Raster Analysis

Ben DeVries, Jan Verbesselt, Loïc Dutrieux

October 21, 2013

Abstract

In this tutorial, we will explore the raster package and other related packages used for typical raster analyses. We will first look at analysis of RasterLayer objects, exploring functions having to do with raster algebra, focal and zonal statistics and other operations. We will then explore spatio-temporal analysis of raster data using RasterBrick objects. Here, we will extract time series data from a RasterBrick object and derive temporal statistics from these data. Since data from the Landsat archive is becoming increasingly important for environmental research and monitoring, this tutorial will focus on the use of these data.

1. perform raster algebra to calculate indices
2. classify a raster layer
3. perform focal operations to sieve a raster
4. parse Landsat scene information from time series data
5. explore a raster brick by plotting layers and layers statistics
6. perform raster brick operations to derive statistics (e.g. %no-data in the time series)
7. extract pixel time series and derive various time series statistics

1 Raster and related packages

The raster package is an essential tool for raster-based analysis in R. Here you will find functions which are fundamental to image analysis. The raster package documentation is a good place to begin exploring the possibilities of image analysis within R. There is also an excellent vignette available at http://cran.r-project.org/web/packages/raster/vignettes/Raster.pdf.

In addition to the raster package, we will be using the rasterVis and ggplot2 packages to make enhanced plots.

```
> # load the necessary packages
> library(rasta)
> library(raster)
> library(rgdal)
> library(sp)
> library(ggplot2)
```

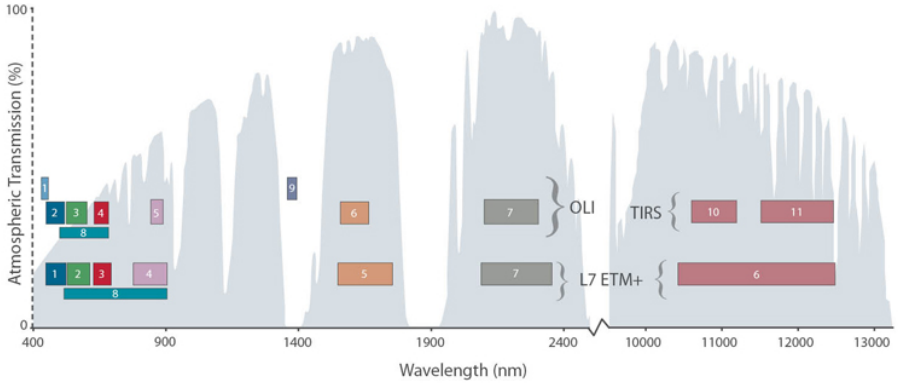


Figure 1: Bands included in the Landsat 7 (ETM+) and Landsat 8 (OLI/TIRS) sensors. Source: NASA.

2 The Landsat archive

Since being released to the public, the Landsat data archive has become an invaluable tool for environmental monitoring. With a historical archive reaching back to the 1970's, the release of these data has resulted in a spur of time series based methods. In this tutorial, we will work with time series data from the Landsat 7 Enhanced Thematic Mapper (ETM+) sensor.

3 Manipulating raster data

Exploring a Landsat scene

Landsat scenes are delivered via the USGS as a number of image layers representing the different bands captured by the sensors. In the case of the Landsat 7 Enhanced Thematic Mapper (ETM+) sensor, the bands are shown in figure xxx. Using different combination of these bands can be useful in describing land features and change processes.

Part of a landsat scene, including bands 2-4 are included as part of the rasta package. These data have been processed using the LEDAPS framework (TODO: insert link), so the values contained in this dataset represent surface reflectance, scaled by 10000 (ie. divide by 10000 to get a reflectance value between 0 and 1).

We will begin exploring these data simply by visualizing them (more methods for data exploration will be covered in Lesson 7).

```
> # load in the data
> data(GewataB2)
> data(GewataB3)
> data(GewataB4)
> # check out the attributes
```

```
> GewataB2
> # some basic statistics using cellStats()
> cellStats(GewataB2, stat=max)
> # ...is equivalent to:
> maxValue(GewataB2)
> # what is the maximum value of all three bands?
> max(c(maxValue(GewataB2), maxValue(GewataB3), maxValue(GewataB4)))
> # plot the histograms of all bands
> hist(GewataB2)
> hist(GewataB3)
> hist(GewataB4)
```

We can improve these plots by adjusting the axis scale, bin size, etc., but the rasterVis package that has enhanced plotting capabilities which make it easier to make more attractive plots. First, to make the comparison easier, we will make a rasterBrick object from these three layers.

```
> library(rasterVis)
> gewata <- brick(GewataB2, GewataB3, GewataB4)
> # view all histograms together with rasterVis
> histogram(gewata)
```

The rasterVis package has several other raster plotting types inherited from the lattice pacakge. For multispectral data, one plot type which is particularly useful for data exploration is the scatterplot matrix, called by the splom() function.

```
> splom(gewata)
```

Calling splom() on a rasterBrick reveals potential correlations between the layers themselves(Figure 2). In the case of bands 2-4 of the gewata subset, we can see that band 2 and 3 (in the visual part of the EM spectrum) are highly correlated, while band 4 contains significant non-redundant information. Given what we know about the location of these bands along the EM spectrum (Figure 1), how could these scatterplots be explained? ETM+ band 4 (nearly equivalent to band 5 in the Landsat 8 OLI sensor) is situated in the near infrared (NIR) region of the EM spectrum and is often used to described vegetation-related features.

The rasterVis package will be demonstrated in more detail in this tutorial the section dealing with raster time series.

Computing new rasters: Raster algebra

In the previous section, we observed a strong correlation between two of the Landsat bands of the gewata subset, but a very different distribution of values in band 4 (NIR). This distribution stems from the fact that vegetation reflects very highly in the NIR range, compared to the visual range of the EM spectrum. A commonly used metric for assessing vegetation

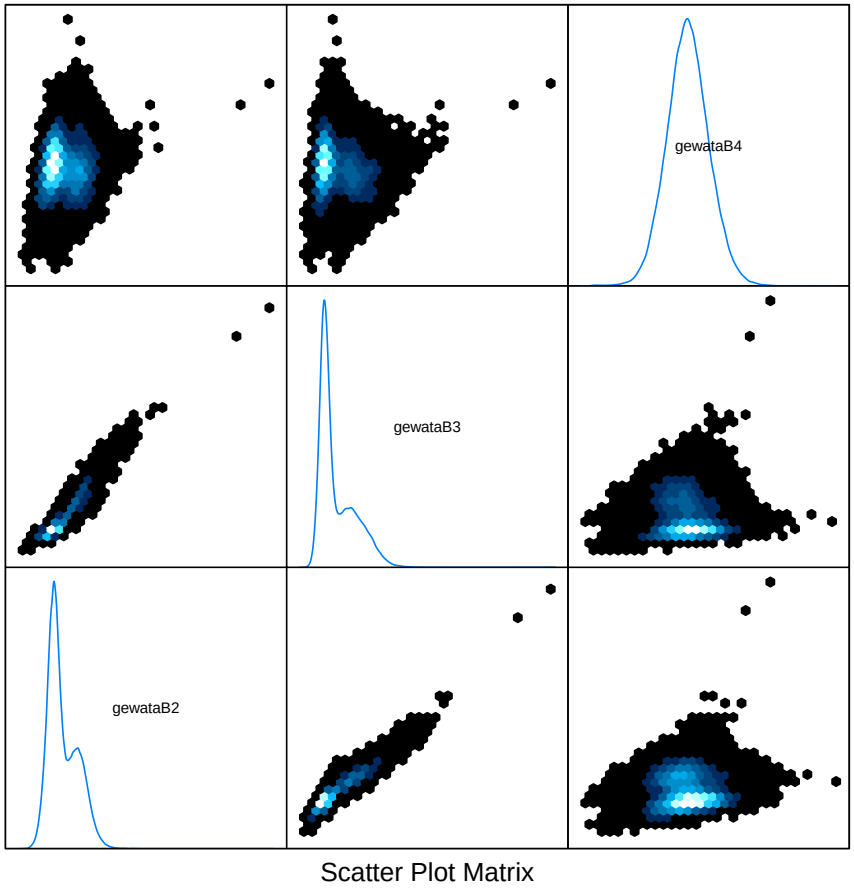


Figure 2: Scatter plot matrix for Landsat bands 2, 3, and 4 of the 'Gewata' subscene.

dynamics, the normalized difference vegetation index (NDVI), takes advantage of this fact and is computed from Landsat bands 3 (red; R) and 4 (NIR) as follows:

$$NDVI = \frac{NIR - R}{NIR + R} \tag{1}$$

Using principles of raster algebra, we can easily perform this calculation in R.

```
> ndvi <- (GewataB4 - GewataB3) / (GewataB4 + GewataB3)
```

Using raster algebra (above) is relatively easy and intuitive. However, with particularly large datasets or complex calculations, it is often desireable to use the overlay() function instead.

```
> ndvi <- overlay(GewataB4, GewataB3, fun=function(x,y){(x-y)/(x+y)})
```

One advantage of this function is the fact that the result can be written immediately to file by including the argument 'filename="..."', thus saving memory (especially important when working with large datasets).

Plotting our new raster, we can immediately see how useful the NDVI metric is for identifying land features. Use the interactive 'drawExtent()' function to zoom into some of the features to inspect them more closely.

```
> # first, plot the raster
> plot(ndvi)
> # call drawExtent() to activate interactive mode
> # and assign the result to an extent object e
> e <- drawExtent()
> # now click 2 points on the plot window
> # these represent the top-right and bottom-left corner of your extent
> # now plot ndvi again, but only the extent you defined interactively
> plot(ndvi, ext=e)
```

4 Classifying raster data

One of the most important tasks in analysis of remote sensing image analysis is image classification. In classifying the image, we take the information contained in the various bands (possibly including other synthetic bands such as NDVI or principle components). In this tutorial we will explore two approaches for image classification: unsupervised (k-means) and supervised (random forest) classification.

Supervised classification: Random Forest[1]

The Random Forest classification algorithm is an ensemble learning method that is used for both classification and regression. In this study, we will use the algorithm to derive land cover classes given a set of training data. Random Forest builds a number of classification trees, and the final class assigned is based on the class with the maximum instances among the final set ('forest') of classes[2].

One major advantage of the Random Forest method is the fact that an 'Out of the Box' (OOB) error estimate and an estimate of variable performace are performed. For each classification tree assembled, a fraction of the training data are left out and used to compute the error for each tree. In addition an importance score is computed for each variable in two forms: the mean decrease in accuracy for each variable, and the Gini impurity criterion.

We should first prepare the data on which the classification will be done. So far, we have prepared three bands from a ETM+ image in 2001 (bands 2, 3 and 4) as a rasterBrick, and have also calculated NDVI. In addition, there is a Vegetation Continuous Field (VCF) product available for the same period (2000)[3]. This product is also based on Landsat ETM+ data, and represents an estimate of tree cover (in %). Since this layer could also be useful in classifying land cover types, we will also include it as a potential covariate in the random forest classification.

```
> # load the data
> data(vcfGewata)
```

[1] This section was written using code contributed by Valerio Avitabile.

[2] For a more complete description of the Random Forests classification method, see http://stat-www.berkeley.edu/users/breiman/RandomForests/cc_home.htm.

[3] For more information on the Landsat VCF product, see http://glcf.umd.edu/data/landsatTreecover/.

```
> plot(vcfGewata)
> histogram(vcfGewata) # or 'hist(vcfGewata)'
```

Note that in the vcfGewata rasterLayer there are some values much greater than 100, which are flags for water, cloud or cloud shadow pixels. To avoid these layers, we can assign a value of NA to these pixels so they are not used in the classification.

```
> vcfGewata[vcfGewata > 100] <- NA
> plot(vcfGewata)
> histogram(vcfGewata)
```

To perform the classification in R, it is best to assemble all covariate layers into one rasterBrick object. In this case, we can simply append these new layers (NDVI and VCF) to our existing rasterBrick (currently consisting of bands 2, 3, and 4). But first, let's rescale the NDVI layer by 10000 (just as the reflectance bands 2, 3, and 4 have been scaled) and store it as an integer raster.

```
> ndvi <- calc(ndvi, fun = function(x) floor(x*10000))
> # change the data type
> # see ?dataType for more info
> dataType(ndvi) <- "INT2U"
> # name this layer to make plots interpretable
> names(ndvi) <- "NDVI"
> # make the covariate rasterBrick
> covs <- addLayer(gewata, ndvi, vcfGewata)
> plot(covs)
```

For this exercise, we will do a very simple classification for 2001 using three classes: forest, cropland and wetland. While for other purposes it is usually better to define more classes (and possibly fuse classes later), a simple classification like this one could be useful, for example, to construct a forest mask for the year 2001.

```
> # load the training polygons
> data(trainingPoly)
> # inspect the data
> trainingPoly@data
> # superimpose training polygons onto ndvi plot
> plot(ndvi)
> plot(trainingPoly, add = TRUE)
```

The training classes are labelled as string labels. For this exercise, we will need to work with integer classes, so we will need to first 'relabel' our training classes. To do this, we will first make a function that will reclassify the strings representing land cover classes into integers 1, 2, and 3 using the conditional assigner ifelse().

```
> # define a reclassification function
> reclass <- function(x){
+   y <- ifelse(x == "forest", 1,
```

```
+               ifelse(x == "cropland", 2, 3)
+               )
+   return(y)
+ }
> # apply this over the trainingPoly data slot
> trainingPoly@data$Code <- sapply(trainingPoly@data$Class,
+                                  FUN = function(x) reclass(x))
```

To train the raster data, we need to convert our training data to the same type using the rasterize() function. This function takes a spatial object (in this case a polygon object) and transfers the values to raster cells defined by a raster object. Here, we will define a new raster containing those values.

```
> classes <- rasterize(trainingPoly, ndvi, field = 'Code')
> dataType(classes) <- "INT1U"
> plot(classes, col = c("dark green", "orange", "light blue"))
```

(Note: there is a handy 'progress="text"' argument, which can be passed to many of the raster package functions and can help to monitor processing).

Our goal in preprocessing these data is to have a table of values representing all layers (covariates) with *known* values/classes. To do this, we will first need to create a version of our rasterBrick only representing the training pixels. Here the mask() function from the raster package will be very useful.

```
> covmasked <- mask(covs, classes)
> plot(covmasked)
> # add the classes layer to this new brick
> names(classes) <- "class"
> trainingbrick <- addLayer(covmasked, classes)
> plot(trainingbrick)
```

Now it's time to add all of these values to a data.frame representing all training data. This data.frame will be used as an input into the RandomForest classification function. We will use getValues() to extract all of the values from the layers of the rasterBrick.

```
> # extract all values into a matrix
> valuetable <- getValues(trainingbrick)
> # convert to a data.frame and inspect the first and last rows
> valuetable <- as.data.frame(valuetable)
> head(valuetable)
> tail(valuetable)
```

In inspecting this training data.frame, you will notice that a significant number of rows has the value NA for the class column, which will be problematic during the training phase. The rows with class=NA represent pixels found outside the training polygons, and these rows should therefore be removed before going ahead with deriving the Random Forest model.

```
> # keep only rows where valuetable$classes has a value
> valuetable <- valuetable[!is.na(valuetable$class),]
> head(valuetable)
> tail(valuetable)
> # convert values in the class column to factors
> valuetable$class <- factor(valuetable$class, levels = c(1:3))
```

Now we have a convenient reference table which contains, for each of the three defined classes, all known values for all covariates. Before proceeding with the classification, let's visualize the distribution of some of these covariates using the ggplot() package.

```
> # 1. NDVI
> ggplot(data = valuetable, aes(x = NDVI)) +
+   geom_histogram() +
+   facet_wrap(~ class) +
+   theme_bw()
> # 2. VCF
> ggplot(data = valuetable, aes(x = vcf2000Gewata)) +
+   geom_histogram() +
+   labs(x = "% Tree Cover") +
+   facet_wrap(~ class) +
+   theme_bw()
> # 3. Bands 3 and 4
> ggplot(data = valuetable, aes(x = gewataB3, y = gewataB4)) +
+   stat_bin2d() +
+   facet_wrap(~ class) +
+   theme_bw()
> # 4. Bands 2 and 3
> ggplot(data = valuetable, aes(x = gewataB2, y = gewataB3)) +
+   stat_bin2d() +
+   facet_wrap(~ class) +
+   theme_bw()
```

We can see from these distributions that these covariates may do well in classifying forest pixels, but we may expect some confusion between cropland and wetland (although the individual bands may help to separate these classes). When performing this classification on large datasets and with a large amount of training data, now may be a good time to save this table using the write.csv() command, in case something goes wrong after this point and you need to start over again.

Now it is time to build the Random Forest model using the training data contained in the table of values we just made. For this, we will use the "randomForest" package in R, which is an excellent resource for building such types of models. Using the randomForest() function, we will build a model based on a matrix of predictors or covariates (ie. the first 5 columns of valuetable) related to the response (the 'class' column of valuetable).

```
> # NA values are not permitted in the covariates/predictor columns
> # which rows have NAs in them?
> delRows <- which(apply(valuetable, 1, FUN = function(x) NA %in% x))
```

```
> # remove these rows from valuetable
> valuetable <- valuetable[-delRows,]

> # construct a random forest model
> # caution: this step takes fairly long!
> library(randomForest)
> modelRF <- randomForest(x = valuetable[,c(1:5)], y = valuetable$class,
+                         importance = TRUE)
```

Since the random forest method involves the building and testing of many classification trees (the 'forest'), it is a computationally expensive step (and could take alot of memory for especially large training datasets). When this step is finished, it would be a good idea to save the resulting object with the save() command. Any R object can be saved as an .rda file and reloaded into future sessions.

The resulting object from the randomForest() function is a specialized object of class "randomForest", which is a large list-type object packed full of information about the model output. Elements of this object can be called and inspected like any list object.

```
> # inspect the structure and element names of the resulting model
> class(modelRF)
> str(modelRF)
> names(modelRF)
> # inspect the confusion matrix of the OOB error assessment
> modelRF$confusion
```

Since we set 'importance=TRUE', we now also have information on the statistical importance of each of our covariates which we can visualize using the varImpPlot() command.

```
> varImpPlot(modelRF)
```

These two plots give two different reports on variable accuracy. In this case, it seems that Gewata bands 3 and 4 have the highest impact on accuracy, while bands 3 and 2 score highest with the Gini impurity criterion. For especially large datasets, it may be helpful to know this information, and leave out less important variables for subsequent runs of the randomForest() function.

Since the VCF layer included NA's (which have also been excluded in our results) and scores relatively low according to the mean accuracy decrease criterion, try to construct an alternate random forest model as above, but excluding this layer. What effect does this have on the overall accuracy of the results (hint: compare the confusion matrices of the original and new outputs). What affect does leaving this variable out have on the processing time (hint: use system.time())?

Now we can apply this model to the rest of the image and assign classes to all pixels. Note that for this step, the names of the raster layers in the input brick (here 'covs') must correspond to the column names of the training table. We will use the predict() function from the raster package to predict class values based on the random forest model we have just constructed. This function uses a pre-defined model to predict values of raster cells based on other raster layers. This model can be derived by a linear regression, for example. In our case, we will use the model provided by the randomForest() function we applied earlier.

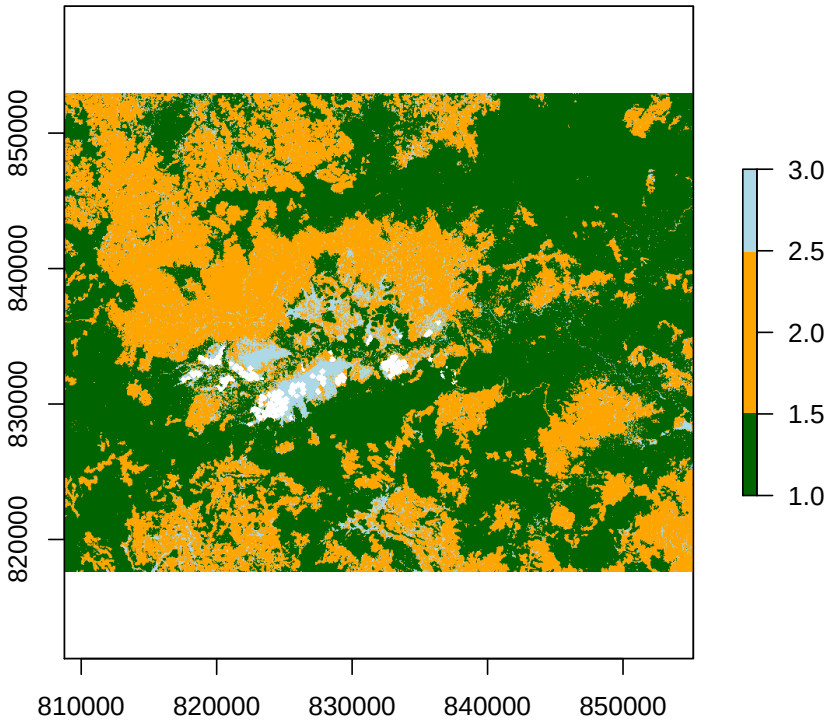


Figure 3: Resulting land cover map using a Random Forest classifier.

```
> # check layer and column names
> names(covs)
> names(valuetable)
> # predict land cover using the RF model
> predLC <- predict(covs, model=modelRF, na.rm=TRUE)
> # plot the results
> # recall: 1 = forest, 2 = cropland, 3 = wetland
> plot(predLC, col=c("dark green", "orange", "light blue"))
```

Note that the predict() function also takes arguments that can be passed to writeRaster() (eg. 'filename = ...'), so it would be a good idea to write to file as you perform this step (rather than keeping all output in memory).

Unsupervised classification: k-means

In the absence of training data, an unsupervised classification can be carried out. Unsupervised classification methods assign classes based on inherent structures in the data without

resorting to training of the algorithm. One such method, the k-means method, divides data into clusters based on Euclidean distances from cluster means in a feature space[4].

We will use the same layers (from the 'covs' rasterBrick) as in the Random Forest classification for this classification exercise. As before, we need to extract all values into a data.frame.

```
> valuetable <- getValues(covs)
> head(valuetable)
```

Now we will construct a kmeans object using the kmeans() function. Like the Random Forest model, this object packages useful information about the resulting class membership. In this case, we will set the number of clusters to three, presumably corresponding to the three classes defined in our random forest classification.

```
> km <- kmeans(na.omit(valuetable), centers = 3, iter.max = 100, nstart = 10)

> # km contains the clusters (classes) assigned to the cells
> head(km$cluster)
> unique(km$cluster) # displays unique values
```

Here, we used the 'na.omit()' argument to avoid any NA values in the valuetable (recall that there is a region of NAs in the VCF layer). These NAs are problematic in the kmeans() function, but omitting them gives us another problem: the resulting vector of clusters (from 1 to 3) is shorter than the actual number of cells in the raster. In other words: how do we know which clusters to assign to which cells? To answer that question, we need to have a kind of 'mask' raster, indicating where the NA values throughout the cov rasterBrick are located.

```
> # create a blank raster with NA values
> rNA <- setValues(raster(covs), NA)
> # loop through layers of covs
> # assign a 1 wherever an NA is enountered
> for(i in 1:nlayers(covs)){
+   rNA[is.na(covs[[i]])] <- 1
+ }
> # convert rNA to an integer vector
> rNA <- getValues(rNA)
> # substitue the NA's for 0's
> rNA[is.na(rNA)] <- 0
```

We now have a vector indicating with a value of 1 where the NA's in the cov brick are. Now that we know where the 'original' NAs are located, we can go ahead and assign the cluster values to a raster. At these 'NA' locations, we will not assign any of the cluster values, instead assigning an NA.

First, we will insert these values into the original valuetable data.frame.

```
> # convert valuetable to a data.frame
> valuetable <- as.data.frame(valuetable)
```

[4]For more information on the theory behind k-means clustering, see http://home.deib.polimi.it/matteucc/Clustering/tutorial_html/kmeans.html#macqueen

```
> plot(classes, col=c("dark green", "orange", "light blue"))
```

Figure 4: Unsupervised land cover classification resulting from k-means method.

```
> # assign the cluster values (where rNA != 1)
> valuetable$class[rNA==0] <- km$cluster
> # assign NA to this column elsewhere
> valuetable$class[rNA==1] <- NA
```

Now we are finally ready to assign these cluster values to a raster. This will represent our final classified raster.

```
> # create a blank raster
> classes <- raster(covs)
> # assign values from the 'class' column of valuetable
> classes <- setValues(classes, valuetable$class)
> plot(classes, col=c("dark green", "orange", "light blue"))
```

These classes are much more difficult to interpret than those resulting from the random forest classification. We can see from Figure 4 that there is particularly high confusion between the (presumably) cropland and wetland classes. Clearly, with a good training dataset, a supervised classification can provide a reasonably accurate land cover classification. However, unsupervised classification methods like k-means are useful for study areas for which little to no *a priori* data exist. Assuming there are no training data available, is there a way we could improve the k-means classification performed in this example? Which one is computationally faster between random forest and k-means (hint: try the system.time() function)?

Applying a raster sieve

Although the land cover raster we created above is limited in the number of thematic classes it has, and we observed some confusion between wetland and cropland classes, it could be useful for constructing a forest mask. To do so, we have to fuse (and remove) non-forest classes, and then clean up the remaining pixels using focal algebra by applying a sieve.

```
> # Make an empty raster based on the LC raster attributes
> formask <- raster(predLC)
> # assign NA to all cells
> formask <- setValues(formask, value = NA)
> # assign 1 to all cells corresponding to the forest class
> formask[predLC==1] <- 1
> plot(formask, col = "dark green", legend = FALSE)
```

We now have a forest mask that can be used to isolate forest pixels for further analysis. For some applications, however, we may only be interested in larger forest areas. We may especially want to remove single forest pixels, as they may be a result of errors, or may not fit our definition of 'forest'.

Figure 5: Zoom of a forest mask before (left) and after (right) application of a sieve.

To remove these pixel 'islands', we will use the focal() function of the raster package, which computes values based on values of neighbouring pixels. The first task is to construct a mask with the focal() function, which will be used to 'clean' up the forest mask we have just produced.

In the first case, we will define 'island' pixels as any pixel not having neighbours in any direction (including diagonals). To do this, we can simply apply a single numeric weight defining the dimensions of the matrix.

```
> # make an empty raster
> sievemask <- setValues(raster(formask), NA)
> # assign a value of 1 for all forest pixels
> sievemask[!is.na(formask)] <- 1
> # sum of all neighbourhood pixels
> sievemask <- focal(sievemask, w=3, fun=sum, na.rm=TRUE)
> sievemask
> histogram(sievemask)
```

We now have a mask whose values are the sum of neighbourhood weights (each equal to 1/9, based on a 3X3 window). In cases where there are no neighbours (ie. the 'island' pixels), this sum will be equal to 1/9 exactly. In cases with one or more neighbours, the sum will be greater than 1/9, up to 1 (ie. if the pixel is completely surrounded by non-NA pixels). To apply the sieve and remove 'island' pixels, we want to select for only cases where sievemask == 1/9 and remove those from the original raster.

```
> # copy the original forest mask
> formaskSieve <- formask
> # assign NA to pixels where the sievemask == 1/9
> formaskSieve[sievemask==1/9] <- NA
> # zoom in to a small extent to check the results
> # Note: you can define your own by using e <- drawExtent()
> e <- extent(c(811744.8, 812764.3, 849997.8, 850920.3))
> par(mfrow=c(1, 2)) # allow 2 plots side-by-side
> plot(formask, ext=e, col="dark green", legend=FALSE)
> plot(formaskSieve, ext=e, col="dark green", legend=FALSE)
> par(mfrow=c(1, 1)) # reset plotting window
```

We have successfully removed all 'island' pixels from the forest mask using the focal() function. Suppose we define 'island' pixels as those have no immediate neighbours, *not* considering diagonal neighbours. In that case, we would have to adjust the weight argument ('w') in focal(), and instead define our own 3X3 matrix which omits diagnoal pixels.

```
> # define a weights matrix
> w <- rbind(c(0, 1, 0),
```

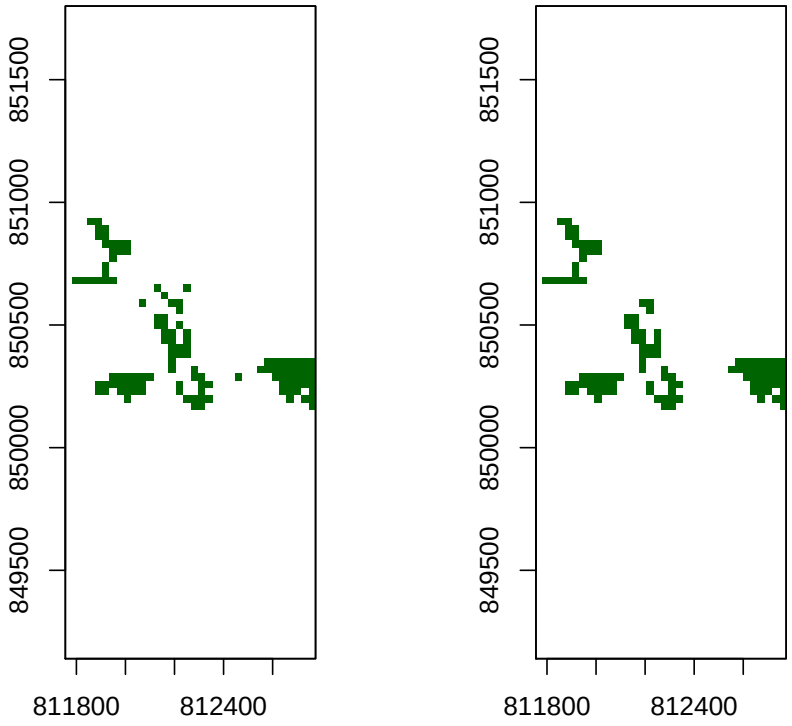


Figure 6: Zoom of a forest mask before (left) and after (right) application of a sieve, without consideration for diagonal neighbours.

```
+            c(1, 1, 1),
+            c(0, 1, 0))
> print(w)
> # alternatively:
> w <- matrix(c(0, 1, 0, 1, 1, 1, 0, 1, 0), nrow = 3)
> # prepare the sievemask (as above)
> sievemask <- setValues(raster(formask), NA)
> sievemask[!is.na(formask)] <- 1
> # sum of all neighbouring pixels, except for diagonals
> sievemask <- focal(sievemask, w = w, fun = sum, na.rm = TRUE)
```

When we define the matrix manually, the weights are not normalized (sum to 1) as they did when we set 'w = 3'. In this case, 'island' pixels would have a value of 1 (no other neighbours), and should be removed according to the sieve. Applying the new sieve as above should give the following results.

In this case, not only pixels with absolutely no neighbours have been removed, but also those pixels with only diagonal neighbours as well. The second case could be considered as a more 'conservative' sieve.

The focal() function can be used for a variety of purposes, including other data filters such as median, Laplacian, Sobel, etc. More complex weight matrices can also be computed, such as a Gaussian using the focalWeight() function. These methods will not be covered in this tutorial. Check out the documentation at ?focal() for more information.

Working with thematic rasters

In some cases, the values of a raster may be categorical, meaning they relate to a thematic class (e.g. 'forest' or 'wetland') rather than a quantitative value (e.g. NDVI or % Tree Cover). The raster dataset 'lulcGewata' is a raster with integer values representing LULC classes from a 2011 classification (using SPOT5 and ASTER source data).

```
> data(lulcGewata)
> # check out the distribution of the values
> freq(lulcGewata)

     value  count
[1,]     1 396838
[2,]     2  17301
[3,]     3    943
[4,]     4  13645
[5,]     5 470859
[6,]     6 104616
[7,]    NA 817794

> hist(lulcGewata)
```

This is a raster with integer values between 1 and 6, but for this raster to be meaningful at all, we need a lookup or attribute table to identify these classes. A .csv file has also been provided as part of the package. Read it in as a data.frame:

```
> data(LUTGewata)
> LUTGewata

  ID             Class
1  1          cropland
2  2            bamboo
3  3         bare soil
4  4 coffee plantation
5  5            forest
6  6           wetland
```

This data.frame represents a lookup table for the raster we just loaded. The ID column corresponds to the values taken on by the lulc raster, and the 'Class' column describes the LULC classes assigned. In R it is possible to add a attribute table to a raster. In order to do this, we need to coerce the raster values to a factor from an integer.

```
> lulc <- as.factor(lulcGewata)
```

If you display the attributes of this raster (just type 'lulc'), it will do so, but will also return an error. This error arises because R expects that a raster with factor values should also have a raster attribute table.

```
> # assign a raster attribute table (RAT)
> levels(lulc) <- LUTGewata
> lulc
```

5 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.

A raster brick from a small area within the Kafa Biosphere Reserve in Southern Ethiopia can be found in the rasta package. Set the working directory to the packages home folder and load the raster brick from file by

```
> 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.

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

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") +
+   theme_bw() +
+   theme(axis.text.x = element_text(angle = 45))
```

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 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(as.character(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
```

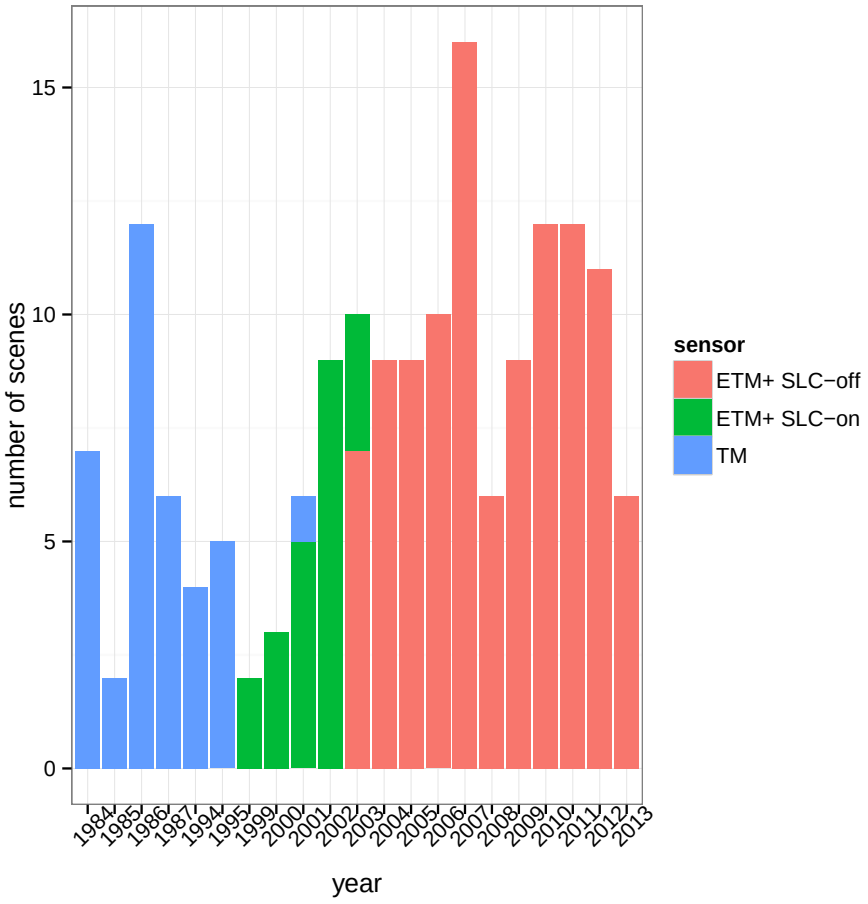


Figure 7: Number of scenes available per year in the time series.

```
> 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[[1:8]], names.attr = sceneinfo$date[1:8])
> # define a more logical colour scale
> library(RColorBrewer)
> # this package has a convenient tool for defining colour palettes
> # ?brewer.pal
> display.brewer.all()
> cols <- brewer.pal(11, 'PiYG')
> # to change the colour scale in levelplot(),
> # we first have to define a rasterTheme object
> # see ?rasterTheme() for more info
> rtheme <- rasterTheme(region = cols)
> levelplot(tura[[1:8]], names.attr = sceneinfo$date[1:8], par.settings = rtheme)
```

This plot 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 analysis 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/

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)
> # 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
> # apply this over all layers in the RasterBrick
> # first, prepare a numeric vector to be 'filled' in
> nas <- vector(mode = 'numeric', length = nlayers(tura))
> for(i in 1:nlayers(tura)){
+   y <- as.data.frame(freq(tura[[i]]))
+   # if there are no NAs, then simply assign a zero
+   # otherwise, grab the # of NAs from the frequency table
+   if(!TRUE %in% is.na(y$value)){
+     nas[i] <- 0
+   } else {
+     nas[i] <- with(y, count[is.na(value)]) / ncell(tura[[i]]) * 100
+   }
+ }
> # add this vector as a column in the sceneinfo data.frame
> #(rounded to 2 decimal places)
> sceneinfo$nodata <- round(nas, 2)
> # plot these values
> ggplot(data = sceneinfo, aes(x = date, y = nodata, shape = sensor)) +
+   geom_point(size = 2) +
+   labs(y = "% nodata") +
+   theme_bw()
>
```

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 as well
> 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)
```

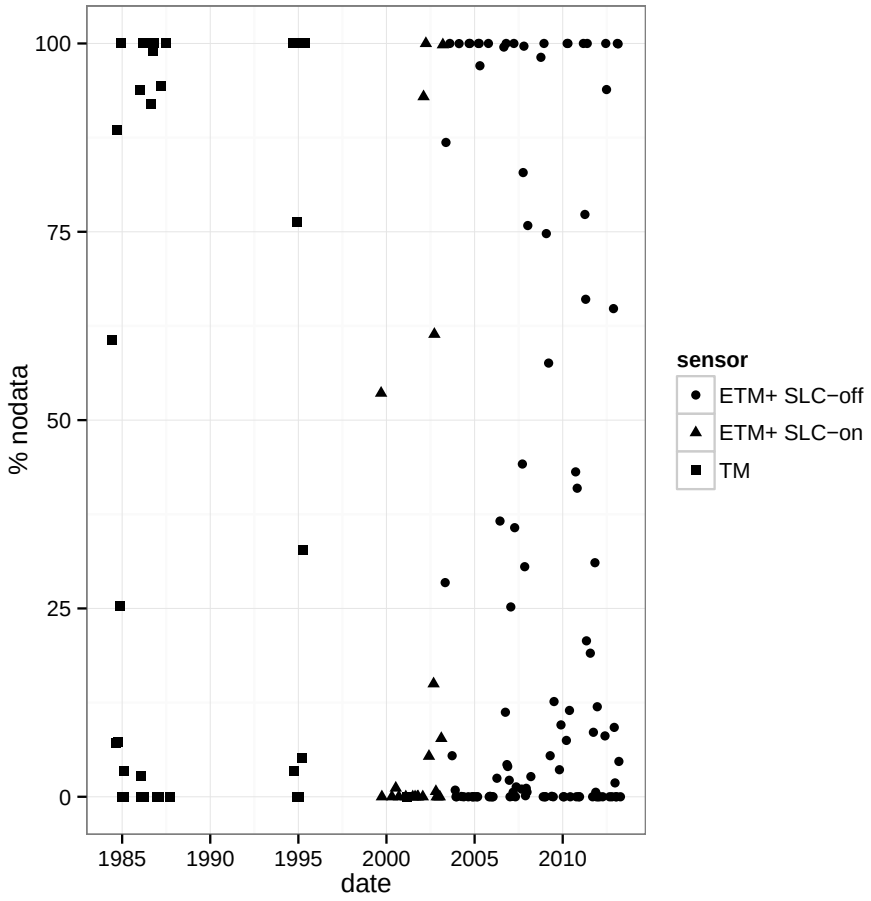


Figure 8: % NAs found in each scene of a Landsat time series.

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. all values of the 53rd pixel in the raster grid:
> y <- as.numeric(tura[53])
> # 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)
```

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)
> click(tura, n = 1)
```

In this case, all data from that pixel are extracted. These time series 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()
> cellFromXY(tura, forest)
> tura[cellFromXY(tura, forest)] # returns a 1-row matrix with all ts values
```

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
> ts <- data.frame(sensor = getSceneinfo(names(tura))$sensor,
+                  date = getSceneinfo(names(tura))$date,
+                  forest = t(tura[cellFromXY(tura, forest)]),
```

```
+                   cropland = t(tura[cellFromXY(tura, cropland)]),
+                   wetland = t(tura[cellFromXY(tura, wetland)])
+                   )
> print(ts)
> # simple plot of forest time series
> plot(ts$date, ts$forest)
> # same thing but using with()
> with(ts, plot(date, forest))
```

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.

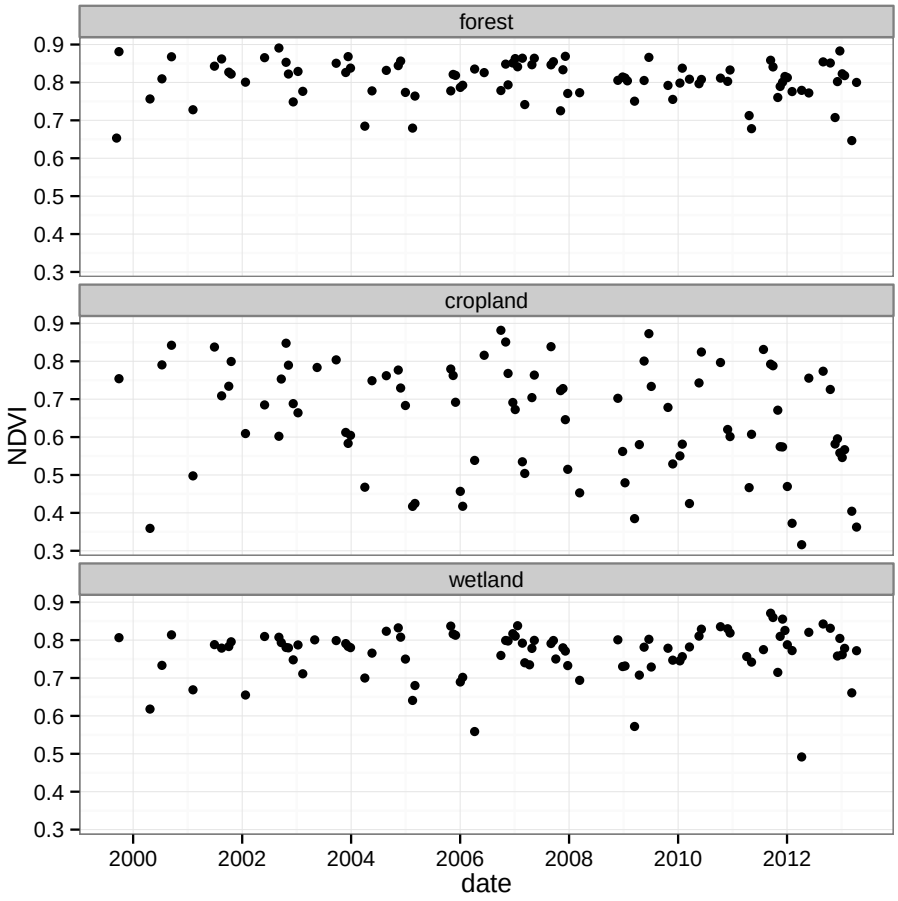
```
> # remove all data from the TM sensor and plot again
> ts <- ts[which(ts$sensor != "TM"), ]
> with(ts, plot(date, forest))
```

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 indeed. An additional column will be used to identify the class (forest, cropland or wetland) of each data point, and this class 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)
> tsmelt <- melt(ts)
> head(tsmelt)
> names(tsmelt) <- c('sensor', 'date', 'class', 'value')
> # convert tsplot$date back to Date class to enable formatting of the plot
> tsmelt$date <- as.Date(tsmelt$date)
> ggplot(data = tsmelt, aes(x = date, y = value / 10000)) +
+   geom_point() +
+   scale_x_date() +
+   labs(y = "NDVI") +
+   facet_wrap(~ class, nrow = 3) +
+   theme_bw()
```

6 Exercise

- to be doable in 3 hours....



- combine concepts from previous lessons as well
- 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