A tutorial on how to use **plantR**

Renato A. F. de Lima[*], Sara R. Mortara[†]and Andrea Sánchez-Tapia[‡]

05 October 2021

Contents

[*] Universidade de São Paulo and Naturalis Biodiversity Center, https://github.com/LimaRAF

[†] Jardim Botânico do Rio de Janeiro, https://github.com/saramortara

[‡] Jardim Botânico do Rio de Janeiro, https://github.com/AndreaSanchezTapia

1 Introduction

This tutorial provides the details and explanations on the functions related to data download, standardization and validation using the **plantR** workflow. For each of the steps of the workflow, we present in more detail the functionalities and their options. The aim is that users can better understand their goals and so adapt functions according to their needs or for their use outside the workflow proposed here.

2 Installing plantR

plantR is currently only available remotely on GitHub. This means that you cannot use R base functions to install it (i.e., `install.packages()`). There are different ways to install packages in R from remote repositories and here we suggest using the package **remotes**. So, first of all, we need to be sure that this package is installed and loaded.

```
install.packages(remotes)
library("remotes")
```

Then we can use **remotes** function `install_github()` to install **plantR** from GitHub.

Like many other R packages, **plantR** depends on other packages to perform some tasks (i.e., dependencies), and these packages often depend on other packages (i.e.m recursive dependencies). Therefore, to install **plantR** you also need to install these dependencies. R should automatically install the packages you don't have (this may be a long list depending on which packages you have already installed). It also will prompt at your screen a list of packages that have more recent versions available for update. It is good to have the most up-to-date versions of all packages, but this is not a necessity to install **plantR** (i.e., you can select to update none of the packages).

```
remotes::install_github("LimaRAF/plantR")
library("plantR")
```

For windows users, you may get a warning while trying to install **plantR**, related to the installation of Rtools. This software needs to be installed in your machine, so that R can build and install packages from their source code on Windows. It became necessary after R version 4.0. So make sure we have Rtools installed, restart your R session and try to install **plantR** again. Check this link on how to install Rtools on Windows.

Some users may still get the error below:

```
Error: package '[name of the package]' was installed before R 4.0.0: please re-install it
```

If this is your case, we suggest installing/updating each of these packages individually, until you finally manage to run **plantR** installation without problems. You may need to restart your R session again to make sure that the all dependencies were installed correctly.

3 Data entry

You can download species records directly from R, using the functions provided by **plantR** or by other R packages. For this tutorial, we are going to use the **plantR** function `rspeciesLink()` and `rgbif2()`, which downloads data from CRIA (Centro de Referência em Informação Ambiental) and for the GBIF (Global Biodiversity Information Facility).

In addition, you can download a DarwinCore Archive (DwC-A) file from the GBIF API or to load a DwC-A from a local directory using the function `readData()`. DwC-A files obtained from GBIF are provided as '.zip' files and contain not only the species records, but also the information on the citations and rights of the database collections.

You can also use your own data, as long as the column names follow the DarwinCore (DwC) standards. If they don't follow the DwC standards, users can still standardize the notation of column names using the function `formatDwc()`. Besides the field name standardization, `formatDwc()` also binds together the records coming from different sources (e.g., CRIA, GBIF, user-provided data).

3.1 Downloading species records with plantR

In this tutorial, we will use species records for three Neotropical tree species:

```
spp <- c("Casearia sylvestris",
         "Euterpe edulis",
         "Trema micrantha")
```

After defining the species names, we proceed to the download of species records. We will first download only herbarium specimens (i.e., vouchers) for the names listed above, including their synonyms accepted in the List of Species of the Brazilian Flora 2020 project. This may take a minute depending on your internet connection:

```
occs_splink <- rspeciesLink(species =  spp,
                            Scope = "plants",
                            basisOfRecord = "PreservedSpecimen",
                            Synonyms = "flora2020")
## Making request to speciesLink...
```

The function `rspeciesLink()` allows a wide variety of ways on how to search for species records. Searches can also be done by locality and collections, using all the filters available in CRIA. Please check the help of the function for more details on the arguments that can be used for this search.

We will download records for the same species from GBIF, using the function `rgbif2()`. This function uses internally the function `occ_search()` from the package **rgbif**, which should be consulted for details on how searches can be performed.

```
occs_gbif <- rgbif2(species =  spp,
                    basisOfRecord = "PRESERVED_SPECIMEN",
                    remove_na = FALSE, n.records = 500000)
## Making request to GBIF for Casearia sylvestris...
## Making request to GBIF for Euterpe edulis...
## Making request to GBIF for Trema micrantha...
```

The occurrence data is now stored in the objects `occs_splink` and `occs_gbif`. We can inspect how many occurrences were downloaded:

```
nrow(occs_splink)
## [1] 18285
nrow(occs_gbif)
## [1] 35849
```

3.2 Preparing the input data: function `formatDwc()`

Before we start to process the species records, we need to combine our input data from CRIA and GBIF and make sure that the input data has all the required fields and that they are in the required format.

We use the function `formatDwc()` to bind data and set up the required and optional fields and to drop fields without essential information for the data processing and validation. We save the output of the function in an object with a different name (i.e., `occs`).

```
occs <- formatDwc(splink_data = occs_splink,
                  gbif_data = occs_gbif, drop = TRUE)
## Dropping fields not essential or recommended to the data cleaning!
```

```
dim(occs)
## [1] 54134    47
```

4 Data standardization

4.1 Standardizing collection codes, people names, collector numbers and dates

Particularly when working with data coming from multiple sources, it is important to standardize the notation used by different collections and sometimes by different collectors within the same collection.

4.1.1 Collection codes

The first step is the standardization of the notation of collection codes, which is stored in the field 'collectionCode'. For some reason, not all collections store in this field their international collection codes (e.g., Index Herbariorum) and sometimes these codes are stored differently across data repositories. For the search of duplicate entries and the production of lists of species (see below) it is important to standardize the codes of collections. In **plantR** this is done using function `getCode`:

```
occs <- getCode(occs)
```

This function returns a new column ('collectionCode.new') which contains the standardized collection codes found in a **plantR** list of herbaria, xylaria and other plant collections. It also returns the column 'collectionObs', which is empty if the collection code was found. But it also provides details if the code is in the (e.g., Index Herbariorum) and it flags the collection codes not found, which probably represent collections other than herbaria, xylaria or carpotheks.

4.1.2 Names of collectors and determiners

There is a lot of variation in the notation of people's names both within and across biological collections. **plantR** offers different tools to edited and standardize those names. Several functions perform different operations, such as standardizing the separation between multiple people's names, the generational suffixes and the removal of Latin characters (function `fixName()`). It also standardizes the format of name notation (i.e., Last Name, First Name(s) or First Name(s), Last Name) and the abbreviation of name initials (functions `prepTDWG()`, `getPrep()` and `getInit()`). These and other functionalities are included in the function `prepName()`, which also performs the removal of treatment prepositions (e.g., Dr., Profa.) and the separation between the first collector/determiner from other collectors/determiners (co-authors or auxiliary names), if present.

We will first standardize the names of collectors, which are stored in the field ‘recordedBy’:

```r
occs$recordedBy.new <- prepName(occs$recordedBy,
                                output = "first",
                                sep.out = "; ")
occs$recordedBy.aux <- prepName(occs$recordedBy,
                                output = "aux",
                                sep.out = "; ")
```

Note that we separated the first and auxiliary names for multiple people’s names. To do so, we set the argument `output` of function `prepName()` to “first” and “aux” (by default `prepName()` edits and returns all people’s names).

We also define that the symbol that will separate multiple people’s names, which in our example will be a semi-colon followed by space (argument `sep.out`).

By default, the function removes accents and special characters from names, but this can be controlled using the argument `special.char`. Also, the default name format is ‘last_init’ (i.e., Last Name, First Names) which is controlled by the argument `format`.

Next, we will do the same operation for the names of determiners, which are stored in the field ‘identifiedBy’:

```r
occs$identifiedBy.new <- prepName(occs$identifiedBy,
                                output = "first",
                                sep.out = "; ")
occs$identifiedBy.aux <- prepName(occs$identifiedBy,
                                output = "aux",
                                sep.out = "; ")
```

The function `prepName()` standardizes the most common types of name notation, but not all of them. We recommend the user to look at the help of functions `fixName()`, `prepTDWG()` and `prepName()` for examples of formats that these functions cannot handle. However, this standardization will be useful for the search of duplicated records across collections (see below).

We also recommend the user to look at all the options of output, which may include accents (e.g., ‘Araújo, J.’ instead of ‘Araujo, J.’), name prepositions (e.g., ‘de Araujo, J.’ or ’ Araujo, J. de’), abbreviations (e.g., ‘Araujo, Jose’) or different name formats (e.g., ‘J. Araujo’).

It is also useful for the validation process to standardize the notation for missing collector and determiner names. To do so, we use the function `missName()` and we set missing names to “s.n.”.

```r
occs$recordedBy.new <- missName(occs$recordedBy.new,
                                 type = "collector", noName = "s.n.")
occs$identifiedBy.new <- missName(occs$identifiedBy.new,
                                   type = "identificator", noName = "s.n.")
```

For some of the validation steps we will perform, it is also useful to extract the last name of the first collector from the formatted field `recordedBy.new`.

```r
occs$last.name <- lastName(occs$recordedBy.new)
```

4.1.3 Collector number

Species records are often associated with a number from a series of collections of a given collector, which is stored in the field ‘recordNumber’. One would expect that this field would only contain numbers but this is not always the case. In addition, collectors may add modifiers to their collection numbers, such as ‘17-A’ and ‘17-B’.

Therefore, the notation of this field also needs standardization, which in **plantR** is performed using the function `colNumber()`:

```
occs$recordNumber.new <- colNumber(occs$recordNumber, noNumb = "s.n.")
```

4.1.4 Collection and identification year

Similar to the collector number, the collection and identification years can have other information than the year itself (i.e. months). To make sure that these fields contain only years, we use the function `getYear()`. We set the character that will represent records without the year of collection and identification ("n.d.") using the argument `noYear`.

```
occs$year.new <- getYear(occs$year, noYear = "n.d.")
occs$yearIdentified.new <- getYear(occs$dateIdentified, noYear = "n.d.")
```

The year of collection is sometimes stored on the field 'eventDate' and not on the field 'year'. In this case, one should replace all missing information from the field 'year' with the information stored in the field 'eventDate'.

4.1.5 The wrapper function `formatOcc()`

All the previous steps are important to understand the editing process of each field. **plantR** can execute all these steps at once using the wrapper function `formatOcc()`:

```
occs1 <- formatDwc(splink_data = occs_splink,
                   gbif_data = occs_gbif, drop = TRUE)
## Dropping fields not essential or recommended to the data cleaning!
occs1 <- formatOcc(occs1)
```

This wrapper will add the new columns containing the edited/formatted information regarding the collection codes, the names of collectors and determiners, the dates of collection and identification, and collector number.

4.2 Editing locality information

One type of information associated with species records is the collection locality, namely "country", "stateProvince", "municipality" and "locality". **plantR** provide tools to edit and validate this type of information, which also is an important step to find missing geographical coordinates and to validate the original coordinates provided with the records.

4.2.1 Formatting locality information

plantR has a function to edit and standardize locality fields: function `fixLoc()`. By default, this function formats all four locality fields simultaneously and returns the input data frame with the new locality columns. But the user can choose to edit one field at a time by changing the function argument 'admin.levels' (in this case, the function returns a vector). However, some of the standardization processes become more complete if all the information is available for the four fields.

```
occs <- fixLoc(occs,
               loc.levels = c("country","stateProvince","municipality","locality"),
               scrap = TRUE)
```

Note that we performed the editing of the locality fields using the argument `scrap = TRUE`. This argument controls the search for missing information from the field 'locality' (i.e., text mining). It also performs some extra editing and cropping of the locality field to obtain more standardized locality descriptions. If the country, state, and municipality fields are given, they remain unaltered. Only missing information in these fields is completed by the editing and scrapping process.

Although the search for missing localities generally results in empty fields (i.e., NAs), it does not mean that the information retrieved from this search is accurate. It depends on whether the information is actually available in the field ‘locality’ and how it is arranged.

4.2.2 Crossing the locality information with the gazetteer

To make sure that the original locality information and the missing information obtained from function `fixLoc()` are valid localities, we need to crosscheck it with a gazetteer. In **plantR**, the search for missing information from a gazetteer is based on a standard locality string. This string simply is the concatenation of the country, state, municipality and locality fields, at the best resolution available. This string is created using the function `strLoc()`:

```
locs <- strLoc(occs)
```

If the municipality exists for a given state and country, then it is deemed as being a valid locality name, and the locality string associated with it is given priority for the validation of the locality information (see below). If there is missing information, then the alternative locality string (obtained using argument `scrap` of function `fixLoc()`) is used as a final attempt to search for missing information in the gazetteer. If both these alternatives do not work, then the upper administrative level available is used as the reference locality information.

The notation of locality information varies a lot, and there are many mistakes regarding the notation of some localities (e.g., “Balneário do Camboriú” instead of “Balneário Camboriú”). Therefore, it is useful to further simplify the locality strings, by reducing all name variants into the same locality string. This simplification is performed by the function `prepLoc()`:

```
locs$loc.string  <- prepLoc(locs$loc.string) # priority string
locs$loc.string1 <- prepLoc(locs$loc.string1) # alternative string
locs$loc.string2 <- prepLoc(locs$loc.string2) # alternative string
```

After the construction and simplification of the locality search strings, we can finally cross it with the gazetteer, using function `getLoc()`.

```
locs <- getLoc(locs, gazet = "plantR")
```

Some of the locality strings were retrieved in the default **plantR** gazetteer at the locality level (e.g., park, farm). The gazetteer is more complete for Latin America and it currently has information at the locality level mostly for Brazil. Therefore, info retrieval at the locality level should be biased towards specimens collected in Brazil. You can use the argument `gazet` to use your personal gazetteer instead of **plantR** default, as long as it contains the same fields (see the help of function `getLoc()` for more details).

Note that the function also returns the geographical coordinates from the gazetteer, if the locality information was found. Note as well that the edited version of the localities is different from the original localities regarding their resolution (see details in the next section, ‘Data validation’).

We then merge the new output of the locality data processing with the main data frame:

```
occs <- cbind.data.frame(occs, locs[, c("loc","loc.correct","latitude.gazetteer","longitude.gazetteer",
```

4.2.3 The wrapper function `formatLoc()`

As before, we provide a function to execute the previous steps at once:

```
occs1 <- formatDwc(splink_data = occs_splink,
                   gbif_data = occs_gbif, drop = TRUE)
## Dropping fields not essential or recommended to the data cleaning!
occs1 <- formatLoc(occs1)
```

4.2.4 Getting correct locality names

It may be useful to trace back the full and valid names of the localities retrieved from the gazetteer. This can be performed using the function `getAdmin()`, which makes the query based on the locality search string retrieved in the gazetteer:

```
head(getAdmin(occs))
```

```
##                             loc.correct NAME_0              NAME_1                  NAME_2 NAME_3 sou
## 1        brazil_rio grande norte_touros Brazil Rio Grande do Norte                  Touros   <NA>
## 2       brazil_rio grande norte_macaiba Brazil Rio Grande do Norte                 Macaíba   <NA>
## 3 brazil_pernambuco_jaboatao guararapes Brazil          Pernambuco Jaboatão dos Guararapes   <NA>
## 4              brazil_parana_nova tebas Brazil              Paraná              Nova Tebas   <NA>
## 5        brazil_espirito santo_linhares Brazil      Espírito Santo                Linhares   <NA>
## 6          brazil_sao paulo_nova odessa Brazil           São Paulo             Nova Odessa   <NA>
```

4.3 Editing geographical coordinates

Previous to the validation of the original geographical coordinates, it is important to make sure that they are provided in the required format: non-zero, non-NA decimal degrees, with decimal digits separated by points.

4.3.1 Preparing geographical coordinates

Function `prepCoord()` transforms the coordinates into decimal degrees and add the columns "decimalLatitude.new" and "decimalLongitude.new", which store the edited coordinates.

```
occs <- prepCoord(occs)
```

4.3.2 Obtaining coordinates from the gazetteer

Geographical coordinates are often missing or are provided in a format that the function `prepCoord()` cannot convert to decimal degrees. For the dataset we are using in this tutorial, we can inspect the proportion of occurrences without geographical coordinates:

```
table(!is.na(occs$decimalLatitude.new))/dim(occs)[1]
```

```
##
##     FALSE      TRUE
## 0.4324454 0.5675546
```

As we can see, ca. 20% of the occurrences records have no coordinates. Therefore, one may want to use the coordinates obtained from the gazetteer to replace the missing coordinates. In **plantR** we use the function `getCoord()` to do this procedure. This function also flags the origin and the probable precision of the original coordinates:

```
occs <- getCoord(occs)
table(!is.na(occs$decimalLatitude.new))/dim(occs)[1]
```

```
##
##      FALSE       TRUE
## 0.02490117 0.97509883
```

Now, missing coordinates were obtained from a gazetteer and there is a coordinate at the best resolution available for almost all occurrences.

4.3.3 The wrapper function `formatCoord()`

Similarly to the functions `formatOcc()` and `formatLoc()` from the previous editing steps, we provide a function that perform all the edits related to the geographical coordinates at once, the function `formatCoord()`.

Because getting missing coordinates from the gazetteer depends on the editing of the locality information, we need to run `formatLoc()` before running `formatCoord()`:

```
occs1 <- formatDwc(splink_data = occs_splink,
                  gbif_data = occs_gbif, drop = TRUE)
## Dropping fields not essential or recommended to the data cleaning!
occs1 <- formatLoc(occs1)
occs1 <- formatCoord(occs1)
```

4.4 Editing species taxonomic information

4.4.1 Editing species names

Although we downloaded occurrences for only three species names, the function `rspeciesLink()` returned (as required) all synonyms accepted by the Brazilian Flora 2020 (http://floradobrasil.jbrj.gov.br/reflora/listaBrasil). In addition, the speciesLink network (http://splink.cria.org.br/) stores the information as provided by the collections, which may vary in the way the scientific names are stored. For data downloaded only from GBIF (https://www.gbif.org/) this step may be unnecessary since GBIF already stores data in a more standardized way.

The **plantR** function `fixSpecies()` performs several edits, including the isolation of taxonomic rank abbreviations, the detection of name modifiers ("cf." and "aff.") and a suggestion for the name status:

```
occs <- fixSpecies(occs)
```

```
sort(table(occs$scientificName))
sort(table(occs$scientificName.new))
```

Note that the number of names has decreased, but there are still many different names than the three names used in the search. The possible statuses of those names are stored in the new column 'scientificNameStatus'.

```
sort(table(occs$scientificNameStatus))
```

```
##
##                    affinis      forma|name_w_authors                subspecies subspecies|name_w_autho
##                          1                         2                         2
##                   conferre    variety|name_w_authors                   variety            infra_speci
##                         17                       780                      1952                      27
##                possibly_ok            name_w_authors
##                      15515                     33138
```

4.4.2 Species synonyms and spelling

Although the function `fixSpecies()` presented above formatted the species names, there are still many more names than the ones we downloaded. Most of them are names at the infra-specific level, but others are probably synonyms, orthographic variants or even typos. There are R packages that check species nomenclature (e.g., **taxize**, **Taxonstand**, and **WorldFlora**). For species listed in the Brazilian Flora 2020, there is also the package `flora`, which follow the accepted nomenclature of the Brazilian Flora 2020 project. **plantR** uses the functionalities of some of these packages to inspect possible suggestions for species names, through the function `prepSpecies()`:

```
occs <- prepSpecies(occs, db = c("bfo", "tpl"), sug.dist = 0.85)
table(occs$suggestedName)
```

```
##
##        Casearia comocladifolia              Casearia decandra            Casearia sylvestris
##                              3                              2                          29326
##          Casselia integrifolia                    Celtis lima                 Celtis riparia
##                              1                              2                              4
##                 Euterpe edulis               Sponia micrantha                Trema micrantha
##                           1416                             19                          22256
## Trema micrantha var. floridana               Trema orientalis                   Trema rugosa
##                            289                              2                              3
```

```
table(occs$tax.notes)
```

```
##
##                                               accepted, but at a higher rank                     repl
##                                 48068                                    35
##                        was misspelled was misspelled|check undefined status      was misspelled|repl
##                                    32                                    19
```

The final output of the function provides the valid names for the names that were retrieved in the two databases selected (column 'suggestedName'). The function also returns which names were replaced or corrected (column 'tax.notes'). In addition, the flag 'check +1 accepted' may represent an indication of possible homonyms, i.e., different species with the same binomial, but different authorities. Together with names there were not found in both databases, those possible homonyms may deserve closer attention.

Note that there are still more names than those used to search for species records. Some of these differences are due to differences in the assignment of synonyms between speciesLink and the Brazilian Flora interface used here, the package **flora**.

For our example, we decided to replace all edited species names (i.e., 'scientificName.new') by the suggested species names from the function `prepSpecies()` (i.e., 'scientificName.new'). However, **plantR** does not execute this step automatically and the decision to do it should be taken by the user:

```
occs$scientificName.new <- occs$suggestedName
```

4.4.3 Botanical family synonyms and standardization

The classification of the taxonomic confidence in species determination of each occurrence is currently performed based on a global list of taxonomists per family (see details below). Because different plant classification systems can use different names to the same family, we need to standardize family names. The standard used by **plantR** to obtain valid family names is the APG IV and the PPG I, which can be obtained using the function `prepFamily()`:

```
occs <- prepFamily(occs)
```

```
## The following family names were automatically replaced:
##
## |Genus    |Old fam.       |New fam.    |
## |:--------|:--------------|:-----------|
## |Casearia |Fabaceae       |Salicaceae  |
## |Casearia |Vivianiaceae   |Salicaceae  |
## |Euterpe  |Palmae         |Arecaceae   |
## |Trema    |Connaraceae    |Cannabaceae |
## |Trema    |Flacourtiaceae |Cannabaceae |
## |Trema    |Ulmaceae       |Cannabaceae |
```

Note that the function returns warnings in the case of conflicts on family names between the original occurrences and the APG IV/PPG I family list. Some are just differences in the list of names accepted from different classification systems, but some may represent errors.

4.4.4 The wrapper function `formatTax()`

As for the other parts of the data editing and standardization, we provide a simple wrapper function that executes the steps above at once:

```
occs1 <- formatDwc(splink_data = occs_splink,
                   gbif_data = occs_gbif, drop = TRUE)
```

```
## Dropping fields not essential or recommended to the data cleaning!
```

```
occs1 <- formatTax(occs1)
```

```
## The following family names were automatically replaced:
##
## |Genus    |Old fam.       |New fam.    |
## |:--------|:--------------|:-----------|
## |Casearia |Fabaceae       |Salicaceae  |
## |Casearia |Vivianiaceae   |Salicaceae  |
## |Euterpe  |Palmae         |Arecaceae   |
## |Trema    |Connaraceae    |Cannabaceae |
## |Trema    |Flacourtiaceae |Cannabaceae |
## |Trema    |Ulmaceae       |Cannabaceae |
```

5 Data validation

The validation steps in **plantR** add new columns to the species data, flagging possible problems. Depending on the study aims and spatial scale, one can use these new columns to try to correct some of the problems or drop them from the final data set.

5.1 Validation of locality information: function `validateLoc()`

The standardization process of locality information and its comparison against the gazetteer return the localities (and coordinates) at the best resolution available. In this comparison, much information provided at the locality level is only retrieved at the municipality level, which is expected due to the completeness of the gazetteer used (not all names of places are stored in the standard **plantR** gazetteer at the locality level). But many of them are only retrieved at the state/province level, meaning that fields containing state and municipality information may need to be checked.

In **plantR** the resolution of the locality information provided with the specimen and the one found in the gazetteer are described using the function `validateLoc()`, which compares the two resolutions and stores the result of this comparison in a new column called 'loc.check':

```
occs <- validateLoc(occs)
```

```
## [1] "Locality resolution in the original data vs. edited data:"
##                 original
## edited           country locality municipality no_info stateProvince
##   country           6489      924           91       0            36
##   locality            41     5446            2       0             0
##   municipality        14    23588         4864       0             4
##   no_info             54       33            0    1287             1
##   stateProvince        2    10003          263       0           992
```

```
unique(occs$loc.check)
```

```
##  [1] "ok_same_resolution"     "check_local.2municip."  "check_local.2state"     "check_local.2country
##  [5] "ok_country2locality"    "check_municip.2state"   "check_country2no.info"  "ok_country2municip.
##  [9] "check_local.2no.info"   "ok_state2municip."      "ok_municip.2locality"   "ok_country2state"
## [13] "check_state2country"    "check_municip.2country" "check_state2no.info"
```

In this column, specimens for which the locality provided could not be found in the gazetteer at the same resolution are flagged with a "check_…". In these cases, the locality resolution is downgraded until a locality is found in the gazetteer. In the case that even the country is not found, then the locality is flagged as "no_info". We also flag all the specimens that retained their resolution (i.e., "ok_same_resolution") and those that could be retrieved at a better locality resolution, also flagged with an "ok_" (e.g., 'ok_state2municipality': a record which had its original resolution at the 'stateProvince' level is now at the 'municipality' level).

Note that many occurrences were flagged with a "check_…". The class "check_local.2municip." should not be a problem, since the internal **plantR** gazetteer does not contain a comprehensive list of names at the locality level for the entire world. However, other "check_…" classes (e.g., "check_local.2country") may indicate missing information or typos that may need some more careful inspection.

5.2 Validation of the geographical coordinates

One important part of the validation process is to verify if the original geographical coordinates provided with the data are valid. In **plantR**, this validation is performed by comparing the locality information retrieved from the gazetteer with the locality information obtained from maps based on the original coordinates. But the validation process also flags inverted/swapped coordinates, records falling near shorelines, borders of neighbour countries, spatial outliers and records from cultivated individuals.

5.2.1 Validate original coordinates at country, state and county level

The first step of the validation of the geographical coordinates is to check the original coordinates against the world (at the country level) and Latin-American maps (at the county level). This step is performed by the function `checkCoord()`, which returns a new column 'geo.check' containing the results:

```
occs <- checkCoord(occs,
                   keep.cols = c("geo.check", "NAME_0", "country.gazet"))
unique(occs$geo.check)
```

```
##  [1] "ok_county"         "ok_county_gazet"   "ok_locality_gazet" "ok_country"        "bad_country"
##  [6] "ok_state"          "sea"               "ok_state_gazet"    "ok_country_gazet"  "no_cannot_chec
## [11] "check_gazetteer"
```

The new column 'geo.check' has some main types of classes. The first type starts with an 'ok_' and is followed by the level at which the coordinate was validated (e.g., country, state, county). It also contains the classes 'sea' (i.e., coordinates falling into the sea or bays), 'bad_country' (i.e., coordinates falling in the land but in the wrong country), 'check_gazetteer' (i.e., locality not found in the gazetteer) and 'no_cannot_check' (i.e., no coordinates available).

The function `checkCoord()` can also return the distance (in meters) between the original coordinates from the county centroid listed in the locality information of the record. This may help to spot coordinates that were not validated but that are close to the reference locality.

5.2.2 Coordinates falling in a neighbouring country

Sometimes geographical coordinates can fall in another country due to rounding or precision issues or simply because the collector was not aware that a country's border was crossed before obtaining the coordinates. The function `checkBorders()` flag those cases, which is only done for the subset of records flagged previously as 'bad_country':

```
occs <- checkBorders(occs, output = 'same.col')
unique(occs$geo.check)
```

```
##  [1] "ok_county"           "ok_county_gazet"        "ok_locality_gazet"      "ok_country"
##  [5] "bad_country[border]" "ok_state"               "sea"                    "ok_state_gazet"
##  [9] "ok_country_gazet"       "no_cannot_check"        "bad_country[inverted?]" "check_gazetteer"
```

Note that for this example, we have set the argument `output` to ‘same.col’. This means that the output of the function will be merged into the existing column ‘geo.check’. But if one wants to store this information in a separate column, this argument can be set to ‘new.col’ and a new column called ‘border.check’ will be created.

5.2.3 Coordinates in the sea but close to the shore

For those geographical coordinates flagged as falling into the sea, we can check if they are close to the shore. Sometimes the coordinate was noted on land, but it falls into the sea due to rounding or precision issues, or simply because the reference map does not contain all islands or contours of the shoreline. In **plantR** this step is performed by the function `checkShore()`:

```
occs <- checkShore(occs, output = 'same.col')
unique(occs$geo.check)
```

```
##  [1] "ok_county"              "ok_county_gazet"        "ok_locality_gazet"      "ok_country"
##  [5] "bad_country[border]"    "ok_state"               "shore"                  "ok_state_gazet"
##  [9] "open_sea"               "ok_country_gazet"       "no_cannot_check"        "bad_country[inverte
## [13] "check_gazetteer"
```

As a result, the function `checkShore()` has re-classified all records flagged as ‘sea’ into ‘shore’ (i.e., near to the shore) and ‘open_sea’ (i.e., far from the shore). By default, this verification is performed by comparing the coordinates against a buffer of 0.5 degrees around continents and major islands (as above). So, open sea records are those more than 0.5 degrees (~ 55 km in the Equator) from continents and major islands. But this verification can also be performed by calculating the distance of each coordinate to the shore and establishing maximum distance as a limit to flag the coordinate as being close to shore (takes much longer).

5.2.4 Inverted and swapped coordinates

There are still coordinates flagged a ‘bad_country’ in our dataset. Some of those records can be due to inversions in the longitude and latitude or to the swapping/transposing between them (i.e., latitude as longitude or vice-versa).

```
occs <- checkInverted(occs, output = 'same.col')
unique(occs$geo.check)
```

```
##  [1] "ok_county"               "ok_county_gazet"         "ok_locality_gazet"       "ok_country"
##  [5] "bad_country[border]"     "ok_state"                "shore"                   "ok_state_gazet"
##  [9] "open_sea"                "ok_country_gazet"        "no_cannot_check"         "bad_country"
## [13] "check_gazetteer"         "ok_country[invert_lat]"  "ok_country[transposed]"  "ok_country[invert
## [17] "ok_country[invert_both]"
```

As a result, some of the records could be validated by inverting or swapping their coordinates. These records are flagged as “ok_country” followed by the combination of inverted and/or swapped coordinates for which validation was acquired in brackets. Note that this function does not perform the validation of the newly arranged coordinates at the state and county levels. So, if deemed necessary, the user can re-include them in the function `checkCoord()`, and step which is automatically executed by the wrapper function `validateCoord()` presented below.

5.2.5 Records from cultivated individuals

Although records from cultivated individuals are not detected using their geographical coordinates, we included the search for cultivated individuals in this section because they are often related to records far away from the species’ original distribution but not always. There are cases when cultivated individuals are within or close to the limits of the species range (i.e. public squares, botanical gardens, home yards) and so they will not be flagged as spatial outliers (see section 'Spatial outliers' below). Flagging these cultivated records may be important for subsetting the data for downstream analysis.

Function `getCult()` flags cultivated or possibly cultivated records in a new column called 'cult.check':

```
occs <- getCult(occs)
table(occs$cult.check)
```

```
## 
##      cultivated prob_cultivated 
##              10             196
```

Note that the function flags two categories: 'cultivated' and 'prob_cultivated'. The first is related to records with descriptions that contain some terms that clearly denote records from cultivated individuals (e.g., 'Cultivated', 'Planted', 'Exotic'). The second category is related to records that contain those terms in their descriptions, but not exclusively (e.g. 'Presence of exotic species'). These cases may lead to spurious flags of cultivated records and so they may need some level of double-checking by the user before they can be safely removed as being true records from cultivated individuals. But as the names of the two categories suggest, there is more certainty that a record was actually taken from a cultivated individual if it is flagged as 'cultivated' than if it is flagged as 'prob_cultivated'.

5.2.6 Spatial outliers

The last step of the validation of geographical coordinates is the search for spatial outliers (i.e., records too far away from species core distributions), which can indicate misidentifications or records obtained from cultivated individuals, but not always (as explained in the previous section). In **plantR**, the search for spatial outliers is based on Mahalanobis distances and is executed using the function `checkOut()`:

```
occs <- checkOut(occs, tax.name = "scientificName.new")
```

Note that since the search for spatial outliers is performed species by species, we need to provide the name of the column containing the taxon names. By default, the function internally removes invalid geographical coordinates and records flagged as cultivated prior to the estimation of the center and co-variance matrix of the geographical coordinates, which are used to get the Mahalanobis distances.

5.2.7 The wrapper function `validateCoord()`

Similar to the other steps of the **plantR** workflow, all the steps related to the validation of the geographical coordinates can be performed at once, using the function `validateCoord()`. Since this function uses information from localities,and taxonomic information, besides the geographical coordinates themselves, all format and validation steps related to this information should have been carried previously.

```
occs1 <- formatDwc(splink_data = occs_splink, 
                   gbif_data = occs_gbif, drop = TRUE)
```

```
## Dropping fields not essential or recommended to the data cleaning!
```

```
occs1 <- formatLoc(occs1)
occs1 <- formatCoord(occs1)
occs1 <- formatTax(occs1)
```

```
## The following family names were automatically replaced:
## 
## |Genus    |Old fam.       |New fam.    |
## |:--------|:--------------|:-----------|
## |Casearia |Fabaceae       |Salicaceae  |
## |Casearia |Vivianiaceae   |Salicaceae  |
## |Euterpe  |Palmae         |Arecaceae   |
## |Trema    |Connaraceae    |Cannabaceae |
## |Trema    |Flacourtiaceae |Cannabaceae |
## |Trema    |Ulmaceae       |Cannabaceae |
```

```
occs1 <- validateLoc(occs1)
```

```
## [1] "Locality resolution in the original data vs. edited data:"
##                original
## edited           country locality municipality no_info stateProvince
##   country           6489      924           91       0            36
##   locality            41     5446            2       0             0
##   municipality        14    23588         4864       0             4
##   no_info             54       33            0    1287             1
##   stateProvince        2    10003          263       0           992
```

```
occs1 <- validateCoord(occs1)
```

5.3 Confidence level of species determinations: function validateTax()

One of the main features of **plantR** is the possibility of validating species identifications against a global list of plant taxonomists. This validation consists of the classification of the species identification of each record into different confidence levels and it is performed using the function `validateTax()`:

```
occs <-  validateTax(occs)
table(occs$tax.check)/dim(occs)[1]
```

As expected, only ca. 15% of the specimens were identified by a family specialist. Note that the function also returns a list with the names in the TDWG format that have many identifications but are not listed as family specialists in the **plantR** database of plant taxonomists. Maybe this name could represent a missing taxonomist from the database. If this is the case, one can add one or more taxonomist names to the validation using the argument `miss.taxonomist`:

```
occs1 <- validateTax(occs, miss.taxonomist = c("Salicaceae_Hatschbach, G."))
```

```
## Top people with many determinations but not in the taxonomist list:
##
## |Identifier    | Records|
## |:-------------|-------:|
## |(mo, N.L.R.L. |       1|
## |(NA)          |       1|
```

```
table(occs1$tax.check)/dim(occs)[1]
```

```
##
##         high          low      unknown
## 5.874312e-03 3.694536e-05 9.940887e-01
```

This is not the optimal example because Gerdt (Guenther) Hatschbach was not a Salicaceae specialist. However, he provided determinations for many different families outside his specialty, often referred to as a generalist. There are some names of generalists in the **plantR** default taxonomist database, and those names can be included in the taxonomic validation through the arguments `generalist`. However, this list of generalist names is biased towards South America, particularly Brazil.

5.4 Duplicated specimens

Another main feature of **plantR** is the search for duplicated records across collections (i.e. same specimen deposited in different biological collections) and the propagation and homogenization of the information within groups of duplicated records.

5.4.1 Create unique record identifiers

First we need to make sure that each records from each collection has it own record identifier. This identifier is used to created the groups of duplicated records. In **plantR**, we create this identifier by concatenating the collection code (edited field 'collectionCode.new') and accession number of each record (field 'catalogNumber') using the function `getTombo()`. And we are going to call it 'numTombo':

```
occs$numTombo <- getTombo(occs[, "collectionCode.new"],
                            occs[, "catalogNumber"])
```

The function returns identifiers such as "UEC_42115" or "CEPEC_1133".

5.4.2 Preparing to search for duplicates

Before searching for the duplicates, we first need to prepare the dataset for the search. The function `prepDup()` basically creates concatenated strings with different sets of information, which will be used in the search. Different combinations of fields can be used to generate these search strings and they should be provided using the argument `comb.fields`:

```
dups <- prepDup(occs,
                 comb.fields =   list(c("family","col.last.name","col.number","col.loc"),
     c("family","col.last.name","col.number","col.year")))
```

In this example, we used two search strings, which differ from the last information (collection locality vs. collection year). The use of two or more search strings increases the chance of finding duplicates even if one or more information is missing.

Note that we used the default arguments of the `prepDup()` function, which exclude from the search the strings without collector name, number or year (e.g., 's.n.' or 's.d.'), as well as missing information for these columns (i.e., NAs). If one decided to include those strings in the search, the number of specimens that will pass the duplicate search will increase but the chances of finding false duplicates increase as well (e.g., Salicaceae + s.n. + s.n. + Ubatuba or Salicaceae + s.n. + s.n. + s.d.).

5.4.3 Finding duplicates across collections

Next, we can actually search for duplicated specimens using the function `getDup()`:

```
dups <- getDup(dups)
head(dups[order(dups$dup.ID),],6)
```

```
##     numTombo                      dup.srch.str1                dup.srch.str2 dup.numb dup.prop dup.I
## 1  HST_23333     Salicaceae_Gaudichaud_3_touros Salicaceae_Gaudichaud_3_2021        0        0     N
## 2  HST_23363       Salicaceae_Riecre_47_macaiba    Salicaceae_Riecre_47_2021        0        0     N
## 3  HST_23291                               <NA>                         <NA>       cc       cc     N
## 4  HCF_31796                               <NA>                         <NA>       cc       cc     N
## 5 CVRD_16660    Salicaceae_Spruce_1334_linhares  Salicaceae_Spruce_1334_2020        0        0     N
## 6 UEC_207122 Salicaceae_Peckolt_680_nova odessa  Salicaceae_Peckolt_680_2020        0        0     N
```

```
table(dups$dup.prop)
```

```
##
##     0    cc
##    12 54122
```

As a result, we found over 6000 records with a high chance of representing true duplicates, i.e., the proportion of search strings shared between them is maximum (dup.prop = 1). Other ~1000 specimens had only one search string in common. Moreover, there is an indication of possible ~7000 unicates, and of other ~5000 specimens we could not check for duplicates (i.e., all search strings used have missing information).

Note that if there are duplicated record identifiers (i.e., same accession number from the same collection) both are listed within the accession number grouped in the new column 'dup.ID'.

It is important to notice that the results of this duplicate search are only indications. First of all, the retrieval of duplicates depends on the dataset itself: if the collection where the duplicate was deposited is not within the dataset (e.g., collection is not in speciesLink or GBIF), the duplicate cannot be retrieved for obvious reasons. But other issues may change the number of unicates and duplicates found. If the search string is too flexible (e.g. only collector name and number), false duplicates may be retrieved if collector names are common (e.g., 'Silva', 'Santos' or 'Lima' for Portuguese family names) and collector numbers are low. So having more than one search string with at least 3 combinations of information decreases the chance of false duplicates. On the other hand, we may have false unicates due to differences in the notation among collections, which could not be solved by **plantR** (e.g., Mattos-Silva, L.A. vs. Silva, L.A.M.), or typos (e.g. Hoffmannsegg, J.C. vs. Hoffmannseg, J.C.). Finally, many duplicates may still exist but due to missing information (e.g., collector number or collection year or locality), they can not enter the duplicate search.

5.4.4 Homogenizing the information within duplicates

Since the amount and quality of the information among collections may vary, one may want to propagate/homogenize the information within the duplicates. This homogenization can be carried out regarding the specimen's identification or their locality and geographical coordinates. First, we need to combine the results of the duplicate search with the main data frame:

```
occs <- cbind.data.frame(occs,
                         dups[,c("dup.ID", "dup.numb", "dup.prop")],
                         stringsAsFactors = FALSE)
```

Next, we use the function `mergeDup()` to perform the propagation and homogenization itself, which aims at expanding the best-quality information found across all specimens of each group of duplicates.

```
occs <- mergeDup(occs)
```

```
## Warning: To merge taxonomic info, the input data must contain the following column(s): tax.check. Sk
## from info to merge.
```

```
table(occs$tax.check, occs$tax.check1)
```

```
## < table of extent 0 x 0 >
```

In this example, we show that the function was able to retrieve high-confidence species identifications for about 6000 specimens with low or unknown confidence levels. There was also an improvement on the stats regarding the geographical and locality checks (i.e., columns 'geo.check' vs. 'geo.check1' and 'loc.check' vs. 'loc.check1').

Note that we used the function defaults, which created new columns for the merged information. If one wants to save the new information in the same columns, the argument `overwrite` should be set to `TRUE`.

5.4.5 Removing duplicates after homogenization

plantR provides a simple function for the (fast) removal of duplicates from the data. By default, only duplicated records from the same collections are removed from the dataset. But if the argument `rm.all` is set to TRUE, all duplicates are removed from the data (i.e., only one record per group of duplicates remains in the dataset). This procedure should only be performed if the user deems it necessary and after the information among groups of duplicates has been homogenized. All the non-homogenized information contained in the other specimens is lost.

```
dim(occs)
## [1] 54134   100
```

```
occs <- rmDup(occs)
## [1] "No duplicate IDs available. Returning the same input data frame"
```

```
dim(occs)
## [1] 54134   100
```

5.4.6 The wrapper function validateDup()

Similar to the other validation steps, all the steps related to duplicated specimens can be performed at once, using the function validateDup(). Since this function uses information from localities, geographical coordinates and taxonomic information, all format and validation steps related to this information should have been carried previously.

6 Data summary and export

Finally, **plantR** can help users to summarize their data sets and the flags of the validation process. It also provides list of species with user-defined numbers of voucher specimens and the export of records by groups (e.g., families, countries, collections).

6.1 Number of specimens, collections, species and other descriptors

The function summaryData() prepares tables containing information on the number of records, collections, species, etc. It also provides summaries of the top collectors and the top countries with more records:

```
summ <- summaryData(occs)
```

```
## =========
##  RECORDS
## =========
## |Type                     | Records|
## |:------------------------|-------:|
## |Unicates                 |      12|
## |Duplicates               |       0|
## |Unknown                  |   54122|
## |Total without duplicates |   54134|
## |Total with duplicates    |   54134|
##
## =============
##  COLLECTIONS
## =============
## Number of biological collections: 332
## Number of collectors’ names: 6
## Collection years: 1760-2021 (>90% and >50% after 1946 and 1990)
##
## Top collections in numbers of records:
## |Collection                  | Records|
## |:---------------------------|-------:|
## |MO                          |    6388|
## |NY                          |    2789|
## |P                           |    2370|
## |http://grbio.org/cool/id-nn |    1928|
## |RB                          |    1852|
##
## Top collectors in numbers of records:
## |Collector     | Records|
## |:-------------|-------:|
## |[NA]          |      12|
## |Regnell, A.F. |       3|
## |Gaudichaud    |       2|
## |Peckolt       |       1|
## |Riecre        |       1|
##
## ==========
##  TAXONOMY
## ==========
## Number of families: 3
## Number of genera: 6
## Number of species: 12
##
## Top richest families:
## |family.new  |     N|  S|
## |:-----------|-----:|--:|
## |Cannabaceae | 22575|  7|
## |Salicaceae  | 30143|  4|
## |Arecaceae   |  1416|  1|
##
## Top richest genera:
## |genus.new |     N|  S|
## |:---------|-----:|--:|
## |Trema     | 22550|  4|
## |Casearia  | 29331|  3|
## |Celtis    |     6|  2|
## |Euterpe   |  1416|  1|
## |NA        |   811|  1|
##
## ===========
##  COUNTRIES
## ===========
## Number of countries: 57
##
## Top countries in numbers of records:
## |Country  | Records| Species|
## |:--------|-------:|-------:|
## |Brazil   |   34313|       6|
## |Mexico   |    4159|       4|
## |Colombia |    1852|       4|
## |Peru     |    1371|       3|
## |Paraguay |    1296|       4|
```

6.2 Summary of validation flags and other issues

plantR also provides summaries of the validation process, such as the number of duplicates found and the validation of the locality, coordinates and species identifications. This is performed by the function `summaryFlags()`:

```
flags <- summaryFlags(occs)
```

```
## ==================
##  DUPLICATE SEARCH
## ==================
## Records per strength of duplicate indication:
##
```

```
## |Strenght               | Records|
## |:----------------------|-------:|
## |0%                     |      12|
## |Cannot check (no info) |   54122|
##
## =====================
##  LOCALITY VALIDATION
## =====================
## Results of the locality validation:
##
## |Validation           | Records|
## |:--------------------|-------:|
## |probably ok          |   23588|
## |ok (same resolution) |   19078|
## |check (downgraded)   |   11317|
## |check (not found)    |      88|
## |ok (upgraded)        |      63|
##
## Details of the validation (original vs. validated localities):
##
## |original.resolution | no_info| country| stateProvince| municipality| locality|
## |:-------------------|-------:|-------:|-------------:|------------:|--------:|
## |no_info             |    1287|       0|             0|            0|        0|
## |country             |      54|    6489|             2|           14|       41|
## |stateProvince       |       1|      36|           992|            4|        0|
## |municipality        |       0|      91|           263|         4864|        2|
## |locality            |      33|     924|         10003|        23588|     5446|
##
## =======================
##  COORDINATE VALIDATION
## =======================
## Valid coordinates per origin:
##
## |Validated |Origin       | Records|
## |:---------|:------------|-------:|
## |yes       |original     |   30699|
## |yes       |gazetter     |   22087|
## |no        |cannot_check |    1348|
##
## Valid coordinates per resolution:
##
## |Validated |Resolution          | Records|
## |:---------|:-------------------|-------:|
## |yes       |ok_county           |   28382|
## |yes       |ok_state            |   13480|
## |yes       |ok_country          |    8451|
## |yes       |ok_locality         |    1869|
## |no        |no_cannot_check     |    1348|
## |yes       |shore               |     273|
## |yes       |open_sea            |     140|
## |yes       |bad_country[border] |      99|
## |yes       |bad_country         |      67|
## |yes       |check_gazetteer     |      25|
##
```

```
## ======================
##  CULTIVATED SPECIMENS
## ======================
## Number of specimens from cultivated individuals:
##
## |Cultivated   | Records|
## |:------------|-------:|
## |probably not |   53928|
## |probably yes |     196|
## |yes          |      10|
```

6.3 Generating species lists or checklists

Another feature of **plantR** is the possibility of building species lists from the dataset of species records. This is done using the function `checkList()` which returns the family, the total number of records and a list of vouchers (in a short or long format). The function also returns the percentage of records with high confidence species identification and with validated geographical coordinates. If the dataset is associated to an specific geographical area, this list of species can be seen as a species checklist.

```
spp.check <- checkList(occs)
```

```
## [1] "No duplicate IDs available. Returning the same input data frame"
```

6.4 Saving the occurrence data

Finally, **plantR** provides the function `saveData()` that saves the species records divided by any group of information, such as the species taxonomy, collections, years and countries. By default, records will be saved as compressed ‘csv’ files in the user working directory. But the user can provide any directory using the argument `dir.name`.

```
saveData(occs, by = "family")
```