**Using R for Time Series Analysis**

(ARIMA Model)

In general, Time Series data is numerical data obtained at regular time intervals. The time intervals can be Annual, Monthly, Quarterly, weekly, daily or hourly. Time series data is mainly based on TIME as an important variable. Time series data are collated to attempt determine the changes in the numeric variable over time, by examining patterns, cycles, trends and irregularities – all about previous time periods, with the help of which we predict the outcome for the future time period.

**Time Series Plot:-**

This is a two-dimensional scatter plot which measures the numeric variable of interest, (ex: revenues) on the vertical Y Axis, corresponding to Time periods in the horizontal Y Axis.

**Time Series Components:**

Time series data are composed of 4 elements. Not all data have all these elements. Most of them are said to have 1,2,3 below.

***1.Trend –*** A long-term general direction of the data, which extends over a period of years, is called TREND. Even though the data above shows upward and downward patterns, the general direction of the data shows upward direction or incremental growth. This is represented by Trend, which can be upward, download, linear or nonlinear.

******

***2) CYCLE -***  Cycles are patterns of highs and lows through which data move over time periods, usually of a couple of years or even longer. Time series data that do not extend over a long period of time may not have enough history to show cyclical effects. Thus cycles are not meant for short-term patterns. Here, the data is over a period of 10 years. They are often measured peak to peak.

 ******

***3) Seasonality -***  Seasonal effects are shorter cycles, many of which occur during a one-year period. Variations in the numeric variables are attributed to the changes in the seasons or buying patterns of the market. These are also upward or downward patterns, but occur within a period of a year.

******

***4) Irregular Fluctuations -***  they are rapid changes which are often unexplained. They can be caused by economical depressions, sudden changes in government policies, natural calamities etc.

***Reading Time Series Data***

The first thing that we want to do to analyse your time series data will be to read it into R, and to plot the time series. we can read data into R using the scan() function, which assumes that data for successive time points is in a simple text file with one column.

For example, the file <http://robjhyndman.com/tsdldata/misc/kings.dat> contains data on the age of death of successive kings of England, starting with William the Conqueror (original source: Hipel and Mcleod, 1994).

The data set looks like this:

**Age of Death of Successive Kings of England**

**#starting with William the Conqueror**

**#Source: McNeill, "Interactive Data Analysis"**

**60**

**43**

**67**

**50**

**56**

**42**

**50**

**65**

**68**

**43**

**65**

**34**

**...**

Only the first few lines of the file have been shown. The first three lines contain some comment on the data, and we want to ignore this when we read the data into R. We can use this by using the “skip” parameter of the scan() function, which specifies how many lines at the top of the file to ignore. To read the file into R, ignoring the first three lines, we type:

**> kings <- scan("http://robjhyndman.com/tsdldata/misc/kings.dat",skip=3)**

 **Read 42 items**

**> kings**

 **[1] 60 43 67 50 56 42 50 65 68 43 65 34 47 34 49 41 13 35 53 56 16 43 69 59 48**

 **[26] 59 86 55 68 51 33 49 67 77 81 67 71 81 68 70 77 56**

In this case the age of death of 42 successive kings of England has been read into the variable ‘kings’.

Once you have read the time series data into R, the next step is to store the data in a time series object in R, so that you can use R’s many functions for analysing time series data. To store the data in a time series object, we use the ts() function in R. For example, to store the data in the variable ‘kings’ as a time series object in R, we type:

**> kingstimeseries <- ts(kings)**

**> kingstimeseries**

 **Time Series:**

 **Start = 1**

 **End = 42**

 **Frequency = 1**

 **[1] 60 43 67 50 56 42 50 65 68 43 65 34 47 34 49 41 13 35 53 56 16 43 69 59 48**

 **[26] 59 86 55 68 51 33 49 67 77 81 67 71 81 68 70 77 56**

Sometimes the time series data set that you have may have been collected at regular intervals that were less than one year, for example, monthly or quarterly. In this case, you can specify the number of times that data was collected per year by using the ‘frequency’ parameter in the ts() function. For monthly time series data, you set frequency=12, while for quarterly time series data, you set frequency=4.

You can also specify the first year that the data was collected, and the first interval in that year by using the ‘start’ parameter in the ts() function. For example, if the first data point corresponds to the second quarter of 1986, you would set start=c(1986,2).

An example is a data set of the number of births per month in New York city, from January 1946 to December 1959 (originally collected by Newton). This data is available in the file <http://robjhyndman.com/tsdldata/data/nybirths.dat> We can read the data into R, and store it as a time series object, by typing:

> births <- scan("http://robjhyndman.com/tsdldata/data/nybirths.dat")

 Read 168 items

**> birthstimeseries <- ts(births, frequency=12, start=c(1946,1))**

**> birthstimeseries**

 **Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec**

 **1946 26.663 23.598 26.931 24.740 25.806 24.364 24.477 23.901 23.175 23.227 21.672 21.870**

 **1947 21.439 21.089 23.709 21.669 21.752 20.761 23.479 23.824 23.105 23.110 21.759 22.073**

 **1948 21.937 20.035 23.590 21.672 22.222 22.123 23.950 23.504 22.238 23.142 21.059 21.573**

 **1949 21.548 20.000 22.424 20.615 21.761 22.874 24.104 23.748 23.262 22.907 21.519 22.025**

 **1950 22.604 20.894 24.677 23.673 25.320 23.583 24.671 24.454 24.122 24.252 22.084 22.991**

 **1951 23.287 23.049 25.076 24.037 24.430 24.667 26.451 25.618 25.014 25.110 22.964 23.981**

 **1952 23.798 22.270 24.775 22.646 23.988 24.737 26.276 25.816 25.210 25.199 23.162 24.707**

 **1953 24.364 22.644 25.565 24.062 25.431 24.635 27.009 26.606 26.268 26.462 25.246 25.180**

 **1954 24.657 23.304 26.982 26.199 27.210 26.122 26.706 26.878 26.152 26.379 24.712 25.688**

 **1955 24.990 24.239 26.721 23.475 24.767 26.219 28.361 28.599 27.914 27.784 25.693 26.881**

 **1956 26.217 24.218 27.914 26.975 28.527 27.139 28.982 28.169 28.056 29.136 26.291 26.987**

 **1957 26.589 24.848 27.543 26.896 28.878 27.390 28.065 28.141 29.048 28.484 26.634 27.735**

 **1958 27.132 24.924 28.963 26.589 27.931 28.009 29.229 28.759 28.405 27.945 25.912 26.619**

 **1959 26.076 25.286 27.660 25.951 26.398 25.565 28.865 30.000 29.261 29.012 26.992 27.897**

Similarly, the file <http://robjhyndman.com/tsdldata/data/fancy.dat> contains monthly sales for a souvenir shop at a beach resort town in Queensland, Australia, for January 1987-December 1993 (original data from Wheelwright and Hyndman, 1998). We can read the data into R by typing:

**> souvenir <- scan("http://robjhyndman.com/tsdldata/data/fancy.dat")**

 **Read 84 items**

**> souvenirtimeseries <- ts(souvenir, frequency=12, start=c(1987,1))**

**> souvenirtimeseries**

 **Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec**

 **1987 1664.81 2397.53 2840.71 3547.29 3752.96 3714.74 4349.61 3566.34 5021.82 6423.48 7600.60 19756.21**

 **1988 2499.81 5198.24 7225.14 4806.03 5900.88 4951.34 6179.12 4752.15 5496.43 5835.10 12600.08 28541.72**

 **1989 4717.02 5702.63 9957.58 5304.78 6492.43 6630.80 7349.62 8176.62 8573.17 9690.50 15151.84 34061.01**

 **1990 5921.10 5814.58 12421.25 6369.77 7609.12 7224.75 8121.22 7979.25 8093.06 8476.70 17914.66 30114.41**

 **1991 4826.64 6470.23 9638.77 8821.17 8722.37 10209.48 11276.55 12552.22 11637.39 13606.89 21822.11 45060.69**

 **1992 7615.03 9849.69 14558.40 11587.33 9332.56 13082.09 16732.78 19888.61 23933.38 25391.35 36024.80 80721.71**

 **1993 10243.24 11266.88 21826.84 17357.33 15997.79 18601.53 26155.15 28586.52 30505.41 30821.33 46634.38 104660.67**

**Plotting Time Series**

**Once you have read a time series into R, the next step is usually to make a plot of the time series data, which you can do with the plot.ts() function in R.**

**For example, to plot the time series of the age of death of 42 successive kings of England, we type:**

**> plot.ts(kingstimeseries)**

****

We can see from the time plot that this time series could probably be described using an additive model, since the random fluctuations in the data are roughly constant in size over time.

Likewise, to plot the time series of the number of births per month in New York city, we type:

**> plot.ts(birthstimeseries)**

****

We can see from this time series that there seems to be seasonal variation in the number of births per month: there is a peak every summer, and a trough every winter. Again, it seems that this time series could probably be described using an additive model, as the seasonal fluctuations are roughly constant in size over time and do not seem to depend on the level of the time series, and the random fluctuations also seem to be roughly constant in size over time.

Similarly, to plot the time series of the monthly sales for the souvenir shop at a beach resort town in Queensland, Australia, we type:

**> plot.ts(souvenirtimeseries)**

****

In this case, it appears that an additive model is not appropriate for describing this time series, since the size of the seasonal fluctuations and random fluctuations seem to increase with the level of the time series. Thus, we may need to transform the time series in order to get a transformed time series that can be described using an additive model. For example, we can transform the time series by calculating the natural log of the original data:

**> logsouvenirtimeseries <- log(souvenirtimeseries)**

**> plot.ts(logsouvenirtimeseries)**

****

Here we can see that the size of the seasonal fluctuations and random fluctuations in the log-transformed time series seem to be roughly constant over time, and do not depend on the level of the time series. Thus, the log-transformed time series can probably be described using an additive model.

**Decomposing Time Series**

Decomposing a time series means separating it into its constituent components, which are usually a trend component and an irregular component, and if it is a seasonal time series, a seasonal component.

**Decomposing Non-Seasonal Data**

A non-seasonal time series consists of a trend component and an irregular component. Decomposing the time series involves trying to separate the time series into these components, that is, estimating the the trend component and the irregular component.

To estimate the trend component of a non-seasonal time series that can be described using an additive model, it is common to use a smoothing method, such as calculating the simple moving average of the time series.

The SMA() function in the “TTR” R package can be used to smooth time series data using a simple moving average. To use this function, we first need to install the “TTR” R package (for instructions on how to install an R package, see [How to install an R package](http://a-little-book-of-r-for-time-series.readthedocs.org/en/latest/src/installr.html#how-to-install-an-r-package)). Once you have installed the “TTR” R package, you can load the “TTR” R package by typing:

**> library("TTR")**

You can then use the “SMA()” function to smooth time series data. To use the SMA() function, you need to specify the order (span) of the simple moving average, using the parameter “n”. For example, to calculate a simple moving average of order 5, we set n=5 in the SMA() function.

For example, as discussed above, the time series of the age of death of 42 successive kings of England appears is non-seasonal, and can probably be described using an additive model, since the random fluctuations in the data are roughly constant in size over time:



Thus, we can try to estimate the trend component of this time series by smoothing using a simple moving average. To smooth the time series using a simple moving average of order 3, and plot the smoothed time series data, we type:

**> kingstimeseriesSMA3 <- SMA(kingstimeseries,n=3)**

**> plot.ts(kingstimeseriesSMA3)**



There still appears to be quite a lot of random fluctuations in the time series smoothed using a simple moving average of order 3. Thus, to estimate the trend component more accurately, we might want to try smoothing the data with a simple moving average of a higher order. This takes a little bit of trial-and-error, to find the right amount of smoothing. For example, we can try using a simple moving average of order 8:

> kingstimeseriesSMA8 <- SMA(kingstimeseries,n=8)

> plot.ts(kingstimeseriesSMA8)



The data smoothed with a simple moving average of order 8 gives a clearer picture of the trend component, and we can see that the age of death of the English kings seems to have decreased from about 55 years old to about 38 years old during the reign of the first 20 kings, and then increased after that to about 73 years old by the end of the reign of the 40th king in the time series.

**Decomposing Seasonal Data**

A seasonal time series consists of a trend component, a seasonal component and an irregular component. Decomposing the time series means separating the time series into these three components: that is, estimating these three components.

To estimate the trend component and seasonal component of a seasonal time series that can be described using an additive model, we can use the “decompose()” function in R. This function estimates the trend, seasonal, and irregular components of a time series that can be described using an additive model.

The function “decompose()” returns a list object as its result, where the estimates of the seasonal component, trend component and irregular component are stored in named elements of that list objects, called “seasonal”, “trend”, and “random” respectively.

For example, as discussed above, the time series of the number of births per month in New York city is seasonal with a peak every summer and trough every winter, and can probably be described using an additive model since the seasonal and random fluctuations seem to be roughly constant in size over time:



To estimate the trend, seasonal and irregular components of this time series, we type:

**> birthstimeseriescomponents <- decompose(birthstimeseries)**

The estimated values of the seasonal, trend and irregular components are now stored in variables birthstimeseriescomponents$seasonal, birthstimeseriescomponents$trend and birthstimeseriescomponents$random. For example, we can print out the estimated values of the seasonal component by typing:

**> birthstimeseriescomponents$seasonal *# get the estimated values of the seasonal component***

 **Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec**

 **1946 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

 **1947 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

 **1948 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

 **1949 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

 **1950 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

 **1951 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

 **1952 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

 **1953 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

 **1954 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

 **1955 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

 **1956 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

 **1957 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

 **1958 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

 **1959 -0.6771947 -2.0829607 0.8625232 -0.8016787 0.2516514 -0.1532556 1.4560457 1.1645938 0.6916162 0.7752444 -1.1097652 -0.3768197**

The estimated seasonal factors are given for the months January-December, and are the same for each year. The largest seasonal factor is for July (about 1.46), and the lowest is for February (about -2.08), indicating that there seems to be a peak in births in July and a trough in births in February each year.

We can plot the estimated trend, seasonal, and irregular components of the time series by using the “plot()” function, for example:

**> plot(birthstimeseriescomponents)**



The plot above shows the original time series (top), the estimated trend component (second from top), the estimated seasonal component (third from top), and the estimated irregular component (bottom). We see that the estimated trend component shows a small decrease from about 24 in 1947 to about 22 in 1948, followed by a steady increase from then on to about 27 in 1959.

**Seasonally Adjusting**

If you have a seasonal time series that can be described using an additive model, you can seasonally adjust the time series by estimating the seasonal component, and subtracting the estimated seasonal component from the original time series. We can do this using the estimate of the seasonal component calculated by the “decompose()” function.

For example, to seasonally adjust the time series of the number of births per month in New York city, we can estimate the seasonal component using “decompose()”, and then subtract the seasonal component from the original time series:

**> birthstimeseriescomponents <- decompose(birthstimeseries)**

**> birthstimeseriesseasonallyadjusted <- birthstimeseries - birthstimeseriescomponents$seasonal**

We can then plot the seasonally adjusted time series using the “plot()” function, by typing:

**> plot(birthstimeseriesseasonallyadjusted)**

****

You can see that the seasonal variation has been removed from the seasonally adjusted time series. The seasonally adjusted time series now just contains the trend component and an irregular component.

The time series of first differences appears to be stationary in mean and variance, and so an ARIMA(p,1,q) model is probably appropriate for the time series of the age of death of the kings of England. By taking the time series of first differences, we have removed the trend component of the time series of the ages at death of the kings, and are left with an irregular component. We can now examine whether there are correlations between successive terms of this irregular component; if so, this could help us to make a predictive model for the ages at death of the kings.

**Selecting a Candidate ARIMA Model**

If your time series is stationary, or if you have transformed it to a stationary time series by differencing d times, the next step is to select the appropriate ARIMA model, which means finding the values of most appropriate values of p and q for an ARIMA(p,d,q) model. To do this, you usually need to examine the correlogram and partial correlogram of the stationary time series.

To plot a correlogram and partial correlogram, we can use the “acf()” and “pacf()” functions in R, respectively. To get the actual values of the autocorrelations and partial autocorrelations, we set “plot=FALSE” in the “acf()” and “pacf()” functions.

**Example of the Ages at Death of the Kings of England**

For example, to plot the correlogram for lags 1-20 of the once differenced time series of the ages at death of the kings of England, and to get the values of the autocorrelations, we type:

> acf(kingtimeseriesdiff1, lag.max=20) *# plot a correlogram*

> acf(kingtimeseriesdiff1, lag.max=20, plot=**FALSE**) *# get the autocorrelation values*

 Autocorrelations of series 'kingtimeseriesdiff1', by lag

 0 1 2 3 4 5 6 7 8 9 10

 1.000 -0.360 -0.162 -0.050 0.227 -0.042 -0.181 0.095 0.064 -0.116 -0.071

 11 12 13 14 15 16 17 18 19 20

 0.206 -0.017 -0.212 0.130 0.114 -0.009 -0.192 0.072 0.113 -0.093



We see from the correlogram that the autocorrelation at lag 1 (-0.360) exceeds the significance bounds, but all other autocorrelations between lags 1-20 do not exceed the significance bounds.

To plot the partial correlogram for lags 1-20 for the once differenced time series of the ages at death of the English kings, and get the values of the partial autocorrelations, we use the “pacf()” function, by typing:

**> pacf(kingtimeseriesdiff1, lag.max=20) # plot a partial correlogram**

**> pacf(kingtimeseriesdiff1, lag.max=20, plot=FALSE) # get the partial autocorrelation values**

 **Partial autocorrelations of series 'kingtimeseriesdiff1', by lag**

 **1 2 3 4 5 6 7 8 9 10 11**

 **-0.360 -0.335 -0.321 0.005 0.025 -0.144 -0.022 -0.007 -0.143 -0.167 0.065**

 **12 13 14 15 16 17 18 19 20**

 **0.034 -0.161 0.036 0.066 0.081 -0.005 -0.027 -0.006 -0.037**



The partial correlogram shows that the partial autocorrelations at lags 1, 2 and 3 exceed the significance bounds, are negative, and are slowly decreasing in magnitude with increasing lag (lag 1: -0.360, lag 2: -0.335, lag 3:-0.321). The partial autocorrelations tail off to zero after lag 3.

Since the correlogram is zero after lag 1, and the partial correlogram tails off to zero after lag 3, this means that the following ARMA (autoregressive moving average) models are possible for the time series of first differences:

* an ARMA(3,0) model, that is, an autoregressive model of order p=3, since the partial autocorrelogram is zero after lag 3, and the autocorrelogram tails off to zero (although perhaps too abruptly for this model to be appropriate)
* an ARMA(0,1) model, that is, a moving average model of order q=1, since the autocorrelogram is zero after lag 1 and the partial autocorrelogram tails off to zero
* an ARMA(p,q) model, that is, a mixed model with p and q greater than 0, since the autocorrelogram and partial correlogram tail off to zero (although the correlogram probably tails off to zero too abruptly for this model to be appropriate)

We use the principle of parsimony to decide which model is best: that is, we assume that the model with the fewest parameters is best. The ARMA(3,0) model has 3 parameters, the ARMA(0,1) model has 1 parameter, and the ARMA(p,q) model has at least 2 parameters. Therefore, the ARMA(0,1) model is taken as the best model.

An ARMA(0,1) model is a moving average model of order 1, or MA(1) model. This model can be written as: X\_t - mu = Z\_t - (theta \* Z\_t-1), where X\_t is the stationary time series we are studying (the first differenced series of ages at death of English kings), mu is the mean of time series X\_t, Z\_t is white noise with mean zero and constant variance, and theta is a parameter that can be estimated.

A MA (moving average) model is usually used to model a time series that shows short-term dependencies between successive observations. Intuitively, it makes good sense that a MA model can be used to describe the irregular component in the time series of ages at death of English kings, as we might expect the age at death of a particular English king to have some effect on the ages at death of the next king or two, but not much effect on the ages at death of kings that reign much longer after that.

**Shortcut: the auto.arima() function**

The auto.arima() function can be used to find the appropriate ARIMA model, eg., type “library(forecast)”, then “auto.arima(kings)”. The output says an appropriate model is ARIMA(0,1,1).

Since an ARMA(0,1) model (with p=0, q=1) is taken to be the best candidate model for the time series of first differences of the ages at death of English kings, then the original time series of the ages of death can be modelled using an ARIMA(0,1,1) model (with p=0, d=1, q=1, where d is the order of differencing required).

**Example of the Volcanic Dust Veil in the Northern Hemisphere**

Let’s take another example of selecting an appropriate ARIMA model. The file file<http://robjhyndman.com/tsdldata/annual/dvi.dat> contains data on the volcanic dust veil index in the northern hemisphere, from 1500-1969 (original data from Hipel and Mcleod, 1994). This is a measure of the impact of volcanic eruptions’ release of dust and aerosols into the environment. We can read it into R and make a time plot by typing:

**> volcanodust <- scan("http://robjhyndman.com/tsdldata/annual/dvi.dat", skip=1)**

 **Read 470 items**

**> volcanodustseries <- ts(volcanodust,start=c(1500))**

**> plot.ts(volcanodustseries)**



From the time plot, it appears that the random fluctuations in the time series are roughly constant in size over time, so an additive model is probably appropriate for describing this time series.

Furthermore, the time series appears to be stationary in mean and variance, as its level and variance appear to be roughly constant over time. Therefore, we do not need to difference this series in order to fit an ARIMA model, but can fit an ARIMA model to the original series (the order of differencing required, d, is zero here).

We can now plot a correlogram and partial correlogram for lags 1-20 to investigate what ARIMA model to use:

**> acf(volcanodustseries, lag.max=20) # plot a correlogram**

**> acf(volcanodustseries, lag.max=20, plot=FALSE) # get the values of the autocorrelations**

 **Autocorrelations of series 'volcanodustseries', by lag**

 **0 1 2 3 4 5 6 7 8 9 10**

 **1.000 0.666 0.374 0.162 0.046 0.017 -0.007 0.016 0.021 0.006 0.010**

 **11 12 13 14 15 16 17 18 19 20**

 **0.004 0.024 0.075 0.082 0.064 0.039 0.005 0.028 0.108 0.182**



We see from the correlogram that the autocorrelations for lags 1, 2 and 3 exceed the significance bounds, and that the autocorrelations tail off to zero after lag 3. The autocorrelations for lags 1, 2, 3 are positive, and decrease in magnitude with increasing lag (lag 1: 0.666, lag 2: 0.374, lag 3: 0.162).

The autocorrelation for lags 19 and 20 exceed the significance bounds too, but it is likely that this is due to chance, since they just exceed the significance bounds (especially for lag 19), the autocorrelations for lags 4-18 do not exceed the signifiance bounds, and we would expect 1 in 20 lags to exceed the 95% significance bounds by chance alone.

**> pacf(volcanodustseries, lag.max=20)**

**> pacf(volcanodustseries, lag.max=20, plot=FALSE)**

 **Partial autocorrelations of series 'volcanodustseries', by lag**

 **1 2 3 4 5 6 7 8 9 10 11**

 **0.666 -0.126 -0.064 -0.005 0.040 -0.039 0.058 -0.016 -0.025 0.028 -0.008**

 **12 13 14 15 16 17 18 19 20**

 **0.036 0.082 -0.025 -0.014 0.008 -0.025 0.073 0.131 0.063**



From the partial autocorrelogram, we see that the partial autocorrelation at lag 1 is positive and exceeds the significance bounds (0.666), while the partial autocorrelation at lag 2 is negative and also exceeds the significance bounds (-0.126). The partial autocorrelations tail off to zero after lag 2.

Since the correlogram tails off to zero after lag 3, and the partial correlogram is zero after lag 2, the following ARMA models are possible for the time series:

* an ARMA(2,0) model, since the partial autocorrelogram is zero after lag 2, and the correlogram tails off to zero after lag 3, and the partial correlogram is zero after lag 2
* an ARMA(0,3) model, since the autocorrelogram is zero after lag 3, and the partial correlogram tails off to zero (although perhaps too abruptly for this model to be appropriate)
* an ARMA(p,q) mixed model, since the correlogram and partial correlogram tail off to zero (although the partial correlogram perhaps tails off too abruptly for this model to be appropriate)

**Shortcut: the auto.arima() function**

Again, we can use auto.arima() to find an appropriate model, by typing “auto.arima(volcanodust)”, which gives us ARIMA(1,0,2), which has 3 parameters. However, different criteria can be used to select a model (see auto.arima() help page). If we use the “bic” criterion, which penalises the number of parameters, we get ARIMA(2,0,0), which is ARMA(2,0): “auto.arima(volcanodust,ic=”bic”)”.

The ARMA(2,0) model has 2 parameters, the ARMA(0,3) model has 3 parameters, and the ARMA(p,q) model has at least 2 parameters. Therefore, using the principle of parsimony, the ARMA(2,0) model and ARMA(p,q) model are equally good candidate models.

An ARMA(2,0) model is an autoregressive model of order 2, or AR(2) model. This model can be written as: X\_t - mu = (Beta1 \* (X\_t-1 - mu)) + (Beta2 \* (Xt-2 - mu)) + Z\_t, where X\_t is the stationary time series we are studying (the time series of volcanic dust veil index), mu is the mean of time series X\_t, Beta1 and Beta2 are parameters to be estimated, and Z\_t is white noise with mean zero and constant variance.

An AR (autoregressive) model is usually used to model a time series which shows longer term dependencies between successive observations. Intuitively, it makes sense that an AR model could be used to describe the time series of volcanic dust veil index, as we would expect volcanic dust and aerosol levels in one year to affect those in much later years, since the dust and aerosols are unlikely to disappear quickly.

If an ARMA(2,0) model (with p=2, q=0) is used to model the time series of volcanic dust veil index, it would mean that an ARIMA(2,0,0) model can be used (with p=2, d=0, q=0, where d is the order of differencing required). Similarly, if an ARMA(p,q) mixed model is used, where p and q are both greater than zero, than an ARIMA(p,0,q) model can be used.

**Forecasting Using an ARIMA Model**

Once you have selected the best candidate ARIMA(p,d,q) model for your time series data, you can estimate the parameters of that ARIMA model, and use that as a predictive model for making forecasts for future values of your time series.

You can estimate the parameters of an ARIMA(p,d,q) model using the “arima()” function in R.

**Example of the Ages at Death of the Kings of England**

For example, we discussed above that an ARIMA(0,1,1) model seems a plausible model for the ages at deaths of the kings of England. You can specify the values of p, d and q in the ARIMA model by using the “order” argument of the “arima()” function in R. To fit an ARIMA(p,d,q) model to this time series (which we stored in the variable “kingstimeseries”, see above), we type:

**> kingstimeseriesarima <- arima(kingstimeseries, order=c(0,1,1)) # fit an ARIMA(0,1,1) model**

**> kingstimeseriesarima**

 **ARIMA(0,1,1)**

 **Coefficients:**

 **ma1**

 **-0.7218**

 **s.e. 0.1208**

 **sigma^2 estimated as 230.4: log likelihood = -170.06**

 **AIC = 344.13 AICc = 344.44 BIC = 347.56**

As mentioned above, if we are fitting an ARIMA(0,1,1) model to our time series, it means we are fitting an an ARMA(0,1) model to the time series of first differences. An ARMA(0,1) model can be written X\_t - mu = Z\_t - (theta \* Z\_t-1), where theta is a parameter to be estimated. From the output of the “arima()” R function (above), the estimated value of theta (given as ‘ma1’ in the R output) is -0.7218 in the case of the ARIMA(0,1,1) model fitted to the time series of ages at death of kings.

**Specifying the confidence level for prediction intervals**

You can specify the confidence level for prediction intervals in forecast.Arima() by using the “level” argument. For example, to get a 99.5% prediction interval, we would type “forecast.Arima(kingstimeseriesarima, h=5, level=c(99.5))”.

We can then use the ARIMA model to make forecasts for future values of the time series, using the “forecast.Arima()” function in the “forecast” R package. For example, to forecast the ages at death of the next five English kings, we type:

**> library("forecast") # load the "forecast" R library**

**> kingstimeseriesforecasts <- forecast.Arima(kingstimeseriesarima, h=5)**

**> kingstimeseriesforecasts**

 **Point Forecast Lo 80 Hi 80 Lo 95 Hi 95**

 **43 67.75063 48.29647 87.20479 37.99806 97.50319**

 **44 67.75063 47.55748 87.94377 36.86788 98.63338**

 **45 67.75063 46.84460 88.65665 35.77762 99.72363**

 **46 67.75063 46.15524 89.34601 34.72333 100.77792**

 **47 67.75063 45.48722 90.01404 33.70168 101.79958**

The original time series for the English kings includes the ages at death of 42 English kings. The forecast.Arima() function gives us a forecast of the age of death of the next five English kings (kings 43-47), as well as 80% and 95% prediction intervals for those predictions. The age of death of the 42nd English king was 56 years (the last observed value in our time series), and the ARIMA model gives the forecasted age at death of the next five kings as 67.8 years.

We can plot the observed ages of death for the first 42 kings, as well as the ages that would be predicted for these 42 kings and for the next 5 kings using our ARIMA(0,1,1) model, by typing:

**> plot.forecast(kingstimeseriesforecasts)**



As in the case of exponential smoothing models, it is a good idea to investigate whether the forecast errors of an ARIMA model are normally distributed with mean zero and constant variance, and whether the are correlations between successive forecast errors.

For example, we can make a correlogram of the forecast errors for our ARIMA(0,1,1) model for the ages at death of kings, and perform the Ljung-Box test for lags 1-20, by typing:

**> acf(kingstimeseriesforecasts$residuals, lag.max=20)**

**> Box.test(kingstimeseriesforecasts$residuals, lag=20, type="Ljung-Box")**

 **Box-Ljung test**

 **data: kingstimeseriesforecasts$residuals**

 **X-squared = 13.5844, df = 20, p-value = 0.851**



Since the correlogram shows that none of the sample autocorrelations for lags 1-20 exceed the significance bounds, and the p-value for the Ljung-Box test is 0.9, we can conclude that there is very little evidence for non-zero autocorrelations in the forecast errors at lags 1-20.

To investigate whether the forecast errors are normally distributed with mean zero and constant variance, we can make a time plot and histogram (with overlaid normal curve) of the forecast errors:

**> plot.ts(kingstimeseriesforecasts$residuals) # make time plot of forecast errors**

**> plotForecastErrors(kingstimeseriesforecasts$residuals) # make a histogram**





The time plot of the in-sample forecast errors shows that the variance of the forecast errors seems to be roughly constant over time (though perhaps there is slightly higher variance for the second half of the time series). The histogram of the time series shows that the forecast errors are roughly normally distributed and the mean seems to be close to zero. Therefore, it is plausible that the forecast errors are normally distributed with mean zero and constant variance.

Since successive forecast errors do not seem to be correlated, and the forecast errors seem to be normally distributed with mean zero and constant variance, the ARIMA(0,1,1) does seem to provide an adequate predictive model for the ages at death of English kings.

**Example of the Volcanic Dust Veil in the Northern Hemisphere**

We discussed above that an appropriate ARIMA model for the time series of volcanic dust veil index may be an ARIMA(2,0,0) model. To fit an ARIMA(2,0,0) model to this time series, we can type:

**> volcanodustseriesarima <- arima(volcanodustseries, order=c(2,0,0))**

**> volcanodustseriesarima**

 **ARIMA(2,0,0) with non-zero mean**

 **Coefficients:**

 **ar1 ar2 intercept**

 **0.7533 -0.1268 57.5274**

 **s.e. 0.0457 0.0458 8.5958**

 **sigma^2 estimated as 4870: log likelihood = -2662.54**

 **AIC = 5333.09 AICc = 5333.17 BIC = 5349.7**

**As mentioned above, an ARIMA(2,0,0) model can be written** as: written as: X\_t - mu = (Beta1 \* (X\_t-1 - mu)) + (Beta2 \* (Xt-2 - mu)) + Z\_t, where Beta1 and Beta2 are parameters to be estimated. The output of the arima() function tells us that Beta1 and Beta2 are estimated as 0.7533 and -0.1268 here (given as ar1 and ar2 in the output of arima()).

Now we have fitted the ARIMA(2,0,0) model, we can use the “forecast.ARIMA()” model to predict future values of the volcanic dust veil index. The original data includes the years 1500-1969. To make predictions for the years 1970-2000 (31 more years), we type:

**> volcanodustseriesforecasts <- forecast.Arima(volcanodustseriesarima, h=31)**

**> volcanodustseriesforecasts**

 **Point Forecast Lo 80 Hi 80 Lo 95 Hi 95**

 **1970 21.48131 -67.94860 110.9112 -115.2899 158.2526**

 **1971 37.66419 -74.30305 149.6314 -133.5749 208.9033**

 **1972 47.13261 -71.57070 165.8359 -134.4084 228.6737**

 **1973 52.21432 -68.35951 172.7881 -132.1874 236.6161**

 **1974 54.84241 -66.22681 175.9116 -130.3170 240.0018**

 **1975 56.17814 -65.01872 177.3750 -129.1765 241.5327**

 **1976 56.85128 -64.37798 178.0805 -128.5529 242.2554**

 **1977 57.18907 -64.04834 178.4265 -128.2276 242.6057**

 **1978 57.35822 -63.88124 178.5977 -128.0615 242.7780**

 **1979 57.44283 -63.79714 178.6828 -127.9777 242.8634**

 **1980 57.48513 -63.75497 178.7252 -127.9356 242.9059**

 **1981 57.50627 -63.73386 178.7464 -127.9145 242.9271**

 **1982 57.51684 -63.72330 178.7570 -127.9040 242.9376**

 **1983 57.52212 -63.71802 178.7623 -127.8987 242.9429**

 **1984 57.52476 -63.71538 178.7649 -127.8960 242.9456**

 **1985 57.52607 -63.71407 178.7662 -127.8947 242.9469**

 **1986 57.52673 -63.71341 178.7669 -127.8941 242.9475**

 **1987 57.52706 -63.71308 178.7672 -127.8937 242.9479**

 **1988 57.52723 -63.71291 178.7674 -127.8936 242.9480**

 **1989 57.52731 -63.71283 178.7674 -127.8935 242.9481**

 **1990 57.52735 -63.71279 178.7675 -127.8934 242.9481**

 **1991 57.52737 -63.71277 178.7675 -127.8934 242.9482**

 **1992 57.52738 -63.71276 178.7675 -127.8934 242.9482**

 **1993 57.52739 -63.71275 178.7675 -127.8934 242.9482**

 **1994 57.52739 -63.71275 178.7675 -127.8934 242.9482**

 **1995 57.52739 -63.71275 178.7675 -127.8934 242.9482**

 **1996 57.52739 -63.71275 178.7675 -127.8934 242.9482**

 **1997 57.52739 -63.71275 178.7675 -127.8934 242.9482**

 **1998 57.52739 -63.71275 178.7675 -127.8934 242.9482**

 **1999 57.52739 -63.71275 178.7675 -127.8934 242.9482**

 **2000 57.52739 -63.71275 178.7675 -127.8934 242.9482**

We can plot the original time series, and the forecasted values, by typing:

**> plot.forecast(volcanodustseriesforecasts)**



One worrying thing is that the model has predicted negative values for the volcanic dust veil index, but this variable can only have positive values! The reason is that the arima() and forecast.Arima() functions don’t know that the variable can only take positive values. Clearly, this is not a very desirable feature of our current predictive model.

Again, we should investigate whether the forecast errors seem to be correlated, and whether they are normally distributed with mean zero and constant variance. To check for correlations between successive forecast errors, we can make a correlogram and use the Ljung-Box test:

**> acf(volcanodustseriesforecasts$residuals, lag.max=20)**

**> Box.test(volcanodustseriesforecasts$residuals, lag=20, type="Ljung-Box")**

 **Box-Ljung test**

 **data: volcanodustseriesforecasts$residuals**

 **X-squared = 24.3642, df = 20, p-value = 0.2268**



The correlogram shows that the sample autocorrelation at lag 20 exceeds the significance bounds. However, this is probably due to chance, since we would expect one out of 20 sample autocorrelations to exceed the 95% significance bounds. Furthermore, the p-value for the Ljung-Box test is 0.2, indicating that there is little evidence for non-zero autocorrelations in the forecast errors for lags 1-20.

To check whether the forecast errors are normally distributed with mean zero and constant variance, we make a time plot of the forecast errors, and a histogram:

**> plot.ts(volcanodustseriesforecasts$residuals) # make time plot of forecast errors**

**> plotForecastErrors(volcanodustseriesforecasts$residuals) # make a histogram**





The time plot of forecast errors shows that the forecast errors seem to have roughly constant variance over time. However, the time series of forecast errors seems to have a negative mean, rather than a zero mean. We can confirm this by calculating the mean forecast error, which turns out to be about -0.22:

**> mean(volcanodustseriesforecasts$residuals)**

 **-0.2205417**

The histogram of forecast errors (above) shows that although the mean value of the forecast errors is negative, the distribution of forecast errors is skewed to the right compared to a normal curve. Therefore, it seems that we cannot comfortably conclude that the forecast errors are normally distributed with mean zero and constant variance! Thus, it is likely that our ARIMA(2,0,0) model for the time series of volcanic dust veil index is not the best model that we could make, and could almost definitely be improved upon!