Raster Time Series

Ben DeVries, Jan Verbesselt

November 5, 2013

1 L

earning Objectives The learning objectives for this lesson are:

1. working with time series rasterBricks
2. parsing Landsat scene information
3. interactively extracting data from raster pixels
4. plotting time series data
5. using spatial joins to summarize raster time series data

2 Working with raster time series

Since the opening of the Landsat data archive, image time series analysis has become a real possibility. In environmental research and monitoring, methods and tools for time series analysis are becoming increasingly important. With the brick and stack object types, the raster package in R allows for certain types of raster-based time series analysis, and allows for integration with other time series packages in R.

Let's start by loading the necessary packages.

A raster brick from a small area within the Kafa Biosphere Reserve in Southern Ethiopia can be found in the rasta package.

```
> data(tura)
> # inspect the data
> class(tura) # the object's class
> projection(tura) # the projection
> res(tura) # the spatial resolution (x, y)
> extent(tura) # the extent of the raster brick
```

Extracting scene information

This RasterBrick was read from a .grd file. One advantage of this file format (over the GeoTIFF format, for example) is the fact that the specific names of the raster layers making up this brick have been preserved, a feature which is important for identifying raster layers, especially when doing time series analysis (where you need to know the values on the time axis). This RasterBrick was prepared from a Landsat 7 ETM+ time series, and the original scene names were inserted as layer names.

```
> # display the names of all layers in the tura RasterBrick
> names(tura)
```

We can parse these names to extract information from them. The first 3 characters indicate which sensor the data come from, with 'LE7' indicating Landsat 7 ETM+ and 'LT5' or 'LT4' indicating Landsat 5 and Landsat 4 TM, respectively. The following 6 characters indicate the path and row (3 digits each), according to the WGS system. The following 7 digits represent the date. The date is formatted in such a way that it equals the year + the julian day. For example, February 5th 2001, aka the 36th day of 2001, would be '2001036'.

```
> # display the 1st 3 characters of the layer names
> sensor <- substr(names(tura), 1, 3)
> print(sensor)
> # display the path and row as numeric vectors in the form (path,row)
> path <- as.numeric(substr(names(tura), 4, 6))
> row <- as.numeric(substr(names(tura), 7, 9))
> print(paste(path, row, sep = ","))
> # display the date
> dates <- substr(names(tura), 10, 16)
> print(dates)
> # format the date in the format yyyy-mm-dd
> as.Date(dates, format="%Y%j")
```

There is a function in the rasta package, getSceneinfo() that will parse these names and output a data.frame with all of these attributes.

```
> # ?getSceneinfo
> sceneinfo <- getSceneinfo(names(tura))
> print(sceneinfo)
> # add a 'year' column to the sceneinfo dataframe and plot #scenes/year
> sceneinfo$year <- factor(substr(sceneinfo$date, 1, 4), levels = c(1984:2013))
> ggplot(data = sceneinfo, aes(x = year, fill = sensor)) +
+   geom_bar() +
+   labs(y = "number of scenes") +
+   scale_y_continuous(limits=c(0, 20)) +
+   theme_bw() +
+   theme(axis.text.x=element_text(angle = 45),
+         legend.position=c(0.9, 0.85))
```

Note that the values along the x-axis of this plot are evenly distributed, even though there are gaps in the values (e.g. between 1987 and 1994). The spacing is due to the fact that we defined sceneinfo$year as a vector of *factors* rather than a numeric vectors. Factors act as thematic classes and can be represented by numbers or letters. In this case, the actual values

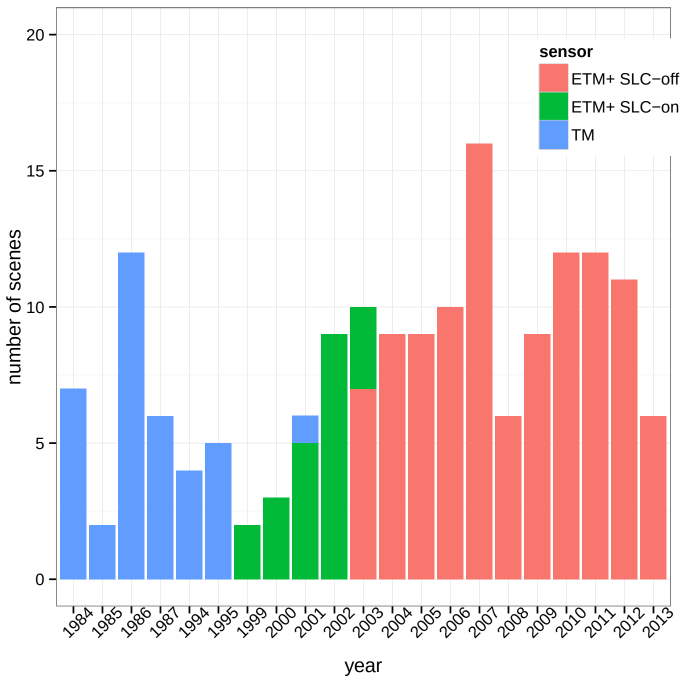

Figure 1: Number of scenes available per year in the time series.

of the factors are not recognized by R. Instead, the levels defined in the factor() function define the hierarchy of the factors (in this case we have defined the levels from 1984 up to 2013, according to the range of acquisition dates). For more information on factors in R, check out `http://www.stat.berkeley.edu/classes/s133/factors.html`.

Try to generate the plot above with the years (x-axis) represented as a numeric vector instead of as a factor. Hint: it is not as straightforward as you might think - to convert a factor x to a numeric vector, try

```
x <- as.numeric(levels(x))
```

Plotting RasterBricks

A RasterBrick can be plotted just as a RasterLayer, and the graphics device will automatically split into panels to accommodate the layers (to an extent: R will not attempt to plot 100 layers at once!). To plot the first 9 layers:

```
> plot(tura, c(1:9))
> # alternatively, you can use [[]] notation to specify layers
> plot(tura[[1:9]])
> # use the information from sceneinfo data.frame to clean up the titles
> plot(tura[[1:9]], main = sceneinfo$date[c(1:9)])
```

Unfortunately, the scale is different for each of the layers, making it impossible to make any meaningful comparison between the raster layers. This problem can be solved by specifying a breaks argument in the plot() function.

```
> # we need to define the breaks to harmonize the scales (to make the plots comparable)
> bks <- seq(0, 10000, by = 2000) # (arbitrarily) define the breaks
> # we also need to redefine the colour palette to match the breaks
> cols <- rev(terrain.colors(length(bks))) # col = rev(terrain.colors(255)) is the default
> # (opt: check out the RColorBrewer package for other colour palettes)
> # plot again with the new parameters
> plot(tura[[1:9]], main = sceneinfo$date[1:9], breaks = bks, col = cols)
```

Alternatively, the rasterVis package has some enhanced plotting functionality for raster objects, including the levelplot() function, which automatically provides a common scale for the layers.

```
> library(rasterVis)
> levelplot(tura[[1:6]])
> # NOTE:
> # for rasterVis plots we must use the [[]] notation for extracting layers
>
> # providing titles to the layers is done using
> # the 'names.attr' argument in place of 'main'
> levelplot(tura[[1:6]], names.attr = sceneinfo$date[1:6])
> # define a more logical colour scale
> library(RColorBrewer)
> # this package has a convenient tool for defining colour palettes
> # ?brewer.pal
> display.brewer.all()
> cols <- brewer.pal(11, 'PiYG')
> # to change the colour scale in levelplot(),
> # we first have to define a rasterTheme object
> # see ?rasterTheme() for more info
> rtheme <- rasterTheme(region = cols)

> levelplot(tura[[1:6]], names.attr = sceneinfo$date[1:6], par.settings = rtheme)
```

This plot (Figure 2) gives us a common scale which allows us to compare values (and perhaps detect trends) from layer to layer. In the above plot, the layer titles do not look very nice – we will solve that problem a bit later.

The rasterVis package has integrated plot types from other packages with the raster package to allow for enhanced visualization of raster data.

```
> # histograms of the first 6 layers
> histogram(tura[[1:6]])
> # box and whisker plot of the first 9 layers
> bwplot(tura[[1:9]])
```

More examples from the rasterVis package can be found @ http://oscarperpinan.github.io/rastervis/

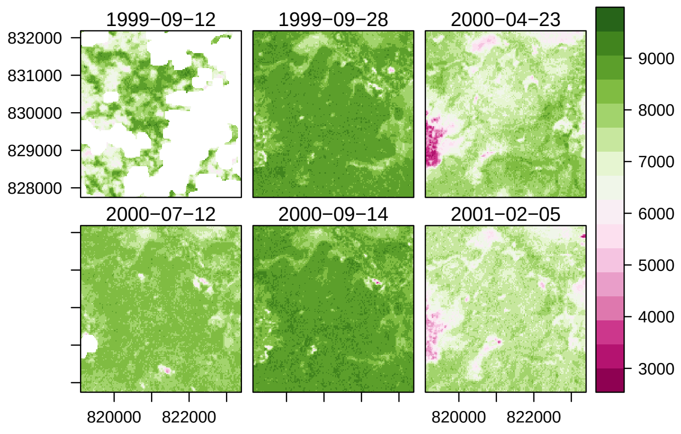

Figure 2: The first six layers of the Tura rasterBrick as visualized using the levelplot() function in rasterVis.

Calculating data loss

In this RasterBrick, the layers have all been individually preprocessed from the raw data format into NDVI values. Part of this process was to remove all pixels obscured by clouds or SLC-off gaps (for any ETM+ data acquired after March 2003). For this reason, it may be useful to know how much of the data has been lost to cloud cover and SLC gaps. First, we will calculate the percentage of no-data pixels in each of the layers using the freq() function. freq() returns a table (matrix) of counts for each value in the raster layer. It may be easer to represent this as a data.frame to access column values.

```
> # try for one layer first
> y <- freq(tura[[1]]) # this is a matrix
> y <- as.data.frame(y)
> print(y)
> # how many NA's are there in this table?
> y$count[is.na(y$value)]
> # alternatively, using the with() function:
> with(y, count[is.na(value)])
> # as a %
> with(y, count[is.na(value)]) / ncell(tura[[1]]) * 100
> # We can compute this more efficiently
> # by using the 'value' argument in freq()
> y <- freq(tura[[1]], value=NA)
> print(y)
> # apply over all layers of the rasterBrick
> y <- freq(tura, value=NA)
> print(y)
> # convert this to % pixels per scene
> y <- freq(tura, value=NA) / ncell(tura) * 100
> # add this vector as a column in the sceneinfo data.frame
> #(rounded to 2 decimal places)
> sceneinfo$nodata <- round(y, 2)

> # plot these values
> ggplot(data = sceneinfo, aes(x = date, y = nodata, shape = sensor)) +
+   geom_point(size = 2) +
+   labs(y = "% nodata") +
+   theme_bw()
```

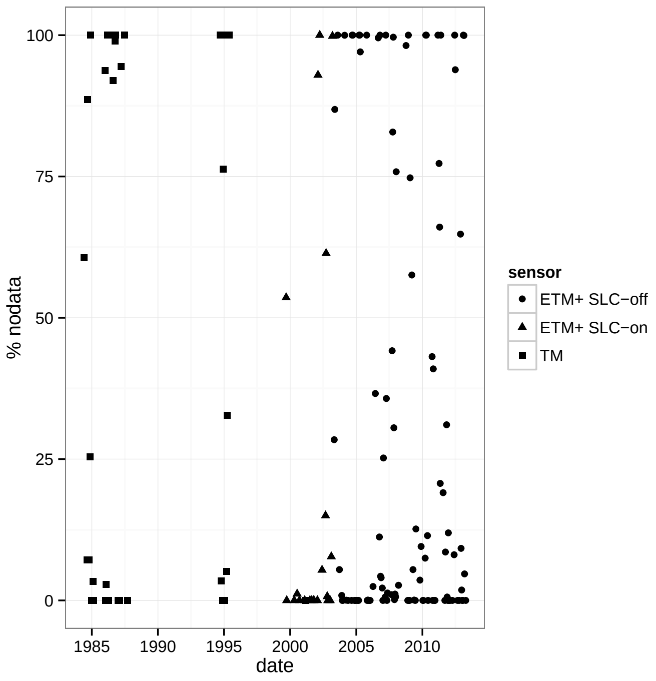

Figure 3: % NAs found in each scene of a Landsat time series.

We have now derived some highly valuable information about our time series. For example, we may want to select an image from our time series with relatively little cloud cover to perform a classification. For further time series analysis, the layers with 100% data loss will be of no use to us, so it may make sense to get rid of these layers.

```
> # which layers have 100% data loss?
> which(sceneinfo$nodata == 100)
> # supply these indices to the dropLayer() command to get rid of these layers
> tura <- dropLayer(tura, which(sceneinfo$nodata == 100))
> # redefine our sceneinfo data.frame so to correspond with rasterBrick
> sceneinfo <- sceneinfo[which(sceneinfo$nodata != 100), ]
> # optional: remake the previous ggplots with this new dataframe
```

With some analyses, it may also be desireable to apply a no-data threshold per scene, in which case layer indices would be selected by:

```
> which(sceneinfo$nodata > some_threshold)
```

In some cases, there may be parts of the study area with more significant data loss due to persistant cloud cover or higher incidence of SLC-off gaps. To map the spatial distribution of data loss, we need to calculate the % of NA in the time series for each *pixel* (ie. looking 'through' the pixel along the time axis). To do this, it is convenient to use the calc() function and supply a special function which will count the number of NA's for each pixel along the time axis, divide it by the total number of data in the pixel time series, and output a percentage. calc() will output a raster with a percentage no-data value for each pixel.

```
> # calc() will apply a function over each pixel
> # in this case, each pixel represents a time series of NDVI values
> # e.g. extract all values of the 53rd pixel in the raster grid
> y <- as.numeric(tura[53])
> print(y) # integer vector from time series values

  [1] 7403 8746 7174 8003 8468 7512 8281 8652 8254 8431 7126   NA 8437 8572   NA
 [16] 8407 8123 7589 7867 7474   NA   NA   NA 8407 8057 8293 8059   NA 6895 7526
 [31] 8227 8382 8487 7737 6304 7664   NA 8264 8312 8286 7702 7758 7919 7138   NA
 [46] 7948 8769 8200 8293 8275 8614 8227 7510 6730 8277 8446 8527 8687   NA   NA
 [61]   NA 8311 8221 7801   NA 7394   NA 8295 8026 7907 7796 7572 6462 8071 8407
 [76]   NA 7687 8007 7865 7982 7637 7664 7939 8226 8127 7429 8324 8253   NA   NA
 [91] 6638   NA 8154 8365 7370 7920 8181 8260 7983 7649 8265 7652   NA 8479 8325
[106] 7194 7939 8329 7805 7858   NA 6796 8047 7438 7142 7904 8104 3645 7681 7361
[121] 6037 6363 6483 6831 7500 8029 7582   NA   NA 7844 7249 6918 5954   NA 7673
[136] 8278 8144   NA 6710 7332 8237 7461

> # how many of these values have been masked (NA)?
> length(y[is.na(y)])

[1] 23

> # as a %
> length(y[is.na(y)]) / length(y) * 100

[1] 16.19718

> # now wrap this in a calc() to apply over all pixels of the RasterBrick
> nodata <- calc(tura, fun = function(x) length(x[is.na(x)]) / length(x) * 100)
> # sometimes it's more readable to define the function first
> percNA <- function(x){
+   y <- length(x[is.na(x)]) / length(x) * 100
+   return(y)
+ }
> nodata <- calc(tura, fun=percNA)
> # plot the nodata raster
> plot(nodata)
```

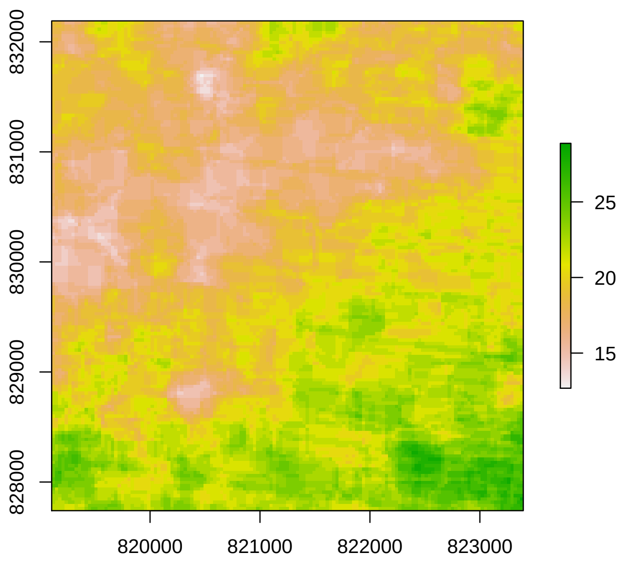

Figure 4: Percent NA throughout the time series for each pixel in the Tura rasterBrick.

From Figure 4 we can now see that after removing the scenes with 100% data loss, our time series is reasonably dense, with a loss of between 10% to 30% of the data due to clouds, cloud shadows and SLC-off gaps. We can also begin to describe spatial patterns of data availability, and while the variance is not very striking in this case, areas with sharp elevation gradients may also show stark differences in data availability. Since availability of data in the temporal domain can be a critical constraint for raster time series analysis, producing plots like Figures 3 and 4 are important initial steps in evaluating the suitability of the data for such analyses.

There is an easy way to interactively extract data from a single pixel using the raster package. First, a raster plot should be made from which to identify the pixel of interest. Then, the click() function allows for extraction of the data contained within that pixel. Simply calling click() with no further arguments will only return the (x, y) coordinates of that point as a 1-row matrix.

```
> plot(tura, 101)
> click()
```

```
            x        y
[1,] 819695.2 830544.3
```

We can extract more meaningful information by passing the name of the object (in this case, a RasterBrick) as the first argument, followed by the argument 'n=1' (or the number of desired points to identify).

```
> plot(tura, 101)
> x <- click(tura, n=1)
> # returns a matrix with n=1 rows
> class(x)
> # convert it to an integer vector
> x <- x[1,]
> # or....
> x <- as.numeric(x)
> # note that the 2nd method removes the names of the vector!
> # plot the vector
> plot(x)
```

In this case, all data from that pixel are extracted. Plotting the values gives a sense of the behaviour of the pixel's values over time, but we need more information to make any real conclusions about the pixel's values over time. To this end, this time series vector could then easily be coerced (or inserted) into a data.frame to plot the data and further analyze them. Let's take a look at a the time series of a few different pixels. Instead of using the interactive click() function to extract these data, suppose that we know the (x, y) coordinates of a land cover type (or other points of interest, from a field campaign or ground truth dataset, for example).

```
> # several pixel coordinate pairs expressed as separate 1-row matrices
> forest <- matrix(data=c(819935, 832004), nrow=1,
+                    dimnames=list(NULL, c('x', 'y')))
> cropland <- matrix(data=c(819440, 829346), nrow=1,
+                      dimnames=list(NULL, c('x', 'y')))
> wetland <- matrix(data = c(822432, 832076), nrow=1,
+                     dimnames=list(NULL, c('x', 'y')))
> # recall that we can extract pixel data if we know the cell #
> # we can easily convert from xy matrix to cell number with cellFromXY()
> forestCell <- cellFromXY(tura, forest)
> print(forestCell)
> croplandCell <- cellFromXY(tura, cropland)
> wetlandCell <- cellFromXY(tura, wetland)
> # return a 1-row matrix with all time series values from this raster cell
> tura[forestCell]
```

Now we are able to extract the time series data given a set of (x, y) coordinates. Let's put the data from these three points into a data.frame to facilitate plotting of the data.

```
> # prepare the data.frame
> # combine data returned earlier from getSceneinfo() with time series values
> ts <- data.frame(sensor = getSceneinfo(names(tura))$sensor,
+                  date = getSceneinfo(names(tura))$date,
+                  forest = t(tura[forestCell]),
+                  cropland = t(tura[croplandCell]),
+                  wetland = t(tura[wetlandCell])
+                  )
> print(ts)
```

Note the use of `t()` to transpose the time series matrices into columns. To see why this is necessary, try remaking the above data.frame without including `t()`. Alternatively, each of the time series matrices could first be coerced to vectors using `x[1,]` or `as.numeric(x)` as we saw earlier before inserting into the data.frame.

```
> # simple plot of forest time series with NDVI scaled back to original values
> # recall that the time series value is NDVI*10000 (see Lesson 6)
> plot(ts$date, ts$forest/10000, xlab="date", ylab="NDVI")
> # same thing, but using with()
> with(ts, plot(date, forest/10000, xlab="date", ylab="NDVI"))
```

Note the two large gaps in the time series during the 1990's, during which time there are no Landsat data available from the USGS. While we could still use these data to understand historical trends, we will only look at time series data from the ETM+ sensor (ie. data acquired after 1999) for the following exercises.

```
> # remove all data from the TM sensor and plot again
> ts <- ts[which(ts$sensor!="TM"), ]
> # alternatively, using subset()
> ts <- subset(ts, sensor!="TM")
> # plot the new time series
> with(ts, plot(date, forest, xlab="date", ylab="NDVI"))
```

A more informative plot would show these time series side by side with the same scale or on the same plot. These are possible with either the base `plot()` function or using ggplot2. Either way, there is some preparation needed, and in the case of ggplot2, this may not be immediately obvious. In the following example, we are going to make a facet_wrap plot. In order to do so, we need to merge the time series columns to make a data.frame with many rows which `ggplot()` can interpret. An additional column will be used to identify the class (forest, cropland or wetland) of each data point, and this class column will be used to 'split' the data into 3 facets. The reshape package has a convenient function `melt()` which will 'automatically' reshape the data.frame to make it passable to the ggplot framework.

```
> library(reshape)
> # convert dates to characters, otherwise melt() returns an error
> ts$date <- as.character(ts$date)
> # 'melt' the data.frame
> tsmelt <- melt(ts)
> # inspect the new data.frame
> head(tsmelt)
> # change the 'variable' column heading to 'class'
> names(tsmelt)[3] <- "class"
> # convert tsplot$date back to Date class to enable formatting of the plot
> tsmelt$date <- as.Date(tsmelt$date)
> # inspect the data.frame
> tsmelt

> ggplot(data = tsmelt, aes(x = date, y = value / 10000)) +
+   geom_point() +
+   scale_x_date() +
+   labs(y = "NDVI") +
+   facet_wrap(~ class, nrow = 3) +
+   theme_bw()
```

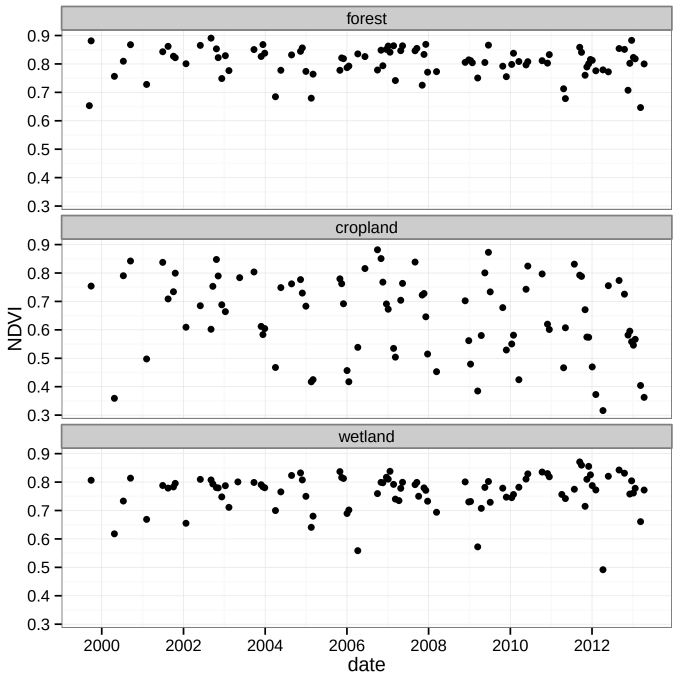

Figure 5: NDVI time series from 1999 to 2012 for three land cover classes: forest, cropland, and wetland.

3 Exercise

- example 1: produce a figure with maximum/minimum/median/mean NDVI per year; figure should have a common scale and be properly labelled
- example 2: compute a time series metric over an entire RasterBrick using the calc() function