Manipulating Time Series Data in R with xts & zoo

Code ▾

Hide

```
library(xts)
library(zoo)
```

1. Introduction to eXtensible Time Series, using xts and zoo for time series

Introducing xts and zoo objects - Video

What is an xts object?

There are many different types of objects in R. With a variety of different features, each has a unique purpose. Some classes inherit behavior from their parents, allowing for custom extensions to existing and well-understood R objects. This makes it easy to adapt existing code to new functionality.

It is for this reason that xts extends the popular zoo class. Knowing this, and what you have seen so far, which of these statements is true?

Possible Answers

- xts objects are matrix objects internally.
- xts objects are indexed by a formal time object.
- Most zoo methods work for xts.
- All of the above. (Correct)

More than a matrix

It is best to think of xts objects as normal R matrices, but with special powers. These powers let you manipulate your data as a function of time, as your data is now self-aware of when it exists in time. Before we can start to exploit these powers, it will be helpful to see how xts objects relate to their base-R relatives.

In this exercise, you will get a feel for xts and how it behaves like a matrix object. The xts object ex_matrix and matrix object core have been pre-loaded for you.

Instructions - Load the xts package using the library() function. - Look at the structure of the sample xts called ex_matrix using str(). - Given that ex_matrix is a matrix, extract the 3rd row and 2nd column. - Now take the matrix core and extract the 3rd row and 2nd column. Notice the difference.

Hide

```
# Load xts
library(xts)

# View the structure of ex_matrix
str(ex_matrix)

# Extract the 3rd observation of the 2nd column of ex_matrix
ex_matrix[3, 2]

# Extract the 3rd observation of the 2nd column of core
core[3, 2]
```

Your first xts object

xts objects are simple. Think of them as a matrix of observations combined with an index of corresponding dates and times.

xts = matrix + times The main xts constructor takes a number of arguments, but the two most important are x for the data and order.by for the index. x must be a vector or matrix. order.by is a vector which must be the same length or number of rows as x, be a proper time or date object (very important!), and be in increasing order.

xts also allows you to bind arbitrary key-value attributes to your data. This lets you keep metadata about your object inside your object. To add these at creation, you simply pass additional name = value arguments to the xts() function.

Since we are focusing here on the mechanics, we'll use random numbers as our data so we can focus on creating the object rather than worry about its contents.

Instructions - Create an object called data that contains five random numbers using rnorm(). - Create a Date class index from "2016-01-01" of length five called dates. - Use the xts constructor to create an object called smith using data and dates as the index. - Create an object called bday which contains a POSIXct date object containing the date - "1899-05-08". - Create an xts object called hayek using data, dates, and a new attribute called born, which should contain the birthday object you just created.

Hide

```
# Create the object data using 5 random numbers
data <- rnorm(5)

# Create dates as a Date class object starting from 2016-01-01
dates <- seq(as.Date("2016-01-01"), length = 5, by = "days")

# Use xts() to create smith
smith <- xts(x = data, order.by = dates)

# Create bday (1899-05-08) using a POSIXct date class object
bday <- as.POSIXct("1899-05-08")

# Create hayek and add a new attribute called born
hayek <- xts(x = data, order.by = dates, born = bday)
```

Deconstructing xts

Now that you can create xts objects, your next task is to examine an xts object from the inside.

At the core of both xts and zoo is a simple R matrix with a few additional attributes. The most important of these attributes is the index. The index holds all the information we need for xts to treat our data as a time series.

When working with time series, it will sometimes be necessary to separate your time series into its core data and index attributes for additional analysis and manipulation. The core data is the matrix portion of xts. You can separate this from the xts object using coredata(). The index portion of the xts object is available using the index() function. Note that both of these functions are methods from the zoo class, which xts extends.

In this exercise you will use these built-in functions to extract both the internal matrix data and internal index from your sample xts object. You will use the hayek time series you created in the last exercise to practice these new functions.

Instructions - Extract the core data of hayek using coredata() and call this hayek_core. - View the class of hayek_core using the class() function. - Extract the date index of hayek using index() and call this hayek_index. - View the class of hayek_index.

Hide

```
# Extract the core data of hayek
hayek_core <- coredata(hayek)

# View the class of hayek_core
class(hayek_core)

# Extract the index of hayek
hayek_index <- index(hayek)

# View the class of hayek_index
class(hayek_index)
```

Time based indices

xts objects get their power from the index attribute that holds the time dimension. One major difference between xts and most other time series objects in R is the ability to use any one of various classes that are used to represent time. Whether POSIXct, Date, or some other class, xts will convert this into an internal form to make subsetting as natural to the user as possible.

a <- xts(x = 1:2, as.Date("2012-01-01") + 0:1) a[index(a)] We'll get into more detail on subsetting xts objects in a later chapter. For now you can simply use date objects to index appropriate rows from your time series. You can think of this as effectively matching the rownames you see in the object. This works as anticipated for time objects because the rownames are really dates!

For this exercise you'll create two time series using two different time classes. You will then subset each object using the other object's index.

Instructions - Create an object of 5 dates called dates starting at "2016-01-01". - Create a time series ts_a using the numbers 1 through 5 as your data, and dates as your - order.by index. - Create a time series ts_b using the numbers 1 through 5 as your data, and the same dates, but - as POSIXct objects. - Use the index from ts_b to extract the dates from ts_a. - Now do the reverse, indexing ts_b using the times from ts_a.

Hide

```
# Create dates
dates <- as.Date("2016-01-01") + 0:4

# Create ts_a
ts_a <- xts(x = 1:5, order.by = dates)

# Create ts_b
ts_b <- xts(x = 1:5, order.by = as.POSIXct(dates))

# Extract the rows of ts_a using the index of ts_b
ts_a[index(ts_a)]

# Extract the rows of ts_b using the index of ts_a
ts_a[index(ts_b)]
```

Importing, exporting and converting time series - Video

Converting xts objects

It is often necessary to convert between classes when working with time series data in R. Conversion can be required for many reasons, but typically you'll be looking to use a function that may not be time series aware or you may want to use a particular aspect of xts with something that doesn't necessarily need to be a full time series.

Luckily, it is quite easy to convert back and forth using the standard as.* style functionality provided in R (for example, as.POSIXct() or as.matrix()).

xts provides methods to convert all of the major objects you are likely to come across. Suitable native R types like matrix, data.frame, and ts are supported, as well as contributed ones such as timeSeries, fts and of course zoo. as.xts() is the workhorse function to do the conversions to xts, and similar functions will provide the reverse behavior.

To get a feel for moving data between classes, let's try a few examples using the Australian population ts object from R named austres.

Instructions Convert the ts class austres data set to an xts and call it au. Then convert the new au xts object into a matrix, am. Inspect the first few entries of this new matrix by using the head() function. Convert the original austres directly into a matrix called am2. Now inspect the first few entries of this new matrix by using the same function. Notice how this time, the conversion didn't preserve the time information?

Hide

```
# Convert austres to an xts object called au
au <- as.xts(austres)

# Then convert your xts object (au) into a matrix am
am <- as.matrix(au)

# Inspect the head of am
head(am)

# Convert the original austres into a matrix am2
am2 <- as.matrix(austres)

# Inspect the head of am2
head(am2)
```

Hide

```
> austres
        Qtr1    Qtr2    Qtr3    Qtr4
1971         13067.3 13130.5 13198.4
1972 13254.2 13303.7 13353.9 13409.3
1973 13459.2 13504.5 13552.6 13614.3
1974 13669.5 13722.6 13772.1 13832.0
1975 13862.6 13893.0 13926.8 13968.9
1976 14004.7 14033.1 14066.0 14110.1
1977 14155.6 14192.2 14231.7 14281.5
1978 14330.3 14359.3 14396.6 14430.8
1979 14478.4 14515.7 14554.9 14602.5
1980 14646.4 14695.4 14746.6 14807.4
1981 14874.4 14923.3 14988.7 15054.1
1982 15121.7 15184.2 15239.3 15288.9
1983 15346.2 15393.5 15439.0 15483.5
1984 15531.5 15579.4 15628.5 15677.3
1985 15736.7 15788.3 15839.7 15900.6
1986 15961.5 16018.3 16076.9 16139.0
1987 16203.0 16263.3 16327.9 16398.9
1988 16478.3 16538.2 16621.6 16697.0
1989 16777.2 16833.1 16891.6 16956.8
1990 17026.3 17085.4 17106.9 17169.4
1991 17239.4 17292.0 17354.2 17414.2
1992 17447.3 17482.6 17526.0 17568.7
1993 17627.1 17661.5

austres <- read.delim("clipboard")
# austres <- as.ts(austres)
austres
```

Importing data

You can now convert data to xts using as.xts(). However, in most real world applications you will often need to read raw data from files on disk or the web. This can be challenging without knowing the right commands.

In the first part of this exercise, you will start by reading a csv file from disk using the base-R read.csv. After you read the data, the next step is to convert it to xts. Here you will be required to use the xts() constructor as well as deal with converting non-standard dates into something that xts understands.

In part two of this exercise, you will read the same data into a zoo object using read.zoo and then convert the zoo object into an xts object.

The data in this exercise are quite simple, but will require some effort to properly import and clean. The full name of the file you will be working with has been saved as the value of tmp_file. On disk, the data look like:

a,b 1/02/2015, 1, 3 2/03/2015, 2, 4

Instructions

- Read the data located at the value of tmp_file using read.csv() to a new variable called dat.
- Convert dat into an xts object using the xts() constructor. Use as.Date() with rownames(dat) - as the first argument.
- Create dat_zoo by using read.zoo() to read in the same tmp_file, and set the argument format - equal to “%m/%d/%Y”.
- Create dat_xts by converting dat_zoo to xts by using as.xts().

Hide

```
tmp_file <- "http://s3.amazonaws.com/assets.datacamp.com/production/course_1127/datasets/tmp_file.csv"
```

Hide

```
# Create dat by reading tmp_file
dat <- read.csv(tmp_file)

# Convert dat into xts
xts(dat, order.by = as.Date(rownames(dat), "%m/%d/%Y"))

# Read tmp_file using read.zoo
dat_zoo <- read.zoo(tmp_file, index.column = 0, sep = ",", format = "%m/%d/%Y")

# Convert dat_zoo to xts
dat_xts <- as.xts(dat_zoo)
```

Exporting xts objects

Now that you can read raw data into xts and zoo objects, it is only natural that you learn how to reverse the process.

There are two main use cases for exporting xts objects. First, you may require an object to persist across sessions for use in later analysis. In this case, it is almost always best to use saveRDS() and readRDS() to serialize single R objects.

Alternatively, you may find yourself needing to share the results of your analysis with others, often expecting the data to be consumed by processes unaware of both R and xts. Most of us would prefer not to think of this horrible fate for our data, but the real world mandates that we at least understand how this works.

One of the best ways to write an xts object from R is to use the zoo function write.zoo(). In this exercise you'll take your temporary data and write it to disk using write.zoo().

Instructions Convert sunspots to xts and save it as sunspots_xts. The temporary file name will be loaded for you. Using write.zoo(), save the sunspots_xts data to the tmp file. Read the tmp file back into R using read.zoo(). Call this sun. Convert sun to xts using the as.xts() function. Call this sun_xts.

Hide

```
data(sunspots)
class(sunspots)
```

Hide

```
# Convert sunspots to xts using as.xts().
sunspots_xts <- as.xts(sunspots)

# Get the temporary file name
tmp <- tempfile()

# Write the xts object using zoo to tmp
write.zoo(sunspots_xts, sep = ",", file = tmp)

# Read the tmp file. FUN = as.yearmon converts strings such as Jan 1749 into a proper time class
sun <- read.zoo(tmp, sep = ",", FUN = as.yearmon)

# Convert sun into xts. Save this as sun_xts
sun_xts <- as.xts(sun)
```

2. First Order of Business - Basic Manipulations

Introducing time based queries - Video

The ISO-8601 standard

The ISO-8601 standard is the internationally recognized and accepted way to represent dates and times. The standard allows for a common format to not only describe dates, but also to represent ranges and repeating intervals.

xts makes use of this standard for all extract and replace operations. This makes code both easy to write and easy to maintain. It also makes for very concise expression of date ranges and intervals.

For xts to work correctly, it is very important to follow the standard exactly. Details can be found in xts subset and .parseISO8601 documentation.

Which is a valid ISO-8601 string acceptable by xts?

Possible Answers

- “2005-01-02”
- “200501”
- “2010/02/22”
- 1. and (2) only (Correct)
- all of the above

Querying for dates

One of the most powerful aspects of working with time series in xts is the ability to quickly and efficiently specify dates and time ranges for subsetting.

Date ranges can be extracted from xts objects by simply specifying the period(s) you want using special character strings in your subset.

A[“20090825”] ## Aug 25, 2009 A[“201203/201212”] ## Mar to Dec 2012 A[“/201601”] ## Up to and including January 2016 For this exercise you will create a simple but very common query. Extract a range of dates using the ISO-8601 feature of xts. After successfully extracting a full year, you will then create a subset of your new object with specific start and end dates using this same notation.

Let’s find some time!

Instructions - Using xts-style time subsetting, select only the year 2016 from the x time series, and call - this x_2016. - Using an explicit start and end date string, get all data from January 1, 2016 through March - 22, 2016. Call this jan_march. - To ensure that you subset all 82 rows, use the length function.

Hide

```
# Select all of 2016 from x
x_2016 <- x["2016"]

# Select January 1, 2016 to March 22, 2016
jan_march <- x["20160101/20160322"]

# Verify that jan_march contains 82 rows
82 == length(jan_march)
```

Extracting recurring intraday intervals

The most common time series data “in the wild” is daily. On occasion, you may find yourself working with intraday data, which contains both dates and times. In this case it is sometimes necessary to view only a subset of time for each day over multiple days. Using xts, you can slice days easily by using special notation in the i = argument to the single bracket extraction (i.e. [i, j]).

As you learned in the previous exercise, the trick to this is to not specify explicit dates, but rather to use the special T/T notation designed for intraday repeating intervals.

Intraday times for all days NYSE[“T09:30/T16:00”]

In this exercise, you will extract recurring morning hours from the time series irreg, which holds irregular data from the month of January 2010. Remember, you can always use the R console to experiment with irreg or to view the help pages with ?xts.

Instructions Using the irregular time series irreg, assign all observations between 8AM and 10AM to morn_2010. Using morn_2010, extract only those observations from the morning of January 13th, 2010.

Hide

```
# Extract all data from irreg between 8AM and 10AM
morn_2010 <- irreg["T08:00/T10:00"]

# Extract the observations in morn_2010 for January 13th, 2010
morn_2010["20100113"]
```

Alternative extraction techniques - Video

Row selection with time objects

Often you may need to subset an existing time series with a set of Dates, or time-based objects. These might be from as.Date(), as.POSIXct(), or a variety of other classes. In this exercise you'll explore how, given an xts object x, it is possible to extract relevant observations using a vector of dates in brackets.

The objects x and dates have been pre-loaded in your workspace.

Instructions - Use the vector dates to subset the object x. - Subset x using dates that are first converted to POSIXct.

Hide

```
# Subset x using the vector dates
x[dates]

# Subset x using dates as POSIXct
x[as.POSIXct(dates)]
```

Update and replace elements

Replacing values in xts objects is just as easy as extracting them. You can use either ISO-8601 strings, date objects, logicals, or integers to locate the rows you want to replace. One reason you may want to do this would be to replace known intervals or observations with NA, say due to a malfunctioning sensor on a particular day or a set of outliers given a holiday.

For individual observations located sporadically throughout your data dates, integers or logical vectors are a great choice. For continuous blocks of time, ISO-8601 is the preferred method.

In this exercise, you'll continue using the vector dates from the previous exercise to modify your x object. Both are already loaded in your workspace.

Instructions - Use the vector dates to replace values in x with NA values. - Replace all values in x for dates (not referring to the vector) from June 9, 2016 onward with - 0. Use ISO-8601 style replacement. - Use the console to look at the value for June 11, 2016 to show that your hard work has paid off!

Hide

```
# Replace the values in x contained in the dates vector with NA
x[dates] <- NA

# Replace all values in x for dates starting June 9, 2016 with 0
x["2016-06-09/"] <- 0

# Verify that the value in x for June 11, 2016 is now indeed 0
x["20160611"]
```

Additional Methods To Find Periods in Your Data - Video

Find the first or last period of time

Sometimes you need to locate data by relative time. Something that is easier said than put into code. This is equivalent to requesting the head or tail of a series, but instead of using an absolute offset, you describe a relative position in time. A simple example would be something like the last 3 weeks of a series, or the first day of current month.

Without a time aware object, this gets quite complicated very quickly. Luckily, xts has the necessary prerequisites built in for you to use with very little learning required. Using the first() and last() functions it is actually quite easy!

For this exercise, you'll extract relative observations from a data set called temps, a time series of summer temperature data from Chicago, IL, USA.

Instructions - Create a new variable lastweek by extracting the last 1 week from temps. - Using your newly created data, extract the last 2 observations without respect to time. - Now extract every day from lastweek except the first two days (this is tricky!).

Hide

```
# Create lastweek using the last 1 week of temps
lastweek <- last(temps, "1 week")

# Print the last 2 observations in lastweek
last(lastweek, 2)

# Extract all but the first two days of lastweek
first(lastweek, "-2 days")
```

Combining first and last

Now that you have seen how to extract the first or last chunk of a time series using natural looking language, it is only a matter of time before you need to get a bit more complex.

In this exercise, you’ll extract a very specific subset of observations by linking together multiple calls to first() and last().

Hide

```
# Last 3 days of first week
last(first(Temps, '1 week'), '3 days')
```

You will reconfigure the example above using the temps data from the previous exercise. The trick to using such a complex command is to work from the inside function, out.

Instructions - Find the first three days of the second week of the temps data set. Use combinations of first() and last() to do this.

Hide

```
# Extract the first three days of the second week of temps
first(last(first(temps, "2 weeks"), "1 week"), "3 days")
```

Math operations in xts - Video

Matrix arithmetic - add, subtract, multiply, and divide in time!

xts objects respect time. By design when you perform any binary operation using two xts objects, these objects are first aligned using the intersection of the indexes. This may be surprising when first encountered.

The reason for this is that you want to preserve the point-in-time aspect of your data, assuring that you don’t introduce accidental look ahead (or look behind!) bias into your calculations.

What this means in practice is that you will sometimes be tasked with handling this behavior if you want to preserve the dimensions of your data.

Your options include:

Use coredata() or as.numeric() (drop one to a matrix or vector). Manually shift index values - i.e. use lag(). Reindex your data (before or after the calculation). In this exercise, you’ll look at the normal behavior, as well as an example using the first option. For now you will use two small objects a and b. Examine these objects in the console before you start.

Instructions Add a and b. Notice the behavior of the dates, which ones remain? Add a with the numeric value of b. b will need to be converted to a numeric for this to work.

Hide

```
# Add a and b
a+b

# Add a with the numeric value of b
a + as.numeric(b)
```

Math with non-overlapping indexes

The previous exercise illustrated the ins and outs of doing basic math with xts objects. At this point you are aware that xts respects time and will only return the intersection of times when doing various mathematical operations.

We alluded to another way to handle this behavior in the last exercise. Namely, re-indexing your data before an operation. This makes it possible to preserve the dimensions of your data by leveraging the same mechanism that xts uses internally in its own Ops method (the code dispatched when you call + or similar).

The third way involves modifying the two series you want by assuring you have some union of dates - the dates you require in your final output. To do this you will need a few functions that won’t be dealt with in depth until Chapter 3, but are very useful here.

merge(b, index(a)) Don’t worry if you aren’t yet familiar with merge(). This exercise may be easier if you just follow along with the instructions.

Instructions Using a and b from the previous exercise, get the value of a + b for each date in a. If no b is available on a given date, the answer should be a on that date. Now add a to b, but this time make sure all values of a are added to the last known value of b in time.

Hide

```
# Add a to b, and fill all missing rows of b with 0
a + merge(b, index(a), fill = 0)

# Add a to b and fill NAs with the last observation
a + merge(b, index(a), fill = na.locf)
```

3. Merging and modifying time series

Merging and modifying time series - Video

Combining xts by column with merge

xts makes it easy to join data by column and row using a few different functions. All results will be correctly ordered in time, regardless of original frequencies or date class. One of the most important functions to accomplish this is merge(). It takes one or more series and joins them by column. It’s also possible to combine a series with a vector of dates. This is especially useful for normalizing observations to a fixed calendar.

merge() takes three key arguments which we will emphasize here. First is the …, which lets you pass in an arbitrary number of objects to combine. The second argument is join, which specifies how to join the series - accepting arguments such as inner or left. This is similar to a relational database join, only here, the index is what we join on. The final argument for this exercise is fill. This keyword specifies what to do with the new values in a series if there is missingness introduced as a result of the merge.

Hide

```
# Basic argument use
merge(a, b, join = "right", fill = 9999)
```

For this exercise, you will explore some of the different join types to get a feel for using merge(). The objects a and b have been pre-loaded into your workspace.

Instructions - Merge a and b using merge() (or cbind()), with the argument join set to “inner”. - Perform a left-join of a and b. Use merge() and set the argument join to the correct value. Fill all missing values with zero (use the fill argument).

Hide

```
# Perform an inner join of a and b
merge(a, b, join = "inner")

# Perform a left-join of a and b, fill missing values with 0
merge(a, b, join = "left", fill = 0)
```

Merging xts objects by column comes in handy when preparing data for time series analysis.

Combining xts by row with rbind

Now that you have merged data by column, you will be happy to know it’s just as easy to add new rows to your data.

xts provides its own S3 method to the base rbind() generic function. The xts rbind function is much simpler than merge(). The only argument that matters is …, which takes an arbitrary number of objects to bind. What is different is that rbind requires a time series, since we need to have timestamps for R to know where to insert new data.

For this exercise you will update your temps data with three new observations. One will be before the series started and two will be after. Pay attention to your function call, does order matter?

In your workspace, the objects temps, temps_june30, temps_july17 and temps_july18 are already loaded.

Instructions - Bind the row from June 30th (temps_june30) to temps, and call this temps2. - Bind the rows from July 17th and 18th to temps2. Call this temps3.

Hide

```
# Row bind temps_june30 to temps, assign this to temps2
temps2 <- rbind(temps, temps_june30)

# Row bind temps_july17 and temps_july18 to temps2, call this temps3
temps3 <- rbind(temps_july17, temps_july18, temps2)
```

Because xts objects are ordered by their time index, the order of arguments in xts’s rbind() command is unimportant.

What types of data can be combined using merge?

Although xts is very flexible when it comes to binding new columns to existing objects, there are still some exceptions. Which of the following is not able to be successfully merged with the xts object a?

Before you answer, try each possibility in the console to your right to see how they behave. We’ve provided two time series, a and b for you to work with.

Possible Answers - xts objects of identical type (e.g. integer + integer). - data.frames with various column types. (Correct) - POSIXct dates vector. - Atomic vectors of the same type (e.g. numeric). - A single NA.

Handling missingness in your data - Video

Fill missing values using last or previous observation

As you’ve encountered already, it’s not uncommon to find yourself with missing values (i.e. NAs) in your time series. This may be the result of a data omission or some mathematical or merge operation you do on your data.

The xts package leverages the power of zoo for help with this. zoo provides a variety of missing data handling functions which are usable by xts.

In this exercise you will use the most basic of these, na.locf(). This function takes the last observation carried forward approach. In most circumstances this is the correct thing to do. It both preserves the last known value and prevents any look-ahead bias from entering into the data.

You can also apply next observation carried backward by setting fromLast = TRUE.

Hide

```
# Last obs. carried forward
na.locf(x)

# Next obs. carried backward
na.locf(x, fromLast = TRUE)
```

Instructions

- Using a subset of temps, fill missing NA observations with the last observation known. Store - them in temps_last.
- Using another subset of temps, backfill missing NA observations with the next observation. - Store them in temps_next.

Hide

```
# Fill missing values in temps using the last observation
temps_last <- na.locf(temps)

# Fill missing values in temps using the next observation
temps_next <- na.locf(temps, fromLast=TRUE)
```

NA interpolation using na.approx()

On occasion, a simple carry forward approach to missingness isn’t appropriate. It may be that a series is missing an observation due to a higher frequency sampling than the generating process. You might also encounter an observation that is in error, yet expected to be somewhere between the values of its neighboring observations.

These are scenarios where interpolation is useful. zoo provides a powerful tool to do this. Based on simple linear interpolation between points, implemented with na.approx() the data points are approximated using the distance between the index values. In other words, the estimated value is linear in time.

For this exercise, you'll use a smaller xts version of the Box and Jenkin's AirPassengers data set that ships with R. We've removed a few months of data to illustrate various fill techniques.

One takeaway, aside from getting a feel for the functions, is to see how various fill techniques impact your data, and especially how it will impact your understanding of it.

Caveat Emptor!

The AirPassengers data set is available in your workspace as AirPass.

Instructions - Fill in missing months in AirPass with linear interpolation using na.approx().

Hide

```
# Interpolate NAs using linear approximation
na.approx(AirPass)
```

Linear interpolation is a straightforward way to account for missingness, although it is up to you to determine its applicability.

Lag operators and difference operations

Combine a leading and lagging time series

Another common modification for time series is the ability to lag a series. Also known as a backshift operation, it's typically shown in literature using LkLk notation, indicating a transformation in time LkX=Xt???kLkX=Xt???k. This lets you see observations like yesterday's value in the context of today.

Both zoo and xts implement this behavior, and in fact extend it from the ts original in R. There are two major differences between xts and zoo implementations that you need to be aware of. One is the direction of the lag for a given k. The second is how missingness is handled afterwards.

For historical reasons in R, zoo uses a convention for the sign of k in which negative values indicate lags and positive values indicate leads. That is, in zoo lag(x, k = 1) will shift future values one step back in time. This is inconsistent with the vast majority of the time series literature, but is consistent with behavior in base R. xts implements the exact opposite, namely for a positive k, the series will shift the last value in time one period forward; this is consistent with intuition, but quite different than zoo.

In this exercise, you will construct a single xts object with three columns. The first column is data one day ahead, the second column is the original data, and the third column is the one day behind - all using xts. A simple xts object, x, has been loaded into your workspace.

Hide

```
# Your final object
cbind(lead_x, x, lag_x)
```

Instructions - Create a one period lead of x called lead_x. - Create a one period lag of x called lag_x. - Using the merge() function, combine lead + x + lag into a new object z.

Hide

```
# Create a leading object called lead_x
lead_x <- lag(x, k = -1)

# Create a lagging object called lag_x
lag_x <- lag(x, k = 1)

# Merge your three series together and assign to z
z = cbind(lead_x, x, lag_x)
```

Generating leads and lags can help you visualize trends in your time series data over time.

Calculate a difference of a series using diff()

Another common operation on time series, typically on those that are non-stationary, is to take a difference of the series. The number of differences to take of a series is an application of recursively calling the difference function n times.

A simple way to view a single (or “first order”) difference is to see it as x(t) - x(t-k) where k is the number of lags to go back. Higher order differences are simply the reapplication of a difference to each prior result.

In R, the difference operator for xts is made available using the diff() command. This function takes two arguments of note. The first is the lag, which is the number of periods, and the second is differences, which is the order of the difference (e.g. how many times diff() is called).

Hide

```
# These are the same
diff(x, differences = 2)
diff(diff(x))
```

In this exercise, you will reuse the AirPass data from earlier in this chapter, though this time you will use the full series from 1948 to 1960.

Instructions

- Construct a first order difference of AirPass by hand, using lag() and subtraction. Save this - as diff_by_hand.
- To verify that your result is identical to using diff(AirPass), combine and inspect the first - few rows of both in your console. Use merge() and head() for this.
- Get the first order 12 month difference of series AirPass. Be sure to specify both the lag and differences arguments in diff().

Hide

```
# Calculate the first difference of AirPass and assign to diff_by_hand
diff_by_hand <- AirPass - lag(AirPass, k=1)

# Use merge to compare the first parts of diff_by_hand and diff(AirPass)
merge(head(diff_by_hand), head(diff(AirPass)))

# Calculate the first order 12 month difference of AirPass
diff(AirPass, lag = 12, differences = 1)
```

As you can see, differencing your series is only one step more complex than generating lags and leads.

What is the key difference in lag between xts and zoo

As you've seen, generating lags and leads are an important tool in your arsenal for handling time series data in R. However, because xts employs slightly different procedures for generating lags and leads compared with zoo of base-R, you have to be very precise in your calls to lag().

Which of the following is FALSE?

Possible Answers - The NA observations in xts resulting from lag() are preserved. - The NA observations in zoo resulting from lag() are dropped. - The k argument in zoo uses positive values for shifting past observations forward. (Correct) - The k argument in xts uses positive values for shifting past observations forward.

xts follows the literature, while zoo continues with the norms of base-R.

4. Apply and aggregate by time

Apply by Time - Video

> temps Temp.Max Temp.Mean Temp.Min 2016-07-01 74 69 60 2016-07-02 78 66 56 2016-07-03 79 68 59 2016-07-04 80 76 69 2016-07-05 90 79 68 2016-07-06 89 79 70 2016-07-07 87 78 72 2016-07-08 89 80 72 2016-07-09 81 73 67 2016-07-10 83 72 64 2016-07-11 93 81 69 2016-07-12 89 82 77 2016-07-13 86 78 68 2016-07-14 89 80 68 2016-07-15 75 72 60 2016-07-16 79 69 60

Find intervals by time in xts

One of the benefits to working with time series objects is how easy it is to apply functions by time.

The main function in xts to facilitate this is endpoints(). It takes a time series (or a vector of times) and returns the locations of the last observations in each interval.

For example, the code below locates the last observation of each year for the AirPass data set.

endpoints(AirPass, on = “years”) [1] 0 12 24 36 48 60 72 84 96 108 120 132 144 The argument on supports a variety of periods, including “years”, “quarters”, “months”, as well as intraday intervals such as “hours”, and “minutes”. What is returned is a vector starting with 0 and ending with the extent (last row) of your data.

In addition to each period, you can find the KthKth period by utilizing the k argument. For example, setting the arguments of your endpoints() call to on = “weeks”, k = 2, would generate the final day of every other week in your data. Note that the last value returned will always be the length of your input data, even if it doesn’t correspond to a skipped interval.

In this exercise you’ll use endpoints() to find two sets of endpoints for the daily temps data.

Instructions - Use endpoints() to locate the end of week in your temps data. - Use endpoints() to locate the end of every second week in your temps data. Remember to use the k argument.

Hide

```
# Locate the weeks
endpoints(temps, on = "weeks")

# Locate every two weeks
endpoints(temps, on = "weeks", k = 2)
```

As you’ll see in the next exercise, locating endpoints can help speed the process of aggregating time series data.

Apply a function by time period(s)

At this point you know how to locate the end of periods using endpoints(). You may be wondering what it is you do with these values.

In the most simple case you can subset your object to get the last values. In certain cases this may be useful. For example, to identify the last known value of a sensor during the hour or get the value of the USD/JPY exchange rate at the start of the day. For most series, you will want to apply a function to the values between endpoints. In essence, use the base function apply(), but used on a window of time.

To do this easily, xts provides the period.apply() command, which takes a time series, an index of endpoints, and a function.

period.apply(x, INDEX, FUN, …) In this exercise you'll practice using period.apply() by taking the weekly mean of your temps data. You'll also look at one of the shortcut functions that does the same thing with slightly different syntax.

Instructions - Calculate the weekly endpoints of the temps series and assign to ep. - Using period.apply(), calculate the weekly mean of the Temp.Mean column of your temps data. Remember that your endpoints were calculated weekly.

Hide

```
# Calculate the weekly endpoints
ep <- endpoints(temps, on = "weeks")

# Now calculate the weekly mean and display the results
period.apply(temps[, "Temp.Mean"], INDEX = ep, FUN = mean)
```

```
       Temp.Mean
```

2016-07-03 67.66667 2016-07-10 76.71429 2016-07-16 77.00000

The period.apply() command allows you to easily calculate complex qualities of your time series data.

Using lapply() and split() to apply functions on intervals

Along the same lines as the previous exercise, xts gives you an additional mechanism to dive into periods of your data. Often it is useful to physically split your data into disjoint chunks by time and perform some calculation on these periods.

For this exercise you'll make use of the xts split() command to chunk your data by time. The split() function creates a list containing an element for each split. The f argument in split() is a character string describing the period to split by (i.e. "months", "years", etc.).

Here you will follow the same process you followed in the previous exercise. However, this time you will manually split your data first, and then apply the mean() function to each chunk. The function lapply() is used for the most efficient calculations. In cases where you don’t want to return a time series, this proves to be very intuitive and effective.

Instructions - Use split() to split your temps data by weeks. Call this temps_weekly - Use lapply() to get the weekly mean of temps_weekly. Call this temps_avg. Print this list.

Hide

```
# Split temps by week
temps_weekly <- split(temps, f = "weeks")

# Create a list of weekly means, temps_avg, and print this list
temps_avg <- lapply(X = temps_weekly, FUN = mean)
temps_avg
```

As you can see, period.apply() is similar to using a combination of split() and lapply()

Selection by endpoints vs. split-lapply-rbind

By now you have seen that even in xts there is more than one way to accomplish a task. In this exercise we’ll highlight this explicitly by tackling the same challenge using two different methods. When you are on your own, you will likely find situations where one or the other will be more intuitive, but for now you should make sure you are able to do both.

Starting with the same daily series temps, the challenge will be to find the last observation in each week.

Note that these functions will always find the dates that are in the closed interval [start of period, end of period] even if there is no observation at the exact start or end. xts represents irregular time series, so it is perfectly valid to have holes in the data where one might expect an observation.

Using the slides and video examples as a reference, find the last observation for each week in our temps data.

Instructions Use the split()-lapply()-rbind() paradigm, given for you in the script, to find the last observation in each week in temps. It is stored in temps_1. Use endpoints() to find the last days of the week in temps. Store the resulting vector in last_day_of_weeks. Create temps_2 by subsetting temps using last_day_of_weeks.

Hide

```
# Use the proper combination of split, lapply and rbind
temps_1 <- do.call(rbind, lapply(split(temps, "weeks"), function(w) last(w, n = "1 day")))

# Create last_day_of_weeks using endpoints()
last_day_of_weeks <- endpoints(temps, "weeks")

# Subset temps using last_day_of_weeks
temps_2 <- temps[last_day_of_weeks]
```

Converting periodicity - Video

Convert univariate series to OHLC data

Aggregating time series can be a frustrating task. For example, in financial series it is common to find Open-High-Low-Close data (or OHLC) calculated over some repeating and regular interval.

Also known as range bars, aggregating a series based on some regular window can make analysis easier amongst series that have varying frequencies. A weekly economic series and a daily stock series can be compared more easily if the daily is converted to weekly.

In this exercise, you'll convert from a univariate series into OHLC series, and then convert your final OHLC series back into a univariate series using the xts function to.period(). This function takes a time-series, x, and a string for the period (i.e. months, days, etc.), in addition to a number of other optional arguments.

to.period(x, period = "months", k = 1, indexAt, name=NULL, OHLC = TRUE, …) You will use a new data set for this exercise, usd_eur, a daily USD/EUR exchange rate from 1999 to August 2016, which has been loaded into your workspace.

Instructions - Convert usd_eur into a weekly OHLC series using to.period(). Call this new series usd_eur_weekly. Note that by default OHLC = TRUE. - Convert usd_eur into a monthly OHLC series. Call this series usd_eur_monthly. - Convert usd_eur into a yearly univariate (no OHLC bars) series. Call this series usd_eur_yearly.

Hide

```
# Convert usd_eur to weekly and assign to usd_eur_weekly
usd_eur_weekly <- to.period(usd_eur, period = "weeks")

# Convert usd_eur to monthly and assign to usd_eur_monthly
usd_eur_monthly <- to.period(usd_eur, period = "months")

# Convert usd_eur to yearly univariate and assign to usd_eur_yearly
usd_eur_yearly <- to.period(usd_eur, period = "years", OHLC = FALSE)
```

Convert a series to a lower frequency

Besides converting univariate time series to OHLC series, to.period() also lets you convert OHLC to lower regularized frequency - something like subsampling your data.

Depending on the chosen frequency, the index class of your data may be coerced to something more appropriate to the new data. For example, when using the shortcut function to.quarterly(), xts will convert your index to the yearqtr class to make periods more obvious.

We can override this behavior by using the indexAt argument. Specifically, using firstof would give you the time from the beginning of the period. In addition, you can change the base name of each column by supplying a string to the argument name.

For this exercise we’ll introduce a new dataset, the edhec hedge fund index data from the PerformanceAnalytics package.

In this exercise you will use the Equity Market Neutral time series from the edhec data, which we’ve assigned to eq_mkt.

Instructions - Convert eq_mkt to quarterly OHLC using the base to.period(). Call this mkt_quarterly. - Convert the original eq_mkt again, this time using to.quarterly() directl y. Change the base name of each OHLC column to edhec_equity and change the index to “firstof”. Call this mkt_quarterly2.

Hide

```
install.packages("PerformanceAnalytics")
library(PerformanceAnalytics)
data(edhec)
dim(edhec)
names(edhec)
eq_mkt <- edhec[, "Equity Market Neutral"]
dim(eq_mkt)
```

Hide

```
# Convert eq_mkt to quarterly OHLC
mkt_quarterly <- to.period(eq_mkt, period = "quarters")

# Convert eq_mkt to quarterly using shortcut function
mkt_quarterly2 <- to.quarterly(edhec, name = "edhec_quity", indexAt = "firstof")
```

Rolling functions - apply windowing functions to time series data - Video

Calculate basic rolling value of series by month

One common aggregation you may want to apply involves doing a calculation within the context of a period, but returning the interim results for each observation of the period.

For example, you may want to calculate a running month-to-date cumulative sum of a series. This would be relevant when looking at monthly performance of a mutual fund you are interested in investing in.

For this exercise, you'll calculate the cumulative annual return using the edhec fund data from the last exercise. To do this, you'll follow the split()-lapply()-rbind() pattern demonstrated below:

x_split <- split(x, f = "months") x_list <- lapply(x_split, cummax) x_list_rbind <- do.call(rbind, x_list) Note the last call uses R's somewhat strange do.call(rbind, …) syntax, which allows you to pass a list to rbind instead of passing each object one at a time. This is a handy shortcut for your R toolkit.

Instructions - Using split(), break up the edhec data into years. Assign this to edhec_years. - Use lapply() to find the cumsum() of the returns per year on edhec_years. Assign this to edhec_ytd. - Use do.call() with rbind to convert your previous list output to a single xts object. Assign this to edhec_xts.

Hide

```
# Split edhec into years
edhec_years <- split(edhec , f = "years")

# Use lapply to calculate the cumsum for each year in edhec_years
edhec_ytd <- lapply(edhec_years, FUN = cumsum)

# Use do.call to rbind the results
edhec_xts <- do.call(rbind, edhec_ytd)
```

Calculate the rolling standard deviation of a time series

Another common requirement when working with time series data is to apply a function on a rolling window of data. xts provides this facility through the intuitively named zoo function rollapply().

This function takes a time series object x, a window size width, and a function FUN to apply to each rolling period. The width argument can be tricky; a number supplied to the width argument specifies the number of observations in a window. For instance, to take the rolling 10-day max of a series, you would type the following:

rollapply(x, width = 10, FUN = max, na.rm = TRUE) Note that the above would only take the 10-day max of a series with daily observations. If the series had monthly observations, it would take the 10-month max. Also note that you can pass additional arguments (i.e. na.rm to the max function) just like you would with apply().

Instructions - Using rollapply(), calculate the 3-month standard deviation of the eq_mkt series. Note that eq_mkt has monthly observations. Call your 3-month rolling standard deviation eq_sd.

Hide

```
# Use rollapply to calculate the rolling 3 period sd of eq_mkt
eq_sd <- rollapply(eq_mkt, width=3, FUN = sd)
```

Rolling values are a useful metric in time series data. Now that you are familiar with the core features of the xts package, the final chapter will explore a few more advanced topics using xts.

5. Extra features of xts

Index, Attributes, and Timezones - Video

Time via index() 50xp For this multiple choice question, you will use the pre-loaded temps data to help you find the correct answer.

Which of the following statements is false?

Possible Answers

- Using the tclass() function on temps returns the same output as does indexClass().
- indexFormat(temps) <- “%b %d, %Y” changes the index of the first entry of the data to Jul 01, 2016.
- Typing help(OlsonNames) into the console will provide R documentation for time zones.
- The time zone of the temps data set is set to “America/New_York”. (Correct)

Class attributes - tclass, tzone, and tformat

xts objects are somewhat tricky when it comes to time. Internally, we have now seen that the index attribute is really a vector of numeric values corresponding to the seconds since the UNIX epoch (1970-01-01).

How these values are displayed on printing and how they are returned to the user when using the index() function is dependent on a few key internal attributes.

The information that controls this behavior can be viewed and even changed through a set of accessor functions detailed here:

The index class using indexClass() (e.g. from Date to chron) The time zone using indexTZ() (e.g. from America/Chicago to Europe/London) The time format to be displayed via indexFormat() (e.g. YYYY-MM-DD) In this exercise, you will practice each of these functions and view the results of your changes. To do so, you’ll once again use the temps data that has been pre-loaded into your workspace.

Instructions View the first three rows of the index in the current temps data. Find the index class of temps using the most relevant command above. Find the time zone of temps, again using the most relevant command above. Change the index format of temps to “%b-%d-%Y”. View the new index format using head().

Hide

```
# View the first three indexes of temps
index(temps)[1:3]

# Get the index class of temps
indexClass(temps)

# Get the timezone of temps
indexTZ(temps)

# Change the format of the time display
indexFormat(temps) <- "%b-%d-%Y"

# View the new format
head(temps)
```

Time Zones (and why you should care!)

One of the trickiest parts to working with time series in general is dealing with time zones. xts provides a simple way to leverage time zones on a per-series basis. While R provides time zone support in native classes POSIXct and POSIXlt, xts extends this power to the entire object, allowing you to have multiple time zones across various objects.

Some internal operation system functions require a time zone to do date math. If a time zone isn’t explicitly set, one is chosen for you! Be careful to always set a time zone in your environment to prevent errors when working with dates and times.

xts provides the function tzone(), which allows you to extract or set time zones.

tzone(x) <- “Time_Zone” In this exercise you will work with an object called times to practice constructing your own xts objects with custom time zones.

Instructions

- Construct an xts time series of the numbers 1 through 10 called times_xts, with tzone set to “America/Chicago”, and indexed by the times object.
- Modify times_xts to show time in “Asia/Hong_Kong”.
- Extract the current time zone as a string.

Hide

```
# Construct times_xts with tzone set to America/Chicago
times_xts <- xts(1:10, order.by = times, tzone = "America/Chicago")

# Change the time zone of times_xts to Asia/Hong_Kong
tzone(times_xts) <- "Asia/Hong_Kong"

# Extract the current time zone of times_xts
indexTZ(times_xts)
```

Periods, Periodicity and Timestamps - Video

Determining periodicity 100xp The idea of periodicity is pretty simple: With what regularity does your data repeat? For stock market data, you might have hourly prices or maybe daily open-high-low-close bars. For macroeconomic series, it might be monthly or weekly survey numbers.

xts provides a handy tool to discover this regularity in your data by estimating the frequency of the observations - what we are referring to as periodicity - using the periodicity() command

In this exercise, you'll try this out on a few sample data sets. In real life you might find yourself doing this as a first step to understanding your data before diving in for further analysis.

Instructions Calculate the periodicity of the temps data set. Calculate the periodicity of the edhec data set. Convert the edhec data to yearly periodicity using to.yearly(). Call this edhec_yearly. Calculate the periodicity of edhec_yearly.

Hide

```
# Calculate the periodicity of temps
periodicity(temps)

# Calculate the periodicity of edhec
periodicity(edhec)

# Convert edhec to yearly
edhec_yearly <- to.yearly(edhec)

# Calculate the periodicity of edhec_yearly
periodicity(edhec_yearly)
```

The periodicity() command combined with the to.period() set of commands gives you a simple way to manipulate your time series data.

Find the number of periods in your data

Often it is handy to know not just the range of your time series index, but also how many discrete irregular periods your time series data covers. You shouldn’t be surprised to learn that xts provides a set of functions to do just that!

If you have a time series, it is now easy to see how many days, weeks or years your data contains. To do so, simply use the function ndays() and its shortcut functions nmonths(), nquarters(), and so forth, making counting irregular periods easy.

Instructions

- Count the months in edhec.
- Count the quarters in edhec.
- Count the years in edhec.

Hide

```
# Count the months
nmonths(edhec)

# Count the quarters
nquarters(edhec)

# Count the years
nyears(edhec)
```

Secret index tools

xts uses a very special attribute called index to provide time support to your objects. For performance and design reasons, the index is stored in a special way. This means that regardless of the class of your index (e.g. Date or yearmon) everything internally looks the same to xts. The raw index is actually a simple vector of fractional seconds since the UNIX epoch.

Normally you want to access the times you stored. index() does this magically for you by using your indexClass. To get to the raw vector of the index, you can use .index(). Note the critical dot before the function name.

More useful than extracting raw seconds is the ability to extract time components similar to the POSIXlt class, which closely mirrors the underlying POSIX internal compiled structure tm. This functionality is provided by a handful of commands such as .indexday(), .indexmon(), .indexyear(), and more.

In this exercise, you’ll take a look at the weekend weather in our pre-loaded temps data using the .indexwday() command. Note that the values range from 0-6, with Sunday equal to 0. Recall that you can use a logical vector to extract elements of an xts object.

Instructions

- Practice extracting the underlying units of your index in the temps data. Use .index() to see the number of seconds, and .indexwday() to see the day of the week of your observations.
- Create index using the which() function to extract weekend observations in temps.
- Select the indexed values from temps.

Hide

```
# Explore underlying units of temps in two commands: .index() and .indexwday()
.index(temps)
.indexwday(temps)


# Create an index of weekend days using which()
index <- which(.indexwday(temps) == 0 | .indexwday(temps) == 6)

# Select the index
temps[index]
```

Modifying timestamps

Most time series we’ve seen have been daily or lower frequency. Depending on your field, you might encounter higher frequency data - think intraday trading intervals, or sensor data from medical equipment.

In these situations, there are two functions in xts that are handy to know.

If you find that you have observations with identical timestamps, it might be useful to perturb or remove these times to allow for uniqueness. xts provides the function make.index.unique() for just this purpose. The eps argument, short for epsilon or small change, controls how much identical times should be perturbed, and drop = TRUE lets you just remove duplicate observations entirely.

On other ocassions you might find your timestamps a bit too precise. In these instances it might be better to round up to some fixed interval, for example an observation may occur at any point in an hour, but you want to record the latest as of the beginning of the next hour. For this situation, the align.time() command will do what you need, setting the n argument to the number of seconds you’d like to round to.

make.index.unique(x, eps = 1e-4) # Perturb make.index.unique(x, drop = TRUE) # Drop duplicates align.time(x, n = 60) # Round to the minute In this exercise, you’ll try the three use cases on an xts object called z.

Instructions - Convert the z series times to a unique series using make.index.unique(), where eps = 1e-4. Save this to z_unique. - Remove duplicate times in z. Save this to z_dup. - Align z to the nearest hour using align.time(). Save this to z_round.

Hide

```
# Make z have unique timestamps
z_unique <- make.index.unique(z, eps = 1e-4)

# Remove duplicate times in z
z_dup <- make.index.unique(z, drop = TRUE)

# Round observations in z to the next hour
z_round <- align.time(z, n = 3600)
```

These final commands should round out your xts knowledge and give you the complete tools to manipulate time series data in R.