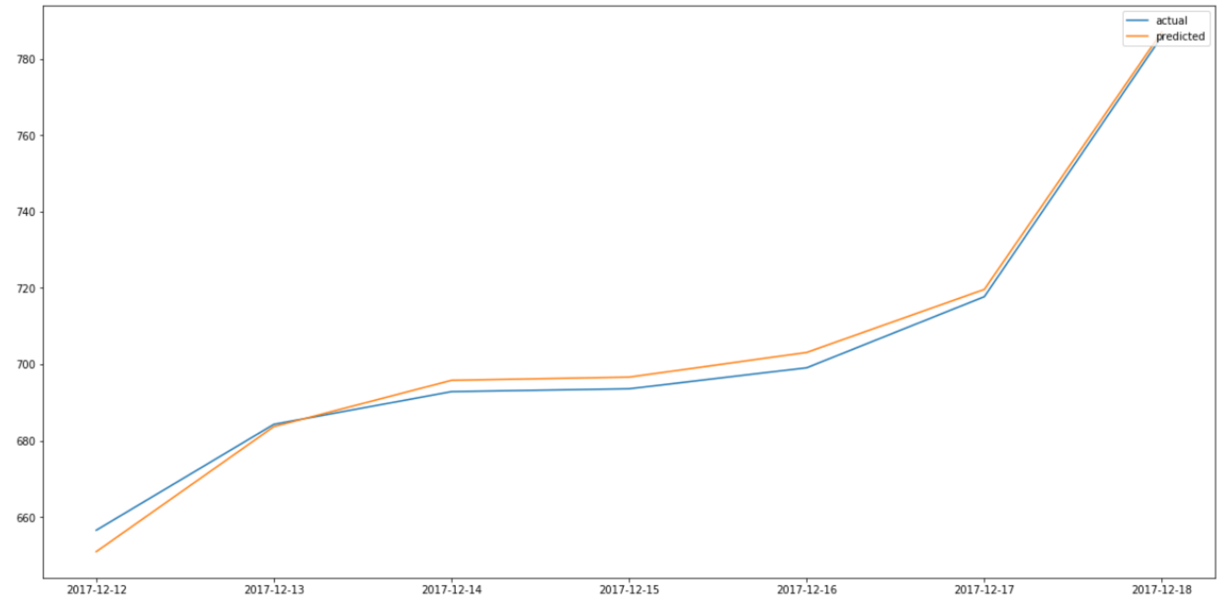


Figure 25: ARIMA 7-Day Forecast




```
actual: [656.52, 684.27, 692.83, 693.58, 699.09, 717.71, 785.99]
predicted: [650.91, 683.67, 695.8, 696.64, 703.11, 719.63, 787.15]
differences: [5.62, 0.61, 2.97, 3.06, 4.02, 1.92, 1.16]
test RMSE: 3.189
```

Figure 26: LSTM RNN 7-Day Forecast

It is clear from the RMSE values of each model, that the LSTM RNN is superior. The results from the LSTM RNN 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 univariate ARIMA benchmark model made a 7-day forecast of Ether price. This is shown in **Figure 27**.

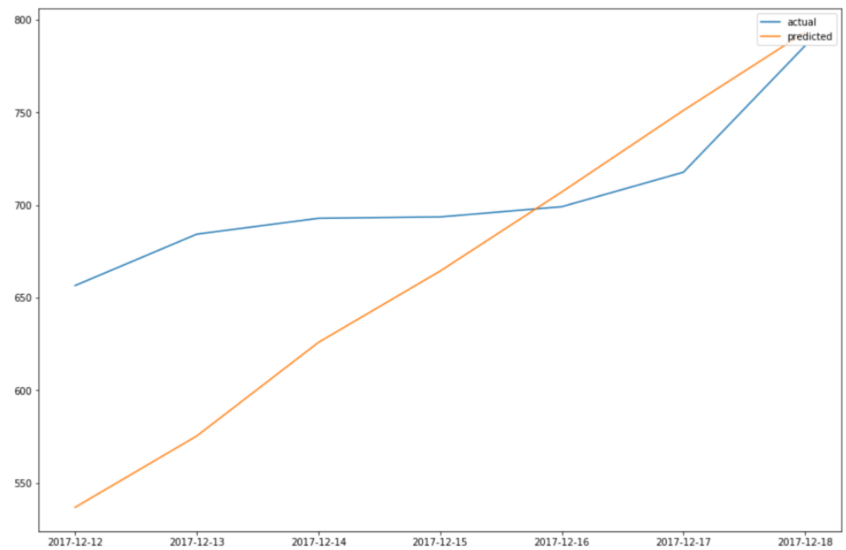


Figure 27: Benchmark Univariate ARIMA Model Ether Price 7-Day Forecast

The LSTM RNN model's training and validation error using the most optimal parameters is shown in **Figure 28**.

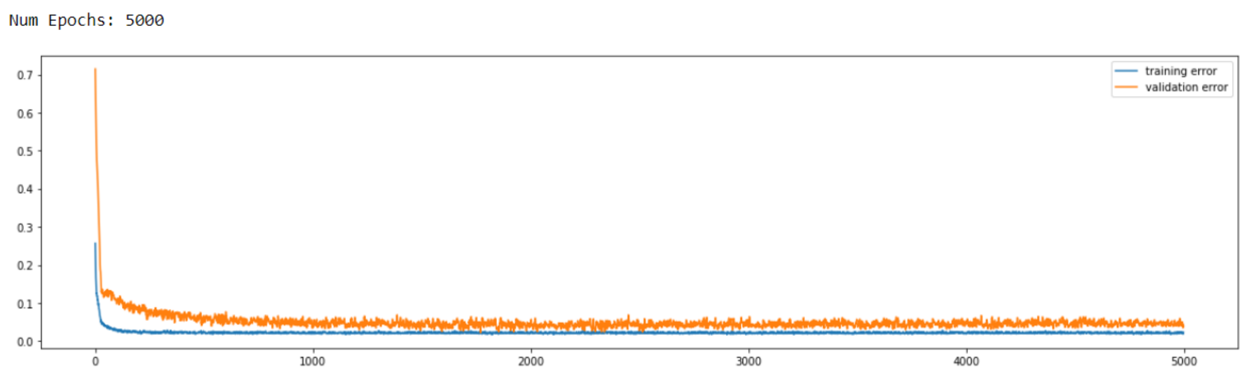
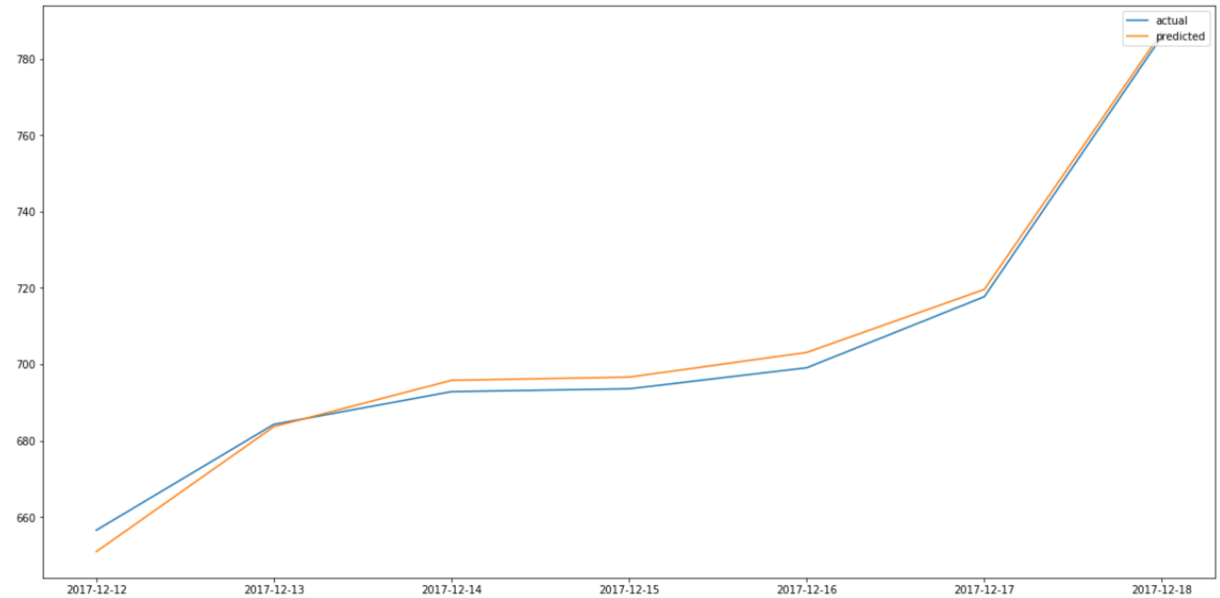


Figure 28: LSTM RNN Model Training and Validation Error over 5,000 Epochs

My LSTM RNN model is shown in **Figure 29**. The RMSE is 3.19.




```
actual: [656.52, 684.27, 692.83, 693.58, 699.09, 717.71, 785.99]
predicted: [650.91, 683.67, 695.8, 696.64, 703.11, 719.63, 787.15]
differences: [5.62, 0.61, 2.97, 3.06, 4.02, 1.92, 1.16]
test RMSE: 3.189
```

Figure 29: LSTM RNN Model 7-Day Ether Price Forecast

Because my LSTM RNN model has a RMSE of 3.19 and my univariate ARIMA model has a RMSE of 68.44, my LSTM RNN is much better as a lower RMSE is more desirable. In addition, my LSTM RNN model generalizes well, because the training error is always lower than the validation error with training and validation errors eventually stabilizing at close values.

Conclusion

5.1 Free-Form Visualization

Figure 30 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 20^{th} lag there is a large positive autocorrelation.

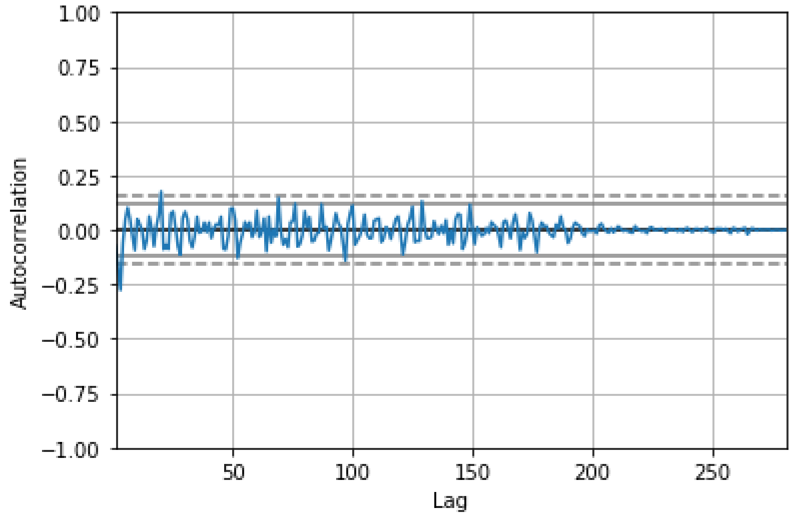


Figure 30: Autocorrelation Plot of Ether Time Series (Mar. 1, 2017 – Dec. 18, 2017)

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. It would also be much more accurate if the data points were more granular rather than per day. In addition, I can try other models such as a CNN sliding-window model, since this model would take only the most recent data points into consideration. This model may perform better than a LSTM RNN, since the volatility of Ether price makes past price values not as important in training than the more recent data points.

6 References

Ariyo AA, Adewumi AO, Ayo CK. Stock Price Prediction Using the ARIMA Model. 2014 UKSim-AMSS 16th International Conference on Computer Modelling and Simulation. 2014.

Blockchain. Wikipedia. 2017 Dec 24 [accessed 2017 Dec 25]. https://en.wikipedia.org/wiki/Blockchain

Brownlee J. A Gentle Introduction to Autocorrelation and Partial Autocorrelation. Machine Learning Mastery. 2017 Aug 15 [accessed 2018 Jan 3]. https://machinelearningmastery.com/gentle-introduction-autocorrelation-partial-autocorrelation/

Brownlee J. How to Use Dropout with LSTM Networks for Time Series Forecasting. Machine Learning Mastery. 2017 Jul 19 [accessed 2018 Jan 3]. https://machinelearningmastery.com/use-dropout-lstm-networks-time-series-forecasting/

Buterin V. Ethereum White Paper. GitHub. [accessed 2017 Dec 25]. https://github.com/ethereum/wiki/wiki/White-Paper

Create a Hello World Contract in ethereum. Ethereum Project. [accessed 2018 Jan 2]. https://ethereum.org/greeter

Demir U, Ergen SC. ARIMA-based time variation model for beneath the chassis UWB channel. EURASIP Journal on Wireless Communications and Networking. 2016;2016(1).

Difference between feedback RNN and LSTM/ GRU. Stack Exchange. [accessed 2017 Dec 25]. https://stats.stackexchange.com/questions/222584/difference-between-feedback-rnn-and-lstm-gru

Doug. How to choose the number of hidden layers and nodes in a feedforward neural network? Stack Exchange. [accessed 2017 Dec 25]. https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw

Ether Price Analysis: China's ICO Ban May Lead to Further Pull-backs. Bitcoin Magazine. 2017 Sep 6 [accessed 2017 Dec 25]. https://bitcoinmagazine.com/articles/ether-price-analysis-chinas-ico-ban-may-lead-further-pull-backs/

Ethereum Charts and Statistics. [accessed 2018 Jan 2]. https://etherscan.io/charts

Ethereum Price Achieves New All-Time High at $518, CryptoKitties Effect? CCN. 2017 Dec 9 [accessed 2017 Dec 25]. https://www.cryptocoinsnews.com/ethereum-price-achieves-new-time-high-518-cryptokitties-effect/

GitHub Developer. [accessed 2018 Jan 2]. https://developer.github.com/v4/

Hinton GE, Srivastava N, Krizhevsky A, Sutskever I, Salakhutdinov RR. Improving neural networks by preventing co-adaptation of feature detectors. 2012 Jul 3 [accessed 2018 Jan 4]

Kane MJ, Price N, Scotch M, Rabinowitz P. Comparison of ARIMA and Random Forest time series models for prediction of avian influenza H5N1 outbreaks. BMC Bioinformatics. 2014 [accessed 2017 Dec 25]. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4152592/

Keskar N, Mudigere D, Nocedal J, Smelanskiy M, Tang P. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. 2017 Feb 9 [accessed 2018 Jan 4]. https://arxiv.org/pdf/1609.04836

Makridakis S, Hibon M. ARMA Models and the Box-Jenkins Methodology. Journal of Forecasting. 1997;16(3):147–163.

Morrison J. Autoregressive Integrated Moving Average Models (ARIMA). [accessed 2018 Jan 2]. http://www.forecastingsolutions.com/arima.html

Nau R. Introduction to ARIMA: nonseasonal models. Statistical forecasting: notes on regression and time series analysis. [accessed 2018 Jan 2]. https://people.duke.edu/~rnau/411arim.htm#sesgrow

Nau R. Stationarity and differencing of time series data. [accessed 2017 Dec 25]. https://people.duke.edu/~rnau/411diff.htm

Nelson DMQ, Pereira ACM, Oliveira RAD. Stock markets price movement prediction with LSTM neural networks. 2017 International Joint Conference on Neural Networks (IJCNN). 2017.

Olah C. Understanding LSTM Networks. Colah's Blog. [accessed 2018 Jan 3]. http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Partial correlation. Wikipedia. 2017 Dec 7 [accessed 2018 Jan 3]. https://en.wikipedia.org/wiki/Partial_correlation

Peck M. The Future of the Web Looks a Lot Like the Bitcoin Blockchain. IEEE Spectrum: Technology, Engineering, and Science News. 2015 Jul 1 [accessed 2017 Dec 25]. https://spectrum.ieee.org/computing/networks/the-future-of-the-web-looks-a-lot-like-bitcoin

Selvin S, Vinayakumar R, Gopalakrishnan EA, Menon VK, Soman KP. Stock price prediction using LSTM, RNN and CNN-sliding window model. 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI). 2017.

Srivastava T. A Complete Tutorial on Time Series Modeling in R. Analytics Vidhya. 2017 Apr 11 [accessed 2018 Jan 2]. https://www.analyticsvidhya.com/blog/2015/12/complete-tutorial-time-series-modeling/

Villegas D. The unit of Root Mean Square Error (RMSE). Stack Exchange. [accessed 2017 Dec 25]. https://stats.stackexchange.com/questions/292059/the-unit-of-root-mean-square-error-rmse

Vincent T. ARIMA Time Series Data Forecasting and Visualization in Python. DigitalOcean. 2017 Nov 2 [accessed 2017 Dec 25]. https://www.digitalocean.com/community/tutorials/a-guide-to-time-series-forecasting-with-arima-in-python-3