Raster Analysis

Ben DeVries, Jan Verbesselt, Loïc Dutrieux, Valerio Avitabile

November 14, 2013

1 Learning Objectives

The learning objectives for this lecture are:

- explore the raster package and related packages used for typical raster analyses
- visualize and summarize raster data using the rasterVis and ggplot packages
- neighbourhood functions using the `focal()` function
- carry out a supervised classification (random forest) on a series of raster layers
- construct a raster sieve using the `clump()` function
- work with thematic rasters

2 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)
> library(rasterVis)
```

3 Exploring raster data

Introduction to Landsat

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. 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 1. Using different combination of these bands can be useful in describing land features and change processes.

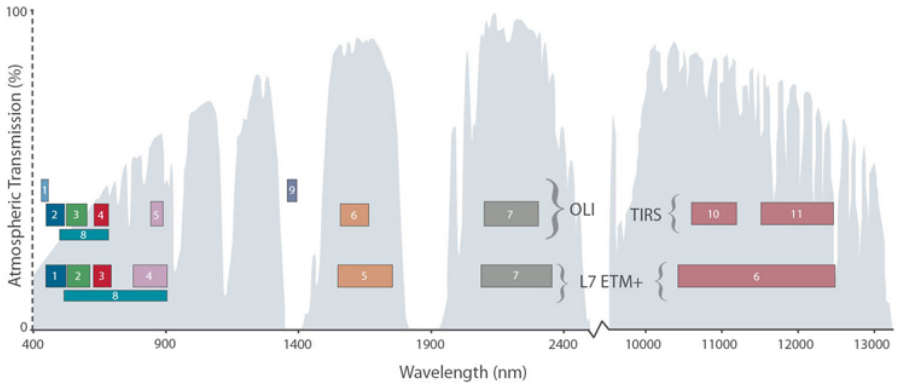

Figure 1: Bands included in the Landsat 7 (ETM+) and Landsat 8 (OLI/TIRS) sensors. Source: NASA.

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[1], 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.

```
> # load in the data
> data(GewataB2)
> data(GewataB3)
> data(GewataB4)
> # check out the attributes
> GewataB2
> # some basic statistics using cellStats()
> cellStats(GewataB2, stat=max)
> cellStats(GewataB2, stat=mean)
> # This is equivalent to:
> maxValue(GewataB2)
```

[1] http://ledaps.nascom.nasa.gov/

```
> # what is the maximum value of all three bands?
> max(c(maxValue(GewataB2), maxValue(GewataB3), maxValue(GewataB4)))
> # summary() is useful function for a quick overview
> summary(GewataB2)
> # plot the histograms of all bands
> hist(GewataB2)
> hist(GewataB3)
> hist(GewataB4)
> # put the 3 bands into a rasterBrick object to summarize together
> gewata <- brick(GewataB2, GewataB3, GewataB4)
> # 3 histograms in one window (automatic, if a rasterBrick is supplied)
> hist(gewata)
```

When we plot the histogram of the rasterBrick, the scales of the axes and the bin sizes are not equivalent, which could be problematic. This can be solved by adjusting these paramters in `hist()`, which would take some extra work and consideration. Alternatively, the rasterVis package has enhanced plotting capabilities which make it easier to make more attractive plots with these considerations in mind.

```
> # view all histograms together with rasterVis
> histogram(gewata)
```

The rasterVis package has several other raster plotting types inherited from the lattice package. 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.

Raster algebra (revisited)

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 dynamics, the normalized difference vegetation index (NDVI), explained in Lesson 5, takes advantage of this fact and is computed from Landsat bands 3 (visible red) and 4 (near infra-red).

In Lesson 5, we explored several ways to calculate NDVI, using direct raster algebra, `calc()` or `overlay()`. Since we will be using NDVI again later in this tutorial, let's calculated it again and store it in our workspace using `overlay()`.

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

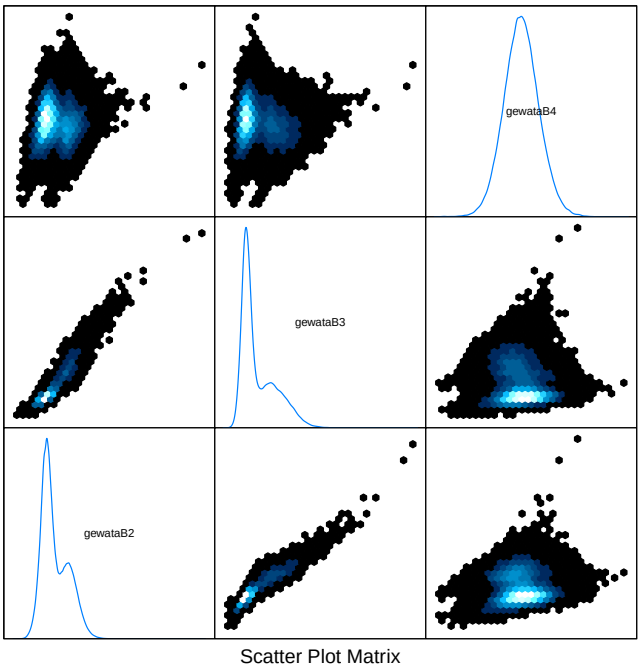

Figure 2: Scatter plot matrix for Landsat bands 2, 3, and 4 of the 'Gewata' sub scene.

Aside from the advantages of `calc()` and `overlay()` regarding memory usage, an additional advantage of these functions is the fact that the result can be written immediately to file by including the `filename="..."` argument, which will allow you to write your results to file immediately, after which you can reload in subsequent sessions without having to repeat your analysis.

Plotting our new NDVI 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
> # this creates an object of the 'extent' class
> class(e)
> print(e)
> # an extent object can be used for a variety of functions,
> # like crop() and plot()
> # now plot ndvi again, but only the extent you defined interactively
> plot(ndvi, ext=e)
> # alternatively, we can create a new raster with this revised extent
```

```
> ndviCrop <- crop(ndvi, e)
> # look at the attributes of the new raster
> ndviCrop
> plot(ndviCrop)
> # remove from the workspace
> rm(ndviCrop)
```

Neighbourhood functions

Neighbourhood or "Moving Window" functions in the raster package are carried out using the `focal()` function. In the help function (`?focal()`) you can find a number of examples of filters and their associated arguments. Applying a simple 3x3 mean filter on our ndvi raster is done as follows.

```
> # to save time, try on a small subset
> # first, define a small extent using drawExtent()
> plot(ndvi)
> e <- drawExtent()
> # make a test raster by cropping
> ndviTest <- crop(ndvi, e)
> # define a 3x3 matrix of equal weights
> w <- matrix(1/9, nc=3, nr=3)
> print(w)
> sum(w) # should be 1
> # pass this matrix to focal()
> ndviMean <- focal(ndviTest, w=w)
> # plot the result compared to original
> opar <- par(mfrow=c(1,2))
> plot(ndviTest, main="NDVI")
> plot(ndviMean, main="NDVI with 3x3 mean filter")
> par(opar)
```

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.

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: supervised (random forest) classification and unsupervised (k-means).

Supervised classification: Random Forest

The Random Forest classification algorithm is an ensemble learning method that is used for both classification and regression. In our case, we will use the method for classification purposes. Here, the random forest method takes random subsets from a training dataset and constructs classification trees using each of these subsets. Trees consist of "branches" and "leaves". Branches represent nodes of the decision trees, which are often thresholds defined for the measured (known) variables in the dataset. Leaves are the class labels assigned at the termini of the trees. Sampling many subsets at random will result in many trees being built. Classes are then assigned based on classes assigned by all of these trees based on a majority rule, as if each class assigned by a decision tree were considered to be a "vote". Figure 3 gives a simple demonstration of how the random forest method works in principle[2]

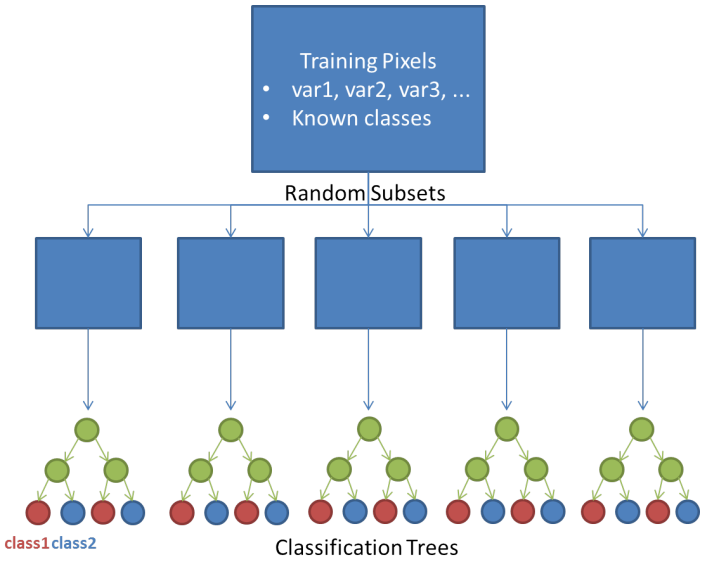

Figure 3: Schematic showing how the Random Forest method constructs classification trees from random subsets of a training dataset. Each tree determines the labels assigned based on the training dataset. Once all trees are assembled, classes are assigned to unknown pixels based on the class which receives the majority of votes based on all the decision trees constructed.

One major advantage of the Random Forest method is the fact that an 'Out of the Bag' (OOB) error estimate and an estimate of variable performance 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 by predicting the class associated with that value and comparing with the already known class. This process results in a confusion matrix, which we will explore in our analysis. 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, which will also be explored in our analysis.

We should first prepare the data on which the classification will be done. So far, we

[2]For a more complete description of the Random Forests classification method, see http://stat-www.berkeley.edu/users/breiman/RandomForests/cc_home.htm. For a "friendlier" introduction, see the presentation at http://www.slideshare.net/0xdata/jan-vitek-distributedrandomforest522013.

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 and check it out
> data(vcfGewata)
> vcfGewata

class       : RasterLayer
dimensions  : 1177, 1548, 1821996  (nrow, ncol, ncell)
resolution  : 30, 30  (x, y)
extent      : 808755, 855195, 817635, 852945  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=utm +zone=36 +datum=WGS84 +units=m +no_defs +ellps=WGS84 +towgs84=0,0,
data source : in memory
names       : vcf2000Gewata
values      : 0, 254  (min, max)

> plot(vcfGewata)
> summary(vcfGewata)

        vcf2000Gewata
Min.                0
1st Qu.            32
Median             64
3rd Qu.            75
Max.              254
NA's             8289

> 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
> # look at revised summary stats
> summary(vcfGewata)

        vcf2000Gewata
Min.                0
1st Qu.            32
Median             64
3rd Qu.            75
Max.              100
NA's            13712
```

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

```
> plot(vcfGewata)
> hist(vcfGewata)
```

To perform the classification in R, it is best to assemble all covariate layers (ie. those layers contaning predictor variable values) 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. In Lesson 5 we encountered different data types for rasters, so we will not go into more detail here.

```
> # multiply all values by 10000
> 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)
> # 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. There are several approaches that could be used to convert these classes to integer codes. In this case, we will first make a function that will reclassify the character strings representing land cover classes into integers based on the existing factor levels.

```
> # inspect the data slot of the trainingPoly object
> trainingPoly@data
```

```
   OBJECTID    Class
0         1  wetland
1         2  wetland
2         3  wetland
3         4  wetland
4         5  wetland
5         6   forest
```

```
6         7   forest
7         8   forest
8         9   forest
9        10   forest
10       11 cropland
11       12 cropland
12       13 cropland
13       14 cropland
14       15 cropland
15       16 cropland
```

```
> # the 'Class' column is actually an ordered factor type
> trainingPoly@data$Class
```

```
 [1] wetland  wetland  wetland  wetland  wetland  forest   forest   forest
 [9] forest   forest   cropland cropland cropland cropland cropland cropland
Levels: cropland forest wetland
```

```
> str(trainingPoly@data$Class)
```

```
 Factor w/ 3 levels "cropland","forest",..: 3 3 3 3 3 2 2 2 2 2 ...
```

```
> # define a reclassification function which substitutes
> # the character label for the factor level (between 1 and 3)
> reclass <- function(x){
+   which(x==levels(trainingPoly@data$Class))
+ }
> # use sapply() to apply this function over each element of the 'Class' column
> # and assign to a new column called 'Code'
> trainingPoly@data$Code <- sapply(trainingPoly@data$Class, FUN=reclass)
```

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.

```
> # assign 'Code' values to raster cells (where they overlap)
> classes <- rasterize(trainingPoly, ndvi, field='Code')
> # set the dataType of the raster to INT1U
> # see ?dataType for more information
> dataType(classes) <- "INT1U"
> # define a colour scale for the classes
> # corresponding to: cropland, forest, wetland
> cols <- c("orange", "dark green", "light blue")
> # plot without a legend
> plot(classes, col=cols, legend=FALSE)
> # add a customized legend
> legend("topright", legend=c("cropland", "forest", "wetland"), fill=cols, bg="white")
```

(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. Try passing this argument to the `rasterize()` command above).

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)
> # Note that in this plot, the 'class' legend is not meaningful
> # This plot is useful simply to check the available layers
```

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 `ggplot()`. (Note: to make the following plots more readable, we will add a column with the class labels as characters. But we will not use this column when performing the classification.)

```
> # add a label column to valuetable
> valuetable$label <- with(valuetable, ifelse(class==1, "cropland",
+                                             ifelse(class==2, "forest", "wetland")))
> # see ?ifelse() for more information
```

```
> # Now make the ggplots using valuetable$label to split the data into facets
>
> # 1. NDVI
> p1 <- ggplot(data=valuetable, aes(x=NDVI)) +
+   geom_histogram(binwidth=300) +
+   facet_wrap(~ label) +
+   theme_bw()
> p1
> # 2. VCF
> p2 <- ggplot(data=valuetable, aes(x=vcf2000Gewata)) +
+   geom_histogram(binwidth=5) +
+   labs(x="% Tree Cover") +
+   facet_wrap(~ label) +
+   theme_bw()
> p2
> # 4. Bands 3 and 4
> p3 <- ggplot(data=valuetable, aes(x=gewataB3, y=gewataB4)) +
+   stat_bin2d() +
+   facet_wrap(~ label) +
+   theme_bw()
> p3
>

> # 4. Bands 2 and 3
> p4 <- ggplot(data = valuetable, aes(x=gewataB2, y=gewataB3)) +
+   stat_bin2d() +
+   facet_wrap(~ label) +
+   theme_bw()
> p4
>
```

We can see from these distributions (e.g. Figure 4) 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
> # keep only the rows with containing no NA's
> valuetable <- na.omit(valuetable)

> # construct a random forest model
> # covariates (x) are found in columns 1 to 5 of valuetable
```

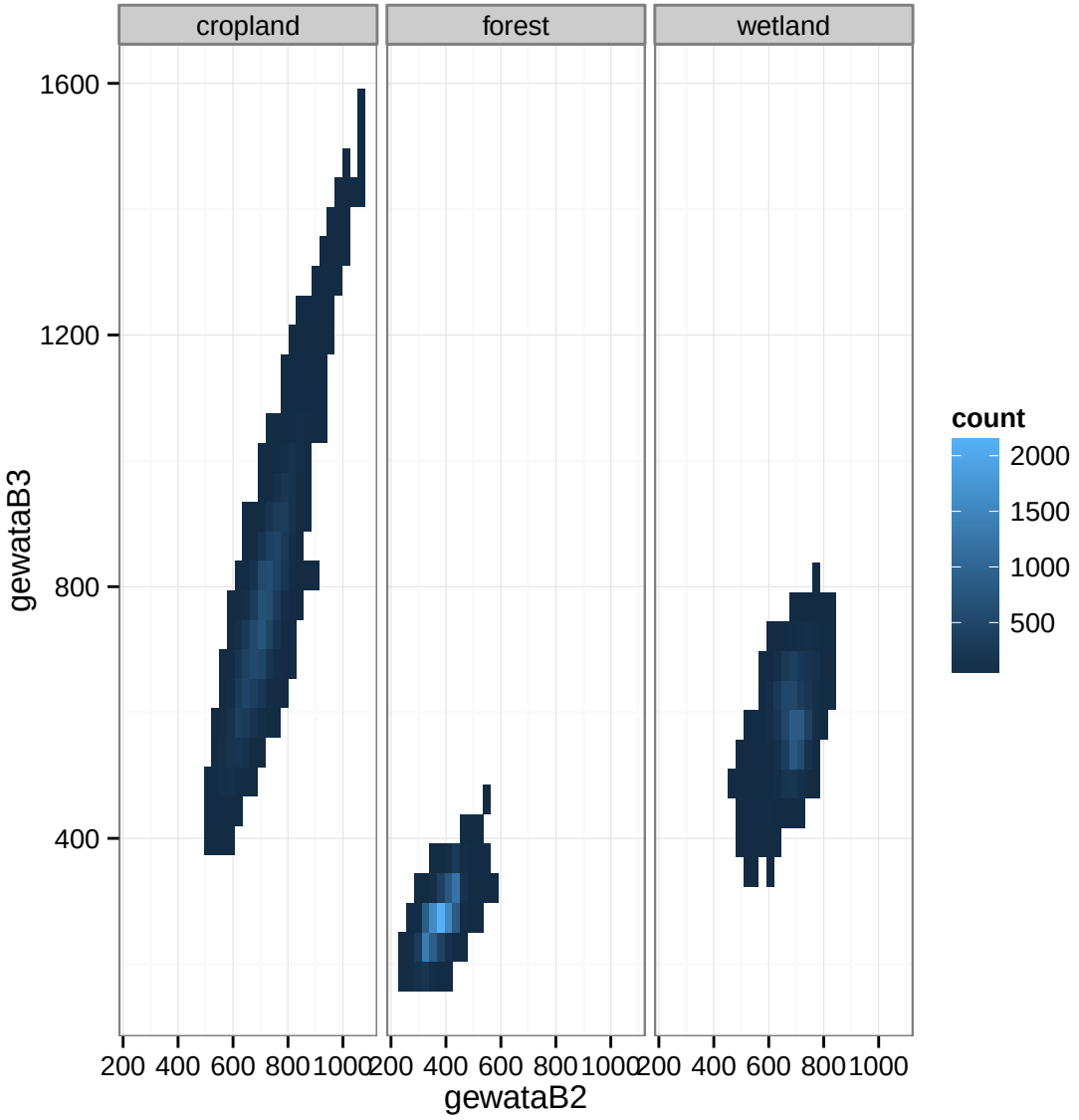

Figure 4: Band2-Band3 scatterplots for the three training classes.

```
> # training classes (y) are found in the 'class' column of valuetable
> # caution: this step takes fairly long!
> # but can be shortened by setting importance=FALSE
> 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
> modelRF
> class(modelRF)
> str(modelRF)
> names(modelRF)
> # inspect the confusion matrix of the OOB error assessment
> modelRF$confusion
> # to make the confusion matrix more readable
> colnames(modelRF$confusion) <- c("cropland", "forest", "wetland", "class.error")
> rownames(modelRF$confusion) <- c("cropland", "forest", "wetland")
> modelRF$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 importance (see `?importance()`). First, the mean decrease in accuracy indicates the amount by which the classification accuracy decreased based on the OOB assessment. Second, the Gini impurity coefficient gives a measure of class homogeneity. More specifically, the decrease in the Gini impurity coefficient when including a particular variable is shown in the plot [4].

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 (Figure 5). 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

[4] From Wikipedia: "Gini impurity is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it were randomly labeled according to the distribution of labels in the subset". See http://en.wikipedia.org/wiki/Decision_tree_learning

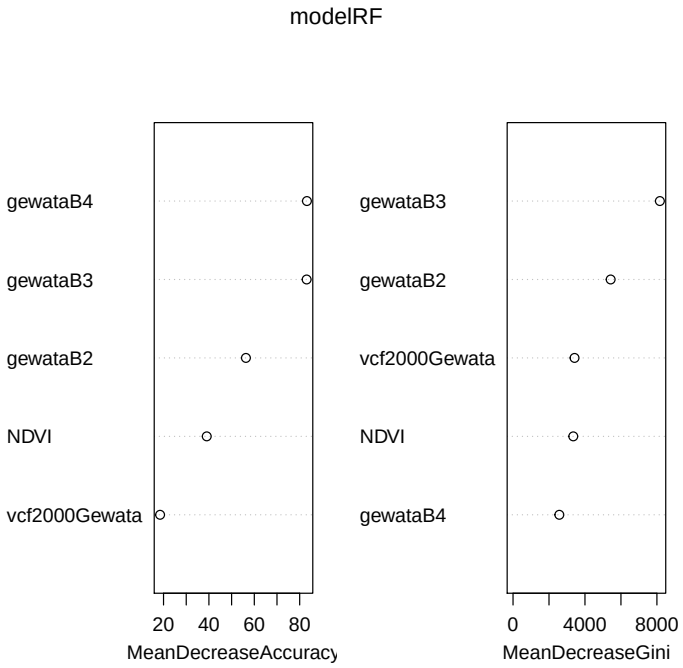

Figure 5: Variable importance plots for a Random Forest model showing the mean decrease in accuracy (left) and the decrease in Gini Impurity Coefficient (right) for each variable.

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.

```
> # 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 = cropland, 2 = forest, 3 = wetland
> cols <- c("orange", "dark green", "light blue")
> plot(predLC, col=cols, legend=FALSE)
> legend("bottomright", legend=c("cropland", "forest", "wetland"), fill=cols, bg="white")
```

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

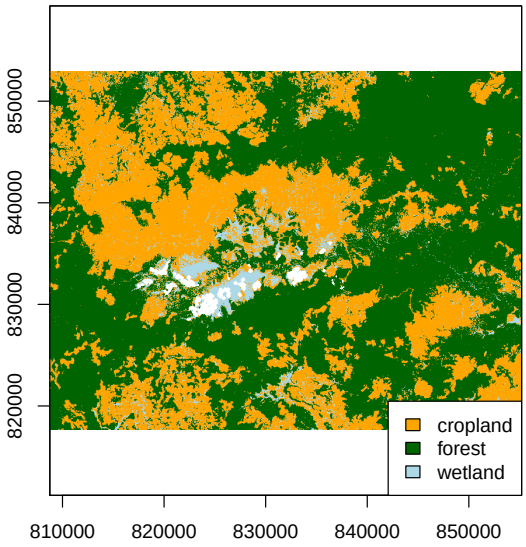

Figure 6: Resulting land cover map using a Random Forest classifier.

this step (rather than keeping all output in memory).

Unsupervised classification: k-means[5]

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[6].

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)
```

[5]This section will not be included in the lecture.

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

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

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

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)
> # 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, legend=FALSE, 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 7 that there is particularly high confusion

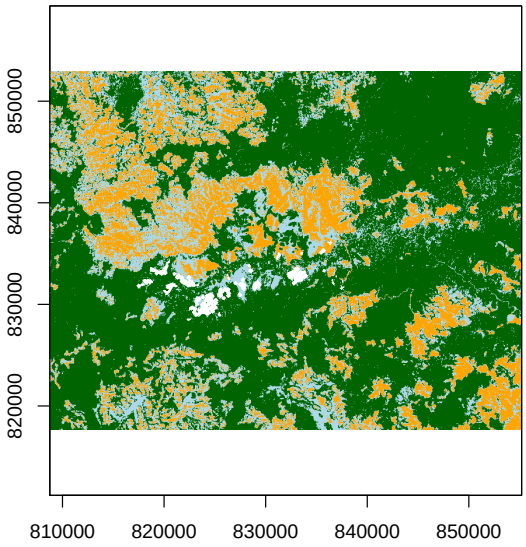

Figure 7: Unsupervised land cover classification resulting from k-means method. At this stage, the classes are undefined, so the colours are somewhat arbitrary.

between (what we might assume to be) the 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 with the Random Forest method 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 by apply a sieve. To do this, we will make use of the `clump()` function in the raster package.

```
> # Make an NA-value raster based on the LC raster attributes
> formask <- setValues(raster(predLC), NA)
> # assign 1 to all cells corresponding to the forest class
> formask[predLC==2] <- 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

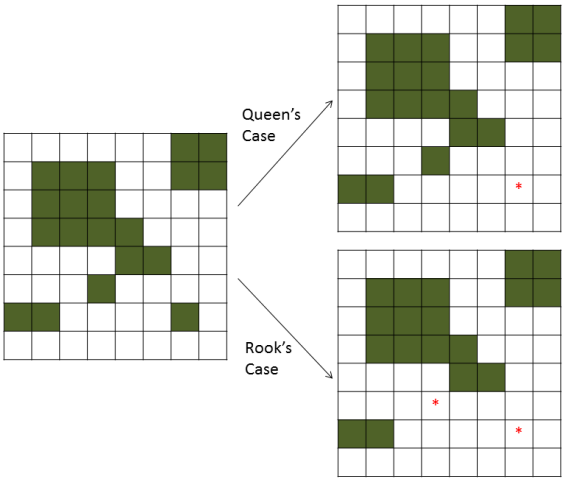

Figure 8: Application of a sieve on a forest mask. In the Queen's case, neighbours in all 8 directions are considered to be adjacent. In the Rook's case, only neighbours in the 4 NESW directions are considered adjacent, and diagonal neighbours are not considered.

especially want to remove single forest pixels, as they may be a result of errors, or may not fit our definition of 'forest'. In this section, we will construct 2 types of sieves to remove these types of pixels, following 2 definitions of "adjacency". In the first approach, the so-called "Queen's Case", neighbours in all 8 directions are considered to be adjacent. If any pixel cell has no neighbours in any of these 8 directions, we will remove that pixel by assigning an NA value. First, we will use the `clump()` function in the raster package to identify clumps of raster cells. This function arbitrarily assigns an ID to these clumps.

```
> # Group raster cells into clumps based on the Queen's Case
> forestclumps <- clump(formask, directions=8)
> plot(forestclumps)
```

When we inspect the frequency table with `freq()`, we can see the number of raster cells included in each of these clump IDs.

```
> # assign freqency table to a matrix
> clumpFreq <- freq(forestclumps)
> head(clumpFreq)
> tail(clumpFreq)
```

We can use the "count" column of this frequency table to select clump ID's with only 1 pixel - these are the pixel "islands" that we want to eventually remove from our original forest mask.

```
> # Coerce freq table to data.frame
> clumpFreq <- as.data.frame(clumpFreq)
> # which rows of the data.frame are only represented by one cell?
> which(clumpFreq$count==1)
```

```
> # which values do these correspond to?
> clumpFreq$value[which(clumpFreq$count==1)]
> # put these into a vector of clump ID's to be removed
> excludeID <- clumpFreq$value[which(clumpFreq$count==1)]
> # make a new forest mask to be sieved
> formaskSieve <- formask
> # assign NA to all clumps whose IDs are found in excludeID
> formaskSieve[forestclumps %in% excludeID] <- NA
> # zoom in to a small extent to check the results
> # Note: you can define your own zoom by using e <- drawExtent()
> e <- extent(c(811744.8, 812764.3, 849997.8, 850920.3))
> opar <- 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(opar) # reset plotting window
```

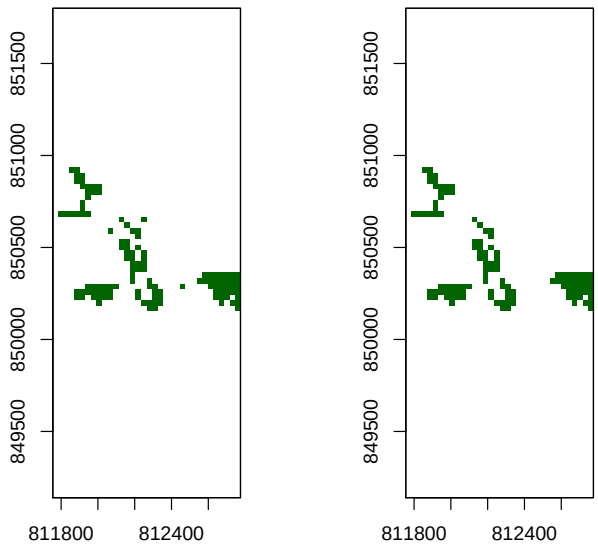

Figure 9: Zoom of a forest mask before (left) and after (right) application of a sieve using the Queen's Case condition.

We have successfully removed all 'island' pixels from the forest mask using the `clump()` function. We can adjust our sieve criteria to only only directly adjacent (NESW) neighbours: the so-called "Rook's Case". To accomplish this, simply repeat the code above, but supply the argument `directions=4` when calling `clump()`. The results of this type of sieve is shown in 10.

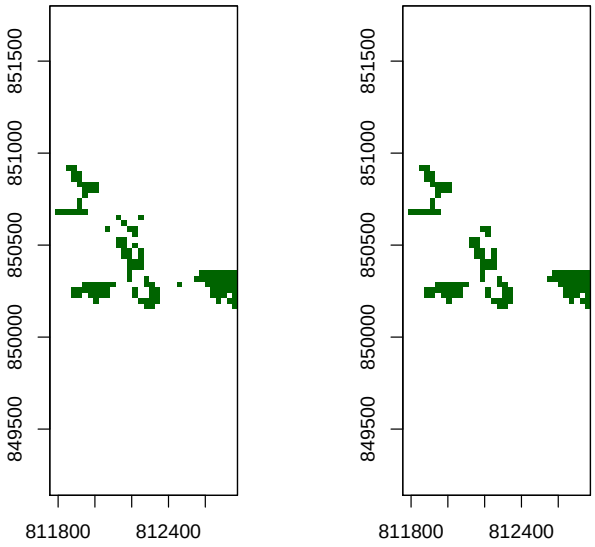

Figure 10: Zoom of a forest mask before (left) and after (right) application of a sieve using the Rook's Case condition.

We could take this approach further and apply a minimum mapping unit (MMU) to our forest mask. How could you adjust the above sieve to remove all forest pixels with area below 0.5 hectares? Consider the fact that Landsat pixels are 30m by 30m, and that one hectare is equal to 10000m^2.

Working with thematic rasters

As we have seen with the land cover rasters we derived using the random forest or k-means methods above, 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 Land Use and Land Cover (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 data.frame defining these classes is also included in the rasta package:

```
> 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
```

In some cases it might be more useulf to visualize only one class at a time. The `layerize()` function in the raster package does this by producing a rasterBrick object with each layer representing the class membership of each class as a boolean.

```
> classes <- layerize(lulc)
> plot(classes)
> # layer names follow the order of classes in the LUT
> names(classes) <- LUTGewata$Class
> plot(classes, legend=FALSE)
```

Now each class is represented by a separate layer representing class membership of each pixel with 0's and 1's. If we want to construct a forest mask as we did above, this is easily done by extracting the fifth layer of this rasterBrick and replacing 0's with NA's.

```
> forest <- raster(classes, 5)
> # is equivalent to:
> forest <- classes[[5]]
> # or (since the layers are named):
> forest <- classes$forest
> # replace 0's (non-forest) with NA's
> forest[forest==0] <- NA
> plot(forest, col="dark green", legend=FALSE)
```

5 Exercise

Perform another Random Forest classification with 6 classes (found in the lulcGewata raster dataset) using all (or some) of the bands used in this tutorial (b2, b3, b4, ndvi and vcf). This time, derive your own training data from the lulcGewata image provided with this package. This can be done interactively by creating polygons using the `drawPoly()` command

Your report on the results and accompanying code should include the following:

- a plot of the original lulcGewata raster with a meaningful legend (ie. classes as characters)
- a plot of your training polygons
- a summary of the resulting randomForest model object
- the resulting thematic map with meaningful legend (as above)
- the output OOB confusion matrix with accuracy per class (with correct row and column headings)
- the variable importance ranks (mean accuracy decrease and mean Gini coefficient decrease)
- **Extra points (if you have time)**: your own confusion matrix derived from an overlay of the original lulcGewata raster with your own LULC raster and resulting accuracy statistics. How do the two confusion matrices compare with each other?

In this exercise, also address the following questions:

1. Which classes have the highest accuracy? Lowest?
2. Is the importance ranking of the input bands different in this case to the 3-class classification we did earlier? If so, how, and what has changed in this case?
3. Can you say something about class separability? Produce a series of facet_wrap plots using `ggplot()` to compare the land cover classes. Are there any classes which show high overlap? Is this consistent with the confusion matrix derived as part of your Random Forest model?

You can address these questions directly in your script by inserting comments (preceded by a #) throughout your script and then submit the exercise as one file.

The following code should help to get you started with selecting training data and proceeding with the supervised classification:

```
> # load in the training classes and Look-Up Table (LUT)
> data(lulcGewata)
> data(LUTGewata)

> # plot lulcGewata with a meaningful legend (see LUTGewata)
> # make sure the classes correspond correctly!
> cols <- c("orange", "light green", "brown", "light pink", "dark green", "light blue")
> plot(lulcGewata, col=cols, legend=FALSE)
> legend("topright", legend=LUTGewata$Class, fill=cols)

> # draw a SpatialPolygons object in area purely represented by cropland
> # Note that drawPoly() doesn't work once you have added a legend() to the plot
> # so, first plot the raster again without the legend
> plot(lulcGewata, col=cols, legend=FALSE)
> cropland <- drawPoly(sp=TRUE)
> # click on "Finish" in the top-right corner (if using Rstudio) when finished
> # this outputs an sp (SpatialPolygons) object with 1 row (feature)
> # if you want more training data for a particular class:
> # append another polygon onto the same object using gUnion() of the rgeos package
> cropland <- gUnion(cropland, drawPoly(sp=TRUE))

> # when you are finished with this class,
> # be sure to set the coordinate reference system (CRS)
> projection(cropland) <- projection(lulcGewata)
> # and check
> projection(cropland)
> plot(lulcGewata)
> plot(cropland, add=TRUE)

> # convert it to a SpatialPolygonsDataFrame (ie. add a @data slot)
> cropland <- SpatialPolygonsDataFrame(cropland, data=data.frame(
+   class="cropland"), match.ID=FALSE)

> # do the same for the other 5 classes...
> # you may have to zoom in first for (e.g.) coffee investment areas
> plot(lulcGewata, col=cols, legend=FALSE)
> e <- drawExtent() # zoom into a coffee area
> plot(lulcGewata, col=cols, legend=FALSE, ext=e)
> # now define a training polygon
> coffee <- drawPoly(sp=TRUE)
> projection(coffee) <- projection(lulcGewata)
> # check
> plot(lulcGewata)
> plot(coffee, add=TRUE)
> # convert to SpatialPolygonsDataFrame
> coffee <- SpatialPolygonsDataFrame(coffee, data=data.frame(
+   class="coffee investment area"), match.ID=FALSE)
```

```
> # once all polygons have been drawn,
> # fusing them into one SpatialPolygons object is problematic,
> # because the Polygon ID's are not unique
> # use spChFIDs() from the sp package to solve this
> # e.g. for the cropland features
> cropland <- spChFIDs(cropland, "cropland")
> forest <- spChFIDs(forest, "forest")
> coffee <- spChFIDs(coffee, "coffee")
> # etc...

> # now they can be bound (2 at a time) as one object using spRbind (maptools)
> trainingPoly <- spRbind(cropland, forest)
> trainingPoly <- spRbind(trainingPoly, coffee)
> # etc...
> # check
> trainingPoly@data
> plot(lulcGewata)
> plot(trainingPoly, add=TRUE)
> ### now proceed with the Random Forest classification
> ### using the Gewata covariates as done earlier in the exercise
```