Spatial R course day 2. Working with biodiveristy databases

Soren Faurby

9/21/2022

Accessing biodiveristy data through web services

(This tutorial is a modification of an earlier version written by Tobias Andermann, most of hard work is his, most of the errors are mine)

Dependencies:

```
library(sp)
library(rgbif)
library(ggplot2)
library(mapr)
library(RColorBrewer)
library(raster)
library(rredlist)
library(rworldmap)
library(taxize)
library(sp)
library(sf)
library(rangeBuilder)
```

Some of you may run into issues installing these libraries (particularly the `taxize` package seems to not be working with some R versions). In general using older versions of R sometimes causes issues and it it generally advisable to have the newest (or at least a new version within created within the last few years installed). To avoid issues with older codes in ongoing projects you can have multiple versions of R installed on the same computer.

In this tutorial we will discuss two main online databases, GBIF and IUCN. For the first half we will work with data from GBIF. We will go through some different functions related to downloading biodiveristy data. The point of this tutorial is to provide you the tools to properly download such data in publication quality (DOI-assigned and thus tracable) and to provide you with some handy plotting functions to display the downloaded data.

1a. Define taxonomy

Taxonomy can be a tricky topic. Several different names exist for many taxa, variations being caused by misspellings, different synonyms and regional differences in common names for species. If you want to extract all records for a certain taxon, you first needs to define a coherent taxonomy and in the worst case you need to sync all available datasets to this chosen taxonomy. This can be a rabbit-hole that can make the collection of large datasets from public databases very time consuming. Luckily GBIF is working with one consistent taxonomy, which most records are assigned to. An important caveat is that GBIF

only guarantees that most records will use one of their accepted names but it does not guarantee that records use the same taxonomic concept. This is however not an issue that is solveable programmatically. It can effectively only be solved by manually inspecting (nearly) all records. We will not go further into it here but just note the issue.

In this tutorial we will work with this GBIF backbone taxonomy (NUB). Check out the description of the GBIF taxonomy under this link to understand how this taxonomy is derived and how to cite it.

The following identifier points to the NUB taxonomy, which will make more sense in a little bit.

```
nub = "d7dddbf4-2cf0-4f39-9b2a-bb099caae36c"
```

1b. Pick a species/genus/family

Pick your own species, genus, or family of interest, for which you want to extract occurrence data from GBIF. If you are feeling sufficiently familiar with R, go ahead and extract occurrence data for multiple taxa, which at the end will yield the most interesting results when plotting the data. In this tutorial we will use an example species but it's strongly encouraged for you to go through the exercise with your own picked taxon (does not have to be a species, can be a smaller or larger taxonomic entity).

```
taxon_name = "Canis lupus"
```

There is a useful search function in rgbif called `name_suggest()`. This will return any matches with your provided taxon name and return the name as well as the taxonomic rank of the match:

```
name_suggest(q=taxon_name)
```

```
## Records returned [41]
## No. unique hierarchies [0]
## Args [q=Canis lupus, limit=100, fields1=key, fields2=canonicalName,
##      fields3=rank]
## # A tibble: 41 x 3
##        key canonicalName           rank
##      <int> <chr>                   <chr>
##  1 5219173 Canis lupus             SPECIES
##  2 6164213 Canis lupus hodophilax  SUBSPECIES
##  3 7193871 Canis lupus lupus       SUBSPECIES
##  4 6164180 Canis lupus beothucus   SUBSPECIES
##  5 6164202 Canis lupus lycaon      SUBSPECIES
##  6 6164203 Canis lupus arctos      SUBSPECIES
##  7 6164205 Canis lupus bernardi    SUBSPECIES
##  8 6164216 Canis lupus arabs       SUBSPECIES
##  9 6164194 Canis lupus rufus       SUBSPECIES
## 10 6164208 Canis lupus columbianus SUBSPECIES
## # ... with 31 more rows
```

1c. Check taxonomic information

Now we use the `name_lookup()` function of the rgbif package to check if our picked taxon exists in the chosen GBIF backbone taxonomy and what information is stored with it. In order for the function to find our taxon in the taxonomy, we need to provide the rank that the taxon name represents (is it a subspecies, species, genus, or family name?). We can extract the correct rank classification from the results of the `name_suggest()` function as shown above. In this example we're working with a taxon name that is on the species level. Accepted ranks are: CLASS, CULTIVAR, CULTIVAR_GROUP, DOMAIN, FAMILY, FORM, GENUS, INFORMAL, INFRAGENERIC_NAME, INFRAORDER, INFRASPECIFIC_NAME, INFRASUBSPECIFIC_NAME, KINGDOM, ORDER, PHYLUM, SECTION, SERIES, SPECIES, STRAIN, SUBCLASS, SUBFAMILY, SUBFORM, SUBGENUS, SUBKINGDOM, SUBORDER, SUBPHYLUM, SUBSECTION, SUBSERIES, SUBSPECIES, SUBTRIBE, SUBVARIETY, SUPERCLASS, SUPERFAMILY, SUPERORDER, SUPERPHYLUM, SUPRAGENERIC_NAME, TRIBE, UNRANKED, VARIETY If you are uncertain about how to parse your taxon name into this function, check the helpfunction by executing `?name_lookup()` in R. Note that in the command below we are using the `datasetKey=nub` settings, which is the GBIF standard taxonomy we defined earlier.

```
library(rgbif)
rank = "species"
taxon_taxonomy_data = name_lookup(query=taxon_name, rank=rank, datasetKey=nub
, limit=1)
taxon_taxonomy_data
```

```
## Records found [22]
## Records returned [1]
## No. unique hierarchies [1]
## No. facets [0]
## No. names [1]
## Args [q=Canis lupus, limit=1, offset=0, rank=species,
##       datasetKey=d7dddbf4-2cf0-4f39-9b2a-bb099caae36c]
## # A tibble: 1 x 37
##       key scientificName   datasetKey   constituentKey   nubKey parentKey
parent
##     <int> <chr>            <chr>        <chr>             <int>     <int>
<chr>
## 1 5219173 Canis lupus Lin~ d7dddbf4-2c~ 7ddf754f-d193-4~ 5.22e6   5219142
Canis
## # ... with 30 more variables: basionymKey <int>, basionym <chr>, kingdom <
chr>,
##   phylum <chr>, order <chr>, family <chr>, genus <chr>, species <chr>,
##   kingdomKey <int>, phylumKey <int>, classKey <int>, orderKey <int>,
##   familyKey <int>, genusKey <int>, speciesKey <int>, canonicalName <chr>
,
##   authorship <chr>, publishedIn <chr>, nameType <chr>, taxonomicStatus <
chr>,
##   rank <chr>, origin <chr>, numDescendants <int>, numOccurrences <int>,
```

```
## #   extinct <lgl>, habitats <chr>, nomenclaturalStatus <lgl>,
## #   threatStatuses <chr>, synonym <lgl>, class <chr>
```

You can see there is a lot of useful data stored in this taxonomy. First we can extract the **numerical taxon id** which we will use in following steps to extract occurrence records for this taxon. The advantage of using a numerical id is that the taxon is unmistakably defined and will not anymore be subject to misspellings and different synonyms from here on.

```
taxon_id = taxon_taxonomy_data$data$key
taxon_id

## [1] 5219173
```

The `name_lookup()` function also provides us the taxon ids of the encompassing taxa higher up in the taxonomic hierarchy. In this example it tells us that the wolf belongs to the genus Canis in the family Canidae etc.

```
taxon_taxonomy_data$hierarchies

## $`5219173`
##   rankkey      name
## 1       1  Animalia
## 2      44  Chordata
## 3     359  Mammalia
## 4     732 Carnivora
## 5    9701   Canidae
## 6 5219142     Canis
```

From the output we can extract the ID of the encompassing genus, which we will be using later on in the tutorial, since we will to work with data of several species. If your genus only has a single species, maybe pick a different genus.

```
genus_ID = taxon_taxonomy_data$data$genusKey
genus_ID

## [1] 5219142
```

Alternatively you can also extract the ID of the parent taxon in general, e.g. if you looked up data for a genus you can extract the family ID like this (in the case of my example here it is the ID of the genus since I looked up the taxonomy of a species, which I stored as `taxon_taxonomy_data`):

```
genus_ID = taxon_taxonomy_data$data$parentKey
genus_ID

## [1] 5219142
```

We can also retrieve a list of popular names (vernacular names) in different languages for our taxon. This list might come in handy if we are to combine the GBIF occurrence data with data from other data-sources, which may not have adopted the same taxonomy. In that case we could search for any matches with this list of vernacular names.

```
taxon_taxonomy_data$names[[1]]$vernacularName
```

```
##   [1] "Wolf"              "grey wolf, wolf"   "loup"
##   [4] "wolf"              "gray wolf"         "Lang"
##   [7] "Wolf"              "Haushund"          "domestic dog"
##  [10] "feral dog"         "guri"              "kuri"
##  [13] "kurio"             "pero"              "peto"
##  [16] "uli"               "ulv"               "varg"
##  [19] "varg [inkl. hund]" "varg/hund"         "gray wolf"
##  [22] "Vlk"               "Ulv"               "Wolf"
##  [25] "Wolf"              "Lobo"              "Hunt"
##  [28] "Susi"              "Susi"              "Loup"
##  [31] "Vuk"               "Farkas"            "Úlfur"
##  [34] "Lupo"              "Wollef"            "Vilkas"
##  [37] "Vilks"             "Lupu"              "Wolf"
##  [40] "Ulv"               "Wilk"              "Lobo"
##  [43] "Lup"               "Vlk dravý"         "Volk"
##  [46] "Ujku"              "Varg"              "Kurt"
##  [49] "Wolf"              "Wolf"              "<U+0412><U+043E><U+0432><U+
043A>"
##  [52] "gumpe"             "Ulv"               "gray wolf"
##  [55] "grey wolf"         "Dheeb"             "<U+0627><U+0644><U+0630><U+
0626><U+0628> <U+0627><U+0644><U+0639><U+0631><U+0628><U+064A>"
##  [58] "Vlk"               "Wolf"              "<U+03BB><U+03CD><U+03BA><U+
03BF><U+03C2>"
##  [61] "Arctic Wolf"       "Common Wolf"       "Gray Wolf"
##  [64] "Mexican Wolf"      "Plains Wolf"       "Timber Wolf"
##  [67] "Tundra Wolf"       "Wolf"              "Grey Wolf"
##  [70] "Hunt"              "Susi"              "Loup"
##  [73] "Loup Gris"         "Loup Vulgaire"     "Vuk"
##  [76] "Farkas"            "Lupo"              "Vilks"
##  [79] "Vilkas"            "Saaral Chono"      "Ulv"
##  [82] "Wilk"              "Lup"               "<U+0432><U+043E><U+043B><U+
043A>"
##  [85] "Vlk"               "Volk"              "Lobo"
##  [88] "Ujku"              "<U+0432><U+0443><U+043A>" "Varg"
##  [91] "Kurt"              "<U+0432><U+043E><U+0432><U+043A>" "Gray Wolf"
##  [94] "Lobo gris"         "Wolf"              "loup"
##  [97] "Canis lupus"       "Lobo gris"         "Wolf"
## [100] "gray wolf"         "loup"              "Wolf"
## [103] "Gray Wolf"         "Gray Wolf"         "Wolf"
## [106] "loup"              "Lobo gris"
```

1d . Download occurrence data

Now we have determined our taxon ID and can proceed downloading occurrence data. Downloading occurrence data from GBIF can be done using the `occ_search()` function of the rgbif package. We already touched upon this in yesterday's tutorial, but we'll explore the gbif `occ_search()` function in some more detail today.

```
library(rgbif)
Wolf=occ_search(taxonKey=taxon_id, hasCoordinate = TRUE, limit=1)
Wolf$meta$count

## [1] 91107
```

You can specify how many records you download with "limit" but no matter how many you download, the full number of records is stored as **$meta$Count.** For speed purposes it might be beneficial to just download a single record as we did here before figuring out how many records there exist since downloading a very large number of records is slow. If you want to restrict your search by searching for a specific locality/region/country, there are several flags that can be used to restrict the search geographically. Some examples are:

```
locality = c("Gothenburg", "Göteborg")

continent = "europe"

country = "SE"
```

In my case I will proceed with donwloading all occurrences for Europe (continent = 'europe'). Feel free to select whatever region or city you're interested in. Before continuing I will however show an important point

```
library(rgbif)
Fake_Wolf=occ_search(taxonKey=taxon_id, hasCoordinate = TRUE, continent = "Oc
eania", limit=50000)
Fake_Wolf$meta$count

## [1] 4044
```

Most of you will probably know that the natural range of wolves is confined to the northern hemisphere but we have here downloaded 4044 records of Australian wolves. What we have guaranteed in our coding is only that we get all records classified under the desired name in GBIF but this does not mean that all the records we have downloaded should be classified as this. (in this particular case the issue is mainly that the australian dingo sometimes is classified as Canis lupus or Canis lupus dingo and we have therefore downloaded a number of dingo records along with our intended wolf records). **We will here focus on analysing the data based on what they are stored as in GBIF. A substantial issue on working with GBIF records later is filtering out taxonomic and spatial errors but that is outside the scope of this tutorial**

In this case I will download all European wolf records.

```
library(rgbif)
records=occ_search(taxonKey=taxon_id, hasCoordinate = TRUE, continent = "Euro
pe", limit=50000)
```

1 e. Plot the occurrence data

There is a cool plotting library ggplot2 which allows easy plotting from dataframes like the one produced from the gbif `occ_search()`command. ggplot2 offers tons of fancy plotting options and is a whole science for itself. Most R users (although not me) tend to rely on ggplot rather than the default plotting functions and for new users of R is is likely beneficial to learn ggplot. For now we just load the map object and we'll then use it in the plotting command later on.

ggplot2 also has integrated commands to load map data. Using the borders() function, we can very easily load a world map with the following command. This stores the polygon information in a ggplot specific format, which is different to the SpatialPolygons format of the world polygons we were loading yesterday. You can set the color of the map and of the filling of landmasses in this command.

```
library(ggplot2)
map_world = borders(database = "world", colour = "gray50", fill = "#383838")
```

It is usually a good idea to crop the world map to the area where our points are occurring. So let's first define the bounding box, which we can then use to restrict the plotting to only our selected area:

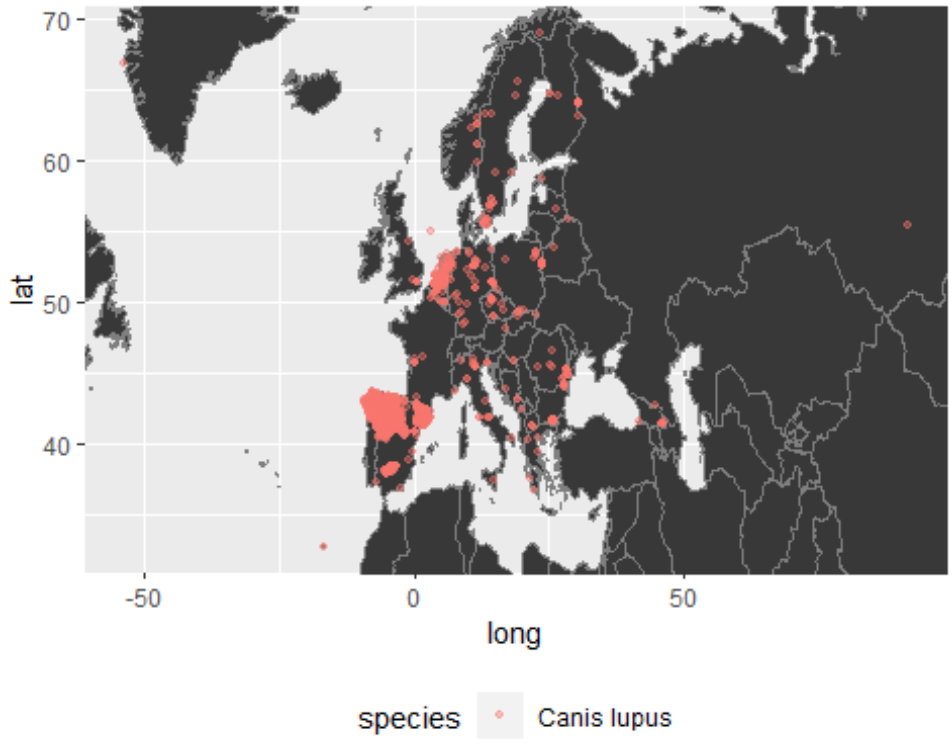
```
xmin = min(records$data$decimalLongitude)
xmax = max(records$data$decimalLongitude)
ymin = min(records$data$decimalLatitude)
ymax = max(records$data$decimalLatitude)
```

Now let's plot our points on the map, colored by species name. The logic of the ggplot syntax is accumulative. That means you can add additonal layers/settings to the command by adding lines connnected with a + sign.

First we just call the function `ggplot()`, then tell it to plot our `map_world` object we created above, then add our gbif dataframe (`records`) using the `geom_point()` function. Within the `geom_point()` command we use `aes()` where we provide the column names in the dataframe that correspond to the x-coordinates (`x=`), y-coordinates (`y=`), and the name of the column we want to color our points by (`colour=`), which in this case is by the species column. Then crop the map around the borders (xmin,xmax,ymin,ymax) defined in the previous step (based on the spread of the data), and finally we add a legend using the `theme()` command.

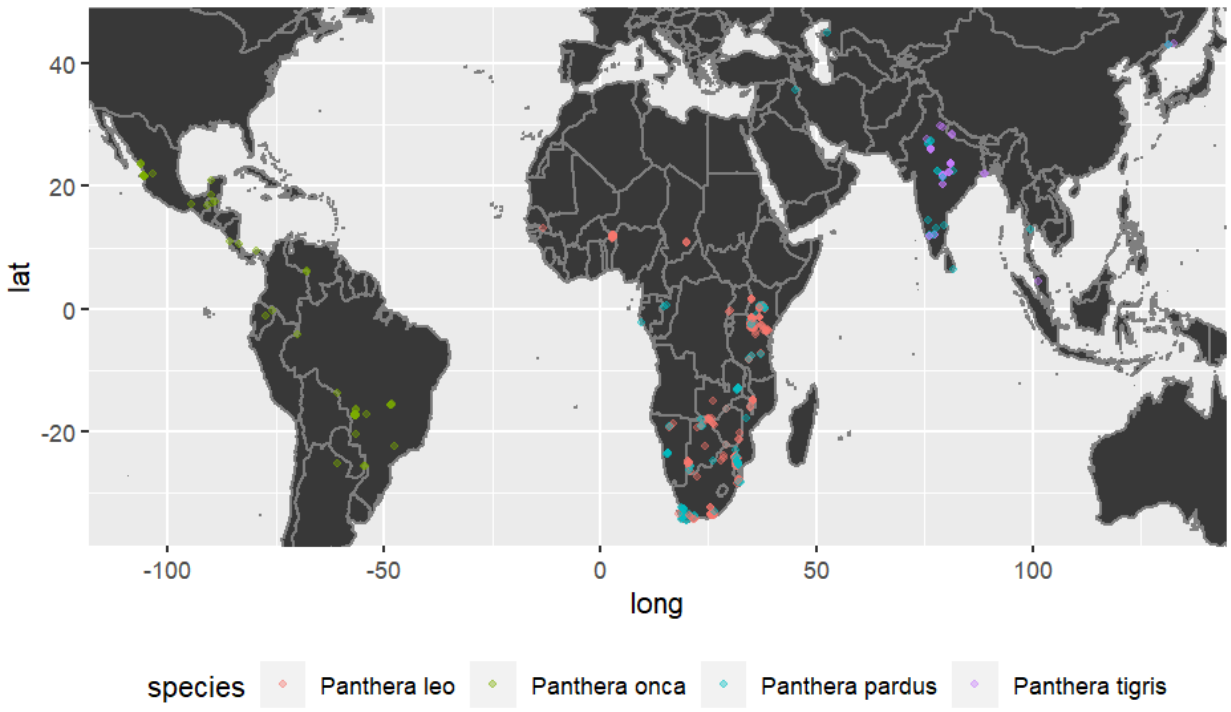
```
library(ggplot2)
ggplot() +
map_world +
geom_point(data = records$data,  # Specify the data for geom_point()
             aes(x = decimalLongitude,  # Specify the x axis as longitude
                 y = decimalLatitude,
                 colour = species),  # Colour the points based on species nam
e
             alpha = 0.4,  # Set point opacity to 40%
             size = 1) +
```

```
coord_cartesian(xlim = c((xmin),(xmax)), ylim = c((ymin),(ymax))) +
theme(legend.position = "bottom")
```



In my plot it is very visible that some areas of the world has few records in GBIF. For instance there are effectively no records from Russia in GBIF.

The plot above looks a bit boring since it only contains the data from my one target species. Recreate the plot for your area and target group of choice. Make sure your dataset contains multiple species, e.g. use the genus_ID we extracted earlier, to download and plot data for a whole genus. Color the points by the species they belong to. I have below created a map of great cats. I chose not continuing with wolves becuase the large number of records caused the waiting time to be very long.



1.f Citing the source data

If you want to be very thorough in your citations, you can export all citations of the data that are present in your downloaded GBIF dataframe (which we stored as `records`). You can easily retrieve this information using the `gbif_citation()` function:

```
library(rgbif)
gbif_citation(records)
#This is very long but I deleted it from the tutorial
```

In some cases this could be a very long list of references. Alternatively/additionally you can create your own official download request at GBIF, which will assign a DOI to your download that can (and should) be cited when publishing. The advantage is that everybody can access the donwload and (hopefully) reproduce your operations on the data. This is the proper way of using GBIF data for publications.

First you need to create a user account at GBIF. This is very simple and fast, just follow this link. Once you created your account and activated it via email, you can execute the lines below in R after replacing the values `GBIF_USER_NAME`, `GBIF_PASSWORD`, and `GBIF_EMAIL` with your account name, passwort, and email address, respectively. You only need to do this once in this session. From here on out the rgbif package will remember your user data, but you'll have to enter them again when you restart R next time.

```
library(rgbif)
options(gbif_user=GBIF_USER_NAME)
```

```
options(gbif_pwd=GBIF_PASSWORD)
options(gbif_email=GBIF_EMAIL)
```

Now let's create a download request for all occurrences associated with your chosen taxon. For this you can use the `occ_download()` command (instead of `occ_search()` which we were using before). I will here download all members of the genus Canis from Africa but you can analyse whichever group you were working on.

```
library(rgbif)
# Get download key
request = occ_download(pred_in("taxonKey", genus_ID), pred_in("continent", "A
frica"))
request

## <<gbif download>>
##   Username: soren.faurby
##   E-mail: soren.faurby@bioenv.gu.se
##   Format: DWCA
##   Download key: 0007992-210819072339941
```

You can get more information about your download request by using the `occ_download_meta()` function or by logging into your gbif user account and checking the download section:

```
library(rgbif)
download_key = occ_download_meta(request[1])
download_key

## <<gbif download metadata>>
##   Status: PREPARING
##   Format: DWCA
##   Download key: 0007992-210819072339941
##   Created: 2021-08-31T18:02:03.029+00:00
##   Modified: 2021-08-31T18:02:03.029+00:00
##   Download link: https://api.gbif.org/v1/occurrence/download/request/00079
92-210819072339941.zip
##   Total records: <NA>
##   Request:
##     type:  and
##     predicates:
##       > type: in, key: TAXON_KEY, value: 5219142
##       > type: in, key: CONTINENT, value: Africa
```

It will likely take 10-20 minutes perhaps longer (depending on the number of records in your query) for this download request to finish compiling. Once the Status: field says `SUCCEEDED` you are ready to retrieve your data. You can do this through R using the `occ_download_get()` function, or you can instead manually download the file by clicking on download on your GBIF webpage. When using the `occ_download_get()` function, provide the path where the zipped folder should be saved (`C C:\\Users\\xfauso\\Documents\\Teaching\\2022\\PhDR\\` in my case, replace with your

path where you want to save, make sure that the folder exists where you are trying to save it).

In my tutorial you will see a # before the occ_download_get line. This is because the waiting time for getting the download done is not easy to implement in the script preparing this tutorial. You should remove the # in your scripts.

In my example below I create a new folder to store the files with dir.create() this is not needed for the codes to run, you might want to instead just save the files in the same folder as everything for this course, it is again not running for my code because I did not download the occurences at the same time as a created this tutorial

At this stage it might be beneficial to start working on part 1g and part 2 of this tutorial and return here afterwards where the data hopefully will be ready.

Some of you might get an error in the occ_download_get() call particular if your version of rgbif is around 1 year old. If you get an error, try updating rgbif.

```
library(rgbif)
key = request[1]
#dir.create("C:\\Users\\xfauso\\Documents\\Teaching\\2022\\PhDR\\output_files\\")
#occ_download_get(key," C:\\Users\\xfauso\\Documents\\Teaching\\2022\\PhDR\\output_files\\")
```

Extract DOI:

We can now use the official GBIF download we requested to cite our data with a unique DOI identifier. The DOI information can be found on the GBIF download webpage, but is also stored in the output of the `occ_download_meta()` function, which we stored as the variable download_key.

The complete citation of these data should be something along these lines:

```
paste0("GBIF Occurrence Download doi:", download_key[2], " accessed via GBIF.org on ", Sys.Date())
```

```
## [1] "GBIF Occurrence Download doi:10.15468/dl.2vzz2q accessed via GBIF.org on 2021-08-31"
```

Load the data:

The main data is stored in a file called `occurrence.txt` in the downloaded zip archive. You can read the data directly form the zip-archive into R, using the unzip function `unz()` together with the `read.table()` function, which reads the data as a dataframe into R. Here we spare our memory by only reading the first 50,000 rows of the data (nrows=50000). Returning to the point about coding from day one, this is an example of calling three separate functions in one line in R.

```
Folder=" C:\\Users\\xfauso\\Documents\\Teaching\\2022\\PhDR\\output_files\"
doi_data = read.table(unz(paste0(Folder, key, ".zip"), "occurrence.txt"),quot
e="",fill = TRUE ,header=T, sep="\t", nrows= 50000 )
```

Plot the data:

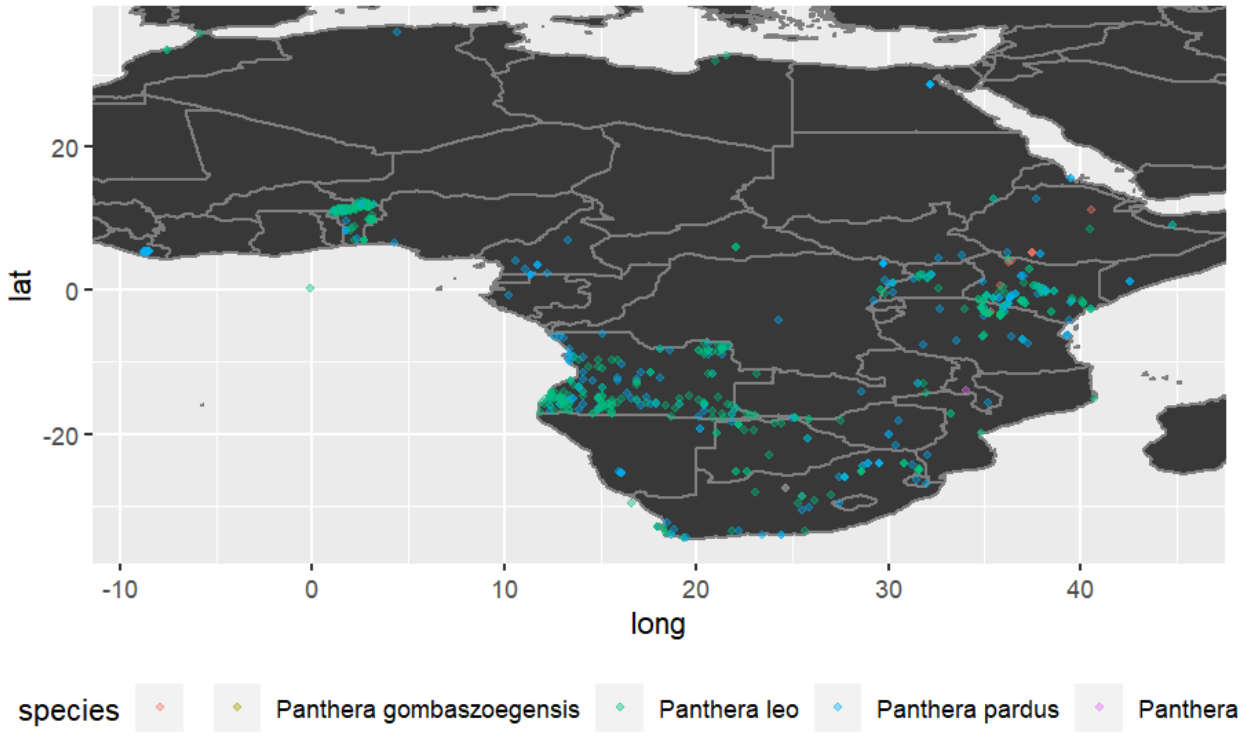
One important note before plotting the data is that the downloaded data could contain some strange coordinates that cannot be properly read and will therefore be coded as `NaN`(meaning Not a Number). These coordinates would cause an error in the plotting function and we therefore need to remove them first. The following two lines take care of that (only selecting lines where the condition `!isna()` is fulfilled, i.e. only those lines that are not `NaN`, as the `!` reverses the statement it is followed by):

```
# remove all rows that have NA data in the coordinates
doi_data=doi_data[!is.na(doi_data$decimalLatitude),]
doi_data=doi_data[!is.na(doi_data$decimalLongitude),]
```

Everything in R can be done multiple ways. An alternative to the statement above is this logical statement.

```
# remove all rows that have NA data in the coordinates
doi_data=doi_data[is.na(doi_data$decimalLatitude)==F,]
doi_data=doi_data[is.na(doi_data$decimalLongitude)==F,]
```

After removing the `NaN` coordinates, plot these data in the same manner as we did above with the data directly downloaded from GBIF through the `occ_search()` function. Make sure you understand the difference between the way we downloaded occurrence data with `occ_search()` (dynamic download through R online portal) vs. the way we did it with `occ_download()` + `occ_download_get()` (API-based DOI-tagged download).



1.g Interactive mapping

As a fun plotting exercise we will plot occurrence data on an interactive OpenStreetView map. The advantage of this is that the user can zoom in and out of the plot and explore large geographic extends in very high detail. Additionally the meta-data (additional values attached to each point, such as species name, etc) of each point can be viewed by clicking on the data point on the map. We can continue working on the dataset we downloaded earlier and called `records`

For plotting interactively we will use the map_leaflet() function as part of the mapr package (The plot might not show, because it can't be displayed by some html viewers. You can view it when downloading this tutorial file in html format and opening it in your html viewer, e.g. Firefox)

```
library(mapr)
map_leaflet(records$data, lon="decimalLongitude", lat="decimalLatitude", size
=5)
```

In the plot you can zoom in and out and you can click on individual points to see which metadata are attached to them.

If you want to play around with different colors, there is a great package called RColorBrewer, which you can use to generate visually pleasing palettes of colors. You can check out the options you can choose from by using the help function

`?colorRampPalette()`. Here we first determine the number of different taxa in our data (in case we have several) and then parse it into the function to create a different color for each species.

```
library(RColorBrewer)
library(mapr)
n_spp = length(unique(records$data$name)) # number of unique taxa in datafram
e (USE spp$name, NOT spp$taxonKey)
myColors = colorRampPalette(brewer.pal(11,"Spectral"))(n_spp) # create color
palette with [n_spp] colors
map_leaflet(records$data, "decimalLongitude", "decimalLatitude", size=5, colo
r=myColors)
```

Subspecies are inconsistently used (especially in GBIF but also elsewhere) and it likely makes sense to color based on species rather than unique name as we did above. To do this we need to change two things in the code above. Firstly we need to change the line specifying the number of colors we created (the one starting with n_spp), secondly we need to specify what map_leaflet should color by. If we do not specify this as we did not above it will color by the column called "name"

```
library(RColorBrewer)
library(mapr)
n_spp = length(unique(records$data$species)) # number of unique taxa in dataf
rame (USE spp$name, NOT spp$taxonKey)
myColors = colorRampPalette(brewer.pal(11,"Spectral"))(n_spp) # create color
palette with [n_spp] colors
map_leaflet(records$data, "decimalLongitude", "decimalLatitude", size=5, colo
r=myColors, name="species")
```

2 Accessing biodiveristy data through web services (IUCN)

In this part of the tutorial we will work with another r-package that can be used to download biodiversity data: The IUCN redlist package `rredlist`.

The IUCN RedList is a collective work of many researchers and conservationists, which compile information about species population trends, habitats, and evaluations of all applicable threats. This information is used to assign a protection status to each species. You can find more information about the IUCN RedList here.

2.a. Get your API token

In order to use this package you first need to apply for an API token. You can do that by filling out this form here.

Since it is hard to predict how long it will take for IUCN to approve your token request (could be minutes or hours), you can get a key from me, just approach me if you're getting to this part of the tutorial and haven't received your token yet.

Then store the API token as a string:

```
iucn_api = YOUR_IUCN_API
```

Extra info: If you're interested you can store this key in your R environment. It's not necessary for this exercise but if you want you can check out the documentation of the functions `rl_use_iucn()` and `edit_r_environ()` by typing `?` followed by the function name.

2.b Pick a taxon and find synonyms

Now you can again pick a species of your choice for which you want to extract species data (such as conservation status, distribution, etc), in this example we're working with the tiger, *Panthera tigris*.

```
taxon_name = "Panthera tigris" # Tiger (remember everything after the hashtag is not interpreted by R so whenever you need to write notes for yourself in your code just start with a hashtag)
```

We can check for synonyms of our species name, using the rl_synonyms() function:

```
library(rredlist)
rl_synonyms(taxon_name,key=iucn_api)

## $name
## [1] "Panthera tigris"
##
## $count
## [1] 1
##
## $result
##   accepted_id   accepted_name        authority      synonym  syn_authority
## 1       15955 Panthera tigris (Linnaeus, 1758) Felis tigris Linnaeus, 1758
```

IUCN standardizes all its data to one main taxonomy, so you will not find any data associated with the synonyms outside of the accepted_name. Therefore make sure you continue working with the name listed in the `accepted_name` column.

Similarly as before in the GBIF tutorial, we can extract popular names (vernacular names) associated with our chosen taxon in the IUCN database:

```
library(rredlist)
rl_common_names(taxon_name,key=iucn_api)

## $name
## [1] "Panthera tigris"
##
## $result
##   taxonname primary language
## 1     Tiger    TRUE      eng
## 2     Tigre   FALSE      fre
```

2.c Habitat information

The IUCN rredlist package offers several functions to extract information about the chosen taxon. You can check out the available functions and explanation in the official package documentation.

For example we can extract information about the habitats the species is found in using the `rl_habitats()` function.

```
library(rredlist)
rl_habitats(taxon_name,key=iucn_api)
```

```
## $name
## [1] "Panthera tigris"
##
## $result
##    code                                                                    ha
bitat
## 1   1.1                                                          Forest - B
oreal
## 2   1.4                                                       Forest - Temp
erate
## 3   1.5                                          Forest - Subtropical/Tropica
l Dry
## 4   1.6                                Forest - Subtropical/Tropical Moist Lo
wland
## 5   1.7 Forest - Subtropical/Tropical Mangrove Vegetation Above High Tide
Level
## 6   1.9                                Forest - Subtropical/Tropical Moist Mo
ntane
## 7   3.5                                       Shrubland - Subtropical/Tropica
l Dry
## 8   3.6                                    Shrubland - Subtropical/Tropical
Moist
## 9   4.5                                       Grassland - Subtropical/Tropica
l Dry
## 10  4.6                    Grassland - Subtropical/Tropical Seasonally Wet/Fl
ooded
##    suitability season majorimportance
## 1     Suitable     NA             Yes
## 2     Suitable     NA             Yes
## 3     Suitable     NA             Yes
## 4     Suitable     NA             Yes
## 5     Suitable     NA             Yes
## 6     Marginal     NA            <NA>
## 7     Suitable     NA             Yes
## 8     Suitable     NA             Yes
## 9     Marginal     NA            <NA>
## 10    Marginal     NA            <NA>
```

2.d. Threat status

One of the most interesting and unique information IUCN has to offer are the RedList assessments. Each species is classified into either one of the following categories:

- Least Concern (LC)
- Near threatened (NT)
- Vulnerable (VU)
- Endangered (EN)
- Critically endangered (CR)
- Extinct in the wild (EW)
- Extinct (EX)
- Data deficient (DD)

Let's see how our species is evaluated. We are using the rl_search() function which will return all sorts of information, but for now we're only interested in the category of the output:

```
library(rredlist)
threat_data = rl_search(taxon_name,key=iucn_api)
threat_data$result$category

## [1] "EN"

threat_data$result$population_trend

## [1] "Decreasing"

threat_data$result$criteria

## [1] "A2abcd "
```

The criteria are a bit more techical to understand but information can be found here In this case tigers are classified as Endangered based on criteria A2. The meaning of this is explained on page 18 in the link. A2 means a *"An observed, estimated, inferred or suspected population size reduction of ≥50% over the last 10 years or three generations"*

We can also view the history of IUCN assessments of our species and see if the trends have improved or worsened. Note that older evaluations may contain different categories, since IUCN has changed its' nomenclature several times in history:

```
library(rredlist)
rl_history(taxon_name,key=iucn_api)

## $name
## [1] "Panthera tigris"
##
## $result
##    year code   category
## 1  2015   EN Endangered
```

```
## 2  2011   EN Endangered
## 3  2010   EN Endangered
## 4  2008   EN Endangered
## 5  2002   EN Endangered
## 6  1996   EN Endangered
## 7  1994    E Endangered
## 8  1990    E Endangered
## 9  1988    E Endangered
## 10 1986    E Endangered
```

We can here see that no changes have happened for tigers (IUCN changed categories betewen 1994 and 1996 which is why the species initially is codes as E and now as EN)

2.e. Extract geographic information

Unfortunately it is not possible to download range maps for your species via the package rredlist (IUCN is a bit particular about making their data available via programming interfaces). However, IUCN has range maps for most species in their database stored on their server. Precise range maps can instead be downloaded manually via the webpage, either for individual species, , or in bulk for whole groups of taxa, e.g. all mammals, via this link, we will return to this at in 2.g.

Even though we can't get the actual range data via the rredlist package, it at least offers a function `rl_occ_country()` which can be used to extract a list of countries where the species exists.

```
library(rredlist)
occurrence_countries = rl_occ_country(taxon_name,key=iucn_api)
occurrence_countries
```

```
## $name
## [1] "Panthera tigris"
##
## $count
## [1] 24
##
## $result
##    code                                country          presence origin
## 1    AF                            Afghanistan Extinct Post-1500 Native
## 2    BD                             Bangladesh            Extant Native
## 3    BT                                 Bhutan            Extant Native
## 4    CN                                  China            Extant Native
## 5    ID                              Indonesia            Extant Native
## 6    IN                                  India            Extant Native
## 7    IR              Iran, Islamic Republic of Extinct Post-1500 Native
## 8    KG                            Kyrgyzstan Extinct Post-1500 Native
## 9    KH                              Cambodia  Possibly Extinct Native
## 10   KP Korea, Democratic People's Republic of  Possibly Extinct Native
## 11   KZ                            Kazakhstan Extinct Post-1500 Native
## 12   LA       Lao People's Democratic Republic            Extant Native
```

```
## 13   MM                                  Myanmar            Extant Native
## 14   MY                                 Malaysia            Extant Native
## 15   NP                                    Nepal            Extant Native
## 16   PK                                 Pakistan Extinct Post-1500 Native
## 17   RU                     Russian Federation            Extant Native
## 18   SG                                Singapore Extinct Post-1500 Native
## 19   TH                                 Thailand            Extant Native
## 20   TJ                               Tajikistan Extinct Post-1500 Native
## 21   TM                             Turkmenistan Extinct Post-1500 Native
## 22   TR                                   Turkey Extinct Post-1500 Native
## 23   UZ                                Uzbekistan Extinct Post-1500 Native
## 24   VN                                  Viet Nam  Possibly Extinct Native
##     distribution_code
## 1  Regionally Extinct
## 2              Native
## 3              Native
## 4              Native
## 5              Native
## 6              Native
## 7  Regionally Extinct
## 8  Regionally Extinct
## 9    Possibly Extinct
## 10   Possibly Extinct
## 11 Regionally Extinct
## 12             Native
## 13             Native
## 14             Native
## 15             Native
## 16 Regionally Extinct
## 17             Native
## 18 Regionally Extinct
## 19             Native
## 20 Regionally Extinct
## 21 Regionally Extinct
## 22 Regionally Extinct
## 23 Regionally Extinct
## 24   Possibly Extinct
```

As you can see, the output also contains the information if the species is extant or extinct (post 1500 AD) in the country where it was once found. In the next step we use this information to plot a world map with countries highlighted where our species has been found, with different coloring depending on if the species is extinct or extant in the respective country.

2.f. Plotting geographic information

First let's turn the `presence` column, which contains the info if the taxon is extinct/extant in each country, into an array of 0 (extinct) and 1 (extant), in order to use this information for plotting. Here we use the notation "!=" which means not equal to:

```
extant_extinct = occurrence_countries$result$presence
extant_extinct[extant_extinct != "Extant"] <- 0
extant_extinct[extant_extinct == "Extant"] <- 1
extant_extinct
```

```
##  [1] "0" "1" "1" "1" "1" "1" "0" "0" "0" "0" "0" "1" "1" "1" "1" "0" "1" "
0" "1"
## [20] "0" "0" "0" "0" "0"
```

Now let's extract the list of countries where our species has been found:

```
theCountries = c(occurrence_countries$result$code) # ISO2 country codes
theCountries
```

```
##  [1] "AF" "BD" "BT" "CN" "ID" "IN" "IR" "KG" "KH" "KP" "KZ" "LA" "MM" "MY"
"NP"
## [16] "PK" "RU" "SG" "TH" "TJ" "TM" "TR" "UZ" "VN"
```

We can now merge the list with our extinct/extant info with the list of the country names into one dataframe:

```
extDF = data.frame(country = c(occurrence_countries$result$code), extant = c(
extant_extinct))
extDF
```

```
##    country extant
## 1       AF      0
## 2       BD      1
## 3       BT      1
## 4       CN      1
## 5       ID      1
## 6       IN      1
## 7       IR      0
## 8       KG      0
## 9       KH      0
## 10      KP      0
## 11      KZ      0
## 12      LA      1
## 13      MM      1
## 14      MY      1
## 15      NP      1
## 16      PK      0
## 17      RU      1
## 18      SG      0
## 19      TH      1
## 20      TJ      0
## 21      TM      0
## 22      TR      0
## 23      UZ      0
## 24      VN      0
```

Now we will use the `rworldmap` package, which provides a function that allows us to find the countries on the world map based on their ISO2 codes (the 2-letter abbrevations in our country list):

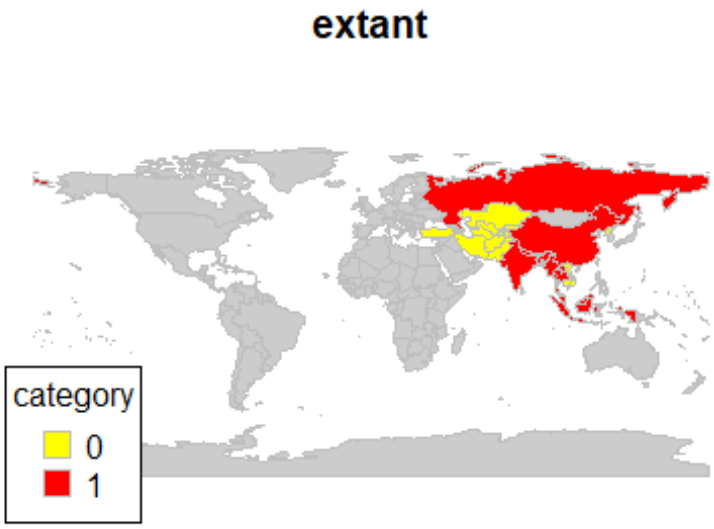
```
library(rworldmap)
extMap = joinCountryData2Map(extDF, joinCode = "ISO2", nameJoinColumn = "coun
try")

## 24 codes from your data successfully matched countries in the map
## 0 codes from your data failed to match with a country code in the map
## 219 codes from the map weren't represented in your data
```

This will join your data from extDF the dataframe with the country map polygons to produce a `SpatialPolygonsDataFrame` object.

Finally we can plot the map with the countries of presence of our species highlighted in red and the countries where the species went extinct in yellow. We'll use the `mapCountryData()` function from the rworldmap package to plot all countries and color them by the values in the target column of our dataframe, specficied with the `nameColumnToPlot=` setting.

```
mapCountryData(extMap, nameColumnToPlot="extant", catMethod = "categorical",
missingCountryCol = gray(.8))
```



2.g. Partial critique of IUCN ranges

As a quick side-note, it is vital to know what is and is not plotted by IUCN. In order to see what is meant by this let us compare the range of tigers from IUCN with the present-natural range from Phylacine which we used yesterday. For this we first need to downlaod the range polygon of tigers from IUCN. You can do this from IUCN. You can search for your species of interest and then click on the blue Download button to the right. Select 'Range data'. You will be prompted to log in. You can just choose your google account to log in or make a new account at IUCN. Then you'll have to write what you want to use the data for (just write 'Needed for a course' or something short like that). You'll also have to agree to their terms and conditions and then a window will pop up that will suggest you to go to the accounts page. From there you can select your requested data under 'Saved downloads'. Click on 'download' (it may take a couple of minutes before your data is available and the button will appear) and store the data in the downloaded github repo data folder.

After this you have the needed data and just need to remember what you learned yesterday. 1) Load the present-natural range of the tiger. 2) Load the shape file into R 3) Change the projection of the shape file

```
## Reading layer `data_0' from data source `C:\\Users\\xfauso\\Documents\\Te
aching\\2022\\PhDR\\Other_data\\Tiger\\data_0.shp' using driver `ESRI Shapefi
le'
## Simple feature collection with 42 features and 15 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 73.20145 ymin: -5.941536 xmax: 140.6644 ymax: 51.2798
## Geodetic CRS:  WGS 84
```

Once this is done we can plot the pattern. For this it is nice to understand the metadata associated with IUCN range maps. For the tiger, we want to distinguish the parts of the ranges based on "Presence". For other species we might e.g. also be interested in "Origin" if we want to remove introduced parts of species ranges.

Below we will plot the Present-natural range in black. On top of this we will plot parts where the species is still found in red and parts where there species presence is uncertain (Possibly Extinct) in blue.

Importantly we here see that the parts coded as possibly extinct or extinct by IUCN is only a small part of where the species would naturally occur. Part of this is because some of the range extractions might have been before 1500 (and tigers for instance went extinct from [Borneo] (https://en.wikipedia.org/wiki/Bornean_tiger) thousands of years ago). A large part of the range decline and it should have been included for IUCN to be completely consistent (and tigers e.g. went extinct from [Java] (https://en.wikipedia.org/wiki/Javan_tiger) post WW2). This therefore highlight that while IUCN generally is a great source for current distributions (which is the only thing IUCN really cares about) their data are generally lacking for historical ranges.

3 Resolving taxon names (taxize)

Often in biology we are confronted with taxon names or lists of taxon names, and we need to somehow retrieve data for these taxa from public databases (as we did in the previous tutorial steps). The challenge here is usually to find the correct species name, since taxonomies can vary between databases and between different authors. Luckily there are great tools to standardize and resolve taxonomic issues in many cases, implemented in the `taxize` package.

The `taxize` package provides access to taxonomic information sourced from many databases, among them the Global Names Resolver (GNR) service provided by the Encyclopedia of Life. The advantage of `taxize` is that it is primarily designed for resolving taxonomic issues and thus has many useful functions for this purpose, compared to some similar but more simplified functionalities of the `rgbif` and `rredlist` packages we have seen in the previous tutorial steps.

Let's first load the package.

```
library(taxize)
```

3. a. Finding correct taxon name for database

Let's say we have a list of species names and we want to know if our species are spelled correctly.

Here, we are searching for two **misspelled species names**:

```
library(taxize)
RESOLVE = gnr_resolve(sci = c("Pantera tigrsi", "Homo saapiens"))
RESOLVE
```

```
## # A tibble: 64 x 5
##    user_supplied_na~ submitted_name matched_name    data_source_title
score
##  * <chr>             <chr>          <chr>           <chr>
<dbl>
##  1 Pantera tigrsi    Pantera tigrsi Panthera tigri~ Catalogue of Life - J~
0.75
##  2 Pantera tigrsi    Pantera tigrsi Panthera tigri~ Wikispecies
0.75
##  3 Pantera tigrsi    Pantera tigrsi Panthera tigri~ Integrated Taxonomic ~
0.75
##  4 Pantera tigrsi    Pantera tigrsi Panthera tigris National Center for B~
0.75
##  5 Pantera tigrsi    Pantera tigrsi Panthera tigri~ Union 4
0.75
##  6 Pantera tigrsi    Pantera tigrsi Panthera tigri~ The Interim Register ~
0.75
##  7 Pantera tigrsi    Pantera tigrsi Panthera tigris Freebase
0.75
##  8 Pantera tigrsi    Pantera tigrsi Panthera tigri~ GBIF Backbone Taxonomy
0.75
##  9 Pantera tigrsi    Pantera tigrsi Panthera tigris Encyclopedia of Life
0.75
## 10 Pantera tigrsi    Pantera tigrsi Panthera tigri~ TaxonConcept
0.75
## # ... with 54 more rows
```

In the output you can find the database you want to download data from and use the according correct species name (can vary between databases).

We might want to only have the names matches to IUCN which we can get by subsetting the data

```
RESOLVE[RESOLVE$data_source_title=="IUCN Red List of Threatened Species",]
```

```
## # A tibble: 2 x 5
##   user_supplied_na~ submitted_name matched_name      data_source_title
```

```
score
##   <chr>             <chr>          <chr>             <chr>
<dbl>
## 1 Pantera tigrsi    Pantera tigrsi Panthera tigris ~ IUCN Red List of Thr~
0.75
## 2 Homo saapiens     Homo saapiens  Homo sapiens Lin~ IUCN Red List of Thr~
0.75
```

Of course the issue might not be misspellings but rather disagreements on the proper species name, such disagreements have multiple causes, included but not limited to: a) disagreements on what genus a species belongs to, b) disagreements on whether a taxon is a subspecies or a full species and c) taxonomic nitpicking. (For taxonomic nitpicking some of the things frequently causing disagreements are whether species names should change from masculine to feminine forms when changing genus and how many "i"'s a species name should end with.

There is no consistent way to deal with inconsistent taxonomy but a few packages have taxon specific solutions. In particular the package *rangeBuilder* has an implementation for terrestrial vertebrates (with the caveat that it will match IUCN taxonomy way better for birds, reptiles and amphibians than for mammals). Let us quickly see how it works for a single bird the crowned comorant which recently changed its genus placement.

```
library(rangeBuilder)
synonymMatch("Phalacrocorax coronatus", db="birds")

##
##  strict match | synonym      1

## [1] "Microcarbo_coronatus"
```

3. b. Getting species list for higher taxa

Let's say we have a taxonomic family name and want to find all species belonging to this family, for example all dogs of the family Canidae.

A number of data sources in taxize provide the capability to retrieve higher taxonomic names,for example the BOLD taxonomy (`db = "bold"`). We can search the taxonomy for taxa belonging to our specified group using the `downstream()` function.

```
library(taxize)
species_output = downstream("Canidae", downto = "Species", db = "bold") #This
might not work on older R-versions. It works on R 4.0.2 and 4.1.2 but not R 3
.5.3

## ==  1 queries  ===============
## v  Found:  Canidae
## ==  Results  =================
##
## * Total: 1
## * Found: 1
## * Not Found: 0
```

```
species_output
```

```
## $Canidae
##                          name     id    rank
## 1       Atelocynus microtis 174664 species
## 2             Canis adustus 174687 species
## 3              Canis anthus 751533 species
## 4              Canis aureus 174686 species
## 5          Canis familiaris  12515 species
## 6             Canis latrans  12513 species
## 7               Canis lupus  12514 species
## 8              Canis lycaon 670110 species
## 9           Canis mesomelas 174683 species
## 10           Canis simensis 174414 species
## 11          Cerdocyon thous  73532 species
## 12    Chrysocyon brachyurus 174685 species
## 13             Cuon alpinus  73506 species
## 14       Lycalopex culpaeus 174672 species
## 15       Lycalopex fulvipes 359592 species
## 16        Lycalopex griseus 174671 species
## 17    Lycalopex gymnocercus 174670 species
## 18       Lycalopex sechurae 174669 species
## 19            Lycalopex sp. 465629 species
## 20        Lycalopex vetulus 174677 species
## 21            Lycaon pictus 174679 species
## 22 Nyctereutes procyonoides 174676 species
## 23        Otocyon megalotis 174674 species
## 24       Speothos venaticus 174667 species
## 25 Urocyon cinereoargenteus  17525 species
## 26       Urocyon littoralis 751860 species
## 27       Vulpes bengalensis 415357 species
## 28             Vulpes chama 796306 species
## 29            Vulpes corsac 170835 species
## 30         Vulpes ferrilata 747543 species
## 31           Vulpes lagopus  64742 species
## 32          Vulpes macrotis 174665 species
## 33             Vulpes velox  16433 species
## 34            Vulpes vulpes  16440 species
## 35             Vulpes zerda 174680 species
##
## attr(,"class")
## [1] "downstream"
## attr(,"db")
## [1] "bold"
```

You can now extract the list of all species belonging to your chosen family:

```
species_list = species_output$Canidae$name
species_list
```

```
##  [1] "Atelocynus microtis"      "Canis adustus"
##  [3] "Canis anthus"             "Canis aureus"
##  [5] "Canis familiaris"         "Canis latrans"
##  [7] "Canis lupus"              "Canis lycaon"
##  [9] "Canis mesomelas"          "Canis simensis"
## [11] "Cerdocyon thous"          "Chrysocyon brachyurus"
## [13] "Cuon alpinus"             "Lycalopex culpaeus"
## [15] "Lycalopex fulvipes"       "Lycalopex griseus"
## [17] "Lycalopex gymnocercus"    "Lycalopex sechurae"
## [19] "Lycalopex sp."            "Lycalopex vetulus"
## [21] "Lycaon pictus"            "Nyctereutes procyonoides"
## [23] "Otocyon megalotis"        "Speothos venaticus"
## [25] "Urocyon cinereoargenteus" "Urocyon littoralis"
## [27] "Vulpes bengalensis"       "Vulpes chama"
## [29] "Vulpes corsac"            "Vulpes ferrilata"
## [31] "Vulpes lagopus"           "Vulpes macrotis"
## [33] "Vulpes velox"             "Vulpes vulpes"
## [35] "Vulpes zerda"
```

Assignment 1

Now where you have a basic overview over the use of the `rgbif`, `rredlist`, and `taxize` packages, you are ready to approach a bioinformatic task based on biodiversity data:

Your supervisor asks you for help with a project about the cat family `Felidae`. Your task is to **create a map of global occurrences** of this family, **colored by species**. Further your supervisor asks you to retrieve a **list of IUCN threat statuses for all species of this family** (if you like a challenge, you can also create a second plot, colored by threat status instead of colored by species).

Since these data are supposed to be used in a publication, your supervisor expects you as a properly trained biodiversity data wizard to provide a DOI assigned dataset of `Felidae` occurrence records to be cited in the study.

Tips for assignment:

There are different ways of solving this task, one approach could be:

- create a download request containing all records assigned to the taxon `Felidae`, using the `occ_download()` function
- download the data and load into R using the info from section 1 of the tutorial
- export the DOI reference of the data download section 1 of the tutorial
- plot occurrences colored by name section 1 of the tutorial
- retrieve a species list of the family `Felidae`, using the `taxize` package
- get the IUCN RedList status for each species, using the `rl_search()` function

Assignment 2

For the very fast and motivated people among you:

You are being asked to plot the actual range maps of all Felidae species (according to IUCN) and on top plot all point occurrences (according to GBIF). For this purpose you need to download the actual range data, which can be downloaded for each species individually, by filtering though the taxonomy in advanced search (in this case by clicking though Animalia -> Chordata -> Mammalia -> Carnivora before finally ticking Felidae) or by selecting or as a big data-package, containing ranges of all mammal species (download here), and then you can extract the ranges of the species of interest. You can decide if you want to plot this all in one plot, or in separate plots, one for each species (doesn't have to be all Felidae species, but maybe a few selected ones).