COSC505 Project: Bitcoin Trading

Shima Amirsadri Deepak Kumar Shambhavi Singh

December 5, 2017

1 Introduction

Bitcoin is one the famous worldwide cryptocurrency available in the market. First launched in 2009 by Satoshi Nakamoto as an open source software, it is a digital payment system which works without a central repository or single administrator. The system is peer to peer and the transactions takes place between users directly without any need of intermediary. It can also be exchanged for other currencies, products and services in most of countries. The historical bitcoin dataset contains the bitcoin trading market data from selected bitcoin exchanges. The data has been collected at an interval of one minute from Jan 2012 to Oct 2017. Bitcoin has seen a recent rise in popularity and its price has skyrocketed in the past 2 years.

This report is divided into three major sections. Section2 covers a brief description of our dataset. Section 3 covers the models that were chosen to fit dataset for bitcoin in USD while section 4 covers the forecasting of bitcoin values using an Autoregressive Integrated Moving Average(ARIMA) model.

2 Dataset

The dataset has been downloaded from Kaggle.com, and contains seven CSV files which stores the minute to minute updates for OHLC(Open, High, Low, Close), Volume in BTC and the corresponding currency, and weighted price of the bitcoin. The time of these transactions have been stored as timestamps according to Unix conventions while the data for timestamps with no trade or activity have been stored as NaNs.

The datasets are as follows:

- **coinbaseUSD_1-min_data_2014-12-01_to_2017-10-20** stores the data obtained from the Coinbase bitcoin exchange in US Dollars from December 1st, 2014, to October 20, 2017.

- **bitstampUSD_1-min_data_2012-01-01_to_2017-10-20** stores the data obtained from the Bitstamp bitcoin exchange in US Dollars from January 1st, 2012, to October 20, 2017.
- **coincheckJPY_1-min_data_2014-10-31_to_2017-10-20** stores the data obtained from the Coincheck bitcoin exchange in Japanese Yen from October 31st, 2014, to October 20, 2017.
- **btceUSD_1-min_data_2012-01-01_to_2017-05-31** stores the data obtained from the BTC-e bitcoin exchange in US Dollars from January 1st, 2012, to May 31, 2017. Currently this exchange has been closed by US Justice Dept from July 26th, 2017.
- **krakenEUR_1-min_data_2014-01-08_to_2017-05-31** stores the bitcoin trade data occurred between January 8, 2014 to May 31st, 2017 on Kraken exchange in Euros.
- **krakenUSD_1-min_data_2014-01-07_to_2017-05-31** stores the data obtained from the Kraken bitcoin exchange in US Dollars from January 7, 2014, to May 31st, 2017.
- **btcnCNY_1-min_data_2012-01-01_to_2017-05-31** stores the data obtained from the Chinese bitcoin market in Chinese Yuan from January 1st, 2012, to May 31, 2017.

3 Model fitting

This section covers the models we tried to fit to our data.

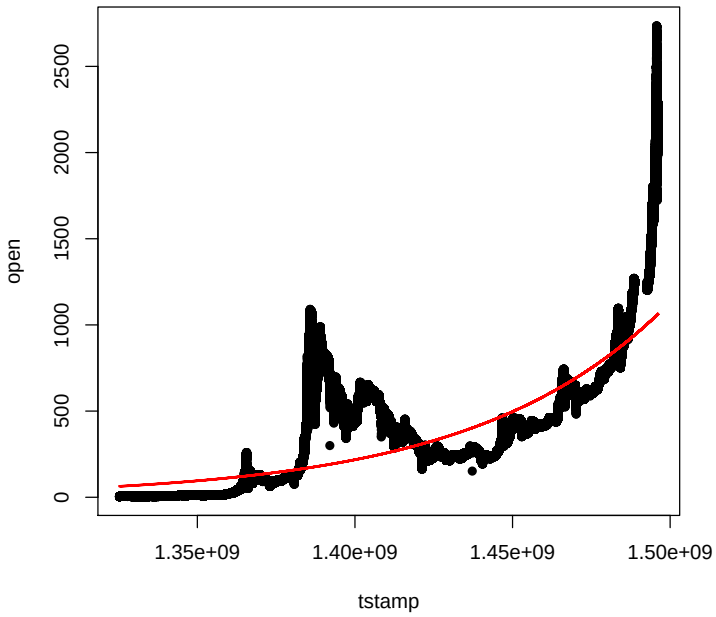
3.1 Exponential Model Fitting

Fitting an exponential model to the Bitcoin opening values In order to fit an exponential model to the Bitcoin opening values, we have taken several steps. The first step is to clean the data. For this purpose, we removed all rows that contain missing values in their opening values. After that, using $qreference()$ method, we found out that our data is far from an exponential model because there are no similar patterns between our data and exponential reference plots.

```
> #install.packages("DAAG")
> qreference<-DAAG::qreference
> csv<-read.csv('btceUSD_1-min_data_2012-01-01_to_2017-05-31.csv')
> csv2<-csv[complete.cases(csv), ]
> open<-csv2[,2]

> # Plotting qrefrence plots for checking if our data follows an exponential model
> qreference(open, xlab=" ",distribution = function(x) qexp(x, rate = 1/mean(open)) )
```

In the next step, we fitted an exponential model to our data and then plotted it. For doing that, we fitted the linear model of log of the opening values, which is the same as fitting an exponential model of timestamp against opening values. From the plot, we realized that the trend of the Bitcoin opening values does not follow an exponential pattern.

```
> #Fitting an exponential model
> tstamp<-csv2$Timestamp
> exponential.model <- lm(log(open)~tstamp)
> summary(exponential.model)

Call:
lm(formula = log(open) ~ tstamp)

Residuals:
    Min      1Q  Median      3Q     Max
-2.8632 -0.3423 -0.1468  0.3296  1.8444

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -1.770e+01  1.915e-02  -924.5   <2e-16 ***
tstamp       1.649e-08  1.342e-11  1228.7   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.7337 on 1870569 degrees of freedom
Multiple R-squared:  0.4466,        Adjusted R-squared:  0.4466
F-statistic: 1.51e+06 on 1 and 1870569 DF,  p-value: < 2.2e-16

> r1<-range(tstamp)
> timevalues <- seq(r1[1], r1[2], 10)
> Counts.exponential2 <- exp(predict(exponential.model,list(tstamp=timevalues)))
> plot(tstamp, open,pch=16)
> lines(timevalues, Counts.exponential2,lwd=2, col = "red", xlab = "Time (s)", ylab = "Count
```



3.2 Poisson Model Fitting

The first assumption that was made was that the Poisson Model would have λ that would be more or less near the highest occurring value in the data, however, even in the initial run, it was clear that the range of Poisson was not very wide hence trying to fit it further seemed

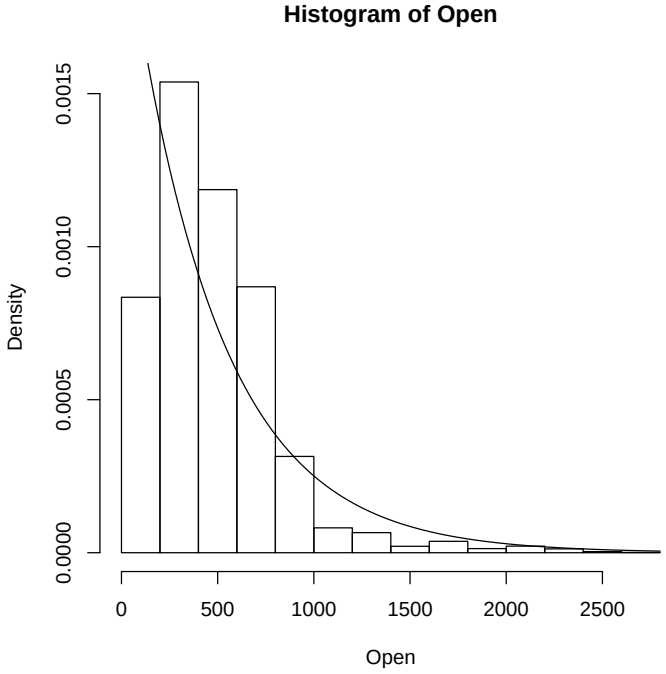
3.3 Simulation with Exponential distribution

After trying to fit the exponential and poisson models to our distribution, we realized that the former seems to fit better compared to the latter. Although we have a peak near the year 2013 which created some residual error but still the error was significantly less compared to others. Hence we chose exponential model to do simulate the bitcoin trading curve.

Let us go through the rationale of doing so. First, we sample the bitstampUSD dataset for 100000 random data points and then we plot the density of the same.

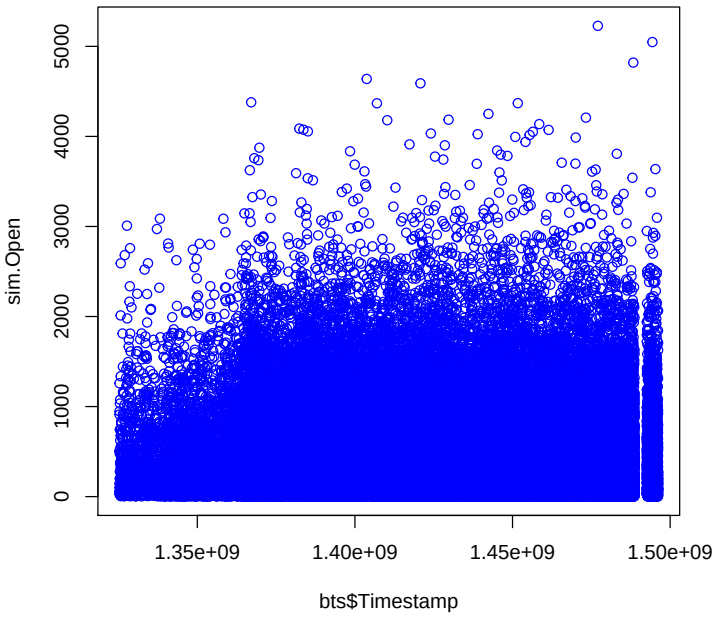
```
> bt <- csv2
> NSamples <- 100000
> bts <- bt[sample(length(bt$Timestamp),NSamples),]
```

```
> Open <- bts$Open
> mean.Open <- mean(Open)     # mean of the opening values of the bitcoin
> hist(Open, freq= FALSE)   # density of the Opening values
> curve(dexp(x,rate=1/mean.Open),add=TRUE)    # overlaying the exponential curve
```



After plotting the density of the opening values of the dataset, we overlay an exponential density function curve on it. The curve seems to fit fairly well to our dataset and so we finally chose to simulate new sample points for our dataset with an exponential density function.

```
> bt <-csv2
> NSamples <- 100000            # number of samples
> bts <- bt[sample(length(bt$Timestamp),NSamples),]    # selecting random sample points
> Open <- bts$Open
> mean.Open <- mean(Open)     # mean of the opening values of the bitcoin
> sim.Open <- rexp(NSamples,rate = 1/mean.Open) # Simulate new observations
> plot(sim.Open~bts$Timestamp,col=4) # plotting new observations
> hist(sim.Open,freq = FALSE)         # density plot of simulated data
> curve(dexp(x,rate=1/mean.Open),add=TRUE)    # overlaying the exponential curve
```



After simulating with exponential model, we observed that although the histogram of the simulated observations fit exponential curve perfectly, the sample observations seems to random. This is due to the fact that our dataset is a time-series observation while the simulation just create observations which follows a particular probability density function. Hence, we require time-series models for simulating our dataset or forecasting future bitcoin values.

3.4 Data clustering

Since our dataset contains a small peak around 2013, so it might be the case that the peaks may belong to different cluster to data. Hence we tried to find the cluster in our dataset using k-means clustering.

```
> ## clusters in dataset
> bt <- csv2
> NSamples <- 100000
> bts <- bt[sample(length(bt$Timestamp),NSamples),]
> # opening times
> xopen <- cbind(TimeStamp = bts$Timestamp, Open = bts$Open)
> cl_open <- kmeans(xopen,2) # find clusters
> plot(xopen,col=cl_open$cluster) # plot the respective clusters
> #closing times
```

```
> xclose <- cbind(TimeStamp = bts$Timestamp, Close = bts$Close)
> cl_close <- kmeans(xclose,2) # find clusters
> plot(xclose,col=cl_close$cluster) # plot the respective clusters
> # Low
> xlow <- cbind(TimeStamp = bts$Timestamp, Low = bts$Low)
> cl_low <- kmeans(xlow,2) # find clusters
> plot(xlow,col=cl_low$cluster) # plot the respective clusters
> # High
> xhigh <- cbind(TimeStamp = bts$Timestamp, high = bts$High)
> cl_high <- kmeans(xhigh,2) # obtain clusters
> plot(xhigh,col=cl_high$cluster) # plot the respective clusters
```

After obtaining the clusters for opening, closing, low and high values of bitcoin we observe that the dataset has can be divided into two clusters, one just at the end of the peak while other containing the exponential growth of bitcoin. Although our data can be divided into two clusters but dividing it and then fitting time-series models to both individually does not seem appropriate. It is because the peak contains some information about the abrupt change in the value of bitcoin affected by external circumstances which is not present in the dataset. Hence fitting models in the two clusters will lead to different models, thereby it would be difficult to draw any inference from the both.

At the end, we decided to do forecasting for the values of bitcoin using an ARIMA model.

4 Forecasting with ARIMA

ARIMA models have a reputation for being transformative and employing historical data for predictions. Here, we will fit a model to our Bitcoin data so that we will be able to do forecasting for future values of bitcoin.

4.1 Introduction to ARIMA

ARIMA is the abbreviation for auto-regressive integrated moving average and its main components are (P, d, q) parameters.

The **auto regressive** ($AR(p)$) component is referring to the use of past values in the regression equation for the output series. The auto-regressive parameter p specifies the number of lags used in the model.

The d represents the degree of differencing in the **integrated** ($I(d)$) component. Differencing a series involves simply subtracting its current from previous values d times.

A **moving average** ($MA(q)$) component represents the error of the model as a combination of previous error terms e_t.

Differencing, autoregressive, and moving average components make up a non-seasonal ARIMA model which can be written as a linear equation:

$$Y_t = c + \phi_1 y_(dt-1) + \phi_p y_(dt-p) + ... + \theta_1 e_(t-1) + \theta_q e_(t-q) + e_t \quad (1)$$

where y_d is Y differenced d times and c is a constant.

4.2 ARIMA model limitations

These models are suitable for long and robust series where the present values are dependent on the past values.

4.3 Outline

In order to fit a model to our data using ARIMA, here is the outline of what we need to do:

- **Data Examination:**
  - Plotting the data in order to find patterns and irregularities
  - Dropping the missing values and outliers (tsclean() is a suitable method for this purpose)
  - Calculating the logarithm of the time series to see a growth trend
- **Data Analyzing:**
  - Finding trends or seasonality in the data if there is any
  - Applying *decompose*() or *stl*() for removing the components of the series
- **Stationarity**
  - Is there any stationarity in the data?
  - Use adf.test(), ACF, PACF plots to determine order of differencing needed
  - **Autocorrelations and selecting model order**
  - Finding out the order of ARIMA using ACF() and PACF() plots
- **Fit an ARIMA model**
- **Assess and repeat**
  - Examine the residuals to make sure there is no pattern and they are normally distributed
  - In case of existing any visible patterns, draw ACF/ PACF plots to figure out if any additional parameters is needed.
  - Refit model if it is necessary. Testing model errors and fit criteria such as AIC or BIC.
  - Compute fitting model by means of the selected model.

4.4 How to fit a model using ARIMA - Description

Step 1: Load R packages

In this study for fitting ARIMA model, we will be using the following libraries. Moreover, we will be utilizing "bitstampUSD_1-min_data_2012-01-01_to_2017-10-20.csv" file as our dataset.

```
> #install.packages("tseries","forecast","ggplot2")
> library('ggplot2')
> library('forecast')
> library('tseries')
> btc<-csv
```

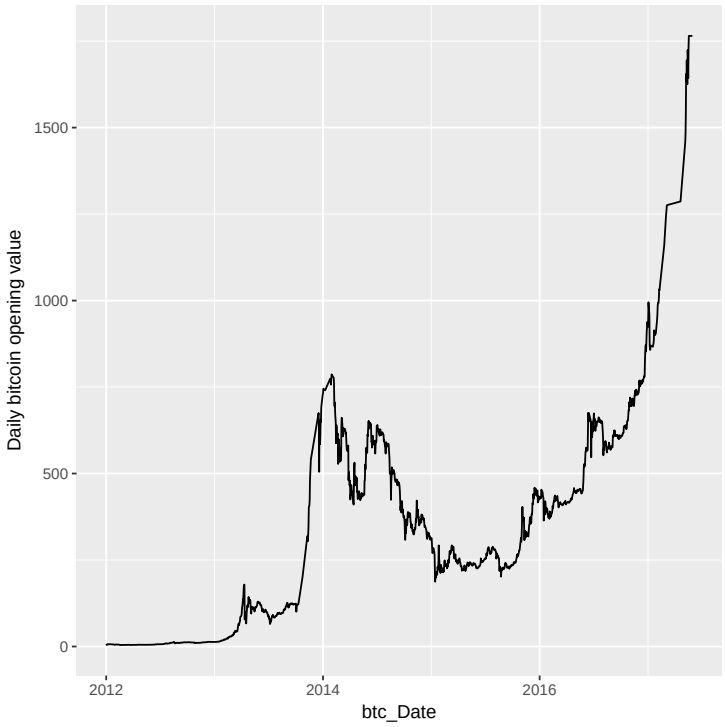
Step 2: Data Cleaning Since our dataset contains too many Nan values, we will be using a method to replace them with their closest values. Then, we changed the intervals of the Bitcoin opening values from minute to day to make our data more visually appealing. Furthermore, we transformed the timestamp to date.

```
> #Replacing the Nan opening values with the nearest values
> replaceNanNearst <- function(dat) {
+   N <- length(dat)
+   na.pos <- which(is.na(dat))
+   if (length(na.pos) %in% c(0, N)) {
+     return(dat)
+   }
+   non.na.pos <- which(!is.na(dat))
+   intervals  <- findInterval(na.pos, non.na.pos,
+                              all.inside = TRUE)
+   left.pos   <- non.na.pos[pmax(1, intervals)]
+   right.pos  <- non.na.pos[pmin(N, intervals+1)]
+   left.dist  <- na.pos - left.pos
+   right.dist <- right.pos - na.pos
+
+   dat[na.pos] <- ifelse(left.dist <= right.dist,
+                         dat[left.pos], dat[right.pos])
+   return(dat)
+ }
> btc$openC<-replaceNanNearst(btc$Open)
> #Changing the interval from minute to minute to day to day
> sq<-seq(1,length(btc$openC),by = 1440)
> btc_date<-btc$Timestamp[sq]
> BTC<- data.frame(btc_open<-btc$openC[sq],btc_Date<-as.Date(as.POSIXct(btc_date, origin = '
> colnames(BTC)<-c("btc_open","btc_Date")
```

Step 3: Analyzing the data Plotting the data to find the outliers and irregularities is of importance.

In some cases, the opening value of Bitcoin increased to 1125 USD on month and dropped to 500 USD the next month. These are doubted outliers that might incline the model by deviating the summaries. Tsclean() in R provides facilities for passing up these outliers. This method replaces outliers applying series smoothing and decomposition. For this purpose, first a time series object should be created using *ts*() command to be passed to *tsclean*():

```
> #Removing the outliers
> count_ts = ts(BTC[, c('btc_open')])
> BTC$clean_open = tsclean(count_ts)
> ggplot() +
+ geom_line(data = BTC, aes(x = btc_Date, y = clean_open)) + ylab("Daily bitcoin opening val
```



After plotting the clean series using ggplot:

It is noticeable that even after ignoring the outliers, there are still fluctuations in the Bitcoin opening values. One of the promising solutions to this issue is through considering a line connecting its bigger bottoms and peaks when smoothing the fluctuations. This concept is known as moving average in time series. This meaning calculates the average of the points across different intervals and therefore makes the data a more predictable series. A moving average

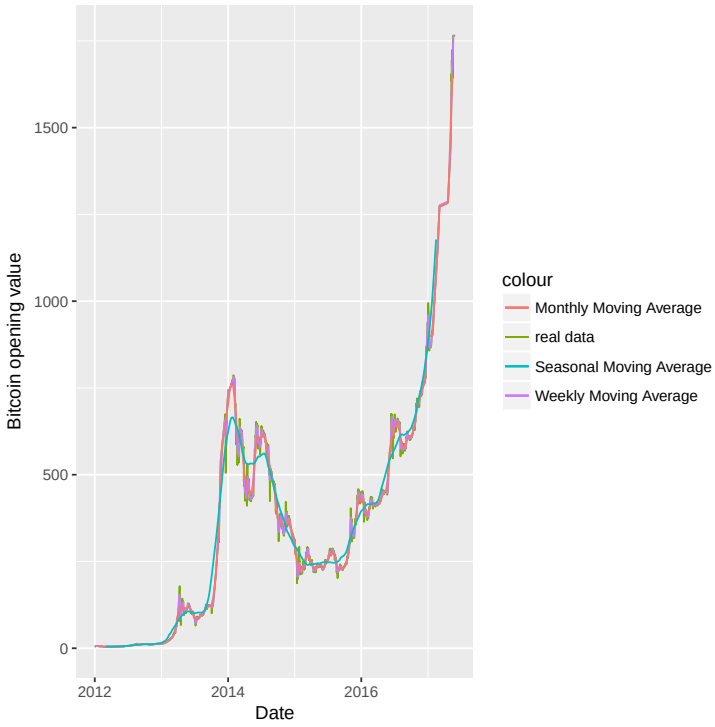
(MA) of order m can be calculated using the following formula:

$$MA = 1/m \sum_{j}^{k} -k^(y_t + j)$$

Where y is the series, k is the number of periods around each point and $m = 2k + 1$. It is worth noting that the moving average here is completely different from the $MA(q)$ component in the definition of ARIMA. $MA(q)$ is related to error lags but the summary statistic of moving average is a part of the data smoothing technique.

We have a smoother function as we expand the size of the window for the moving average. If we take monthly (or seasonally) instead of weekly moving average, we can have a more predictable series.

```
> #Making the clean data(The data with no outliers) smoother using the moving average
> BTC$open_ma = ma(BTC$clean_open, order=7)
> BTC$open_ma30 = ma(BTC$clean_open, order=30)
> BTC$open_ma120 = ma(BTC$clean_open, order=120)
> ggplot() +
+   geom_line(data = BTC, aes(x = btc_Date, y = clean_open, colour = "real data")) +
+   geom_line(data = BTC, aes(x = btc_Date, y = open_ma,   colour = "Weekly Moving Average")
+   geom_line(data = BTC, aes(x = btc_Date, y = open_ma30, colour = "Monthly Moving Average'
+   geom_line(data = BTC, aes(x = btc_Date, y = open_ma120, colour = "Seasonal Moving Averag
+   ylab("Bitcoin opening value")
```



As can be seen, the Seasonal Moving Average offers the smoothest data and we will model it (as shown by green line above) for simplicity.

Step 4: Decompose the data

The corner stones for analyzing a time series are seasonality, trend and cycle. These components detect the historical patterns of the series. In case of being available, these components can contribute in analyzing the behavior of the series in order to be able to create a forecasting model.

Seasonal component is related to the fluctuations in the data in seasonal cycles. For example, in which seasons, the Bitcoin value is larger and in which seasons in has smaller values. Trend component is assigned to the whole pattern of the series. Does the series have an increasing or decreasing trend over all of the intervals?

Cycle component is about all of the increasing or decreasing trends that are not seasonal. In general, trend and cycle components are considered together as one. The last part of the series that cannot be referred to any of the other components is called residual or error.

The process of taking out these components is called decomposition.

For example, if Y is the Bitcoin opening value, we can decompose the series by applying an additive or multiplicative model:

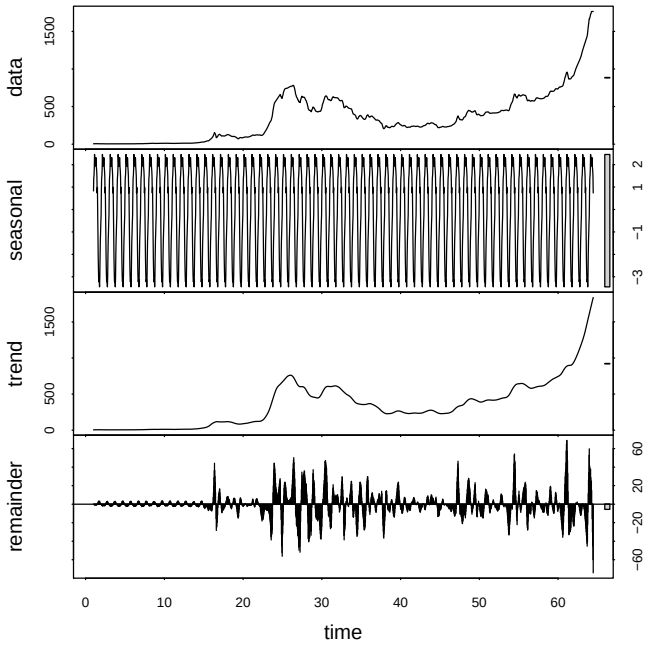
$$Y = S_t * T_t * E_t \tag{2}$$

Where S, T and E are seasonal, trend and error components, respectively.

An additive model is more appropriate for the conditions that seasonal or trend components are not proportional to the level of series. In these situations, we can just overlay the components together to rebuild the series. In contrast, if the seasonality is dependent to trend of the series, a multiplicative model can be more efficient.

As fitting a model to seasonal data is a complicated process, we will explain how to de-seasonalize the series and use a "vanilla" non-seasonal ARIMA model. In order to do this, we first calculate seasonal component of the data using *stl*(). The *Stl*() is a function for decomposing and forecasting the series. It calculates the seasonal component of the series using smoothing, and configuring the series by subtracting seasonality.

```
> #Decomposion
> count_ma = ts(na.omit(BTC$open_ma), frequency=30)
> decomp = stl(count_ma, s.window="periodic")
> deseasonal_open <- seasadj(decomp)
> plot(decomp)
```



Seasadj() is a method for removing the seasonality. We set the periodicity of the data (i.e., the number of observations per period) for the frequency parameter in *ts*() object. As we utilize smoothed daily data, we have 30 observations per month.

Step 5: Stationarity

Fitting an ARIMA model requires series to be stationary which means that its mean, variance and autocovariance are independent from time. One of the formal statistical tests for stationarity is the augmented Dickey-Fuller (ADF). The null hypothesis here is that the series is non-stationary. ADF checks if the change in the Y values can be described by lagged value and a linear trend. If values of Y are related to lagged values and a trend component emerges, the series is considered to be non-stationary and null hypothesis will not be rejected.

```
> #ADF Test for testing stationarity
> adf.test(count_ma, alternative = "stationary")

        Augmented Dickey-Fuller Test

data:  count_ma
Dickey-Fuller = 0.85419, Lag order = 12, p-value = 0.99
alternative hypothesis: stationary
```

The opening values of Bitcoin are non-stationary. The average of opening values of Bitcoin change through time. The ADF test does not reject the NULL hypothesis and corroborates our guess. The non-stationary series can be transformed to stationary ones by deducting one period's values from the previous period's values. Differencing can assist in deleting the trends or cycles from the data and makes it stationary:

$$Y_(d_t) = Y_t - Y_(t-1) \tag{3}$$

In a similar vein, differencing can be used for removing seasonal patterns at specific lags:

$$Y_(d_t) = (Y_t - Y_(t-s)) - (Y_(t-1) - Y_(t-s-1)) \tag{4}$$

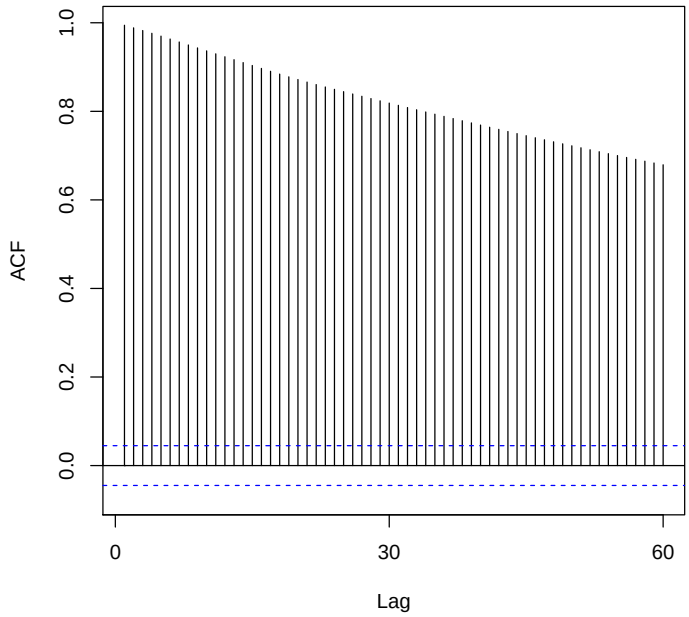
where d component of ARIMA represents the number of carried out differences.

Step 6: Autocorrelations and Choosing Model Order

In this section, the methods for choosing the order of differencing will be explained. Auto Correlation Function (ACF) is an efficient method for determining if a series is stationary. It is also beneficial in choosing the order of the parameters of the ARIMA model. In case of existing a correlation between a variable and its lags, there are some trends and therefore, the series changes over time.

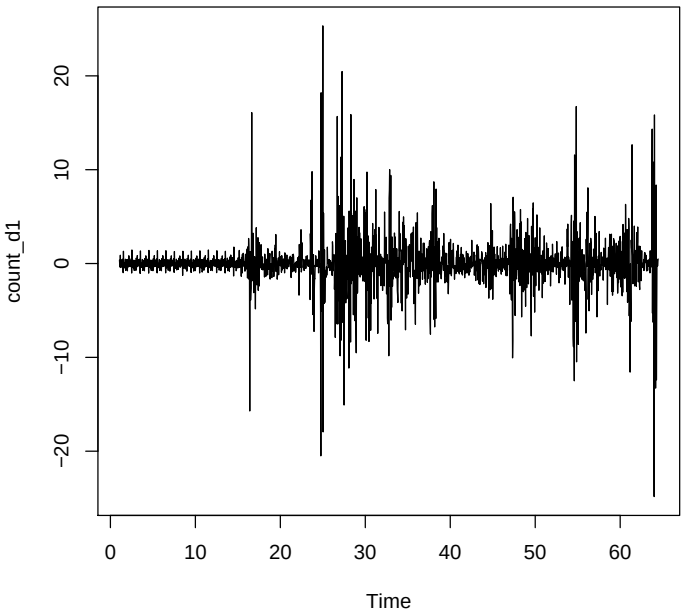
ACF plots show the correlation between a series and its lags. Partial autocorrelation plots (PACF) show the correlation between a variable and its lags that is not included by previous lags. As can be seen, there are autocorrelations with many lags in Bitcoin opening values by the ACF plot. This is because of the carry-over correlation from the early lags. However, the PACF plot does not show any spikes.

```
> #Plotting ACF and PACF plots for displaying the auto correlation
> Acf(count_ma, main='')
> Pacf(count_ma, main='')
```



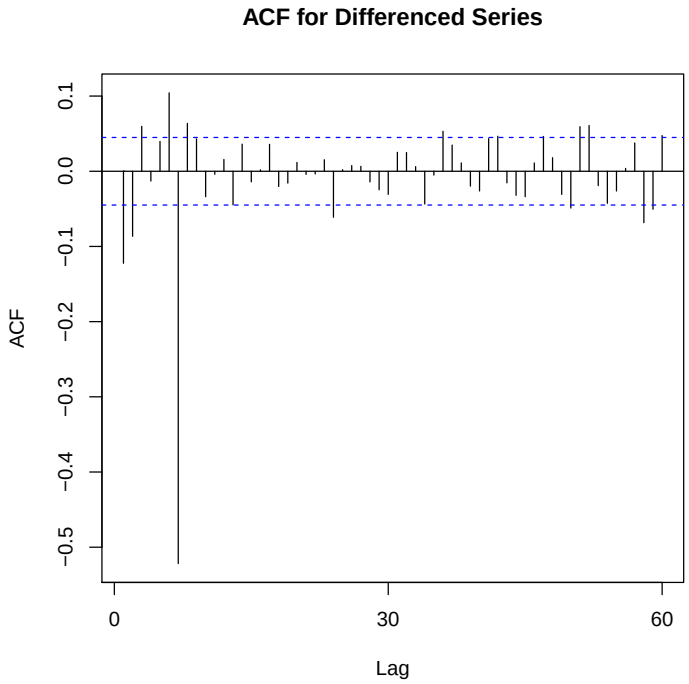
We will start from $d = 1$ and reevaluate if more differencing is needed. The null hypothesis of being non-stationary is rejected with the *adf* test. Considering the plotted differenced series, there is an oscillating pattern around 0 with no visible strong trend. Therefore, differencing of order 1 terms seems to be sufficient but differencing of order 2 decreases the number spikes in ACF so it should be included in the model.

```
> #Plotting the differenced series for choosing the order of differencing
> count_d1 = diff(deseasonal_open, differences = 2)
> plot(count_d1)
> adf.test(count_d1, alternative = "stationary")

        Augmented Dickey-Fuller Test

data:  count_d1
Dickey-Fuller = -13.738, Lag order = 12, p-value = 0.01
alternative hypothesis: stationary
```



Next, spikes at particular lags of the differenced series could be of benefit when it comes to choosing the p or q for our model:

```
> #Plotting ACF and PACF plots of the differencing series to choose p and q for the model
> Acf(count_d1, main='ACF for Differenced Series')
> Pacf(count_d1, main='PACF for Differenced Series')
```



There are noticeable auto correlations at lag1, 2 and 7. In PACF plot, there are significant spikes at lag 1, 2 and 7 and 14. Therefore, we would test models with AR or MA components of order 1, 2, 7 or 14. A spike at lag 7 and 14 may imply that there is a seasonal pattern available, perhaps as day of the week.

Step 7: Fitting an ARIMA model

We should use the forecast package to apply the *arima*() function for determining order of the model. We can also automatically specify a set of optimal (p, d, q) using *auto.arima*().This function searches through combinations of order parameters and picks the set that optimizes model fit criteria.

There are some criteria for evaluating the quality of fitting models. Among the most reputable ones, we can mention Akaike information criteria (AIC) and Baysian information criteria (BIC). These criteria can examine the amount of information lost for the chosen model. The objective here is that the selected minimizes AIC and BIC.

For applying *auto.arima*(), it is necessary to complete steps 1-5 to comprehend the series and analyzing model results. We can also determine maximum order of (p, d, q) using *auto.arima*() that is by default equal to 5.

```
> #Fitting the model
> auto.arima(deseasonal_open, seasonal=FALSE)

Series: deseasonal_open
ARIMA(2,2,2)
```

```
Coefficients:
          ar1      ar2     ma1     ma2
      -1.1647  -0.8839  1.0949  0.7582
s.e.   0.0392   0.0237  0.0564  0.0308

sigma^2 estimated as 7.191:  log likelihood=-4575.56
AIC=9161.11   AICc=9161.14   BIC=9188.87
```

The offered parameters (2, 2, 2) by the automated procedure are compatible with our expectations from the explained steps. The model incorporates differencing of degree 2, and applies an autoregressive term of the second lag and a moving average model of order 2.

Our model regarding the ARIMA notation can be defined as follows:

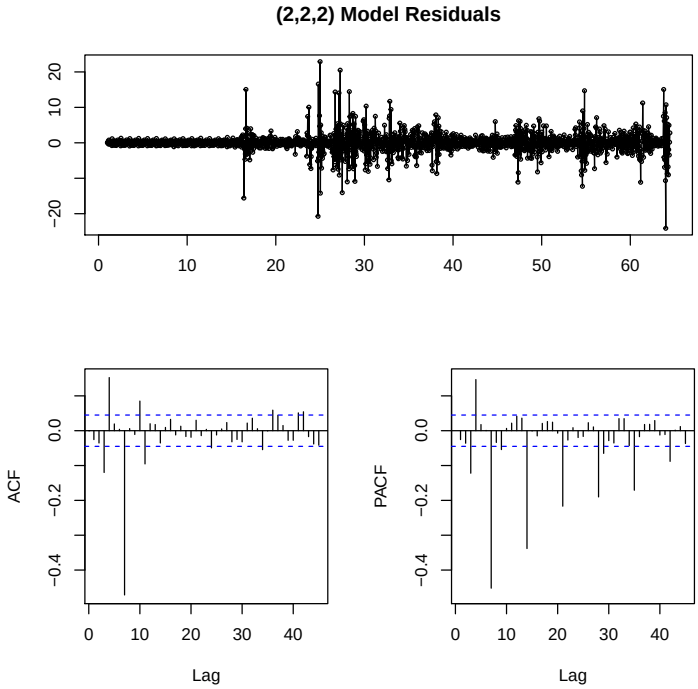
$$\tilde{Y}_{d_t} = 0.551 Y_{t-1} - 0.2496 e_{t-1} + E \tag{5}$$

Here, E is some error and the original series is differenced with order 2.

Step 8: Evaluate and Iterate

So far, we fitted a model to our data that can be used for forecasting, but how much we can trust it? For this purpose, we can test ACF and PACF plots for model residuals. In case of correctly selecting model order parameters and structure, there should not be any significant autocorrelations.

```
> #Testing the model using model residuals
> fit<-auto.arima(deseasonal_open, seasonal=FALSE)
> tsdisplay(residuals(fit), lag.max=45, main='(2,2,2) Model Residuals')
```

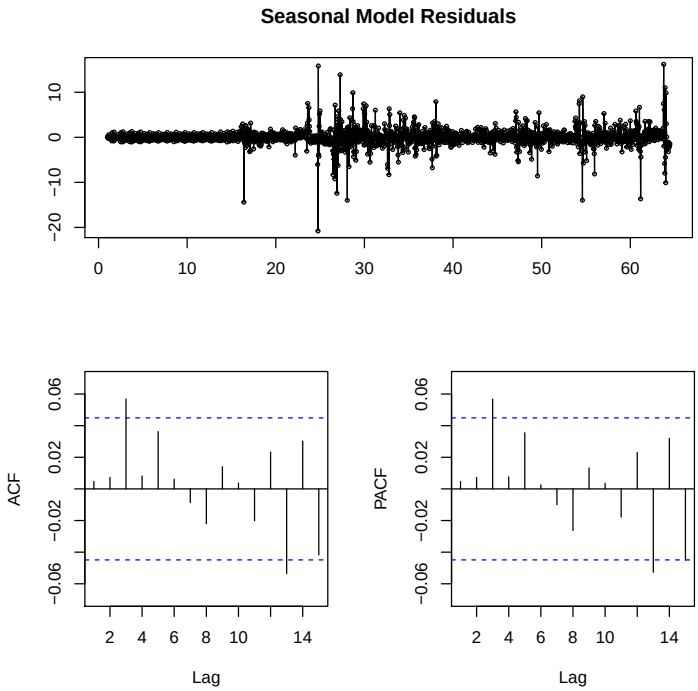


From the $ACF/PACF$ and model residuals plots , we can see that there is a repeating pattern repeating at lag 7. This suggests that our model might work better with a different configuration, such as $p = 7$ or $q = 7$.

We can do the fitting process from the beginning for $MA(7)$ and test the model again. In the new model, there are no observable autocorrelations. In case of being correctly selected, the model does not show any patterns in the model residuals. If the residuals are normally distributed, it means that the model correctly fits the series. Tsdisplay() function is applicable in this situation. Residuals plots display a smaller error range, more or less centered around 0. As can be seen, the AIC is smaller when we select (2,2, 7) for the parameters.

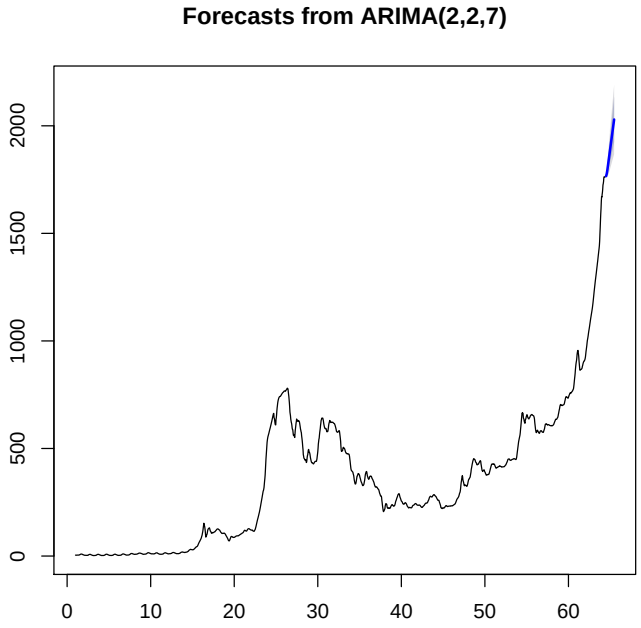
```
> #Fitting the model with new parameters
> fit2 = arima(deseasonal_open, order=c(2,2,7))
> tsdisplay(residuals(fit2), lag.max=15, main='Seasonal Model Residuals')
> arima(x = deseasonal_open, order = c(2, 2, 7))

Call:
arima(x = deseasonal_open, order = c(2, 2, 7))

Coefficients:
          ar1      ar2     ma1     ma2     ma3     ma4     ma5     ma6      ma7
      -0.1386  -0.0933  0.0145  0.0468  0.0161  0.0424  0.0260  0.0326  -0.9603
```

```
s.e.   0.0244   0.0243  0.0097  0.0095  0.0094  0.0092  0.0088  0.0091   0.0096

sigma^2 estimated as 3.863:  log likelihood = -4000.62,  aic = 8021.24
```



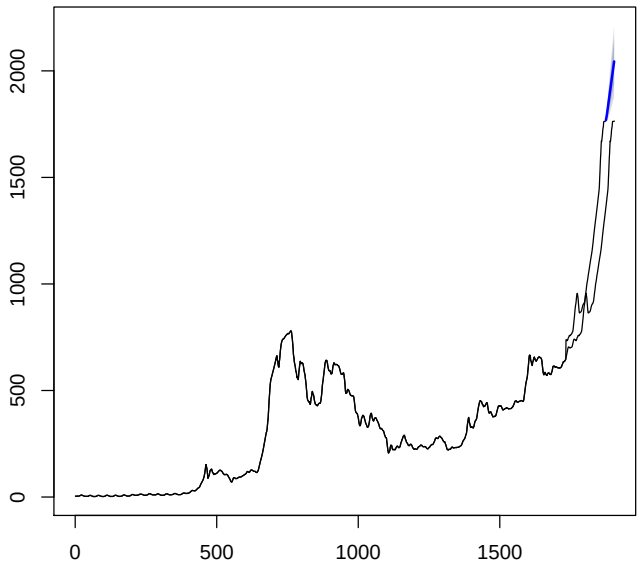
It is easy to do the prediction by a fitted model in R. We just need to determine forecast horizon h periods ahead for making the predictions.

```
> #Forecasting
> fcast <- forecast(fit2, h=30)
> plot(fcast)
```



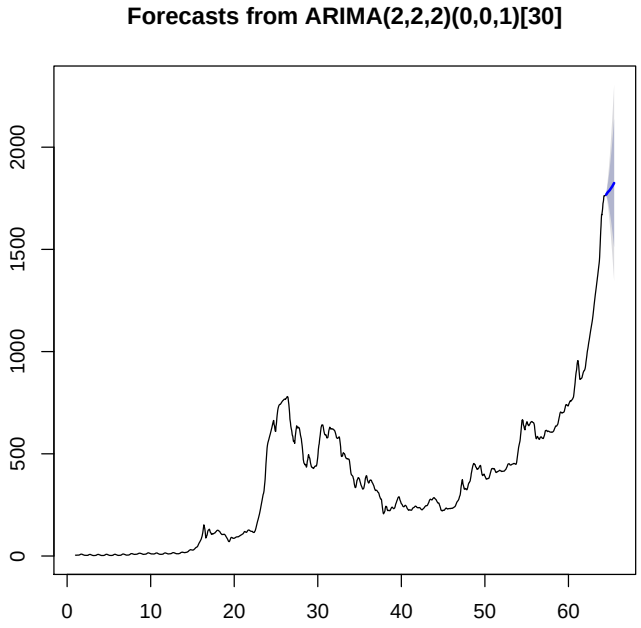
In order to find out how the model will work in future, we can fit the model to a part of the data (hold-out set) and then compare the forecast with the actual values.

```
> #Testing the performance of the model
> hold <- window(ts(deseasonal_open), start=1734)
> fit_no_holdout = arima(ts(deseasonal_open[-c(1734:1764)]), order=c(2,2,7))
> fcast_no_holdout <- forecast(fit_no_holdout,h=30)
> plot(fcast_no_holdout, main=" ")
> lines(ts(deseasonal_open))
```



The blue line that shows forecast has an upward trend which shows the same behavior as the series. This model can be used as a benchmark against more complicated models. Forecasting improvement We can improve the forecast by adding back the seasonal component to the model or by setting (P, D, Q) components in the auto.arima() function.

```
> #Forecasting improvement
>  fit_w_seasonality = auto.arima(deseasonal_open, seasonal=TRUE)
>  fit_w_seasonality

Series: deseasonal_open
ARIMA(2,2,2)(0,0,1)[30]

Coefficients:
          ar1      ar2     ma1     ma2     sma1
      -1.1775  -0.8810  1.1138  0.7552  -0.0358
s.e.   0.0414   0.0235  0.0600  0.0308   0.0242

sigma^2 estimated as 7.186:  log likelihood=-4574.47
AIC=9160.93   AICc=9160.98   BIC=9194.24

>  seas_fcast <- forecast(fit_w_seasonality, h=30)
>  plot(seas_fcast)
```



The 80% and 95% confidence limits are shown with the darker and lighter grey respectively. It is inevitable that long-run forecasts cause a high level of uncertainty. This is due to the current value of Y is dependent on previous ones. We can evaluate the model residuals again. As shown, the autocorrelation with lag 7 recommends a higher order component for the model.

It should be noted that we followed the tutorial to fit the ARIMA model to our data.

5 Conclusions