BIOM4031 Tutorial 3: Raster Basics (Single Layers)

Figure 1: Snow leopard (*Panthera uncia*) caught on camera trap

Introduction

In Tutorials 1 and 2, you learned how to interact with and manipulate spatial vector data in R. In this tutorial, we will introduce the other main spatial data model: raster data. We will demonstrate how spatial raster data are loaded, stored and visualized in R and learn how to perform a range of common operations involving single raster layers, including global statistics, focal statistics (terrain analysis), and raster classification.

Today's tutorial is loosely based on a study by Watts et al. (2019) that investigated the habitat preferences of snow leopards (*Panthera uncia*) in the Himalayan border region of Ladakh in Northern India. The snow leopard is one of the world's most elusive big cats found in remote, mountainous regions of Central Asia. Its population is declining as a result of habitat loss, poaching and climate change and the species is currently regarded as 'Vulnerable' by the IUCN. Leopards also predate domestic livestock leading to conflict with humans and retaliatory killings in some areas.

Watts et al. (2019) used data from camera traps and direct observations to model snow leopard distributions in Ladakh based on six habitat characteristics. They then investigated whether human settlements in areas of high leopard suitability were at greater risk of livestock predation or benefited from enhanced ecotourism opportunities. We won't go as far as replicating species distribution models. However, this case study presents an opportunity to work with a range of common raster data types in an interesting location!

Learning objectives

By the end of the tutorial you will know how to:

1. Import raster data into R and be familiar with the structure of spatial raster objects.
2. Visualize spatial rasters using `ggplot`
3. Calculate summary statistics for individual raster layers (i.e. global statistics)
4. Derive new raster layers through terrain analysis (a type of 'focal statistics')
5. Perform basic raster classification based on pixel values
6. Measure areas in raster layers.

Data

All of the data used in today's tutorial come from the public domain. In today's data folder you will find:

leopards.gpkg Snow leopard camera trap and observation data obtained from Watts et al. (2019)

villages.gpkg Locations of villages in the Ladakh region obtained from Watts et al. (2019)

dem.tif A 90m resolution digital elevation model of Ladakh from Shuttle Radar Topography Mission and accessed through the R package `geodata`.*

landcover.tif A land cover map of Ladakh obtained from ESA WorldCover.

ladakh.shp A polygon vector of the Ladakh region (generated by me as borders in this region are disputed China/India/Pakistan and unbiased shapefiles are hard to come by!).

India.shp A basemap of India obtained through the R package `geodata`*

*R code used to access elevation and landcover data through R can be found in the Appendix of this worksheet.

Load packages

For this tutorial we will need the `sf` and `tidyverse` packages that we used previously. We will also need to load the `terra` package for interacting with spatial rasters and the `tidyterra` package which is used for linking `terra` to tidyverse functions (including `ggplot` for data visualisation). Finally, we will load the package `colorspace` which is one of the best R packages for adding custom color scales to plots.

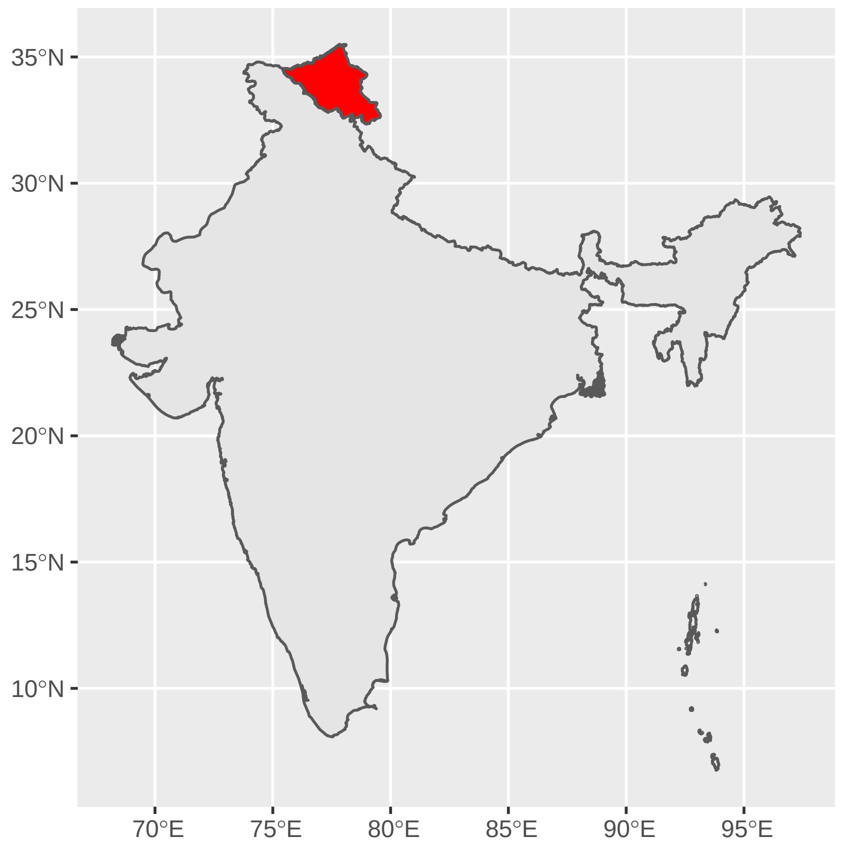
```
library(sf)
library(tidyverse)
library(terra)
library(tidyterra)
library(colorspace)
```

1. Importing rasters in R

To provide some geographic context for our case study, let's first read in the India basemap and polygon of the Ladakh region from the Tutorial 3 data folder and plot them using `ggplot`:

```
india = st_read("data/india.gpkg")
ladakh = st_read("data/ladakh.gpkg")
```

```
ggplot() +
  geom_sf(data = india) +
  geom_sf(data = ladakh, fill = 'red')
```



Our study site is located at the northern extreme of India, at the Himalayan border with China and Pakistan. It is a mountainous and rugged region: perfect habitat for snow leopards. To he terrain better and familiarize ourselves with raster data in R, we'll start of by load in our digital elevation map (DEM) of Ladakh and taking a look at it. The DEM is stored in a `geotiff` file called `dem.tif`. We can read this into R using the `rast` function from package `terra`:

```
dem = rast('data/dem.tif')
dem
```

```
## class       : SpatRaster
## dimensions  : 379, 496, 1  (nrow, ncol, nlyr)
## resolution  : 0.008333333, 0.008333333  (x, y)
## extent      : 75.43333, 79.56667, 32.34167, 35.5  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## source      : dem.tif
## name        : IND_elv_msk
## min value   :        2575
## max value   :        7181
```

If we open up `dem` you can see that `terra` stores raster datasets in a class of object called a `SpatRaster`. As with our `sf` objects the summary of a `SpatRaster` contains several key pieces of information. It tells us the dimensions of the raster: it has 379 rows and 496 columns and consists of a single layer. It tells us that the resolution (i.e. dimensions of each cell) is approximately 0.008 x 0.008 degrees, or ~ 90m. It gives us the coordinates of the rectangle around the raster (the extent) and tells us that these coordinates are in WGS84 latitude and longitude. It also tells us the name of each layer and prints a summary of the minimum and maximum cell values.

There are several utility functions in `terra` that allow us to extract these bits of information from the raster summary:

```
# Number of rows/columns/layers
nrow(dem)
ncol(dem)
nlyr(dem)

# Number of cells/pixels in each layer
ncell(dem)

# Resolution
res(dem)

# Extent
ext(dem)

#Layer names
names(dem)
```

If you want, you can even extract the cell values from the raster using `values`, which gives us a column for each layer and a row for each cell (186984 in this case).

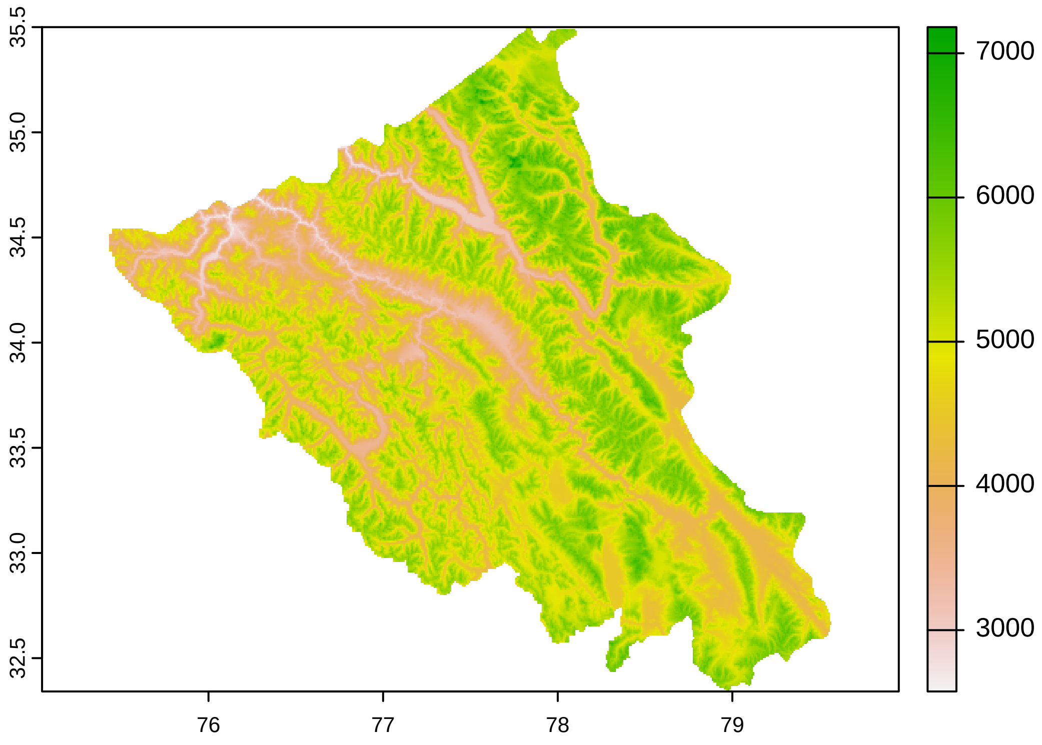
```
# These are equivalent:
values(dem)
dem[]
```

You can see that our `dem` mainly contains missing (NA) values for these first cells. Let's plot the data to have a look at it.

2. Plotting raster data

As with sf, there is a default plotting method for `SpatRaster` objects which we call using `plot`:

```
plot(dem)
```



This function is a bit more useful than the `sf` equivalent. It is optimised for speedy rendering of large rasters so it can be very useful for inspecting your data. However, it lacks the versatility of dedicated graphics packages like `ggplot` for styling and overlaying multiple layers. We can plot `SpatRaster` objects with `ggplot` using a dedicated 'geom' called `geom_spatraster` that is imported by package `tidyterra`:

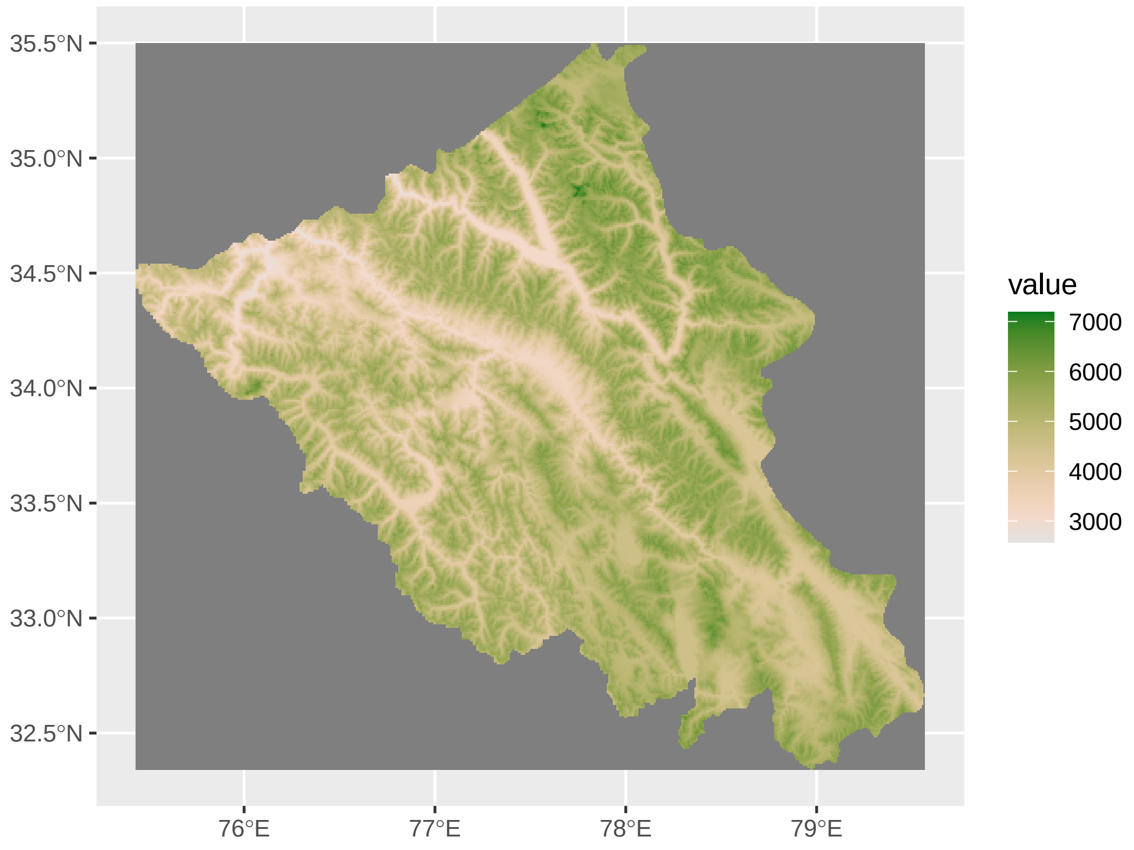
```r
dem.plot =
ggplot() +
  geom_spatraster(data=dem)
```

You'll notice ggplot is a bit 'stickier' and slower to render than the base function. We don't need to specify any fill aesthetics in `geom_spatraster` because it knows to fill the cells according to their values. There are a couple of other things to notice. Firstly, `ggplot` plots the missing (NA) values as dark gray. This is useful to remind you that all rasters are rectangular. However, we generally don't want to plot the missing values! As with `geom_sf`, adding a `geom_spatraster` also adds a spatial (fixed aspect ratio) coordinate reference system. Finally, you can see that `ggplot` has applied its default blue colour gradient for continuous data. We probably want to change this as it's not particularly intuitive for elevation data.

2b. Styling raster layers

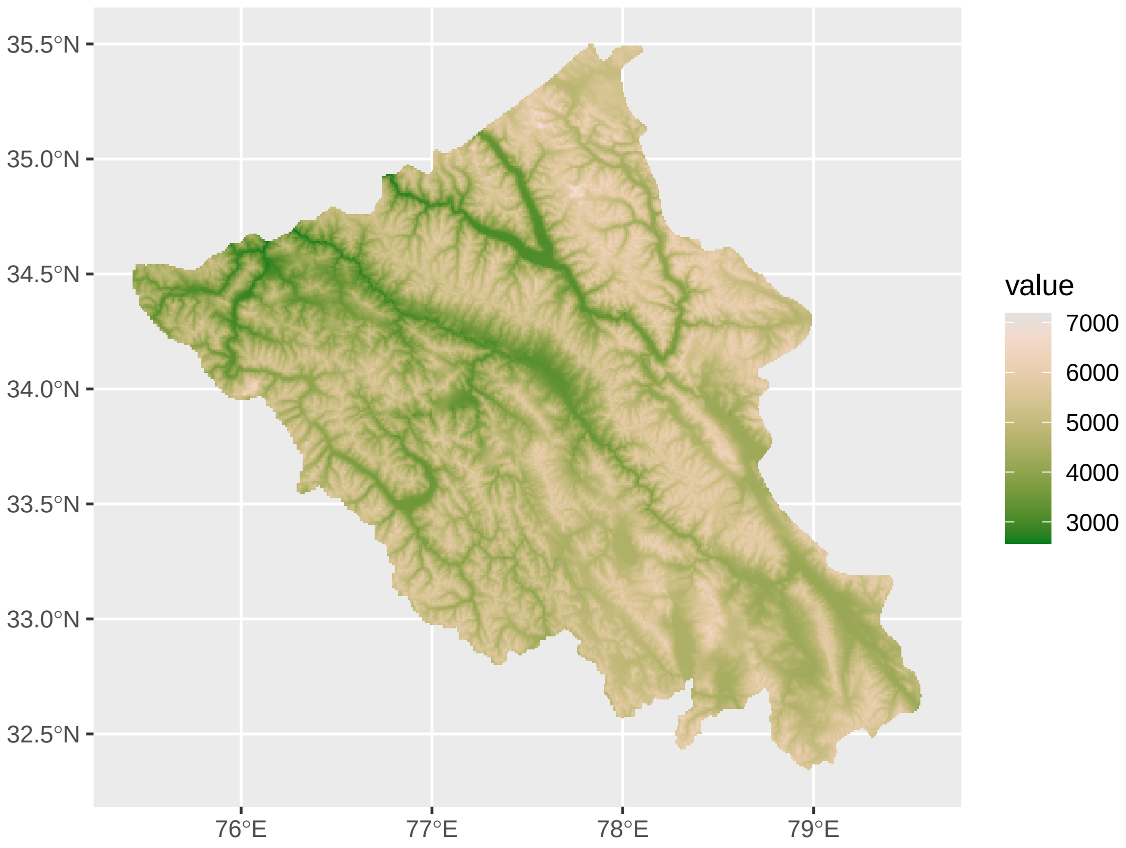
In order to change the colour gradient of a raster and hide NA values, we need to apply a custom fill scale to our `ggplot`.As we saw in Tutorials 1 and 2, there are several ways of doing this. Package `colorspace` has lots of nice predefined palettes which we can view using `hcl_palettes(plot = T)`. Let's try changing our fill scale of our DEM to something more appropriate like "Terrain 2". Because we have continuous data and want a sequential palette we use `scale_fill_continuous_sequential`.

```
dem.plot +
  scale_fill_continuous_sequential('Terrain 2')
```



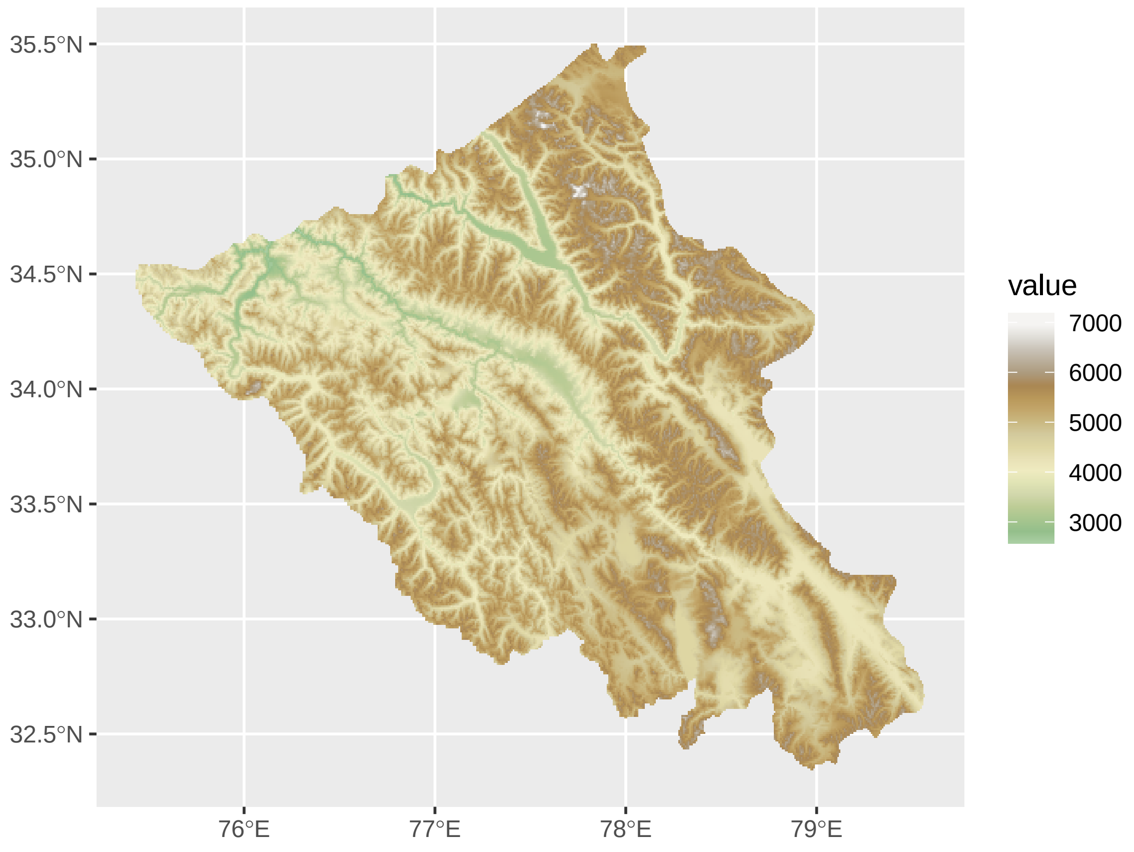
If we don't want to plot the NA values then we can set the `na.value` to transparent. We might also want to invert our fill gradient as the mountains in this area certainly aren't forested (Weirdly a lot of `colorspace` palettes are reversed by default so that higher values get darker colours, so we need to set `rev=FALSE` to invert it)

```
dem.plot +
  scale_fill_continuous_sequential('Terrain 2',na.value='transparent',rev=F)
```



That looks a bit better. However, for elevation data there are some stunning palettes called 'hypsometric tints' that are provided with `tidyterra` and are designed to replicate the kind of natural tones you would see in atlases. You can browse the palettes that are available here and add them to the plot with `scale_fill_hypso_c` (with the 'c' at the end for continuous data):

```
dem.plot =
dem.plot +
  scale_fill_hypso_c('wiki-2.0_hypso')

dem.plot
```



That looks even better!

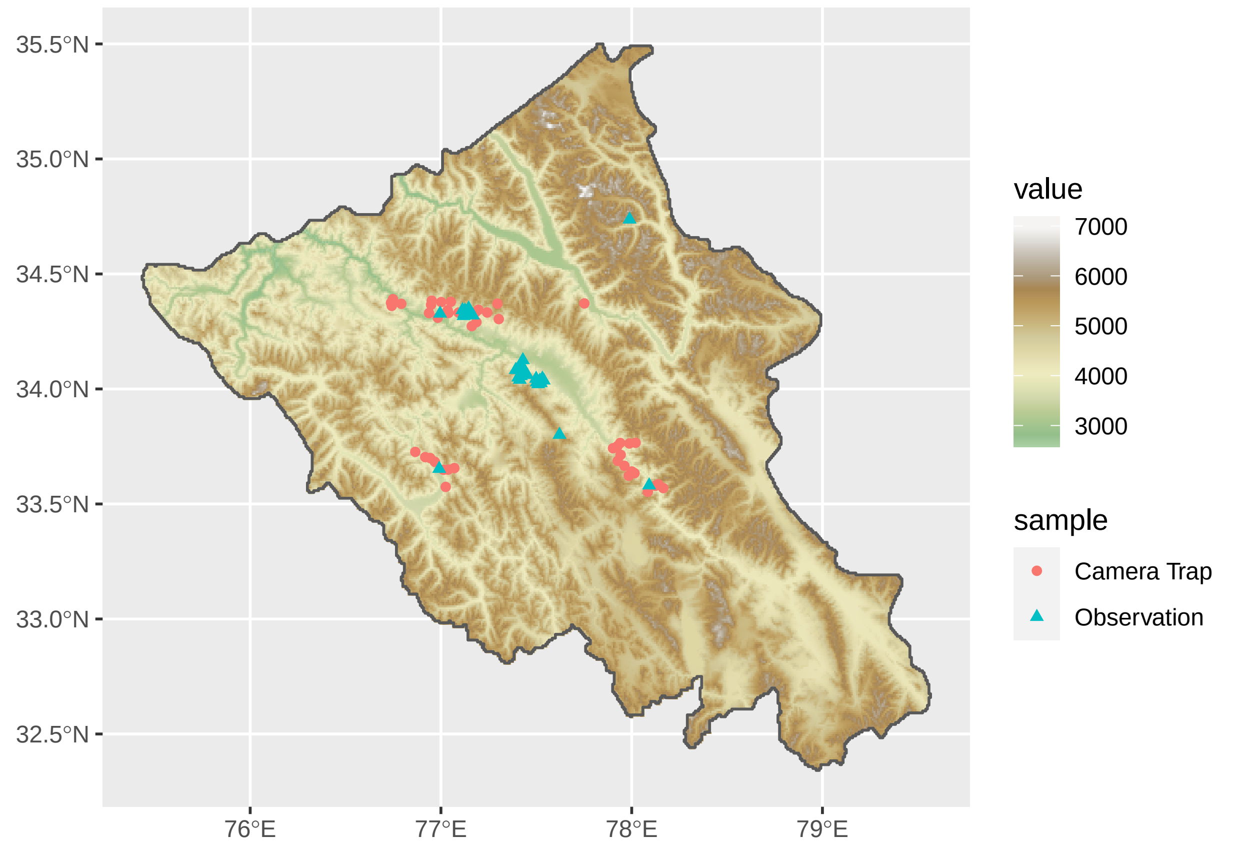
2b. Adding additional layers

Using `ggplot` we can add layers to our map in the same way as previous tutorials, including spatial vectors read in using `sf`. To demonstrate, let's load in our snow leopard observations from the Tutorial 3 data folder, along with the polygon boundary of the Ladakh region.

```
leopards = st_read('data/leopards.gpkg')
ladakh = st_read('data/ladakh.gpkg')
```

We add new spatial vector layers to our `dem.plot` exactly as we have done previously using `geom_sf`. If we open up the `leopards` dataframe you can see we have a column called `sample` specifying whether each observation was from a camera trap or direct observation, so we could also colour our points based on the sampling method:

```
dem.plot +
  geom_sf(data = leopards, mapping = aes(colour = sample,shape=sample)) +
  geom_sf(data = ladakh, fill = 'transparent')
```



So now we have an elevation map of our study area, we might want to calculate some summary statistics from the raster layers we have imported and derive some additional layers for use in our analysis.

3. Global statistics

Global statistics take a raster and produce a single value summarizing the data in each layer. In `terra` they are called with the `global` function. For example, if we want to find the average elevation in Ladakh we could do:

```
#Note that we need to tell the function to ignore NA values using na.rm = TRUE
global(dem, fun=mean, na.rm=TRUE)
```

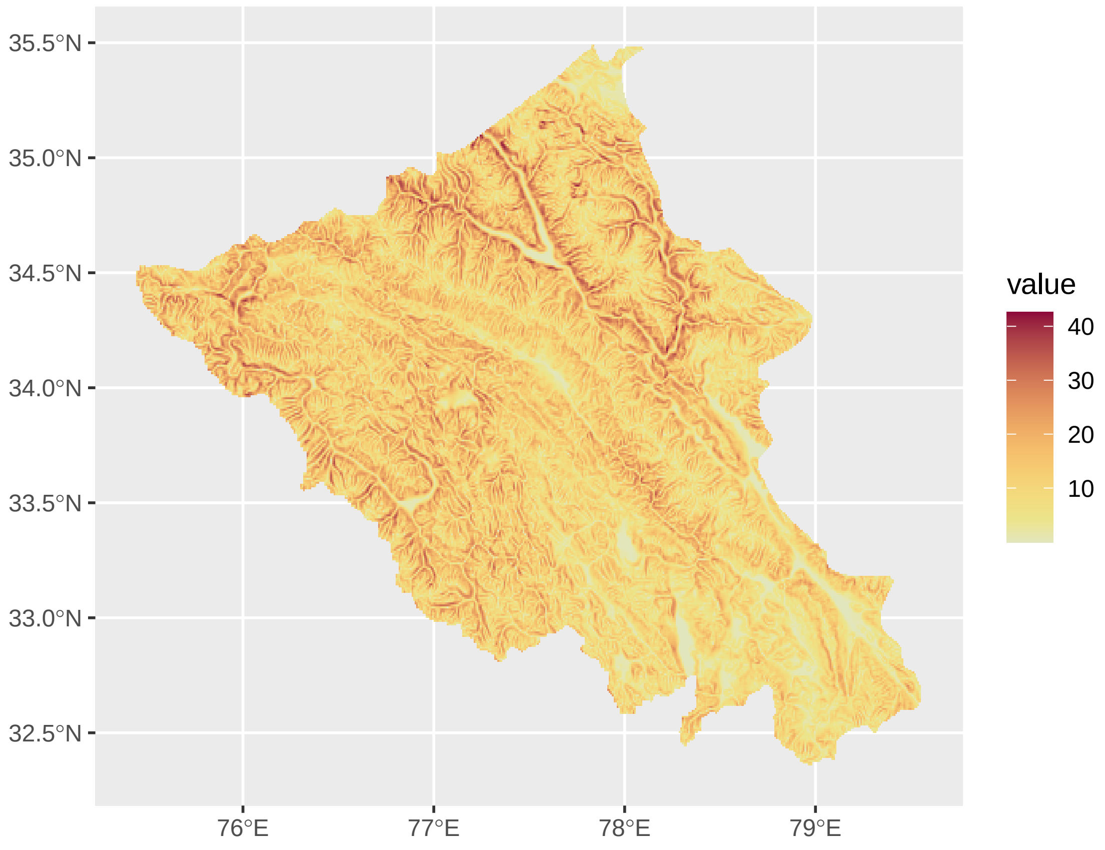
The average elevation is 4846 m - pretty high! We could do the same for minimum and maximum elevation using `fun=min` or `fun=max`, although there is a handy function called `minmax` that does this for us:

```
minmax(dem)
```

4. Focal statistics (Terrain analysis)

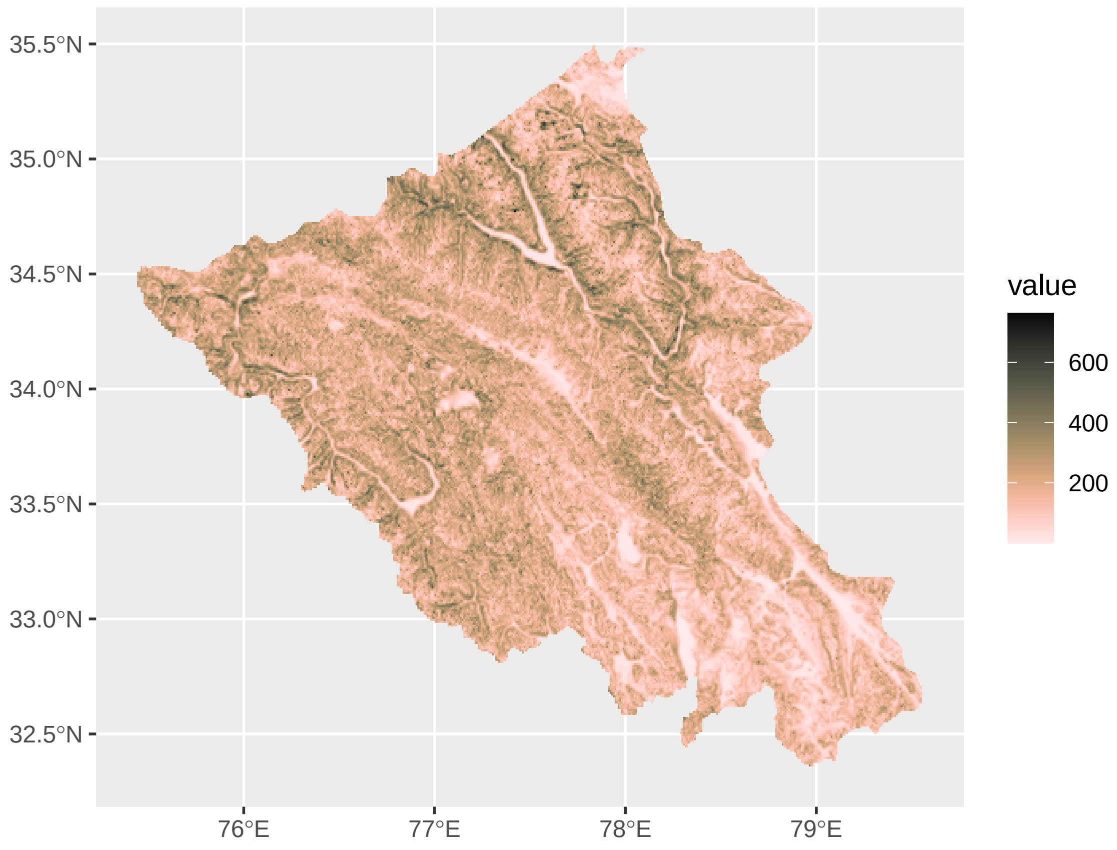
In addition to global statistics, there are also raster operations called 'focal statistics' where a new value is calculated for each cell based on the values of the cells that surround it. They are described in more detail in Section 4.3.4 of Geocomputation with R. However, there are some useful inbuilt ones in `terra` that we can use to calculate landscape metrics like slope, aspect and the 'terrain ruggedness index' (TRI) which were used by Watt et al. (2019) for predicting snow leopard habitat preferences. We apply them using the `terrain` function. For example to calculate slope angle we use:

```
slope = terrain(dem, v = 'slope')

ggplot() +
  geom_spatraster(data=slope) +
  scale_fill_continuous_sequential('Heat', na.value = 'transparent')
```



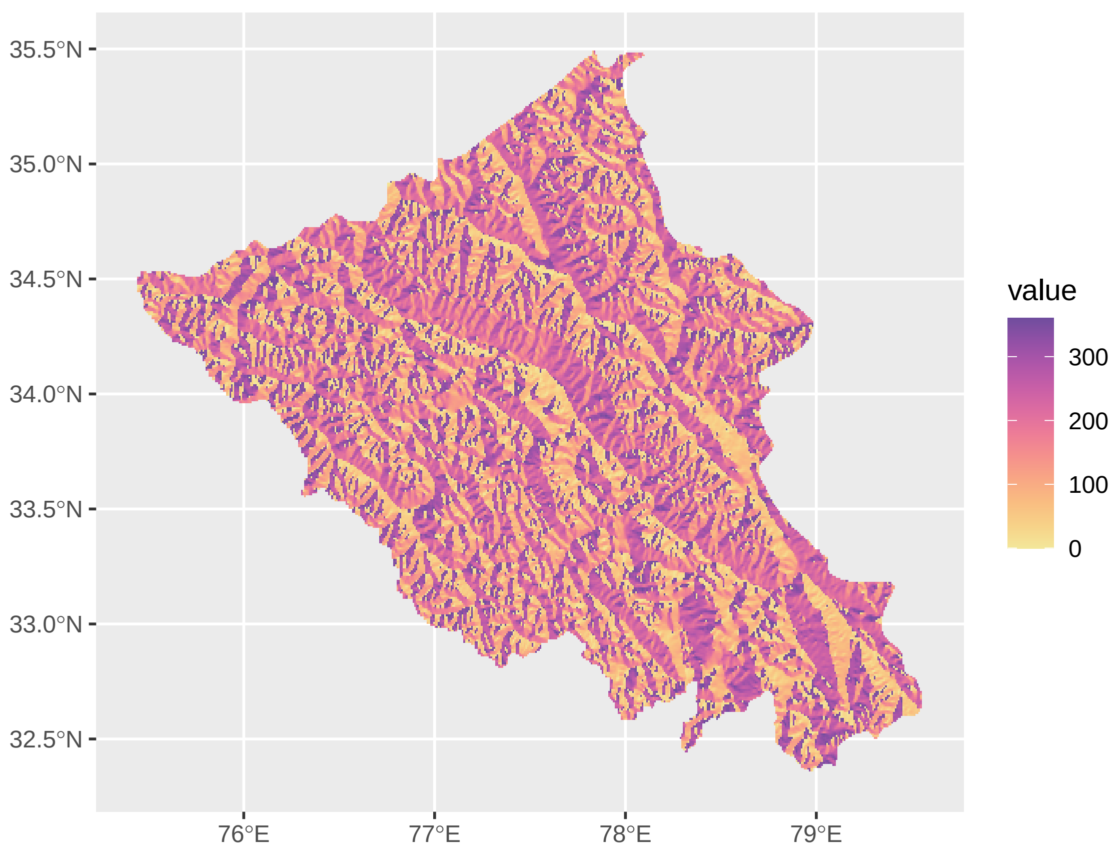
To calculate the topographic ruggedness index (a measure of how different each cell is from those surrounding it):

```
TRI = terrain(dem, v = 'TRI')

ggplot() +
  geom_spatraster(data=TRI) +
  scale_fill_continuous_sequential('Turku', na.value = 'transparent', rev = F)
```



To calculate the aspect (i.e. direction that a slope faces):

```
aspect = terrain(dem, v = 'aspect')

ggplot() +
  geom_spatraster(data=aspect) +
  scale_fill_continuous_sequential('Sunset', na.value = 'transparent')
```

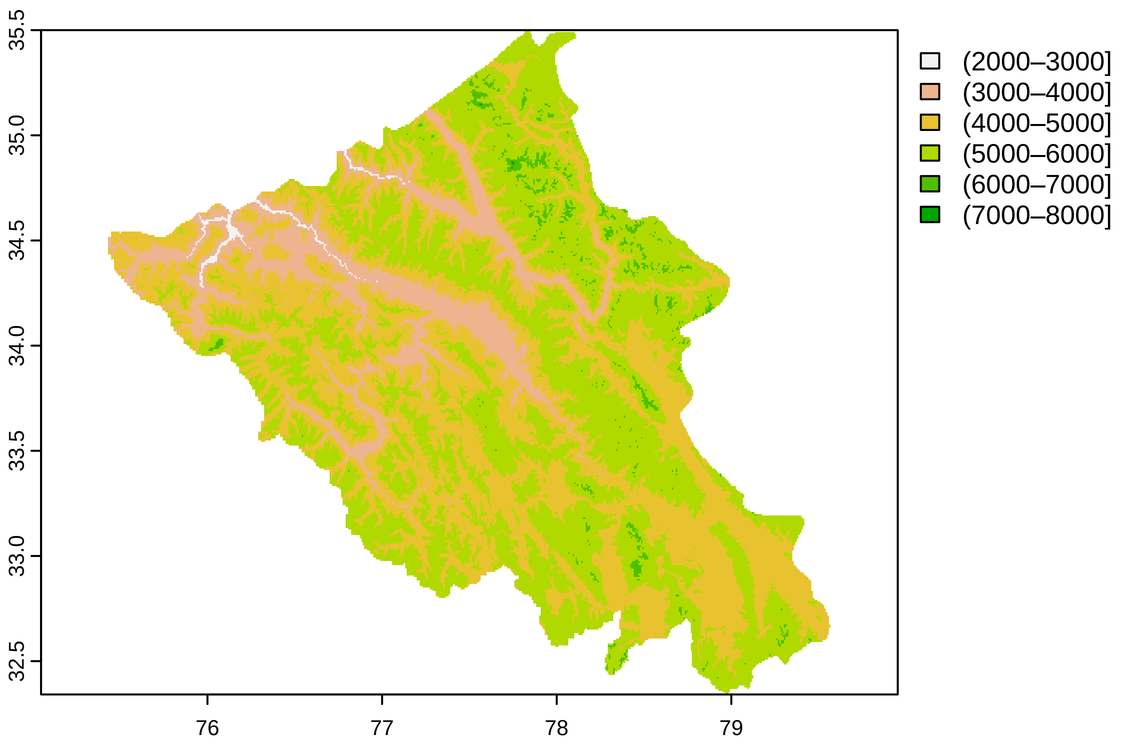


You can see that in these examples, the aspect is hard to plot properly on a continuous scale because it is a circular variable (in degrees), meaning that the values at the extremes of the colour scale (0 and 360) are actually identical. In order to plot aspect, we might want to classify our cells into discrete categories e.g. slopes facing North, East, South and West.

5. Raster classification

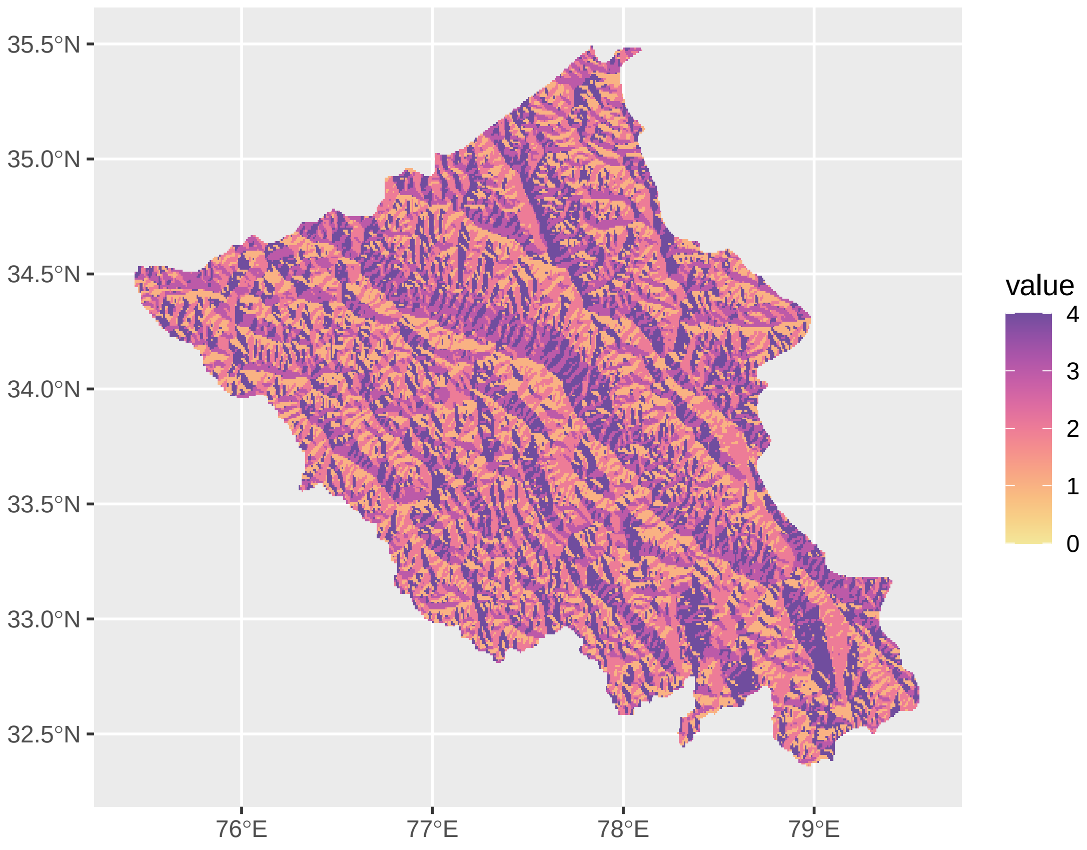
Raster classification involves grouping cells within certain ranges of values into defined classes. In `terra`, we do it using the `classify` function. One of the easiest ways to classify a raster is to just give `classify` a sequence of break points to use for grouping the data. For example, this will classify our elevation data into 1000m bands:

```
classify(dem,rcl = seq(2000,8000,1000)) %>% plot
```



If we want to do anything more sophisticated we need to build a reclassification table with three columns (from - to - becomes) containing the ranges for each of the classes we want to split the data into and the new value for labeling that class. So, if we want to classify our `aspect` raster into 90 degree intervals around the quadrants of a compass (N, E, S, W) we would do:

```r
rcl = matrix(c(0,45,1,
               45,135,2,
               135,225,3,
               225,315,4,
               315,360,1),
             ncol=3, byrow=T)

aspect = classify(aspect,rcl=rcl)

ggplot() +
  geom_spatraster(data=aspect) +
  scale_fill_continuous_sequential('Sunset', na.value = 'transparent')
```



SpatRasters are only able to store numbers, so we have had to give our new classes slightly uninformative numeric values. However, we can provide a 'raster attribute table' (RAT), or dataframe, which specifies how values in the raster map to categorical variables. The RAT needs to contain an id column which contains the cell values and then any number of other columns containing the categorical variables. In our case we want an aspect column:

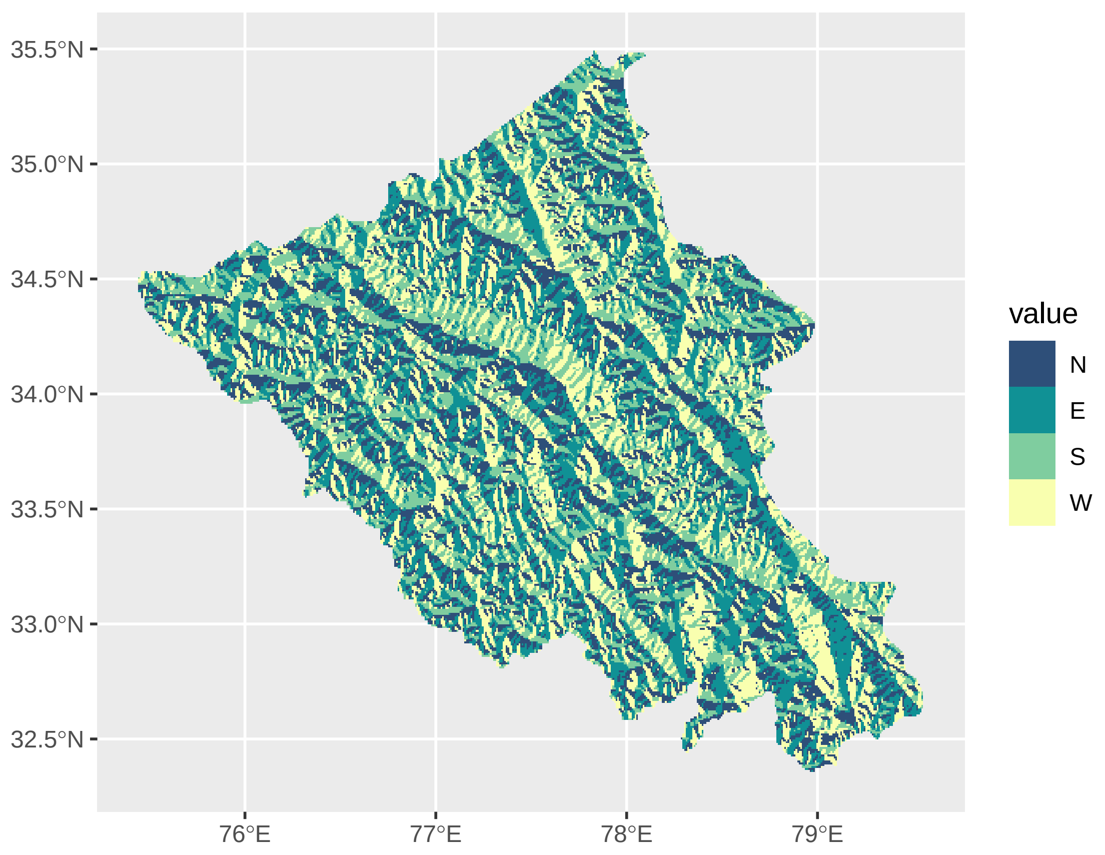
```
rat = data.frame(id = 1:4, aspect = c('N','E','S','W'))
```

We then set the levels in our aspect raster to be the RAT we have just created.

```
levels(aspect) = rat
```

If we plot aspect now you can see that we have a categorical legend with the labels are shown (note that we use scale_fill_discrete this time as we have discrete values and use na.translate rather than na.value to suppress plotting of NA values):

```
ggplot() +
  geom_spatraster(data=aspect) +
  scale_fill_discrete_sequential('BluYl', na.translate = FALSE,rev=F)
```

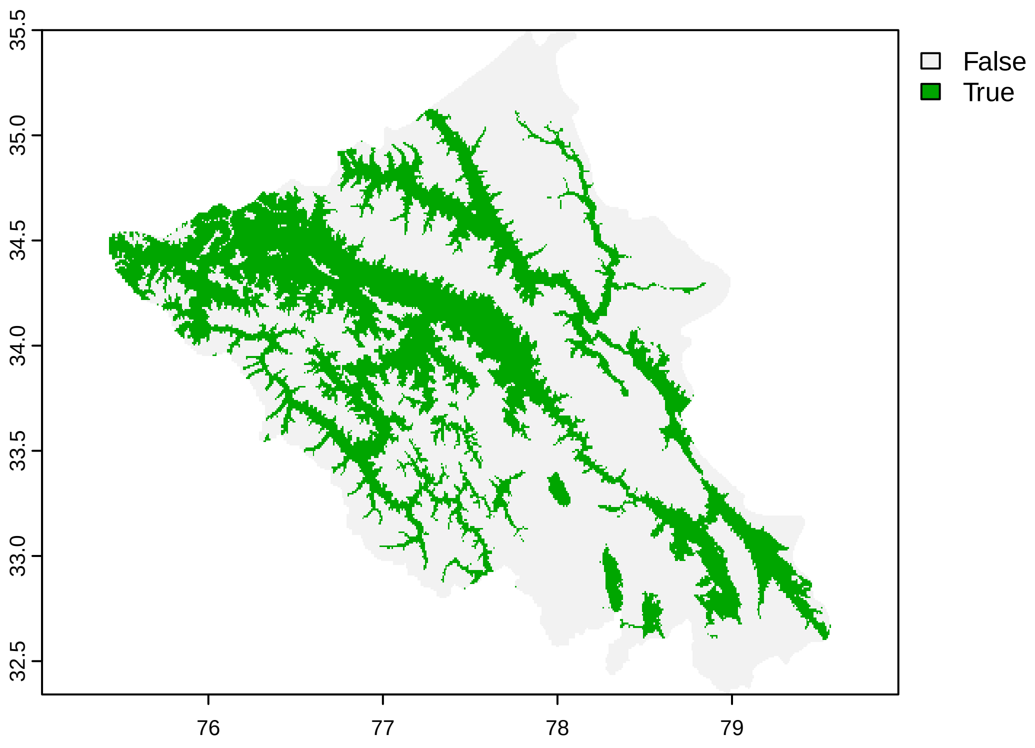


In addition to the classification methods above, we can also do simple binary (TRUE/FALSE) classifications. For example, Watts et al. (2019) reported that habitat that was likely to be "highly suitable" for snow leopards ranged between 2800 and 4600 m elevation. This will make cell values `TRUE` (or 1) for all cells lower than 4600 m and `FALSE` (or 0) for cells greater than or equal to 4600m:

```
plot(dem < 4600)
```

This will test whether each cell is greater than 2800 AND less than 4600 m and return a TRUE/FALSE value:

```
suitable = dem > 2800 & dem < 4600
plot(suitable)
```

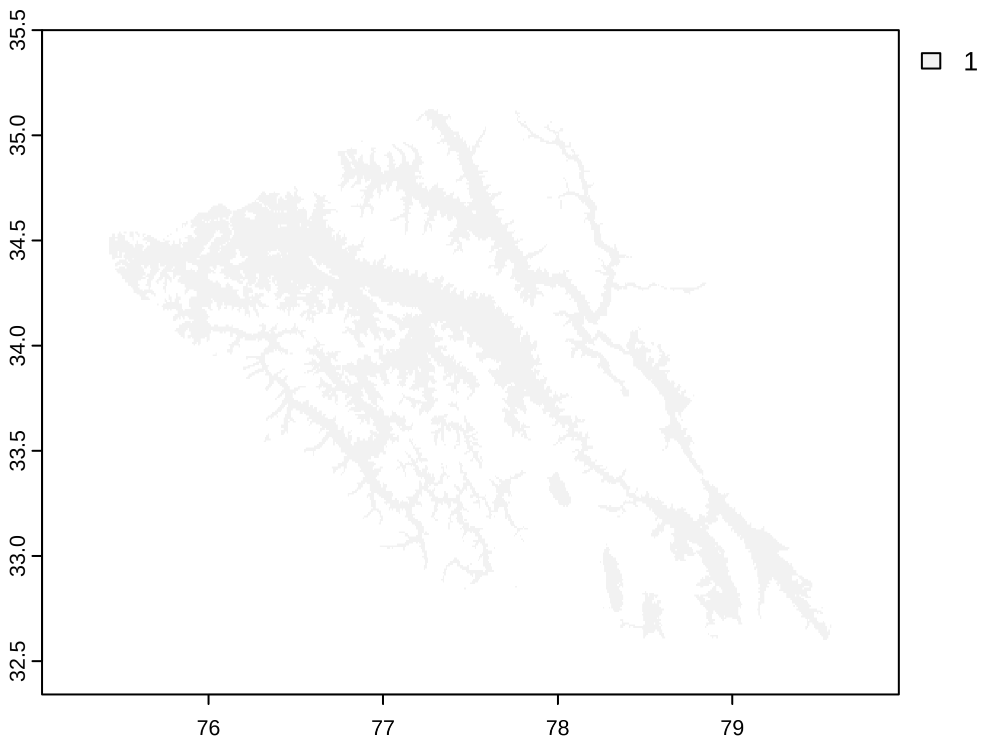


So what if we wanted to subset out just the ‘suitable’ cells and measure their area?

6. Subsetting cells

While there isn’t really a subset operation for single raster layers, it is possible to select just cells with just the values you want by setting everything else to NA using square bracket notation. So for example we could set cells that aren’t in the ‘suitable’ elevation zone to NA by doing:

```
suitable[suitable==0] <- NA
plot(suitable)
```



Watts et al. (2019) also reported that snow leopards tended to avoid north facing slopes (probably pretty chilly in this area!) so we could make a raster of just the north facing slopes like this:

```
northface = aspect
northface[aspect!=1] <- NA
plot(northface)
```

7. Measuring areas

Now let's measure the area of habitat in Ladakh that falls within the 'suitable' elevation range. In `terra` we measure the total area of pixels that are not 'NA' using the function `expanse`:

```
# Set the units to be km2 here
expanse(suitable,unit='km')

# Calculate proportion of total area
expanse(suitable)/expanse(dem)
```

This tells that there is approximately 20000 km^2 of habitat in the 'suitable' elevation zone, as defined by Watts et al. (2019), representing about 33% of the total land area of Ladakh.

8. Exporting rasters

As a final step, we may want to export the raster of 'suitable' habitat we have calculated for future use. We do that using the function `writeRaster`, in this case saving as a Geotiff file.

```
writeRaster(suitable, 'suitable_habitat.tif')
```

9. Saving your workspace

As an alternative to saving individual layers we could also just save our entire R Workspace containing all the objects that we have created, which is often really useful if you plan to pick up again on the analysis later or want to use the layers in another R analysis. We will use some of our snow leopard layers in another tutorial so let's save our workspace using `save.image` (note that the file has a .RData extension)

```
save.image('Tutorial 3 - Snow leopard.RData')
```

Appendix

Code used for downloading the SRTM digital elevation model of Ladakh (downloaded for India and then cropped and masked to Ladakh polygon)

```
elevation = geodata::elevation_30s('IND',path = tempdir())
elevation = crop(elevation,ladakh) %>% mask(ladakh)
writeRaster(elevation,'dem.tif')
```

To get the India basemap

```
geodata::gadm('IND',level=0,path = 'data/india.gpkg')
```