Time Series: Issues with R

Andrius Buteikis, andrius.buteikis@mif.vu.lt
http://web.vu.lt/mif/a.buteikis/

R Issues

There are various (sometimes little, sometimes more serious) issues and function specifics which may do always seem to make sense. A number of them are mentioned at the homepage of Time Series, 4th Edition

Issue No. 1: Functions with the same names override one another

There are two quite popular packages are `plyr` and `dplyr`. First, look at what happens when you load ‘dplyr:

```
library(dplyr)
```

```
## Warning: package 'dplyr' was built under R version 3.5.2
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

As you can see, functions like `filter` and `lag` are overloaded with the new ones from the package.

Now assume that we want to load `plyr`:

```
library(plyr)
```

```
## ------------------------------------------------------------------------------
```

```
## You have loaded plyr after dplyr - this is likely to cause problems.
## If you need functions from both plyr and dplyr, please load plyr first, then dplyr:
## library(plyr); library(dplyr)
```

```
## ------------------------------------------------------------------------------
```

```
##
## Attaching package: 'plyr'
```

```
## The following objects are masked from 'package:dplyr':
##
##     arrange, count, desc, failwith, id, mutate, rename, summarise,
##     summarize
```

In addition to function overloading we now get an additional warning specifically for this package.

- If we want to have both `plyr` and `dplyr`, we need to **firstly load `plyr` and THEN `dplyr`**;
- We need to be careful in regards to function names - we might override functions from some packages with newer ones from different packages.
- You can load specific functions by referencing the package, like `plyr::some_function(...)` and `dplyr::another_function()` - this prevents errors, though for more advanced functions and/or expressions, additional functions from the package may need to be loaded, see import package.
- `plyr` and `dplyr` may have additional conflicts with some **Multivariate AR** (**VAR**) model packages.

Issue No. 2: `dplyr` and the `lag()` function

Assume that our data Y_t starts at time $t = 1$

```
x <- ts(1:5)
x
```

```
## Time Series:
## Start = 1
## End = 5
## Frequency = 1
## [1] 1 2 3 4 5
```

Assume that we want to automatically create Y_{t-1}.
By default the `lag()` function is a **forward** shift (i.e. into the future):

```
cbind(x, lag(x), lag(x, -1))
```

```
## Time Series:
## Start = 0
## End = 6
## Frequency = 1
##    x lag(x) lag(x, -1)
## 0 NA      1         NA
## 1  1      2         NA
## 2  2      3          1
## 3  3      4          2
## 4  4      5          3
## 5  5     NA          4
## 6 NA     NA          5
```

- `lag(x)` - in this case, instead of lagging the values back, we append the series at time $t = 0$. At time $t = 1$, we have $Y_1 = 1$ and we would assume that $Y_{t-1} = Y_0 = NA$. However, by default we would get the future value, i.e. $Y_{t+1} = Y_2 = 2$.
- `lag(x, -1)` - in this case at $t = 1$ we get an NA value - a **BACKWARD** shift - as we would expect.

Firstly, if we wanted to load dplyr and run the same function, we would get an **error**:

```
suppressPackageStartupMessages({# supress the package warnings if we are ABSOLUTELY sure about what we are doing
  library(dplyr)
})
lag(x)
```

```
## Error: `x` must be a vector, not a ts object, do you want `stats::lag()`?
```

```
# dplyr::lag(x) # alternatively, without loading
```

This does not work for `ts(...)` variables - it results in an error. If we use the value as a vector:

```
lag(c(x))
```

```
## [1] NA  1  2  3  4
```

We get what would be the equivalent of `stats::lag(x, -1)` - a **BACKWARD** shift

Issue No. 3 Overreliance on auto.arima

```r
suppressPackageStartupMessages({
  library(astsa)
  library(forecast)
})
tmp_mdl <- auto.arima(UnempRate)
```

```r
tmp_mdl
```

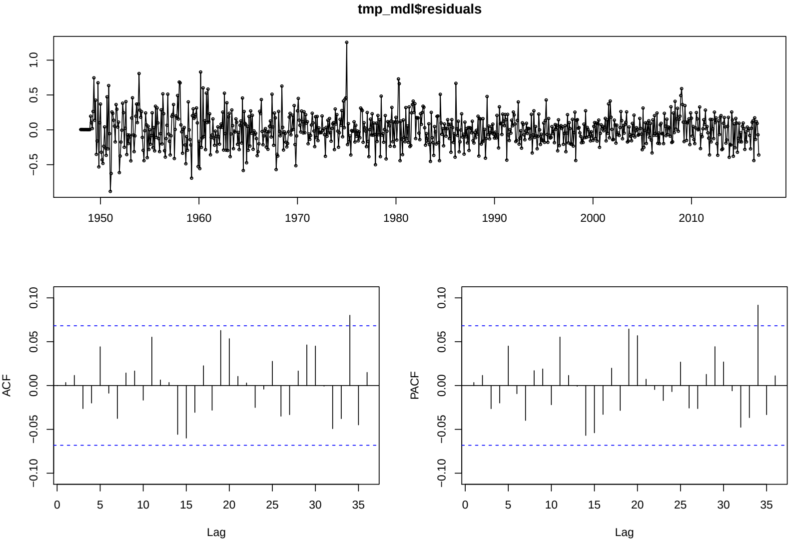
```
## Series: UnempRate
## ARIMA(3,0,1)(2,1,2)[12]
##
## Coefficients:
```

```
## Warning in sqrt(diag(x$var.coef)): NaNs produced
```

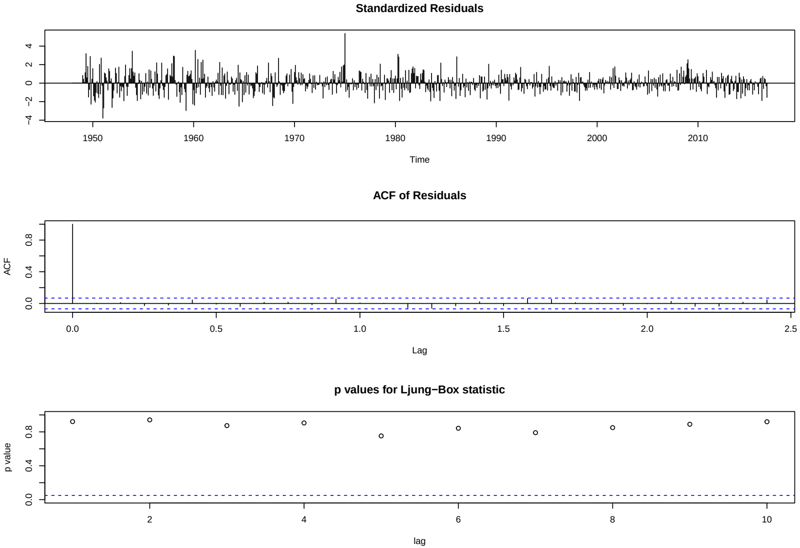
```
##          ar1      ar2      ar3      ma1      sar1    sar2     sma1     sma2
##       1.6852  -0.5763  -0.1204  -0.6106  -0.2849  0.0376  -0.4645  -0.2408
## s.e.  0.0379   0.0657   0.0394   0.0461   0.0464  0.0510   0.0392      NaN
##
## sigma^2 estimated as 0.05465:  log likelihood=25.52
## AIC=-33.03   AICc=-32.81   BIC=9.3
```

We even get a warning - the standard errors for the last coefficient are not calculated.

```r
forecast::tsdisplay(tmp_mdl$residuals)
```



```r
tsdiag(tmp_mdl)
```



We can manually specify a model:

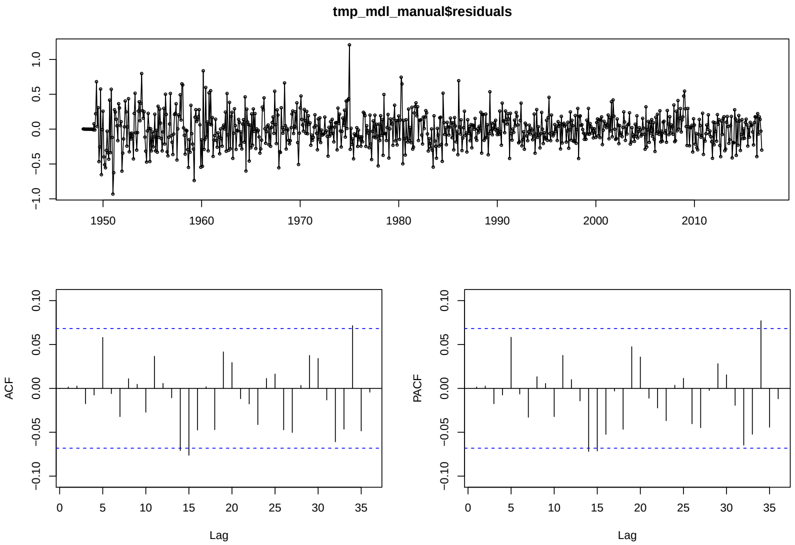
```
tmp_mdl_manual <- forecast::Arima(UnempRate, order = c(2, 1, 1), seasonal = c(0, 1, 1))
tmp_mdl_manual
```

```
## Series: UnempRate
## ARIMA(2,1,1)(0,1,1)[12]
##
## Coefficients:
##          ar1     ar2      ma1     sma1
##       0.5897  0.1342  -0.4831  -0.7676
## s.e.  0.1105  0.0465   0.1090   0.0254
##
## sigma^2 estimated as 0.05587:  log likelihood=15.69
## AIC=-21.38   AICc=-21.3   BIC=2.13
```

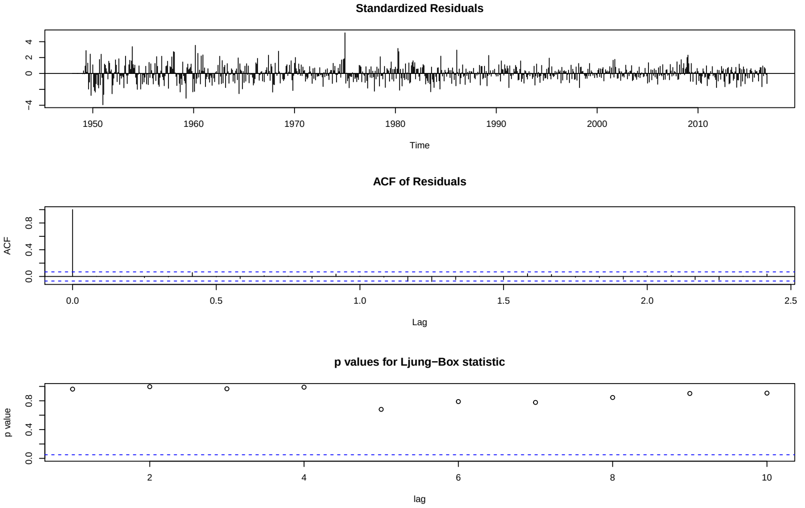
Note:

- The BIC is smaller for the manually specified model - the penalty for including more variables in the `auto.arima` model is harsher.
- The AIC is smaller for the `auto.arima` model.

```r
forecast::tsdisplay(tmp_mdl_manual$residuals)
```



```r
tsdiag(tmp_mdl_manual)
```



The residuals are not that much different in terms of their similarity to a *WN* process.

Issue # 4: `intercept` vs `mean`

Remember that:

- for $\mathrm{MA(q)}$ models: $Y_t = \alpha + \theta(L)\epsilon_t$, the mean is the intercept: $\mathbb{E}(Y_t) = \alpha$.
- for stationary $\mathrm{AR(p)}$ or $\mathrm{ARMA(p, q)}$ models $\phi(L)Y_t = \alpha + \theta(L)\epsilon_t$, the mean is $\mu = \alpha/(1 - \phi_1 - ... - \phi_p)$, or $\alpha = \mu \cdot (1 - \phi_1 - ... - \phi_p)$.

In other words when there is an AR term in the model, the intercept is **NOT** the mean.

```
set.seed(1)
x = arima.sim(list(order = c(1,0,0), ar= 0.4), n = 1000) + 10
```

The true mean of the process is 50, which translates to $\alpha = 10 \cdot (1 - 0.4) = 6$.

```
mean(x)
```

```
## [1] 9.978789
```

Verify this via manual simulation:

```
set.seed(1)
epsilon <- rnorm(1000, mean = 0, sd = 1)
x    <- NULL
x[1] <- 6 + epsilon[1]
for(j in 2:length(epsilon)){
  x[j] <- 6 + 0.4 * x[j - 1] + epsilon[j]
}
```

```
mean(x)
```

```
## [1] 9.974864
```

If we estimate using `stats::arima`:

```
mdl_1 <- arima(x, order = c(1, 0, 0))
mdl_1
```

```
##
## Call:
## arima(x = x, order = c(1, 0, 0))
##
## Coefficients:
##          ar1  intercept
##       0.3541     9.9717
## s.e.  0.0299     0.0510
##
## sigma^2 estimated as 1.085:  log likelihood = -1459.74,  aic = 2925.48
```

The naming is **incorrect** - the `intercept` is actually the `mean`.

On the other hand, if we use `forecast::Arima`:

```
mdl_2 <- forecast::Arima(x, order = c(1, 0, 0))
mdl_2
```

```
## Series: x
## ARIMA(1,0,0) with non-zero mean
##
## Coefficients:
##          ar1    mean
##       0.3541  9.9717
## s.e.  0.0299  0.0510
##
## sigma^2 estimated as 1.087:  log likelihood=-1459.74
## AIC=2925.48   AICc=2925.51   BIC=2940.21
```

The naming appears correct ... unless we extract the coefficients:

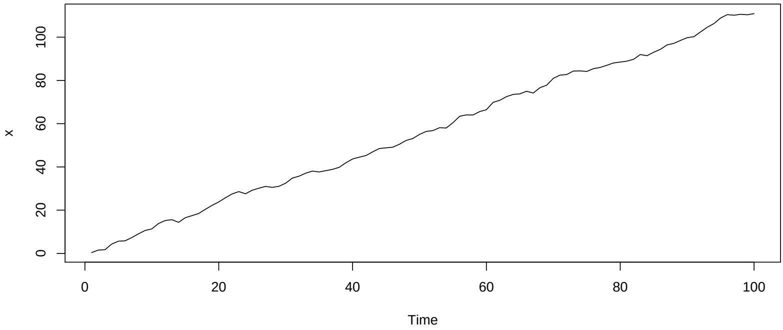
```
coef(mdl_2)
```

```
##       ar1 intercept
## 0.3541427 9.9716752
```

Then the naming is again **incorrect**.

Issue # 5.1: `stats::arima` vs `forecast::Arima` for $Y_t \sim I(d)$

```
set.seed(1)
v = rnorm(100, mean = 1, sd = 1)
x = ts(cumsum(v))
plot.ts(x)
```



The `stats` package is one of the default packages, which come with the installation of `R`.

```
mdl_stats <- arima(x, order = c(1, 1, 0), include.mean = TRUE)
mdl_stats
```

```
##
## Call:
## arima(x = x, order = c(1, 1, 0), include.mean = TRUE)
##
## Coefficients:
##          ar1
##       0.6031
## s.e.  0.0793
##
## sigma^2 estimated as 1.294:  log likelihood = -153.46,  aic = 310.91
```

```
mdl_forcast <- forecast::Arima(x, order = c(1, 1, 0), include.drift = TRUE)
mdl_forcast
```

```
## Series: x
## ARIMA(1,1,0) with drift
##
## Coefficients:
##           ar1   drift
##       -0.0031  1.1163
## s.e.   0.1002  0.0897
##
## sigma^2 estimated as 0.8178:  log likelihood=-129.51
## AIC=265.01   AICc=265.26   BIC=272.8
```

Note that when we are differencing the series:

- `stats::arima()` fits $\Delta Y_t = \phi \Delta Y_{t-1} + \epsilon_t$ (no constant);
- `forecast::Arima()` fits $\Delta Y_t = \alpha + \phi \Delta Y_{t-1} + \epsilon_t$ (constant).

Consequently, if we want to fit a $I(d)$ series with a drift **using** `stats::arima()`, there are two ways to go about this:

- fit the differenced series, `diff(x)`, with a constant:

```
arima(diff(x), order = c(1, 0, 0), include.mean = TRUE)
```

```
##
## Call:
## arima(x = diff(x), order = c(1, 0, 0), include.mean = TRUE)
##
## Coefficients:
##           ar1  intercept
##       -0.0031     1.1163
## s.e.   0.1002     0.0897
##
## sigma^2 estimated as 0.8012:  log likelihood = -129.51,  aic = 265.01
```

In this case, for the AR model, the `intercept` is actually the **mean** of `diff(x)`.

- specify the constant as an exogeneous variable:

```
arima(x, order = c(1, 1, 0), xreg = 1:length(x))
```

```
##
## Call:
## arima(x = x, order = c(1, 1, 0), xreg = 1:length(x))
##
## Coefficients:
##           ar1  1:length(x)
##       -0.0031       1.1163
## s.e.   0.1002       0.0897
##
## sigma^2 estimated as 0.8012:  log likelihood = -129.51,  aic = 265.01
```

In this case, the exogeneous variable is the **intercept** of `diff(x)`.

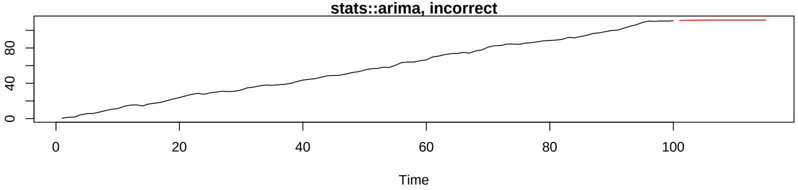
- alternatively, use `forecast::Arima()` (where it is named `drift`).

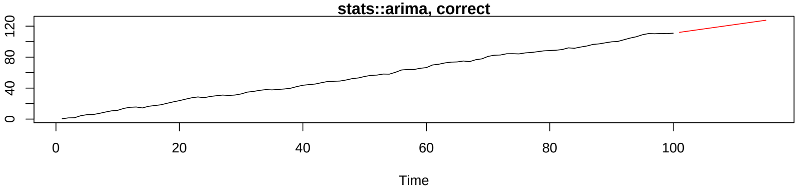
Issue # 5.2: Forecasting with $Y_t \sim I(d)$

Assume that we want to forecast via the **default** `predict()` function and we *ignore* the fact that we do not include a drift in our model:

```
mdl_stats_wrong   <- arima(x, order = c(1, 1, 0), include.mean = TRUE)
mdl_stats_correct <- arima(x, order = c(1, 1, 0), xreg = 1:length(x))
#
forc_1 <- predict(mdl_stats_wrong, 15)
forc_2 <- predict(mdl_stats_correct, 15,
                  newxreg = (length(x) + 1):(length(x) + 15))
```

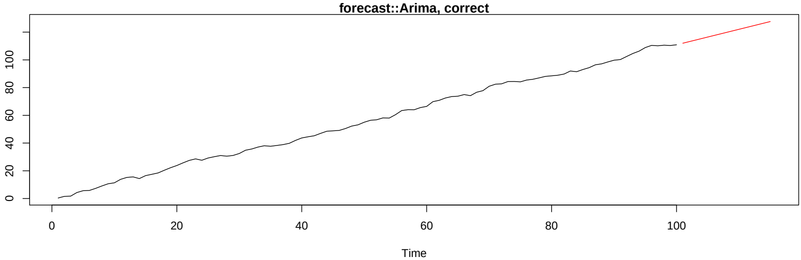
```
par(mfrow = c(2, 1), mai = c(1, 0.5, 0.2, 0.2))
ts.plot(x, forc_1$pred, col = 1:2, main = "stats::arima, incorrect")
ts.plot(x, forc_2$pred, col = 1:2, main = "stats::arima, correct")
```

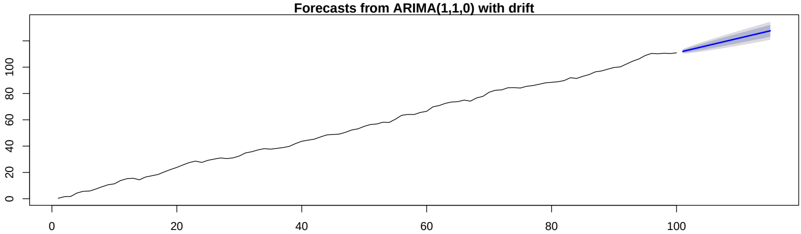




```r
mdl_forcast <- forecast::Arima(x, order = c(1, 1, 0), include.drift = TRUE)
#
forc_3 <- forecast(mdl_forcast, 15)
```

```r
par(mfrow = c(2, 1), mai = c(1, 0.5, 0.2, 0.2))
ts.plot(x, forc_3$mean, col = 1:2, main = "forecast::Arima, correct")
plot(forc_3, cmain = "stats::arima, correct")
```





Issue # 6: Tests for Residuals autocorrelation of an $\mathrm{ARMA(p,q)}$ model

The null hypothesis:

$$H_0 : \rho(1) = ... = \rho(k) = 0$$
$$H_1 : \exists j : \rho(j) \neq 0$$

The Ljung-Box test:

$$Q(k) = T(T+2) \sum_{m=1}^{k} \frac{\widehat{\rho}^2(m)}{T-m}$$

Box-Pierce test:

$$Q_{\mathrm{BP}} = T \sum_{m=1}^{k} \widehat{\rho}^2(m)$$

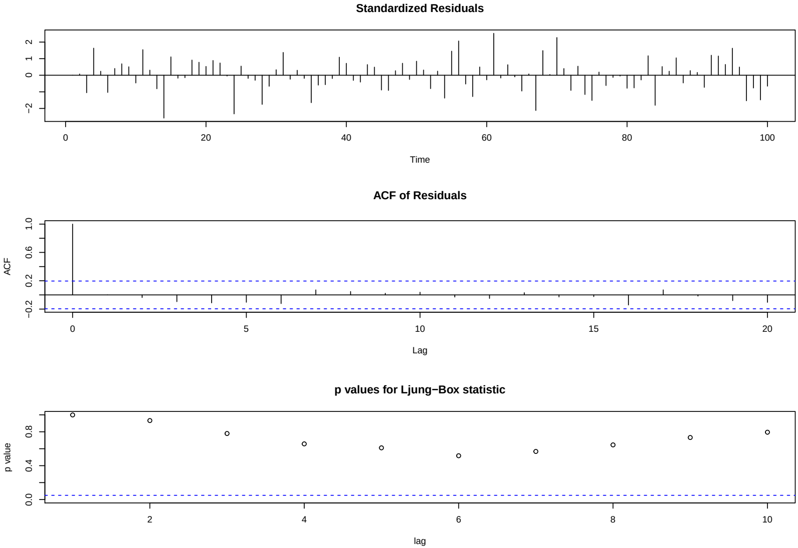
- If we are testing whether Y_t exhibits significant autocorrelation, then we reject the null hypothesis if

$$Q(k) > \chi^2_{1-\alpha,\ k}$$

  where k are the degrees of freedom.
- If we are testing the residuals of an estimated $\mathrm{ARIMA}(\mathrm{p}, \mathrm{q})$ model (**without constant**), the degrees of freedom need to be adjusted to reflect the parameter estimation. In such cases the degrees of freedom should be set to $k - p - q > 0$.

The same applies to Q_{BP}

```r
tsdiag(mdl_forcast)
```



```
out <- data.frame(INCORRECT = rep(NA, 7), CORRECT = rep(NA, 7))
for(i in 3:10){
  out[i - 2, 1] <- Box.test(mdl_forcast$residuals, lag = i,
                            type = "Ljung-Box")$p.value
  out[i - 2, 2] <- Box.test(mdl_forcast$residuals, lag = i,
                            fitdf = length(coef(mdl_forcast)),
}
rownames(out) <- paste0("Lag_", 3:10)
print(out)
```

```
##        INCORRECT   CORRECT
## Lag_3  0.7796620 0.2966303
## Lag_4  0.6572870 0.2967712
## Lag_5  0.6105057 0.3098529
## Lag_6  0.5171249 0.2663823
## Lag_7  0.5677989 0.3300712
## Lag_8  0.6452965 0.4212575
## Lag_9  0.7323974 0.5310294
## Lag_10 0.7948932 0.6205997
```