Modelos ARIMA no R

Regis Augusto Ely

23 de maio de 2014 (v. 1.1)

Introdução

Vamos ver alguns exemplos de implementação dos modelos ARIMA que estudamos em aula. A lógica é reconstruir a dinâmica por trás dos modelos e entender melhor os resultados teóricos que vimos durante o curso de séries temporais no PPGOM/UFPel. Para isso vamos simular e estimar os modelos na "mão", sem utilizar funções prontas do R. Já para a realização do diagnóstico e previsão iremos utilizar algumas funções que nos facilitam o cálculo. Os códigos são feitos para rodar em qualquer computador com R instalado, basta copiar e colar todos os blocos de código em um arquivo .R e então rodar tudo.

Simulação

A primeira etapa será a simulação de modelos ARMA no R. Lembre que não vale utilizar funções como a arima.sim, pois devemos reconstruir a dinâmica das séries. Para isso, primeiro devemos definir os parâmetros desejados. Ao rodar o código você pode alterar ou incluir os coeficientes da maneira que quiseres.

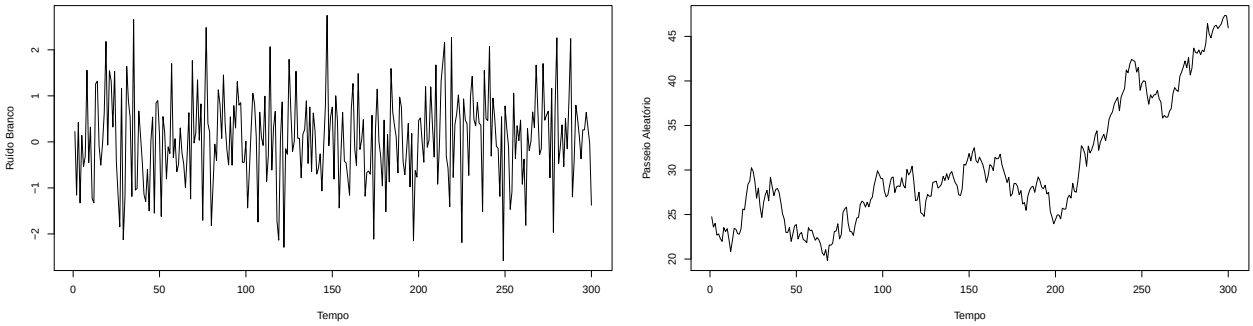
```
# Definição dos parâmetros
n <- 400 # Total de observações simuladas
nx <- 100 # Total de observações descartadas
c <- 0.3 # Constante do AR
phi <- c(0.5,-0.8) # Vetor de parâmetros do AR
mu <- 0.7 # Média do MA
theta <- c(0.8,0.3,-0.2) # Vetor de parâmetros do MA
ca <- 0.2 # Constante do ARMA
phia <- c(0.6) # Vetor de parâmetros AR do ARMA
thetaa <- c(0.3) # Vetor de parâmetros MA do ARMA
```

Devemos criar os vetores em que armazenaremos os dados, além do ruído branco que será o componente aletório dos modelos. As ordens p e q dos processos também são calculadas de acordo com o comprimento dos vetores dos coeficientes.

```
# Criação dos vetores e cálculo dos valores de p, d e q.
set.seed(12345)
rb <- rnorm(n) # Ruído branco
yar <- numeric(n) # Vetor do AR
yma <- numeric(n) # Vetor do MA
yarma <- numeric(n) # Vetor do ARMA
p <- length(phi) # Ordem do AR
q <- length(theta) # Ordem do MA
pa <- length(phia) # Ordem AR do ARMA
qa <- length(thetaa) # Ordem MA do ARMA
```

Primeiro simularemos um ruído branco e um passeio aleatório. As séries tem n observações, sendo que as nx primeiras são descartadas. Os gráficos são plotados no final do comando.

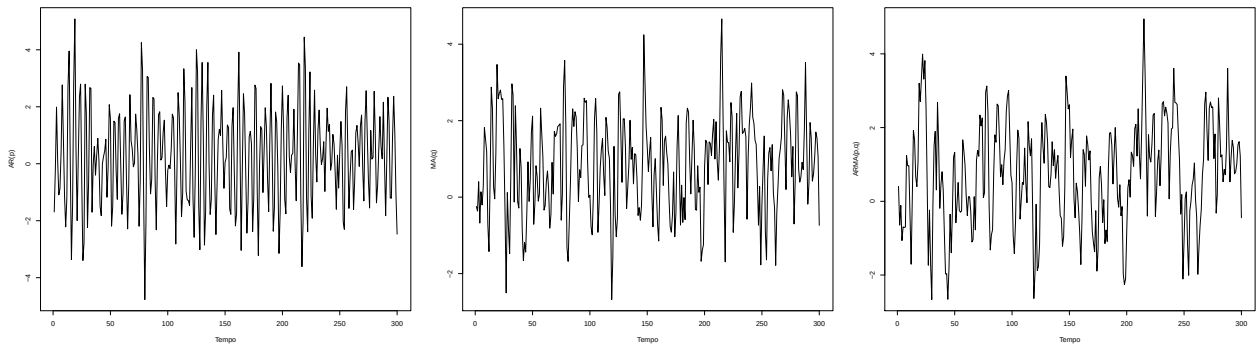
```r
# Simulação do ruído branco e do passeio aleatório
ruido <- rb[-1:-nx] # Ruído Branco
rw <- cumsum(rb)
passeio <- rw[-1:-nx] # Passeio aleatório
par(mfrow=c(1,2))
ts.plot(ruido, gpars=list(xlab="Tempo", ylab="Ruído Branco"))
ts.plot(passeio, gpars=list(xlab="Tempo", ylab="Passeio Aleatório"))
```



Agora vamos simular os processos AR, MA e ARMA. As operações serão feitas elemento por elemento (poderíamos utilizar operações com vetores também). Note que fazemos dois loops em cada série, o primeiro para simular os n elementos e o segundo para calcular as p ou q defasagens de cada valor de y. No final plotamos os gráficos destes processos.

```r
### Simulação do processo AR(p)
for(i in (p+1):n) {
        psum <- 0
    for(j in 1:p){
        psum <- psum + phi[j]*yar[i-j]
    }
    yar[i] <- psum + c + rb[i]
}
ar <- yar[-1:-nx]

### Simulação do processo MA(q)
for(i in (q+1):n) {
        psum <- 0
    for(k in 1:q){
        psum <- psum + theta[k]*rb[i-k]
    }
    yma[i] <- psum + mu + rb[i]
}
ma <- yma[-1:-nx]

### Simulação do processo ARMA(p,q)
for(i in (pa+1):n) {
    psum <- 0
    for(j in 1:pa){
        psum <- psum + phia[j]*yarma[i-j]
    }
    for(k in 1:qa){
        psum <- psum + thetaa[k]*rb[i-k]
    }
    yarma[i] <- psum + ca + rb[i]
```

```
}
arma <- yarma[-1:-nx]

# Plotando os resultados
par(mfrow=c(1,3))
ts.plot(ar, gpars=list(xlab="Tempo", ylab="AR(p)"))
ts.plot(ma, gpars=list(xlab="Tempo", ylab="MA(q)"))
ts.plot(arma, gpars=list(xlab="Tempo", ylab="ARMA(p,q)"))
```



Note que estas séries parecem apresentar uma persistência maior dos choques do que a série do ruído branco. É esta persistência que podemos modelar.

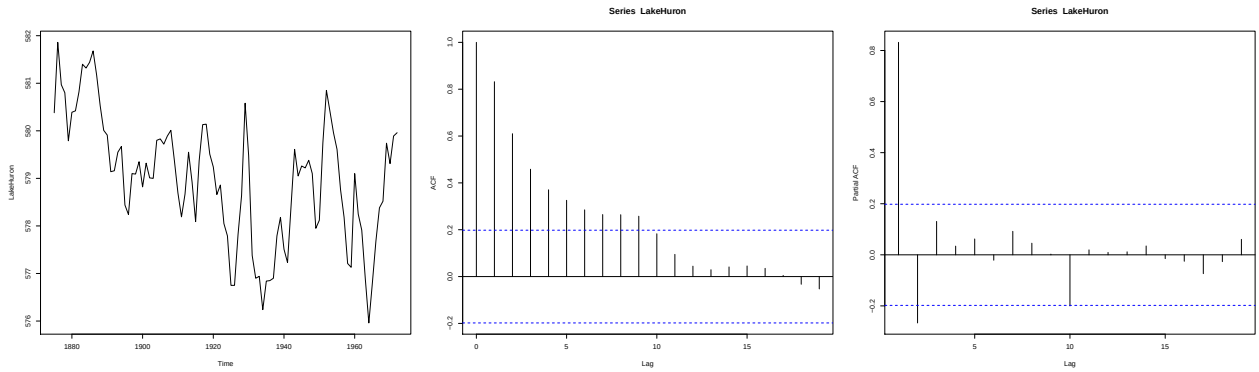
Se quiserem podem gravar os dados gerados em um arquivo .csv.

```
# Gravar dados no arquivo "simulations.csv"
export <- cbind(ruido, passeio, ar, ma, arma)
write.csv(export, file="simulations.csv")
```

Identificação

Para realizarmos o procedimento de identificação de Box-Jenkins vamos carregar alguns dados reais. A primeira série será composta pelos níveis do lago Huron durante os anos de 1875-1972. Lembrem que ao estudar a dinâmica destes processo descobrimos como devem se comportar as funções de autocorrelação e autocorrelação parcial.

```
data(LakeHuron)
par(mfrow=c(1,3))
plot(LakeHuron)
acf(LakeHuron)
pacf(LakeHuron)
```

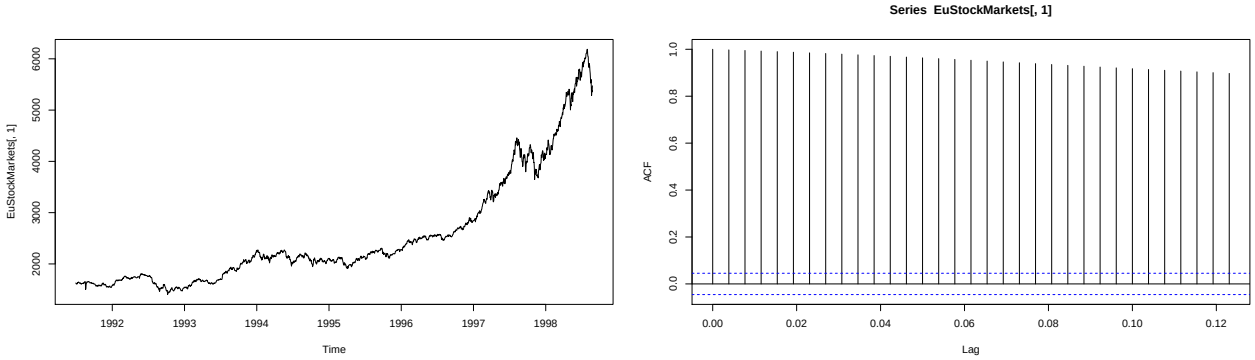


Notem que a função acf do R plota o lag zero da autocorrelação (que será sempre um). Isso é um incômodo, pois podemos nos confundir na identificação, além de ser difícil de visualizar os outros lags. A dica (dada pelo Cláudio Shikida) é utilizar a função Acf (com A maiúsculo), que faz parte do pacote forecast. Há ainda a função acf2 do pacote astsa, que plota ambas as fac e facp juntas.

Repare que a fac da nossa série parece decair exponencialmente até se tornar estatisticamente insignificante. Já a facp tem dois picos no lag 1 e 2. Vimos que esse comportamento é típico de um AR(2).

A segunda série que iremos analisar é do valor diário de fechamento do índice de ações DAX durante 1991 a 1998, que é composto pelas 30 maiores companhias alemãs da Bolsa de valores de Frankfurt.

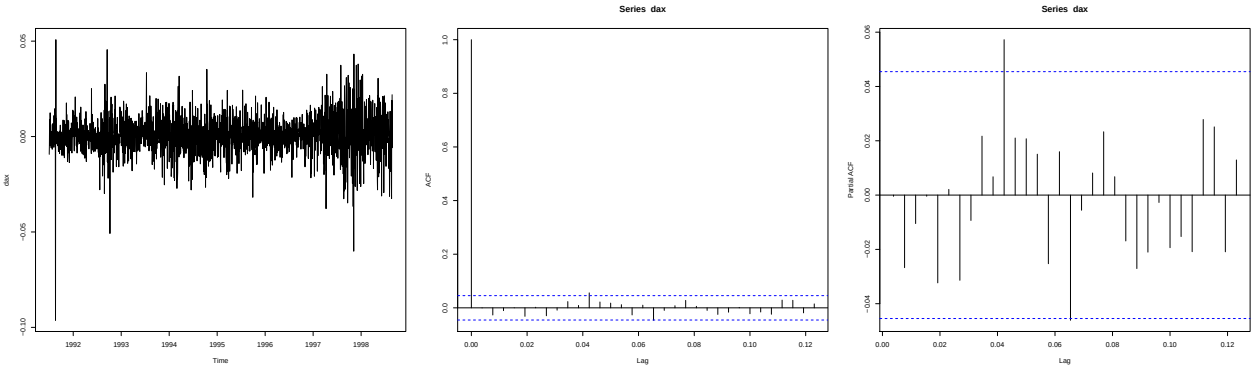
```
data(EuStockMarkets)
par(mfrow=c(1,2))
plot(EuStockMarkets[,1])
acf(EuStockMarkets[,1])
```



Observe que este índice tem um comportamento bem errático, que lembra o passeio aleatório que simulamos anteriormente. A fac não decai exponencialmente, sinalizando que a série não parece ser estacionária.

Vamos lidar com esta não estacionariedade tirando a primeira diferença. Lembrem que a primeira diferença remove tendências estocásticas ou lineares de uma série (embora possa adicionar correlação na mesma). Antes de diferenciarmos, vamos aplicar o log, pois então ganhamos a interpretação de retorno continuamente composto, além de estabilizar um pouco a variância e transformar a clara tendência exponencial em linear.

```
dax <- diff(log(EuStockMarkets[,1]))
par(mfrow=c(1,3))
plot(dax)
acf(dax)
pacf(dax)
```

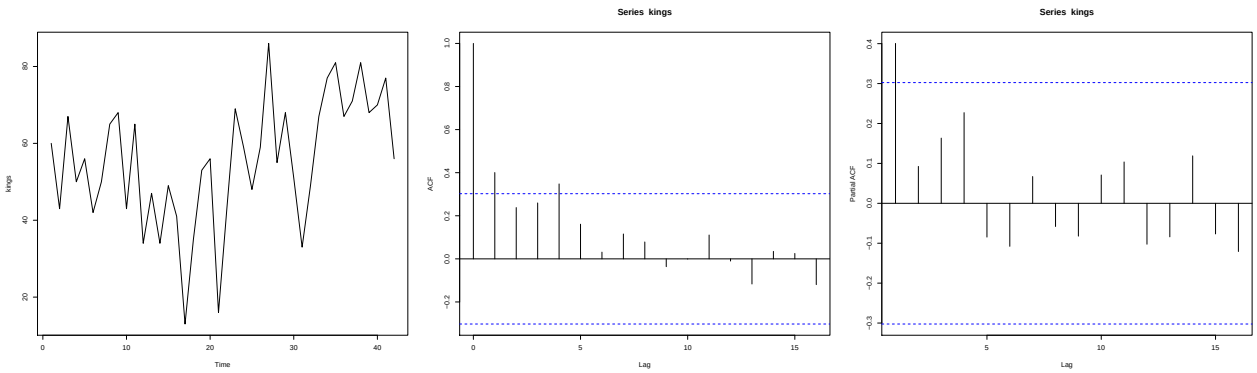


Ops, nossa série parece um ruído branco agora. Pelo jeito não há nenhuma correlação entre retornos passados e presentes do índice DAX. Bem eficiente este mercado alemão hein! Parece que nossos modelos ARIMA não

são suficientes para extrair algum dinheiro dos alemães. Mais tarde veremos alguns modelos não lineares que nos dizem algo sobre esses agrupamentos de volatilidade que observamos no gráfico do retorno.

Vamos identificar mais uma série. Desta vez ela será composta pela idade da morte de 42 reis sucessivos da Inglaterra. Obteremos estes dados pela internet (uma conexão é necessária ao rodar o comando). Os dados estão no site do Rob J Hyndman, que é professor de estatística da Monash University na Austrália. Ele também é autor do pacote forecast no R, que utilizaremos mais a frente. Em seu site há muito material sobre séries e sobre o R, principalmente no tópico de previsão, sua especialidade.

```
kings <- ts(scan("http://robjhyndman.com/tsdldata/misc/kings.dat",skip=3))
par(mfrow=c(1,3))
plot(kings)
acf(kings)
pacf(kings)
```



Repare que apesar de parecer estar tudo certo com a fac e facp, a média da série claramente mudou longo do tempo. Isso é um indicativo de não estacionariedade. Aqui iremos avançar um pouco no conteúdo e introduzir o teste de Dickey-Fuller aumentado para verificarmos essa nossa hipótese de não estacionariedade. Para isso precisaremos instalar um pacote novo no R, chamado tseries, que contém a função adf.test.

```
if(!require(tseries)) {
        install.packages("tseries", repos = "http://cran.rstudio.com/")
}
library(tseries)
adf.test(kings)
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  kings
## Dickey-Fuller = -2.1132, Lag order = 3, p-value = 0.529
## alternative hypothesis: stationary
```

O p-valor é alto, logo aceitamos a hipótese de uma raíz unitária, o que nos diz que a série não é estacionária (veremos mais a frente no curso como funciona o teste ADF). Vamos então tirar a primeira diferença para remover essa não estacionariedade.

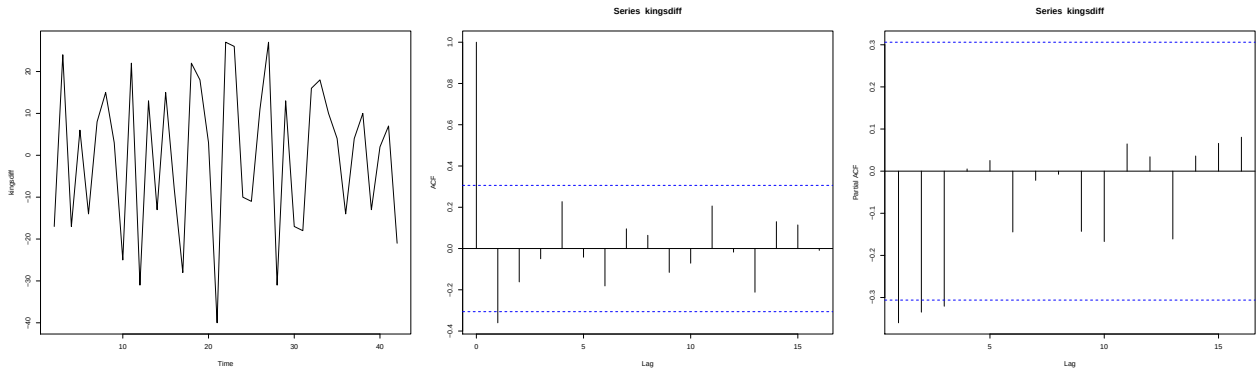
```
kingsdiff <- diff(kings, 1)
adf.test(kingsdiff)
```

```
##
```

```
##  Augmented Dickey-Fuller Test
##
## data:  kingsdiff
## Dickey-Fuller = -4.0754, Lag order = 3, p-value = 0.01654
## alternative hypothesis: stationary
```

Agora sim! Rejeitamos a hipótese nula e parece que a série é estacionária. Vamos olhar para os gráficos da fac e facp.

```
par(mfrow=c(1,3))
plot(kingsdiff)
acf(kingsdiff)
pacf(kingsdiff)
```



Esta série está um pouco complicada. Temos algumas possibilidades diferentes. Poderia ser um AR(3) ou um MA(1). Vamos com o MA(1) por ser mais parcimônio (menos coeficientes).

Agora que aprendemos a simular os modelos ARIMA e identificarmos séries de tempo univariadas vamos estimar os modelos.

Estimação

Lembrem que o objetivo é entender a estrutura teórica por trás dos modelos. Para isso vamos construir a estimativa de máxima verossimilhança conforme vimos em aula. Primeiro temos que criar uma função para calcular a log-verossimilhança e então carregá-la dentro da função optim do R, que utiliza métodos de otimização numérica para achar os parâmetros que minimizam o valor negativo da função. Podemos então comparar os nossos resultados com a funçao arima do R, que estima tudo isso automaticamente.

Vamos lá! Primeiro iremos estimar o MA(1) que identificamos na série da idade da morte dos reis da Inglaterra. A primeira etapa é carregar uma função que nos dê a log-verossimilhança de um MA(1) para um conjunto de dados e parâmetros. Utilizaremos a função de log-verossimilhança condicional, conforme vimos em aula:

$$L(\theta; Y_t) = -\frac{T}{2}\ln(2\pi) - \frac{T}{2}\ln(\sigma^2) - \sum_{t=1}^{T}\frac{\varepsilon_t^2}{2\sigma^2}$$

E lembrem que $\varepsilon_t = y_t - \mu - \theta\varepsilon_{t-1}$ no caso de um MA(1). Iremos também supor que $\varepsilon_0 = 0$.

```
loglik.ma <- function(param, data) {
        T <- length(data)
        res <- numeric(T)
        res[1] <- data[1] - param[1]
```

```
        c1 <- -(T/2)*log(2*pi)
        for(i in 2:T) {
                res[i] <- data[i] - param[1] - param[2]*res[i-1]
        }
        c2 <- -(T/2)*log(param[3])
        c3 <- -sum((res^2)/(2*param[3]))
        L <- c1 + c2 + c3
        -L
}
```

Agora vamos chutar parâmetros iniciais e então mandar a função optim encontrar os parâmetros que minimizam essa funçao acima. Note que os nossos parâmetros são (μ, θ, σ^2). A função irá retornar alguns warnings pelo fato de tentar valores negativos para σ^2, mas por enquanto isso não é problema para nós (a opção method = "L-BFGS-B" lida com otimizações condicionadas). Também podemos calcular os erros dos coeficientes obtendo os elementos da diagonal da raíz quadrada da matriz inversa da hessiana. Comparamos os resultados com a função arima do R.

```
# Dando um chute inicial e estimando o MA(1)
param_ma <- c(0.5, 0.1, 10)
estma <- optim(par = param_ma, fn = loglik.ma, method = "BFGS", hessian = TRUE,
               data = kingsdiff)

# Calculando o erro padrão dos coeficiente
estma_se <- sqrt(diag(solve(estma$hessian)))

# Estimando pela função arima e comparando os resultados numa matriz
estma_arima <- arima(kingsdiff, order = c(0,0,1))
result_ma <- matrix(c(estma_arima$coef[2:1], estma_arima$sigma2,
                      sqrt(diag(estma_arima$var.coef))[2:1], NA), 2, 3, byrow=TRUE)
ma_comp <- rbind(estma$par, result_ma[1,], estma_se, result_ma[2,])
colnames(ma_comp) <- c("mu", "theta", "sigma2")
rownames(ma_comp) <- c("optim.coef", "arima.coef", "optim.se", "arima.se")
ma_comp
```

```
##                   mu      theta    sigma2
## optim.coef 0.3291954 -0.7417389 209.94612
## arima.coef 0.3881635 -0.7462804 228.24163
## optim.se   0.6198672  0.1105213  42.35326
## arima.se   0.6521644  0.1278466        NA
```

Os resultados são parecidos não? Alguma diferença existe, pois o método da função arima utiliza como chutes iniciais a estimativa de minímos quadráticos (que é um chute inicial melhor do que o nosso). Além disso a implementação da função de log-verossimilhança dessa função é muito mais eficiente que a nossa (feita em C). Note que a função arima não calcula a variância do erro, por isso o NA no último elemento.

Agora vamos estimar um modelo AR(1) utilizando a expressão da log-verossimilhança condicional que vimos na aula:

$$L(\phi, Y_t) = -\left[\frac{T-1}{2}\right]\ln(2\pi) - \left[\frac{T-1}{2}\right]\ln(\sigma^2) - \sum_{t=2}^{T}\frac{\varepsilon_t^2}{2\sigma^2}$$

Lembrando que nesse caso $\varepsilon_t = y_t - c - \phi y_{t-1}$. Seguiremos as mesmas etapas, primeiro carregando a função de log-verossimilhança. Note que teremos de novo 3 parâmetros (c, ϕ, σ^2), e iremos supor que $y_0 = 0$.

```
loglik.ar <- function(param, data) {
        T <- length(data)
        res <- numeric(T)
        res[1] <- data[1] - param[1]
        c1 <- -((T-1)/2)*log(2*pi)
        for(i in 2:T) {
                res[i] <- data[i] - param[1] - param[2]*data[i-1]
        }
        c2 <- -((T-1)/2)*log(param[3])
        c3 <- -sum((res^2)/(2*param[3]))
        L <- c1 + c2 + c3
        -L
}
```

Para realizar a estimação, iremos gerar uma série AR(1) com valores $c = 0.7$, $\phi = 0.8$ e $\sigma^2 = 1$. Então comparamos os resultados com a função arima do R.

```
#Criando um AR(1)
set.seed(12345)
yt <- numeric(1000)
eps <- rnorm(1000)
c <- 0.7
phi <- 0.8
for(i in 2:1000){
        yt[i] <- c + phi*yt[i-1] + eps[i]
}

# Dando um chute inicial e estimando o AR(1)
param_ar <- c(1, 0.5, 0.5)
estar <- optim(par = param_ar, fn = loglik.ar, method = "BFGS", hessian = TRUE, data = yt)

# Calculando o erro padrão dos coeficiente
estar_se <- sqrt(diag(solve(estar$hessian)))

# Estimando pela função arima e comparando os resultados numa matriz
estar_arima <- arima(yt, order = c(1,0,0))
result_ar <- matrix(c(estar_arima$coef[2:1], estar_arima$sigma2,
                      sqrt(diag(estar_arima$var.coef))[2:1], NA), 2, 3, byrow=TRUE)
ar_comp <- rbind(estar$par, result_ar[1,], estar_se, result_ar[2,])
colnames(ar_comp) <- c("c", "phi", "sigma2")
rownames(ar_comp) <- c("optim.coef", "arima.coef", "optim.se", "arima.se")
ar_comp
```

```
##                     c        phi    sigma2
## optim.coef 0.70171554 0.81163352 0.9973541
## arima.coef 3.69315335 0.81375535 1.0005474
## optim.se   0.07517156 0.01837277 0.0446249
## arima.se   0.16912177 0.01842791        NA
```

Os resultados são semelhantes, com exceção do coeficiente c. Isso porque a função arima calcula o intercepto como sendo μ e não c. Mas lembre que $\mu = \frac{c}{1-\phi}$. Assim, podemos calcular o c do modelo estimado pela função arima e comparar com o nosso.

```r
ar_comp[2,1]*(1-ar_comp[2,2])
```

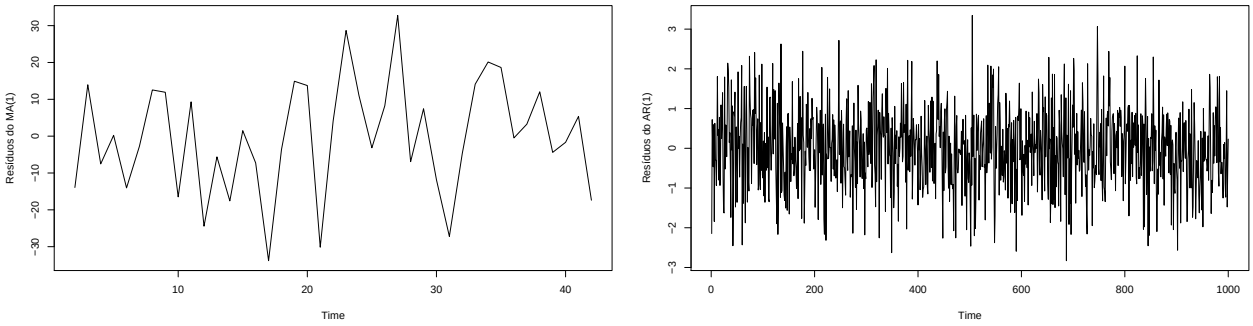
```
## [1] 0.6878301
```

Agora sim, o resultado está próximo. Está na hora de vermos se a especificação do nosso modelo está boa, ou seja, se estamos conseguindo remover completamente a correlação dos resíduos.

Diagnóstico

Para verificarmos se o modelo está bem especificado, devemos olhar para os resíduos da estimação.

```r
par(mfrow=c(1,2))
plot(estma_arima$residuals, ylab = "Resíduos do MA(1)")
plot(estar_arima$residuals, ylab = "Resíduos do AR(1)")
```



Parecem estacionários, mas para sabermos se eliminamos completamente a autocorrelação residual devemos fazer o teste de Ljung-Box. Vimos em aula que este teste tem uma distribuição assintótica qui-quadrada com T graus de liberdade, mas quando o aplicamos aos resíduos de um modelo ARIMA, perdemos $p + q$ graus de liberdade. Esse ajuste é feito através da opção fitdf da função Box.test. Calculamos o teste de Ljung-Box para 15 defasagens, especificando fitdf=1 pois trata-se de um MA(1) e um AR(1).

```r
Box.test(estma_arima$residuals, lag = 15, type = "Ljung-Box", fitdf = 1)
```

```
##
##  Box-Ljung test
##
## data:  estma_arima$residuals
## X-squared = 10.142, df = 14, p-value = 0.7518
```

```r
Box.test(estar_arima$residuals, lag = 15, type = "Ljung-Box", fitdf = 1)
```

```
##
##  Box-Ljung test
##
## data:  estar_arima$residuals
## X-squared = 15.599, df = 14, p-value = 0.3385
```

Tudo certo, aceitamos a hipótese nula de que os resíduos não são autocorrelacionados. Agora temos um bom modelo linear para descrever nossos dados. Outras opções de diagnóstico que vimos em aula são os testes de Akaike (AIC), Schwart (BIC) e Hannan-Quin (HQC). Temos no R algumas funções que escolhem automaticamente a defasagem dos modelos ARIMA através destes critérios (de uma olhada na função auto.arima do pacote forecast). Vamos agora realizar alguma previsão para estas variáveis.

Previsão

O melhor pacote de previsão do R é o forecast. Irei carregar ele e então realizar uma previsão de 10 horizontes para o MA e o AR. Este pacote utiliza uma função interna para a estimação chamada Arima. Antes de fazer a previsão vamos estimar os modelos de novo através dessa função.

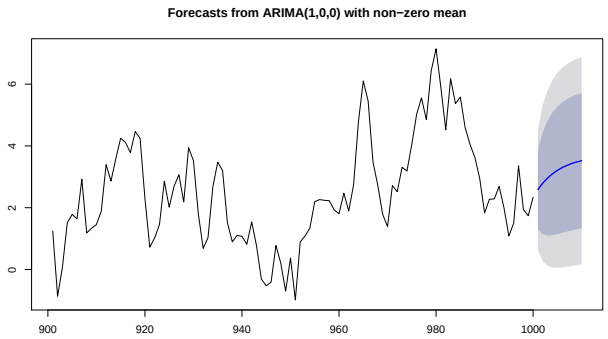
```
if(!require(forecast)) {
        install.packages("forecast", repos = "http://cran.rstudio.com/")
}
library(forecast)
Ma <- Arima(kingsdiff, order = c(0, 0, 1))
Ar <- Arima(yt, order = c(1, 0, 0))

prev_ma <- forecast.Arima(Ma, h = 10); prev_ma
```

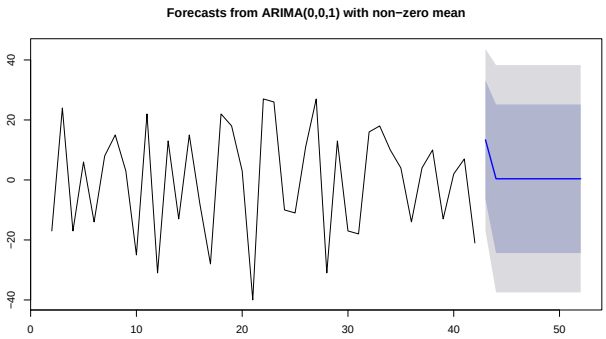
```
##    Point Forecast      Lo 80    Hi 80     Lo 95    Hi 95
## 43     13.3629503  -6.488541 33.21444 -16.99729 43.72319
## 44      0.3881635 -24.381967 25.15829 -37.49448 38.27081
## 45      0.3881635 -24.381967 25.15829 -37.49448 38.27081
## 46      0.3881635 -24.381967 25.15829 -37.49448 38.27081
## 47      0.3881635 -24.381967 25.15829 -37.49448 38.27081
## 48      0.3881635 -24.381967 25.15829 -37.49448 38.27081
## 49      0.3881635 -24.381967 25.15829 -37.49448 38.27081
## 50      0.3881635 -24.381967 25.15829 -37.49448 38.27081
## 51      0.3881635 -24.381967 25.15829 -37.49448 38.27081
## 52      0.3881635 -24.381967 25.15829 -37.49448 38.27081
```

```
prev_ar <- forecast.Arima(Ar, h = 10); prev_ar
```

```
##      Point Forecast    Lo 80    Hi 80      Lo 95    Hi 95
## 1001       2.589164 1.305978 3.872350 0.62670016 4.551628
## 1002       2.794776 1.140412 4.449140 0.26464512 5.324907
## 1003       2.962094 1.102269 4.821919 0.11773741 5.806451
## 1004       3.098250 1.114044 5.082456 0.06366832 6.132832
## 1005       3.209048 1.146600 5.271495 0.05480669 6.363288
## 1006       3.299210 1.186546 5.411874 0.06816871 6.530251
## 1007       3.372580 1.227309 5.517850 0.09167101 6.653488
## 1008       3.432285 1.265692 5.598878 0.11876696 6.745802
## 1009       3.480870 1.300273 5.661468 0.14593408 6.815806
## 1010       3.520407 1.330585 5.710229 0.17136296 6.869450
```

```
par(mfrow=c(1,2))
plot(prev_ar, include = 100)
plot(prev_ma, include = 100)
```





Note que no MA(1), a previsão converge para a média após o primeiro período, enquanto que no AR, a previsão irá lentamente convergir para a média. Exatamente como vimos em aula quando calculamos isso!