Taking Time Seriously: An Introduction to Time-Series Analysis

Dr. Niccole M. Pamphilis

This handout is meant to be a guide for those who are new to working with time-series data. Please note that there are many packages for carrying out time-series analysis in R and some are better fits for more complex models, while others are better for particular tests you may wish to run. Odds are, if there is a particular model you want to run or test you would like to use that is not covered in this introduction to time-series, a package for it exists out there.

The examples in this handout correspond to those used in a talk previously given, but the basic code itself can be applied to other data sets, which is exactly what you will be doing throughout the talk and can be used later.

Open data

Opening your dataset can be done in several ways depending on how the data is formatted. The data being used here is saved as a Stata file (.dta). As such, the "foreign" library needs to be used so R can recognised the data. Additionally, you can name your dataset anything you like within R, I tend to work with the basic name "data", but if you are working with multiple data sets at one time you may choose to be more creative.

```
library(foreign)
```

```
## Warning: package 'foreign' was built under R version 3.4.4
```

```
data<-read.dta(file.choose())
```

Time Setting your Dataset

The dataset, once loaded, needs to be set as time series data in order for R to recognise as the same observation(s) repeatedly viewed over time. This can be done using the "ts" command. Inside the parentheses the first argument represents the variable in the dataset that indicates time. Next you will indicate the start value, and the frequency of observation (for example if monthly it would be 12, if quarterly 4).

ts_data<-ts(Dependent_Variable, start=, freq=12)

```
approval<-ts(data$approval, start=c(1978,1), freq=12)
```

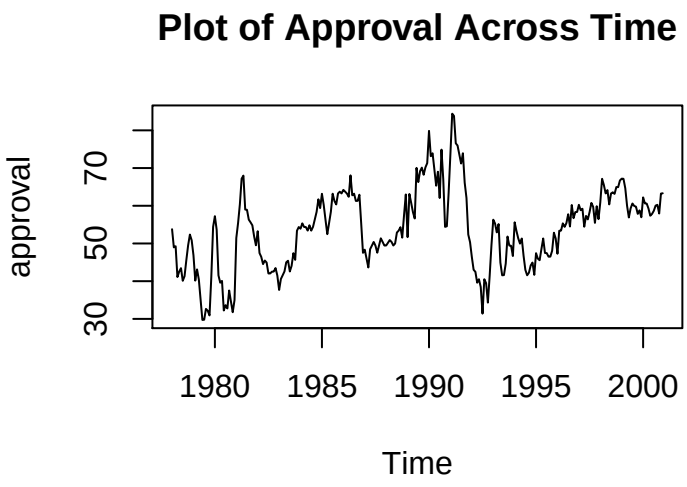
Looking at your time series

It is best to start any analysis by looking over your data. Below is a basic plot of the dependent variable of interest observed across time. Notice that time is on the x-axis and our variable of interest is on the y-axis.

The code to execute the plot is "plot.ts" with the dependent variable placed within the parentheses.

plot.ts(approval)

```
plot.ts(approval,main="Plot of Approval Across Time")
```



Generating ACF and PACF

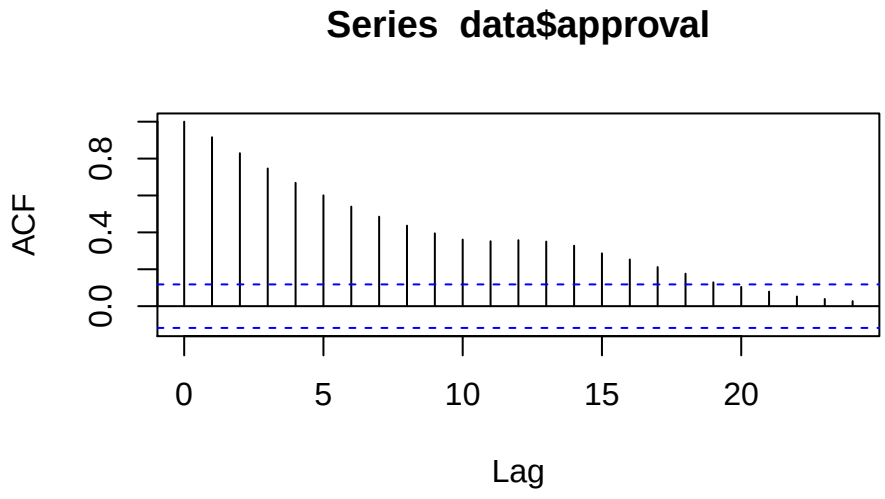
To determine what time components may be present in your data you make choose to look at the ACF and PACF. This can be done with "acf" and "pacf" commands. After creating the acf and pacf look to see either graph shows decay over time or presents statistically significant spikes to help diagnose time components.

To generate the Auto-correlation Function Plot:

acf(dependent_variable)

In the ACF below, do you see decay or spikes?

```
acf(data$approval)
```

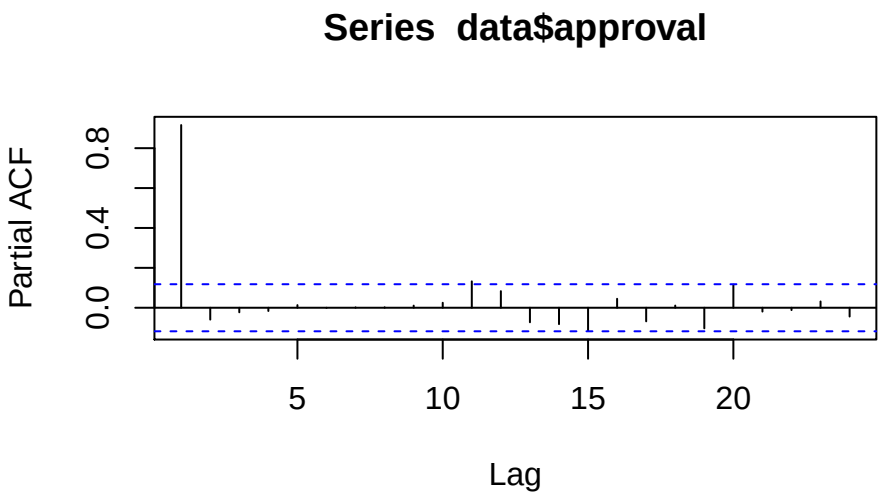


To generate the Partial Auto-correlation Function Plot:

pacf(dependent_variable)

In the PACF below do you see any decay or spikes?

```
pacf(data$approval)
```



Run ARMA Models

When working with time-series data you will likely find that you will need to run several possible/plausible models to see which is the best fit for your data. The code to execute an AR, MA, ARMA, or ARIMA model is presented below.

To run an ARMA model you need to first determine how many AR lags you need (p) and how many MA lags you need (q). The generic code is:

arima(dependent_variable, order=c(p,d,q))

The d in the code above refers to how the data should be differenced, which if it is just an ARMA model will be 0. We will discuss when d is not 0 in a little bit.

Running an AR(1) Model:

The command to run an AR model is "arima", here we are just including the dependent variable. Notice it is in the order argument that you indicate it is an AR model by changing the value of the "p" component. Here I have saved the results as an object "Approval100" and I need to call Approval100 to see the output. If I just execute the first line of code the model would be run and the results stored, but not printed.

```
   Approval100<-arima(data$approval, order=c(1,0,0))
   Approval100
```

```
##
## Call:
## arima(x = data$approval, order = c(1, 0, 0))
##
## Coefficients:
##          ar1  intercept
```

```
##       0.9155    54.0692
## s.e.  0.0235     2.8186
##
## sigma^2 estimated as 16.88:  log likelihood = -782.52,  aic = 1571.05
```

Running an MA(1) Model:

To run the MA model we use the same code as the AR model above, but this time we alter the value for the "q" component in the order argument.

```
    Approval001<-arima(data$approval, order=c(0,0,1))
    Approval001
```

```
##
## Call:
## arima(x = data$approval, order = c(0, 0, 1))
##
## Coefficients:
##          ma1  intercept
##       0.7642    53.7302
## s.e.  0.0277     0.7044
##
## sigma^2 estimated as 44.15:  log likelihood = -914.74,  aic = 1835.48
```

Running an AR(1) MA1(1) Model:

Finally, to run an ARMA model, we simply adjust the values for both the AR and MA components.

```
    Approval101<-arima(data$approval, order=c(1,0,1))
    Approval101
```

```
##
## Call:
## arima(x = data$approval, order = c(1, 0, 1))
##
## Coefficients:
##          ar1     ma1  intercept
##       0.9059  0.0604    54.0529
## s.e.  0.0270  0.0634     2.6916
##
## sigma^2 estimated as 16.82:  log likelihood = -782.07,  aic = 1572.15
```

Testing for White Noise

To determine which model is the best fit, you will need to undertake a few steps including testing for white noise. In this situation white noise implies we have removed all the time elements from our model and now we are in a position to determine the relationship between our independent variables of interest and the dependent variable.

First step in testing for White Noise is to generate residuals from your models. If you have saved your model results as above you can simply use the command "residual" on each set of model results.

Make sure to save the residuals from each model for future use as separate objects.

residual(model_name)

```
    approval100resid<-residuals(Approval100)
    approval001resid<-residuals(Approval001)
    approval101resid<-residuals(Approval101)
```

After generating residuals you can then test if there are any remaining correlations across time using a test for White Noise called the Ljung-Box Test. For this test, the null hypothesis is: All Auto-correlations for lags are jointly 0. Interpretation: if Q is not significant, series is white noise, which is what we want for our final model.

Box.test(model_residuals, type="Ljung-Box")

```
Box.test(approval100resid, type="Ljung-Box")
```

```
##
##  Box-Ljung test
##
## data:  approval100resid
## X-squared = 0.92318, df = 1, p-value = 0.3366
```

```
Box.test(approval001resid, type="Ljung-Box")
```

```
##
##  Box-Ljung test
##
## data:  approval001resid
## X-squared = 81.609, df = 1, p-value < 2.2e-16
```

```
Box.test(approval101resid, type="Ljung-Box")
```

```
##
##  Box-Ljung test
##
## data:  approval101resid
## X-squared = 0.0073273, df = 1, p-value = 0.9318
```

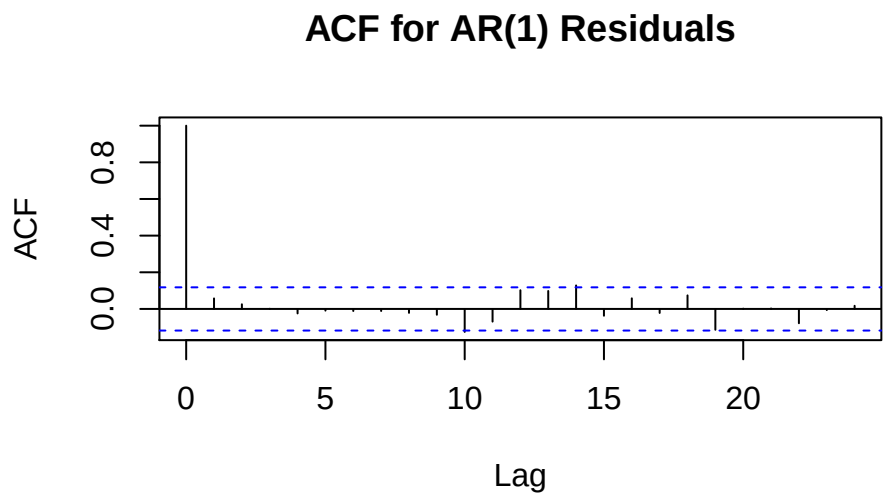
Visually, if you wanted to look at the remaining correlations across time in the residuals you could look at the acf for the model residuals.

What you would expect to see if you have controlled for the time processes would be one correlation at lag 0 that was statistically significant (i.e., the value at time t is perfectly correlated with itself) and the remaining lagged correlations failing to reach statistical significance.

acf(model_residuals):

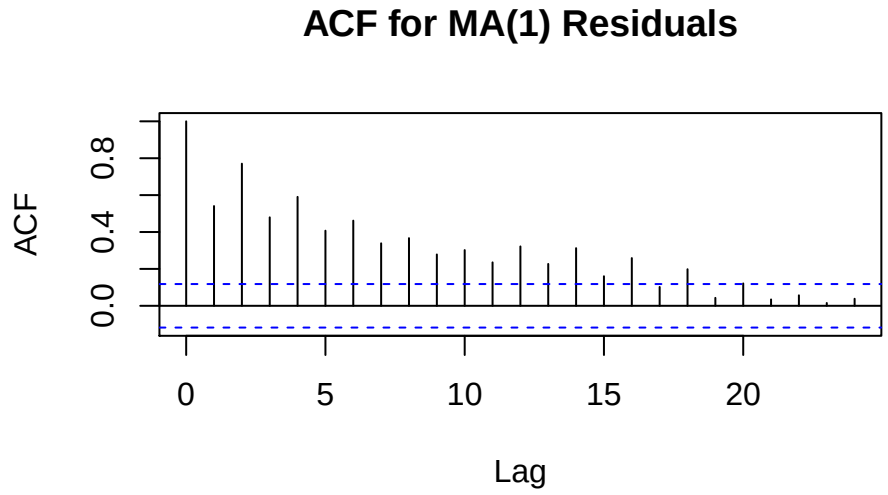
How does the ACF for the AR(1) model from our example look below?

```
   acf(approval100resid, main="ACF for AR(1) Residuals")
```



What about the residuals for the MA(1) model?

```
    acf(approval001resid, main="ACF for MA(1) Residuals")
```



Testing for Stationarity

First, open library with stationarity checks; this may need to be installed if you have not used it before.

```
 library(tseries)
```

```
## Warning: package 'tseries' was built under R version 3.4.4
```

Augmented Dickey-Fuller Test## Null:Unit Root, not stationary

adf.test(dependent_variable)

```
adf.test(data$approval)
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  data$approval
## Dickey-Fuller = -3.533, Lag order = 6, p-value = 0.04
## alternative hypothesis: stationary
```

Phillips-Perron test Null hypothesis: unit root, not stationary

PP.test(depdent_variable)

```
    PP.test(data$approval)
```

```
##
##  Phillips-Perron Unit Root Test
##
## data:  data$approval
## Dickey-Fuller = -3.958, Truncation lag parameter = 5, p-value =
## 0.01155
```

KPSS Test Null is that the variable it (level or trend) stationary

kpss.test(dependent_variable)

```
  kpss.test(data$approval)
```

```
## Warning in kpss.test(data$approval): p-value smaller than printed p-value
```

```
##
##  KPSS Test for Level Stationarity
##
## data:  data$approval
## KPSS Level = 1.2466, Truncation lag parameter = 3, p-value = 0.01
```

Differencing the data:

If your data are integrated you may choose to difference your data. This can be done with the command "diff", where the first argument is the variable you want to difference and the second argument is how many times you want this done.

Warning: Do not difference your data more than once, as this will induce more issues.

diff(dependent_variable, differences=1)

```
approvaldiff<-diff(data$approval, differences=1)
```

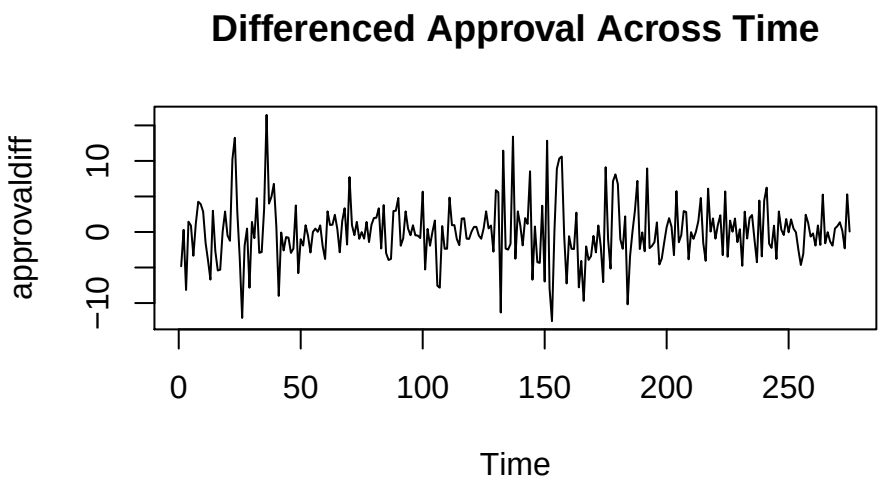
Retesting for Stationarity:

Once you have differenced your data, check to see if the issue of non-stationarity has been addressed.

Below is our approval data that has been differenced one time.

Visually, what does the differenced data look like?

```
plot.ts(approvaldiff, main="Differenced Approval Across Time")
```



You can also run the tests for stationarity on the differenced data as we did earlier.

```
adf.test(approvaldiff)
```

```
## Warning in adf.test(approvaldiff): p-value smaller than printed p-value
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  approvaldiff
## Dickey-Fuller = -7.1101, Lag order = 6, p-value = 0.01
## alternative hypothesis: stationary
```

```
pp.test(approvaldiff)
```

```
## Warning in pp.test(approvaldiff): p-value smaller than printed p-value
```

```
##
##  Phillips-Perron Unit Root Test
##
## data:  approvaldiff
## Dickey-Fuller Z(alpha) = -254.59, Truncation lag parameter = 5,
## p-value = 0.01
## alternative hypothesis: stationary
```

```
kpss.test(approvaldiff)
```

```
## Warning in kpss.test(approvaldiff): p-value greater than printed p-value
```

```
##
##  KPSS Test for Level Stationarity
##
## data:  approvaldiff
## KPSS Level = 0.024109, Truncation lag parameter = 3, p-value = 0.1
```