1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

Time Series Analysis in R

Code

1 Exploratory time series data analysis

1.1 Introduction

1.1.1 Exploring raw time series

The most common first step when conducting time series analysis is to display your time series dataset in a visually intuitive format. The most useful way to view raw time series data in R is to use the print() command, which displays the Start, End, and Frequency of your data along with the observations.

Another useful command for viewing time series data in R is the length() function, which tells you the total number of observations in your data.

Some datasets are very long, and previewing a subset of data is more suitable than displaying the entire series. The head(***, n =***) and tail(***, n =***) functions, in which n is the number of items to display, focus on the first and last few elements of a given dataset respectively.

In this exercise, you'll explore the famous River Nile annual streamflow data, Nile. This time series dataset includes some metadata information. When calling print(Nile), note that Start = 1871 indicates that 1871 is the year of the first annual observation, and End = 1970 indicates 1970 is the year of the last annual observation.

Hide

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

```
# Print the Nile dataset
print(Nile)

# List the number of observations in the Nile dataset
length(Nile)

# Display the first 10 elements of the Nile dataset
head(Nile, n = 10)

# Display the last 12 elements of the Nile dataset
tail(Nile, n = 12)
```

1.1.2 Basic time series plots

While simple commands such as print(), length(), head(), and tail() provide crucial information about your time series data, another very useful way to explore any data is to generate a plot.

In this exercise, you will plot the River Nile annual streamflow data using the plot() function. For time series data objects such as Nile, a Time index for the horizontal axis is typically included. From the previous exercise, you know that this data spans from 1871 to 1970, and horizontal tick marks are labeled as such. The default label of “Time” is not very informative. Since these data are annual measurements, you should use the label “Year”. While you’re at it, you should change the vertical axis label to “River Volume (1e9 m^{3})”.

Additionally, it helps to have an informative title, which can be set using the argument main. For your purposes, a useful title for this figure would be “Annual River Nile Volume at Aswan, 1871-1970”.

Finally, the default plotting type for time series objects is “l” for line. Connecting consecutive observations can help make a time series plot more interpretable. Sometimes it is also useful to include both the observations points as well as the lines, and we instead use “b” for both.

Hide

```
# Plot the Nile data with xlab, ylab, main, and type arguments
plot(Nile, type = "b", xlab = "Year", ylab = "River Volume (1e9 m^{3})", main = "Ann
ual River Nile Volume at Aswan, 1871-1970")
```

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

The plot() command is one of the most versatile commands in R. When used with time series data, this command automatically plots your data over time.

1.1.3 What does the time index tell us?

Some data are naturally evenly spaced by time. The time series discrete_data shown in the top figure has 20 observations, with one observation appearing at each of the discrete time indices 1 through 20. Discrete time indexing is appropriate for discrete_data.

The time series continuous_series shown in the bottom figure also has 20 observations, it is following the same periodic pattern as discrete_data, but its observations are not evenly spaced. Its first, second, and last observations were observed at times 1.210322, 1.746137, and 20.180524, respectively. Continuous time indexing is natural for continuous_series, however, the observations are approximately evenly spaced, with about 1 observation observed per time unit. Let's investigate using a discrete time indexing for continuous_series.

Hide

```
discrete_time_index <- c(1:20)

plot(continuous_time_index, discrete_data, type = "b")
plot(discrete_time_index, discrete_data, type = "b")
```

Hide

```
# Plot the continuous_series using continuous time indexing
par(mfrow=c(2,1))
plot(continuous_time_index,continuous_series, type = "b")

# Make a discrete time index using 1:20
discrete_time_index <- c(1:20)

# Now plot the continuous_series using discrete time indexing
plot(discrete_time_index,continuous_series, type = "b")
```

1.2 Sampling frequency

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

1.2.1 Identifying the sampling frequency

In addition to viewing your data and plotting over time, there are several additional operations that can be performed on time series datasets.

The start() and end() functions return the time index of the first and last observations, respectively. The time() function calculates a vector of time indices, with one element for each time index on which the series was observed.

The deltat() function returns the fixed time interval between observations and the frequency() function returns the number of observations per unit time. Finally, the cycle() function returns the position in the cycle of each observation.

In this exercise, you'll practice applying these functions to the AirPassengers dataset, which reports the monthly total international airline passengers (in thousands) from 1949 to 1960.

Hide

```
# Plot AirPassengers
plot(AirPassengers)

# View the start and end dates of AirPassengers
start(AirPassengers)
end(AirPassengers)

# Use time(), deltat(), frequency(), and cycle() with AirPassengers
time(AirPassengers)
deltat(AirPassengers)
frequency(AirPassengers)
cycle(AirPassengers)
```

These commands provide considerable descriptive information about the structures and patterns in your time series data. It may help to keep these commands handy when working with your own time series data.

1.2.2 When is the sampling frequency exact?

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

The sampling frequency is often only approximate and the interval between observations is not quite a fixed unit. For example, there are usually 365 days in a year based on the Gregorian calendar. However, (almost) every four years there are 366 days (leap years). This compensates for the fact that the Earth completes a rotation around Sol, the sun, in approximately 365.2422 days, on average.

As a simplifying assumption, we often ignore these small discrepancies and proceed as though the sampling frequency and observation intervals are fixed constants. Typically, our results will not be sensitive to approximation when the underlying process is not changing too quickly.

1.2.3 Missing values

Sometimes there are missing values in time series data, denoted NA in R, and it is useful to know their locations. It is also important to know how missing values are handled by various R functions. Sometimes we may want to ignore any missingness, but other times we may wish to impute or estimate the missing values.

Let’s again consider the monthly AirPassengers dataset, but now the data for the year 1956 are missing. In this exercise, you’ll explore the implications of this missing data and impute some new data to solve the problem.

The mean() function calculates the sample mean, but it fails in the presence of any NA values. Use mean(___, na.rm = TRUE) to calculate the mean with all missing values removed. It is common to replace missing values with the mean of the observed values. Does this simple data imputation scheme appear adequate when applied the the AirPassengers dataset?

Hide

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

```
# Plot the AirPassengers data
plot(AirPassengers_NA)

# Compute the mean of AirPassengers
mean(AirPassengers_NA, na.rm = TRUE)

# Impute mean values to NA in AirPassengers
AirPassengers_NA[85:96] <- mean(AirPassengers_NA, na.rm = TRUE)

# Generate another plot of AirPassengers
plot(AirPassengers_NA)

# Add the complete AirPassengers data to your plot
#rm(AirPassengers_NA)
points(AirPassengers_NA, type = "l", col = 2, lty = 3)
AirPassengers_NA
```

Based on your plot, it seems that simple data imputation using the mean is not a great method to approximate what's really going on in the AirPassengers data.

1.3 Basic time series objects

1.3.1 Creating a time series object with ts()

The function ts() can be applied to create time series objects. A time series object is a vector (univariate) or matrix (multivariate) with additional attributes, including time indices for each observation, the sampling frequency and time increment between observations, and the cycle length for periodic data. Such objects are of the ts class, and represent data that has been observed at (approximately) equally spaced time points. Now you will create time series objects yourself.

The advantage of creating and working with time series objects of the ts class is that many methods are available for utilizing time series attributes, such as time index information. For example, as you've seen in earlier exercises, calling plot() on a ts object will automatically generate a plot over time.

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

In this exercise, you'll familiarize yourself with the ts class by encoding some time series data (saved as data_vector) into ts and exploring the result. Your time series data_vector starts in the year 2004 and has 4 observations per year (i.e. it is quarterly data).

Hide

```
# Use print() and plot() to view data_vector
print(data_vector)
plot(data_vector)

# Convert data_vector to a ts object with start = 2004 and frequency = 4
time_series <- ts(data_vector, start = 2004, frequency = 4)

# Use print() and plot() to view time_series
print(time_series)
plot(time_series)
```

As you can see, ts objects are treated differently by commands such as print() and plot(). For example, automatic use of the time-index in your calls to plot() requires a ts object.

1.3.2 Testing whether an object is a time series

When you work to create your own datasets, you can build them as ts objects. Recall the dataset data_vector from the previous exercise, which was just a vector of numbers, and time_series, the ts object you created from data_vector using the ts() function and information regarding the start time and the observation frequency. As a reminder, data_vector and time_series are shown in the plot on the right.

When you use datasets from others, such as those included in an R package, you can check whether they are ts objects using the is.ts() command. The result of the test is either TRUE when the data is of the ts class, or FALSE if it is not.

Hide

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

```
# Check whether data_vector and time_series are ts objects
is.ts(data_vector)
is.ts(time_series)

# Check whether Nile is a ts object
is.ts(Nile)

# Check whether AirPassengers is a ts object
is.ts(AirPassengers)
```

is.ts() is a simple command for determining whether or not you're working with a ts object. As you can see, the Nile and AirPassengers datasets you worked with earlier in the chapter are both encoded as ts objects.

1.3.3 Plotting a time series object

It is often very useful to plot data we are analyzing, as is the case when conducting time series analysis. If the dataset under study is of the ts class, then the plot() function has methods that automatically incorporate time index information into a figure.

Let's consider the eu_stocks dataset (available in R by default as EuStockMarkets). This dataset contains daily closing prices of major European stock indices from 1991-1998, specifically, from Germany (DAX), Switzerland (SMI), France (CAC), and the UK (FTSE). The data were observed when the markets were open, so there are no observations on weekends and holidays. We will proceed with the approximation that this dataset has evenly spaced observations and is a four dimensional time series.

Hide

```
eu_stocks <- EuStockMarkets
```

Hide

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

```
# Check whether eu_stocks is a ts object
is.ts(eu_stocks)

# View the start, end, and frequency of eu_stocks
start(eu_stocks)
end(eu_stocks)
frequency(eu_stocks)

# Generate a simple plot of eu_stocks
plot(eu_stocks)

# Use ts.plot with eu_stocks
ts.plot(eu_stocks, col = 1:4, xlab = "Year", ylab = "Index Value", main = "Major Eur
opean Stock Indices, 1991-1998")

# Add a legend to your ts.plot
legend("topleft", colnames(eu_stocks), lty = 1, col = 1:4, bty = "n")
```

2 Predicting the future

2.1 Trend spotting!

2.1.1 Removing trends in variability via the logarithmic transformation

The logarithmic function log() is a data transformation that can be applied to positively valued time series data. It slightly shrinks observations that are greater than one towards zero, while greatly shrinking very large observations. This property can stabilize variability when a series exhibits increasing variability over time. It may also be used to linearize a rapid growth pattern over time.

Hide

```
plot(rapid_growth)
```

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

Hide

```
# Log rapid_growth
linear_growth <- log(rapid_growth)

# Plot linear_growth using ts.plot()
 ts.plot(linear_growth)
```

As you can see, the logarithmic transformation helps stabilize your data by inducing linear growth over time. Remember to adjust your interpetation of the data accordingly.

2.1.2 Removing trends in level by differencing

The first difference transformation of a time series $z[t]$ consists of the differences (changes) between successive observations over time, that is $z[t] - z[t-1]$.

Differencing a time series can remove a time trend. The function diff() will calculate the first difference or change series. A difference series lets you examine the increments or changes in a given time series. It always has one fewer observations than the original series.

Hide

```
# Generate the first difference of z
dz <- diff(rapid_growth)

# Plot dz
ts.plot(dz)

# View the length of z and dz, respectively
length(rapid_growth)
length(dz)
```

By removing the long-term time trend, you can view the amount of change from one observation to the next.

2.1.3 Removing seasonal trends with seasonal differencing

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

For time series exhibiting seasonal trends, seasonal differencing can be applied to remove these periodic patterns. For example, monthly data may exhibit a strong twelve month pattern. In such situations, changes in behavior from year to year may be of more interest than changes from month to month, which may largely follow the overall seasonal pattern.

The function diff(…, lag = s) will calculate the lag s difference or length s seasonal change series. For monthly or quarterly data, an appropriate value of s would be 12 or 4, respectively. The diff() function has lag = 1 as its default for first differencing. Similar to before, a seasonally differenced series will have s fewer observations than the original series.

Hide

```
ts.plot(x)
```

Hide

```
# Generate a diff of x with lag = 4. Save this to dx
dx <- diff(x, lag = 4)

# Plot dx
ts.plot(dx)

# View the length of x and dx, respectively
length(x)
length(dx)
```

Once again differencing allows you to remove the longer-term time trend - in this case, seasonal volatility - and focus on the change from one period to another.

2.2 The white noise (WN) model

2.2.1 Simulate the white noise model

The white noise (WN) model is a basic time series model. It is also a basis for the more elaborate models we will consider. We will focus on the simplest form of WN, independent and identically distributed data.

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

The arima.sim() function can be used to simulate data from a variety of time series models. ARIMA is an abbreviation for the autoregressive integrated moving average class of models we will consider throughout this course.

An ARIMA(p, d, q) model has three parts, the autoregressive order p, the order of integration (or differencing) d, and the moving average order q. We will detail each of these parts soon, but for now we note that the ARIMA(0, 0, 0) model, i.e., with all of these components zero, is simply the WN model.

Hide

```
# Simulate a WN model with list(order = c(0, 0, 0))
white_noise <- arima.sim(model = list(order = c(0, 0, 0)), n = 100)

# Plot your white_noise data
ts.plot(white_noise)

# Simulate from the WN model with: mean = 100, sd = 10
white_noise_2 <- arima.sim(model = list(order = c(0, 0, 0)), n = 100, mean = 100, sd
= 10)

# Plot your white_noise_2 data
ts.plot(white_noise_2)
```

The arima.sim() command is a useful way to quickly simulate time series data with the qualities you specify.

2.2.2 Estimate the white noise model

For a given time series y we can fit the white noise (WN) model using the arima(…, order = c(0, 0, 0)) function. Recall that the WN model is an ARIMA(0,0,0) model. Applying the arima() function returns information or output about the estimated model. For the WN model this includes the estimated mean, labeled intercept, and the estimated variance, labeled sigma^2.

In this exercise, you'll explore the qualities of the WN model. What is the estimated mean? Compare this with the sample mean using the mean() function. What is the estimated variance? Compare this with the sample variance using the var() function.

1 Exploratory time series data analysis
1.1 Introduction
1.2 Sampling frequency
1.3 Basic time series objects
2 Predicting the future
3 Correlation analysis and the autocorrelation function
4 Autoregression

The time series y has already been loaded, and is shown in the adjoining figure.

Hide

```
ts.plot(y)
```

Hide

```
# Fit the WN model to y using the arima command
arima(y, order = c(0, 0, 0))

# Calculate the sample mean and sample variance of y
mean(y)
var(y)
```

From the comparisons you can see that the arima() function estimates are very close to the sample mean and variance estimates, in fact identical for the mean.

2.3 The random walk (RW) model

2.3.1 Simulate the random walk model

The random walk (RW) model is also a basic time series model. It is the cumulative sum (or integration) of a mean zero white noise (WN) series, such that the first difference series of a RW is a WN series. Note for reference that the RW model is an ARIMA(0, 1, 0) model, in which the middle entry of 1 indicates that the model's order of integration is 1.

The arima.sim() function can be used to simulate data from the RW by including the model = list(order = c(0, 1, 0)) argument. We also need to specify a series length n. Finally, you can specify a sd for the series (increments), where the default value is 1.

Hide

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

```
# Generate a RW model using arima.sim
random_walk <- arima.sim(model = list(order = c(0, 1, 0)), n = 100)

# Plot random_walk
ts.plot(random_walk)

# Calculate the first difference series
random_walk_diff <- diff(random_walk)

# Plot random_walk_diff
ts.plot(random_walk_diff)
```

As you can see, the first difference of your random_walk data is white noise data. This is because a random walk is simply recursive white noise data. By removing the long-term trend, you end up with simple white noise.

2.3.2 Simulate the random walk model with a drift

A random walk (RW) need not wander about zero, it can have an upward or downward trajectory, i.e., a drift or time trend. This is done by including an intercept in the RW model, which corresponds to the slope of the RW time trend.

For an alternative formulation, you can take the cumulative sum of a constant mean white noise (WN) series, such that the mean corresponds to the slope of the RW time trend.

To simulate data from the RW model with a drift you again use the arima.sim() function with the model = list(order = c(0, 1, 0)) argument. This time, you should add the additional argument mean = … to specify the drift variable, or the intercept.

Hide

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

```
# Generate a RW model with a drift uing arima.sim
rw_drift <- arima.sim(model = list(order = c(0, 1, 0)), n = 100, mean = 1)

# Plot rw_drift
ts.plot(rw_drift)

# Calculate the first difference series
rw_drift_diff <- diff(rw_drift)

# Plot rw_drift_diff
ts.plot(rw_drift_diff)
```

Once again, taking the first difference of your random walk data transformed it back into white noise data, regardless of the presence of your long-term drift.

2.3.3 Estimate the random walk model

For a given time series y we can fit the random walk model with a drift by first differencing the data, then fitting the white noise (WN) model to the differenced data using the arima() command with the order = c(0, 0, 0)) argument.

The arima() command displays information or output about the fitted model. Under the Coefficients: heading is the estimated drift variable, named the intercept. Its approximate standard error (or s.e.) is provided directly below it. The variance of the WN part of the model is also estimated under the label sigma^2.

Hide

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

```
# Difference your random_walk data
rw_diff <- diff(rw_drift)

# Plot rw_diff
ts.plot(rw_diff)

# Now fit the WN model to the differenced data
model_wn <- arima(rw_diff, order = c(0, 0, 0))

# Store the value of the estimated time trend (intercept)
int_wn <- model_wn$coef

# Plot the original random_walk data
ts.plot(rw_drift)

# Use abline(0, ...) to add time trend to the figure
abline(0, int_wn)
```

The arima() command correctly identified the time trend in your original random-walk data.

2.4 Stationary processes

2.4.1 Are the white noise model or the random walk model stationary?

The white noise (WN) and random walk (RW) models are very closely related. However, only the RW is always non-stationary, both with and without a drift term. This is a simulation exercise to highlight the differences.

Recall that if we start with a mean zero WN process and compute its running or cumulative sum, the result is a RW process. The cumsum() function will make this transformation for you. Similarly, if we create a WN process, but change its mean from zero, and then compute its cumulative sum, the result is a RW process with a drift.

Hide

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

```
# Use arima.sim() to generate WN data
white_noise <- arima.sim(model = list(order = c(0,0,0)), n = 100)

# Use cumsum() to convert your WN data to RW
random_walk <- cumsum(white_noise)

# Use arima.sim() to generate WN drift data
wn_drift <- arima.sim(model = list(order = c(0,0,0)), n = 100, mean = 0.4)

# Use cumsum() to convert your WN drift data to RW
rw_drift <- cumsum(wn_drift)

# Plot all four data objects
plot.ts(cbind(white_noise, random_walk, wn_drift, rw_drift))
```

As you can see, it is easy to reverse-engineer the RW data by simply generating a cumulative sum of white noise data.

3 Correlation analysis and the autocorrelation function

3.1 Scatterplots

3.1.1 Asset prices vs. asset returns

The goal of investing is to make a profit. The revenue or loss from investing depends on the amount invested and changes in prices, and high revenue relative to the size of an investment is of central interest. This is what financial asset returns measure, changes in price as a fraction of the initial price over a given time horizon, for example, one business day.

Let’s again consider the eu_stocks dataset. This dataset reports index values, which we can regard as prices. The indices are not investable assets themselves, but there are many investable financial assets that closely track major market indices, including mutual funds and exchange traded funds.

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

Log returns, also called continuously compounded returns, are also commonly used in financial time series analysis. They are the log of gross returns, or equivalently, the changes (or first differences) in the logarithm of prices.

The change in appearance between daily prices and daily returns is typically substantial, while the difference between daily returns and log returns is usually small. As you'll see later, one advantage of using log returns is that calculating multi-period returns from individual periods is greatly simplified - you just add them together!

In this exercise, you'll further explore the eu_stocks dataset, including plotting prices, converting prices to (net) returns, and converting prices to log returns.

Hide

```
# Plot eu_stocks
plot(eu_stocks)

# Use this code to convert prices to returns
returns <- eu_stocks[-1,] / eu_stocks[-1860,] - 1

# Convert returns to ts
returns <- ts(returns, start = c(1991, 130), frequency = 260)

# Plot returns
plot(returns)

# Use this code to convert prices to log returns
logreturns <- diff(log(eu_stocks))

# Plot logreturns
plot(logreturns)
```

Daily net returns and daily log returns are two valuable metrics for financial data.

3.1.2 Characteristics of financial time series

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

Daily financial asset returns typically share many characteristics. Returns over one day are typically small, and their average is close to zero. At the same time, their variances and standard deviations can be relatively large. Over the course of a few years, several very large returns (in magnitude) are typically observed. These relative outliers happen on only a handful of days, but they account for the most substantial movements in asset prices. Because of these extreme returns, the distribution of daily asset returns is not normal, but heavy-tailed, and sometimes skewed. In general, individual stock returns typically have even greater variability and more extreme observations than index returns.

In this exercise, you'll work with the eu_percentreturns dataset, which is the percentage returns calculated from your eu_stocks data. For each of the four indices contained in your data, you'll calculate the sample mean, variance, and standard deviation.

Notice that the average daily return is about 0, while the standard deviation is about 1 percentage point. Also apply the hist() and qqnorm() functions to make histograms and normal quantile plots, respectively, for each of the indices.

Hide

```
eu_percentreturns <-  returns * 100
```

Hide

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

```
# Generate means from eu_percentreturns
colMeans(eu_percentreturns)

# Use apply to calculate sample variance from eu_percentreturns
apply(eu_percentreturns, MARGIN = 2, FUN = var)

# Use apply to calculate standard deviation from eu_percentreturns
apply(eu_percentreturns, MARGIN = 2, FUN = sd)

# Display a histogram of percent returns for each index
par(mfrow = c(2,2))
apply(eu_percentreturns, MARGIN = 2, FUN = hist, main = "", xlab = "Percentage Retur
n")

# Display normal quantile plots of percent returns for each index
par(mfrow = c(2,2))
apply(eu_percentreturns, MARGIN = 2, FUN = qqnorm, main = "")
qqline(eu_percentreturns)
```

Note that the vast majority of returns are near zero, but some daily returns are greater than 5 percentage points in magnitude. Similarly, note the clear departure from normality, especially in the tails of the distributions, as evident in the normal quantile plots.

3.1.3 Plotting pairs of data

Time series data is often presented in a time series plot. For example, the index values from the eu_stocks dataset are shown in the adjoining figure.

Hide

```
plot(eu_stocks)
```

Recall, eu_stocks contains daily closing prices from 1991-1998 for the major stock indices in Germany (DAX), Switzerland (SMI), France (CAC), and the UK (FTSE).

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

It is also useful to examine the bivariate relationship between pairs of time series. In this exercise we will consider the contemporaneous relationship, that is matching observations that occur at the same time, between pairs of index values as well as their log returns. The plot(a, b) function will produce a scatterplot when two time series names a and b are given as input.

To simultaneously make scatterplots for all pairs of several assets the pairs() function can be applied to produce a scatterplot matrix. When shared time trends are present in prices or index values it is common to instead compare their returns or log returns.

In this exercise, you'll practice these skills on the eu_stocks data. Because the DAX and FTSE returns have similar time coverage, you can easily make a scatterplot of these indices. Note that the normal distribution has elliptical contours of equal probability, and pairs of data drawn from the multivariate normal distribution form a roughly elliptically shaped point cloud. Do any of the pairs in the scatterplot matrices exhibit this pattern, before or after log transformation?

Hide

```
# Make a scatterplot of DAX and FTSE
plot(eu_stocks[,"DAX"], eu_stocks[,"FTSE"])

# Make a scatterplot matrix of eu_stocks
pairs(eu_stocks)

# Convert eu_stocks to log returns
logreturns <- diff(log(eu_stocks))

# Plot logreturns
plot(logreturns)

# Make a scatterplot matrix of logreturns
pairs(logreturns)
```

As you can see, the pairs() command is a useful way to quickly check for relationships between your indices.

3.2 Covariance and correlation

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

3.2.1 Calculating sample covariances and correlations

Sample covariances measure the strength of the linear relationship between matched pairs of variables. The cov() function can be used to calculate covariances for a pair of variables, or a covariance matrix when a matrix containing several variables is given as input. For the latter case, the matrix is symmetric with covariances between variables on the off-diagonal and variances of the variables along the diagonal.

Covariances are very important throughout finance, but they are not scale free and they can be difficult to directly interpret. Correlation is the standardized version of covariance that ranges in value from -1 to 1, where values close to 1 in magnitude indicate a strong linear relationship between pairs of variables. The cor() function can be applied to both pairs of variables as well as a matrix containing several variables, and the output is interpreted analogously.

Hide

```
DAX_logreturns <- diff(log(eu_stocks[, "DAX"]))
FTSE_logreturns <- diff(log(eu_stocks[, "FTSE"]))
```

Hide

```
# Use cov() with DAX_logreturns and FTSE_logreturns
cov(DAX_logreturns, FTSE_logreturns)

# Use cov() with logreturns
cov(logreturns)

# Use cor() with DAX_logreturns and FTSE_logreturns
cor(DAX_logreturns, FTSE_logreturns)

# Use cor() with logreturns
cor(logreturns)
```

The cov() and cor() commands provide a simple and intuitive output for comparing the relationships between your indices, especially when a scatterplot matrix is difficult to interpret.

3.3 Autocorrelation

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

3.3.1 Calculating autocorrelations

Autocorrelations or lagged correlations are used to assess whether a time series is dependent on its past. For a time series x of length n we consider the n-1 pairs of observations one time unit apart. The first such pair is (x[2],x[1]), and the next is (x[3],x[2]). Each such pair is of the form (x[t],x[t-1]) where t is the observation index, which we vary from 2 to n in this case. The lag-1 autocorrelation of x can be estimated as the sample correlation of these (x[t], x[t-1]) pairs.

In general, we can manually create these pairs of observations. First, create two vectors, x_t0 and x_t1, each with length n-1, such that the rows correspond to (x[t], x[t-1]) pairs. Then apply the cor() function to estimate the lag-1 autocorrelation.

Luckily, the acf() command provides a shortcut. Applying acf(…, lag.max = 1, plot = FALSE) to a series x automatically calculates the lag-1 autocorrelation.

Finally, note that the two estimates differ slightly as they use slightly different scalings in their calculation of sample covariance, 1/(n-1) versus 1/n. Although the latter would provide a biased estimate, it is preferred in time series analysis, and the resulting autocorrelation estimates only differ by a factor of (n-1)/n.

Hide

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

```
n = 100
# Define x_t0 as x[-1]
x_t0 <- x[-1]

# Define x_t1 as x[-n]
x_t1 <- x[-n]

# Confirm that x_t0 and x_t1 are (x[t], x[t-1]) pairs
head(cbind(x_t0, x_t1))

# Plot x_t0 and x_t1
plot(x_t0, x_t1)

# View the correlation between x_t0 and x_t1
cor(x_t0, x_t1)

# Use acf with x
acf(x, lag.max = 1, plot = FALSE)

# Confirm that difference factor is (n-1)/n
cor(x_t1, x_t0) * (n-1)/n
```

As you can see, the acf() command is a helpful shortcut for calculating autocorrelation.

3.3.2 The autocorrelation function

Autocorrelations can be estimated at many lags to better assess how a time series relates to its past. We are typically most interested in how a series relates to its most recent past.

The acf(…, lag.max = …, plot = FALSE) function will estimate all autocorrelations from 0, 1, 2,…, up to the value specified by the argument lag.max. In the previous exercise, you focused on the lag-1 autocorrelation by setting the lag.max argument to 1.

In this exercise, you'll explore some further applications of the acf() command.

Hide

```
x <- arima.sim(model = list(order = c(0, 1, 0)), n = 100)
```

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

Hide

```
# Generate ACF estimates for x up to lag-10
acf(x, lag.max = 10, plot = FALSE)

# Type the ACF estimate at lag-10
0.604

# Type the ACF estimate at lag-5
0.209
```

Since autocorrelations may vary by lag, we often consider autocorrelations as a function of the time lag. Taking this view, we have now estimated the autocorrelation function (ACF) of x from lags 0 to 10.

3.3.3 Visualizing the autocorrelation function

Estimating the autocorrelation function (ACF) at many lags allows us to assess how a time series x relates to its past. The numeric estimates are important for detailed calculations, but it is also useful to visualize the ACF as a function of the lag.

In fact, the acf() command produces a figure by default. It also makes a default choice for lag.max, the maximum number of lags to be displayed.

The time series x shows strong persistence, meaning the current value is closely relatively to those that proceed it. The time series y shows a periodic pattern with a cycle length of approximately four observations, meaning the current value is relatively close to the observation four before it. The time series z does not exhibit any clear pattern.

Hide

```
plot.ts(cbind(x, y, z))
```

In this exercise, you'll plot an estimated autocorrelation function for each time series. In the plots produced by acf(), the lag for each autocorrelation estimate is denoted on the horizontal axis and each autocorrelation estimate is indicated by the height of the vertical bars. Recall that the ACF at lag-0 is always 1.

1 Exploratory time series data analysis
1.1 Introduction
1.2 Sampling frequency
1.3 Basic time series objects
2 Predicting the future
3 Correlation analysis and the autocorrelation function
4 Autoregression

Finally, each ACF figure includes a pair of blue, horizontal, dashed lines representing lag-wise 95% confidence intervals centered at zero. These are used for determining the statistical significance of an individual autocorrelation estimate at a given lag versus a null value of zero, i.e., no autocorrelation at that lag.

Hide

```
# View the ACF of x
acf(x)

# View the ACF of y
acf(y)

# View the ACF of z
acf(z)
```

Plotting the estimated ACF of x shows large positive correlations for several lags which quickly decay towards zero. Plotting the estimated ACF of y shows large positive correlations at lags which are multiples of four, although these also decay towards zero as the lag multiple increases. Finally, the estimated ACF of z is near zero at all lags. It appears the series z is not linearly related to its past, at least through lag 20.

4 Autoregression

4.1 The autoregressive model

4.1.1 Simulate the autoregressive model

The autoregressive (AR) model is arguably the most widely used time series model. It shares the very familiar interpretation of a simple linear regression, but here each observation is regressed on the previous observation. The AR model also includes the white noise (WN) and random walk (RW) models examined in earlier chapters as special cases.

The versatile arima.sim() function used in previous chapters can also be used to simulate data from an AR model by setting the model argument equal to list(ar = phi) , in which phi is a slope parameter from the interval (-1, 1). We also need to specify a series length n.

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

Hide

```
# Simulate an AR model with 0.5 slope
x <- arima.sim(model = list(ar = 0.5), n = 100)

# Simulate an AR model with 0.9 slope
y <- arima.sim(model = list(ar = 0.9), n = 100)

# Simulate an AR model with -0.75 slope
z <- arima.sim(model = list(ar = -.75), n = 100)

# Plot your simulated data
plot.ts(cbind(x, y, z))
```

As you can see, your x data shows a just a moderate amount of autocorrelation while your y data shows a large amount of autocorrelation. Alternatively, your z data tends to oscillate considerably from one observation to the next.

4.1.2 Estimate the autocorrelation function (ACF) for an autoregression

What if you need to estimate the autocorrelation function from your data? To do so, you'll need the acf() command, which estimates autocorrelation by exploring lags in your data. By default, this command generates a plot of the relationship between the current observation and lags extending backwards.

In this exercise, you'll use the acf() command to estimate the autocorrelation function for three new simulated AR series (x, y, and z). These objects have slope parameters 0.5, 0.9, and -0.75.

Hide

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

```
# Calculate the ACF for x
acf(x)

# Calculate the ACF for y
acf(y)

# Calculate the ACF for z
acf(z)
```

The plots generated by the acf() command provide useful information about each lag of your time series. The first series x has positive autocorrelation for the first couple lags, but they quickly approach zero. The second series y has positive autocorrelation for many lags, but they also decay to zero. The last series z has an alternating pattern, as does its autocorrelation function (ACF), but its ACF still quickly decays to zero in magnitude.

4.1.3 Persistence and anti-persistence

Autoregressive processes can exhibit varying levels of persistence as well as anti-persistence or oscillatory behavior. Persistence is defined by a high correlation between an observation and its lag, while anti-persistence is defined by a large amount of variation between an observation and its lag.

4.1.4 Compare the random walk (RW) and autoregressive (AR) models

The random walk (RW) model is a special case of the autoregressive (AR) model, in which the slope parameter is equal to 1. Recall from previous chapters that the RW model is not stationary and exhibits very strong persistence. Its sample autocovariance function (ACF) also decays to zero very slowly, meaning past values have a long lasting impact on current values.

The stationary AR model has a slope parameter between -1 and 1. The AR model exhibits higher persistence when its slope parameter is closer to 1, but the process reverts to its mean fairly quickly. Its sample ACF also decays to zero at a quick (geometric) rate, indicating that values far in the past have little impact on future values of the process.

Hide

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

```
# Simulate and plot AR model with slope 0.9
x <- arima.sim(model = list(ar = 0.9), n = 200)
ts.plot(x)
acf(x)

# Simulate and plot AR model with slope 0.98
y <- arima.sim(model = list(ar = 0.98), n = 200)
ts.plot(y)
acf(y)

# Simulate and plot RW model
z <-arima.sim(model = list(order = c(0, 1, 0)), n = 200)
ts.plot(z)
acf(z)
```

As you can see, the AR model represented by series y exhibits greater persistence than series x, but the ACF continues to decay to 0. By contrast, the RW model represented by series z shows considerable persistence and relatively little decay in the ACF.

4.2 AR model estimation and forecasting

4.2.1 Estimate the autoregressive (AR) model

For a given time series x we can fit the autoregressive (AR) model using the arima() command and setting order equal to c(1, 0, 0). Note for reference that an AR model is an ARIMA(1, 0, 0) model.

In this exercise, you'll explore additional qualities of the AR model by practicing the arima() command on a simulated time series x as well as the AirPassengers data. This command allows you to identify the estimated slope (ar1), mean (intercept), and innovation variance (sigma^2) of the model.

Hide

1 Exploratory time series data analysis
1.1 Introduction
1.2 Sampling frequency
1.3 Basic time series objects
2 Predicting the future
3 Correlation analysis and the autocorrelation function
4 Autoregression

```
# Fit the AR model to x
arima(x, order = c(1, 0, 0))

# Copy and paste the slope (ar1) estimate
0.8954

# Copy and paste the slope mean (intercept) estimate
-0.2025

# Copy and paste the innovation variance (sigma^2) estimate
1.046

# Fit the AR model to AirPassengers
AR <- arima(AirPassengers, order = c(1, 0, 0))
print(AR)

# Run the following commands to plot the series and fitted values
ts.plot(AirPassengers)
AR_fitted <- AirPassengers - residuals(AR)
points(AR_fitted, type = "l", col = 2, lty = 2)
```

By fitting an AR model to the AirPassengers data, you've succesfully modeled the data in a reproducible fashion. This allows you to predict future observations based on your AR_fitted data.

4.2.2 Simple forecasts from an estimated AR mode

Now that you've modeled your data using the arima() command, you are ready to make simple forecasts based on your model. The predict() function can be used to make forecasts from an estimated AR model. In the object generated by your predict() command, the $pred value is the forecast, and the $se value is the standard error for the forecast.

To make predictions for several periods beyond the last observations, you can use the n.ahead argument in your predict() command. This argument establishes the forecast horizon (h), or the number of periods being forecast. The forecasts are made recursively from 1 to h-steps ahead from the end of the observed time series.

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

In this exercise, you'll make simple forecasts using an AR model applied to the Nile data, which records annual observations of the flow of the River Nile from 1871 to 1970.

Hide

```
# Fit an AR model to Nile
AR_fit <- arima(Nile, order  = c(1, 0, 0))
print(AR_fit)

# Use predict() to make a 1-step forecast
predict_AR <- predict(AR_fit)

# Obtain the 1-step forecast using $pred[1]
predict_AR$pred[1]

# Use predict to make 1-step through 10-step forecasts
predict(AR_fit, n.ahead = 10)

# Run to plot the Nile series plus the forecast and 95% prediction intervals
ts.plot(Nile, xlim = c(1871, 1980))
AR_forecast <- predict(AR_fit, n.ahead = 10)$pred
AR_forecast_se <- predict(AR_fit, n.ahead = 10)$se
points(AR_forecast, type = "l", col = 2)
points(AR_forecast - 2*AR_forecast_se, type = "l", col = 2, lty = 2)
points(AR_forecast + 2*AR_forecast_se, type = "l", col = 2, lty = 2)
```

Your predictions of River Nile flow from 1971 to 1980 make sense based on the data you have. The relatively wide band of confidence (represented by the dotted lines) is a result of the low persistence in your Nile data.

4.3 The simple moving average model

4.3.1 Simulate the simple moving average model

The simple moving average (MA) model is a parsimonious time series model used to account for very short-run autocorrelation. It does have a regression like form, but here each observation is regressed on the previous innovation, which is not actually observed. Like the autoregressive (AR)

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

model, the MA model includes the white noise (WN) model as special case.

As with previous models, the MA model can be simulated using the arima.sim() command by setting the model argument to list(ma = theta), where theta is a slope parameter from the interval (-1, 1). Once again, you also need to specifcy the series length using the n argument.

Hide

```
# Generate MA model with slope 0.5
x <- arima.sim(model = list(ma = 0.5), n = 100)

# Generate MA model with slope 0.9
y <- arima.sim(model = list(ma = 0.9), n = 100)

# Generate MA model with slope -0.5
z <- arima.sim(model = list(ma = -0.5), n = 100)

# Plot all three models together
plot.ts(cbind(x, y, z))
```

Note that there is some very short-run persistence for the positive slope values (x and y), and the series has a tendency to alternate when the slope value is negative (z).

4.3.2 Estimate the autocorrelation function (ACF) for a moving average

Now that you've simulated some MA data using the arima.sim() command, you may want to estimate the autocorrelation functions (ACF) for your data. As in the previous chapter, you can use the acf() command to generate plots of the autocorrelation in your MA data.

In this exercise, you'll use acf() to estimate the ACF for three simulated MA series, x, y, and z. These series have slope parameters of 0.4, 0.9, and -0.75, respectively, and are shown in the figure on the right.

Hide

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

```
# Calculate ACF for x
acf(x)

# Calculate ACF for y
acf(y)

# Calculate ACF for z
acf(z)
```

As you can see from your ACF plots, the series x has positive sample autocorrelation at the first lag, but it is approximately zero at other lags. The series y has a larger sample autocorrelation at its first lag, but it is also approximately zero for the others. The series z has an alternating pattern, and its sample autocorrelation is negative at the first lag. However, similar to the others, it is approximately zero for all higher lags.

4.4 MA model estimation and forecasting

4.4.1 Estimate the simple moving average model

Now that you've simulated some MA models and calculated the ACF from these models, your next step is to fit the simple moving average (MA) model to some data using the arima() command. For a given time series x we can fit the simple moving average (MA) model using arima(…, order = c(0, 0, 1)). Note for reference that an MA model is an ARIMA(0, 0, 1) mode.

Hide

1 Exploratory time series data analysis
1.1 Introduction
1.2 Sampling frequency
1.3 Basic time series objects
2 Predicting the future
3 Correlation analysis and the autocorrelation function
4 Autoregression

```
# Fit the MA model to x
arima(x, order = c(0, 0, 1))

# Paste the slope (ma1) estimate below
0.7928

# Paste the slope mean (intercept) estimate below
0.1589

# Paste the innovation variance (sigma^2) estimate below
0.9576

# Fit the MA model to Nile
MA <- arima(Nile, order = c(0, 0, 1))
print(MA)

# Plot Nile and MA_fit
ts.plot(Nile)
MA_fit <- Nile - resid(MA)
points(MA_fit, type = "l", col = 2, lty = 2)
```

By fitting an MA model to your Nile data, you're able to capture variation in the data for future prediction. Based on the plot you've generated, does the MA model appear to be a strong fit for the Nile data?

4.4.2 Simple forecasts from an estimated MA model

Now that you've estimated a MA model with your Nile data, the next step is to do some simple forecasting with your model. As with other types of models, you can use the predict() function to make simple forecasts from your estimated MA model. Recall that the $pred value is the forecast, while the $se value is a standard error for that forecast, each of which is based on the fitted MA model.

Once again, to make predictions for several periods beyond the last observation you can use the n.ahead = h argument in your call to predict(). The forecasts are made recursively from 1 to h-steps ahead from the end of the observed time series. However, note that except for the 1-step forecast, all forecasts from the MA model are equal to the estimated mean (intercept).

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

Hide

```
# Make a 1-step forecast based on MA
predict_MA <- predict(MA)

# Obtain the 1-step forecast using $pred[1]
predict_MA$pred[1]

# Make a 1-step through 10-step forecast based on MA
predict(MA, n.ahead = 10)

# Plot the Nile series plus the forecast and 95% prediction intervals
ts.plot(Nile, xlim = c(1871, 1980))
MA_forecasts <- predict(MA, n.ahead = 10)$pred
MA_forecast_se <- predict(MA, n.ahead = 10)$se
points(MA_forecasts, type = "l", col = 2)
points(MA_forecasts - 2*MA_forecast_se, type = "l", col = 2, lty = 2)
points(MA_forecasts + 2*MA_forecast_se, type = "l", col = 2, lty = 2)
```

Note that the MA model can only produce a 1-step forecast. For additional forecasting periods, the predict() command simply extends the original 1-step forecast. This explains the unexpected horizontal lines after 1971.

4.5 Compare AR and MA models

4.5.1 AR vs MA models

As you've seen, autoregressive (AR) and simple moving average (MA) are two useful approaches to modeling time series. But how can you determine whether an AR or MA model is more appropriate in practice?

To determine model fit, you can measure the Akaike information criterion (AIC) and Bayesian information criterion (BIC) for each model. While the math underlying the AIC and BIC is beyond the scope of this course, for your purposes the main idea is these these indicators penalize models with more estimated parameters, to avoid overfitting, and smaller values are preferred. All factors being equal, a model that produces a lower AIC or BIC than another model is considered a better fit.

1 Exploratory time series data analysis

1.1 Introduction

1.2 Sampling frequency

1.3 Basic time series objects

2 Predicting the future

3 Correlation analysis and the autocorrelation function

4 Autoregression

To estimate these indicators, you can use the AIC() and BIC() commands, both of which require a single argument to specify the model in question.

In this exercise, you'll return to the Nile data and the AR and MA models you fitted to this data. These models and their predictions for the 1970s (AR_fit) and (MA_fit) are depicted in the plot on the right.

Hide

```
AR <- arima(Nile, order = c(1, 0, 0))
MA <- arima(Nile, order = c(0, 0, 1))


AR_fit <- Nile - resid(AR)
MA_fit <- Nile - resid(MA)
# Find correlation between AR_fit and MA_fit
cor(AR_fit, MA_fit)

# Find AIC of AR
AIC(AR)

# Find AIC of MA
AIC(MA)

# Find BIC of AR
BIC(AR)

# Find BIC of MA
BIC(MA)
```

Although the predictions from both models are very similar (indeed, they have a correlation coeffiicent of 0.94), both the AIC and BIC indicate that the AR model is a slightly better fit for your Nile data.