A Simple Workflow

2025-01-15

## Introduction

This document is intended to act as a primer for the use of the new Neotoma R package, neotoma2 and is the companion to the [*Introduction to Neotoma* presentation](https://docs.google.com/presentation/d/1Fwp5yMAvIdgYpiC04xhgV7OQZ-olZIcUiGnLfyLPUt4/edit?usp=sharing). Some users may be working with this document as part of a workshop for which there is a Binder instance. The Binder instance will run RStudio in your browser, with all the required packages installed.

If you are using this workflow on its own, or want to use the package directly, [the neotoma2 package](https://github.com/NeotomaDB/neotoma2) is available on CRAN by running:

install.packages('neotoma2')  
library(neotoma2)

Your version should be at or above 1.0.5.

This workshop will also require other packages. To maintain the flow of this document we’ve placed instructions at the end of the document in the section labelled “[Installing packages on your own](#localinstall)”. Please install these packages, and make sure they are at the lastest version.

## Learning Goals

In this tutorial you will learn how to:

1. [Site Searches](#Xb5d6e9ac3a71ce9ee21fb4adf696179f0c676dd): Search for sites using site names and geographic parameters.
2. [Filter Results](#Xc137b8f049b608263a1e85cabf1f29f9416ab97): Filter results using temporal and spatial parameters.
3. [Explore Data](#Xb514ed3696a9385b48ff9718814784d686e88a0): Obtain sample information for the selected datasets.
4. [Visualize Data](#Xcb865a7d131d2bec6508c2bd3408f80abc5dead): Perform basic Stratigraphic Plotting

## Background

### Getting Help with Neotoma

If you’re planning on working with Neotoma, please join us on [Slack](https://join.slack.com/t/neotomadb/shared_invite/zt-cvsv53ep-wjGeCTkq7IhP6eUNA9NxYQ) where we manage a channel specifically for questions about the R package (the *#it\_r* channel, or *#it\_r\_es* for R help in Spanish and *#it\_r\_jp* in Japanese). You may also wish to join the Neotoma community through our Google Groups mailing lists; please [see the information on our website](https://www.neotomadb.org/about/join-the-neotoma-community) to be added.

### Understanding Data Structures in Neotoma

Data in the Neotoma database itself is structured as a set of linked relationships to express different elements of paleoecological analysis:

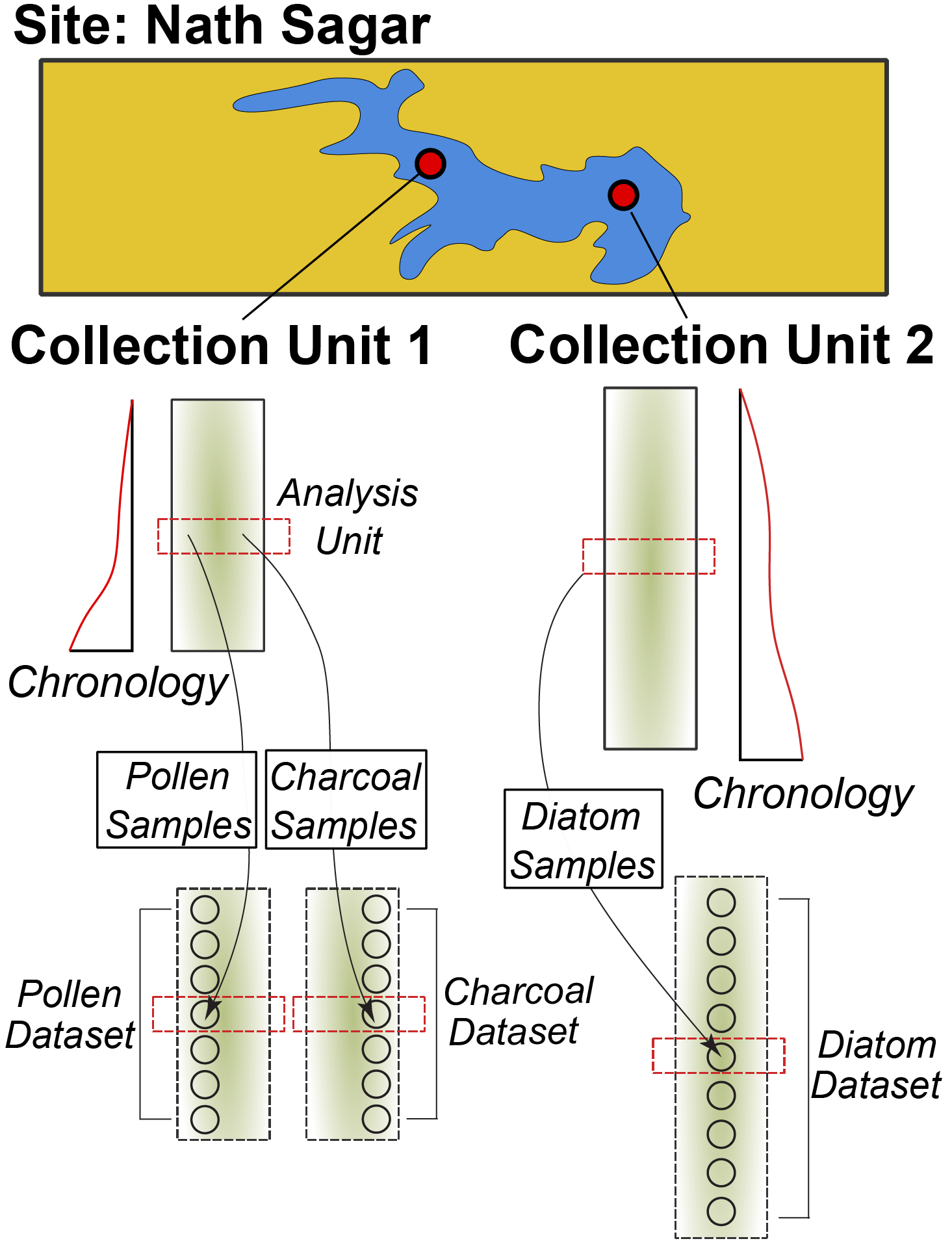
* space and time
  + Where is a sample located in latitude and longitude?
  + Where is a sample along a depth profile?
  + What is the estimated age of that sample?
  + What is the recorded age of elements within or adjacent to the sample?
* observations
  + What is being counted or measured?
  + What units are being used?
  + Who observed it?
* scientific methods
  + What statistical model was used to calculate age?
  + What uncertainty terms are used in describing an observation?
* conceptual data models
  + How do observations in one sample relate to other samples within the same collection?
  + How does an observation of a fossil relate to extant or extinct relatives?

These relationships can be complex because paleoecology is a broad and evolving discipline. As such, the database itself is highly structured, and normalized, to allow new relationships and facts to be added, while maintaining a stable central data model. If you want to better understand concepts within the database, you can read the [Neotoma Database Manual](https://open.neotomadb.org/manual), or take a look at [the database schema itself](https://open.neotomadb.org/dbschema).

In this workshop we want to highlight two key structural concepts:

1. The way data is structured conceptually within Neotoma (Sites, Collection Units and Datasets).
2. The way that this structure is adapted within the neotoma2 R package.

#### Data Structure in the Neotoma Database



**Figure**. *The structure of sites, collection units, samples, and datasets within Neotoma. A site contains one or more collection units. Chronologies are associated with collection units. Samples with data of a common type (pollen, diatoms, vertebrate fauna) are assigned to a dataset.*

Data in Neotoma is associated with **sites** – specific locations with latitude and longitude coordinates.

Within a **site**, there may be one or more [**collection units**](https://open.neotomadb.org/manual/dataset-collection-related-tables-1.html#CollectionUnits) – locations at which samples are physically collected within the site:

* an archaeological **site** may have one or more **collection units**, pits within a broader dig site
* a pollen sampling **site** on a lake may have multiple **collection units** – core sites within the lake basin.
* A bog sample **site** may have multiple **collection units** – a transect of surface samples within the bog.

Collection units may have higher resolution GPS locations than the site location, but are considered to be part of the broader site.

Data within a **collection unit** is collected at various [**analysis units**](https://open.neotomadb.org/manual/sample-related-tables-1.html#AnalysisUnits).

* All sediment at 10cm depth in the depth profile of a cutbank (the collection unit) along an oxbow lake (the site) is one analysis unit.
* All material in a single surface sample (the collection unit) from a bog (the site) is an analysis unit.
* All fossil remains in a buried layer from a bone pile (the collection unit) in a cave (the site) is an analysis unit.

Any data sampled within an analysis unit is grouped by the dataset type (charcoal, diatom, dinoflagellate, etc.) and aggregated into a [**sample**](https://open.neotomadb.org/manual/sample-related-tables-1.html#Samples). The set of samples for a collection unit of a particular dataset type is then assigned to a [**dataset**](https://open.neotomadb.org/manual/dataset-collection-related-tables-1.html#Datasets).

* A sample would be all diatoms (the dataset type) extracted from sediment at 12cm (the analysis unit) in a core (the collection unit) obtained from a lake (the site).
* A sample would be the record of a single mammoth bone (sample and analysis unit, dataset type is vertebrate fauna) embeded in a riverbank (here the site, and collection unit).

#### Data Structures in neotoma2

**Figure**. *Neotoma R Package UML diagram. Each box represents a data class within the package. Individual boxes show the class object, its name, its properties, and functions that can be applied to those objects. For example, a sites object has a property sites, that is a list. The function plotLeaflet() can be used on a sites object.*

If we look at the [UML diagram](https://en.wikipedia.org/wiki/Unified_Modeling_Language) for the objects in the neotoma2 R package we can see that the data structure generally mimics the structure within the database itself. As we will see in the [Site Searches section](#Xb5d6e9ac3a71ce9ee21fb4adf696179f0c676dd), we can search for these objects, and begin to manipulate them (in the [Simple Analysis section](#Xcb865a7d131d2bec6508c2bd3408f80abc5dead)).

It is important to note: *within the neotoma2 R package, most objects are sites objects, they just contain more or less data*. There are a set of functions that can operate on sites. As we add to sites objects, using get\_datasets() or get\_downloads(), we are able to use more of these helper functions.

## Site Searches

### get\_sites()

There are several ways to find sites in neotoma2, but we think of sites as being spatial objects primarily. They have names, locations, and are found within the context of geopolitical units, but within the API and the package, the site itself does not have associated information about taxa, dataset types or ages. It is simply the container into which we add that information. So, when we search for sites we can search by:

| Parameter | Description |
| --- | --- |
| sitename | A valid site name (case insensitive) using % as a wildcard. |
| siteid | A unique numeric site id from the Neotoma Database |
| loc | A bounding box vector, geoJSON or WKT string. |
| altmin | Lower altitude bound for sites. |
| altmax | Upper altitude bound for site locations. |
| database | The constituent database from which the records are pulled. |
| datasettype | The kind of dataset (see get\_tables(datasettypes)) |
| datasetid | Unique numeric dataset identifier in Neotoma |
| doi | A valid dataset DOI in Neotoma |
| gpid | A unique numeric identifier, or text string identifying a geopolitical unit in Neotoma |
| keywords | Unique sample keywords for records in Neotoma. |
| contacts | A name or numeric id for individuals associuated with sites. |
| taxa | Unique numeric identifiers or taxon names associated with sites. |

All sites in Neotoma contain one or more datasets. It’s worth noting that the results of these search parameters may be slightly unexpected. For example, searching for sites by sitename, latitude, or altitude will return all of the datasets for the particular site. Searching for terms such as datasettype, datasetid or taxa will return the site, but the only datasets returned will be those matching the dataset-specific search terms. We’ll see this later.

#### Site names: sitename="%ø%"

We may know exactly what site we’re looking for (“Lake Solsø”), or have an approximate guess for the site name (for example, we know it’s something like “Solsø”, but we’re not sure how it was entered specifically), or we may want to search all sites that have a specific term, for example, *ø*.

We use the general format: get\_sites(sitename="%ø%") for searching by name.

PostgreSQL (and the API) uses the percent sign as a wildcard. So "%ø%" would pick up [“Lake Solsø”](https://data.neotomadb.org/4445) for us (and picks up “Isbenttjønn” and “Lake Flåfattjønna”). Note that the search query is also case insensitive.

##### Code

denmark\_sites <- neotoma2::get\_sites(sitename = "%ø%")  
plotLeaflet(denmark\_sites)

##### Result

#### Location: loc=c()

The original neotoma package used a bounding box for locations, structured as a vector of latitude and longitude values: c(xmin, ymin, xmax, ymax). The neotoma2 R package supports both this simple bounding box, but also more complex spatial objects, using the [sf package](https://r-spatial.github.io/sf/). Using the sf package allows us to more easily work with raster and polygon data in R, and to select sites from more complex spatial objects. The loc parameter works with the simple vector, [WKT](https://arthur-e.github.io/Wicket/sandbox-gmaps3.html), [geoJSON](http://geojson.io/#map=2/20.0/0.0) objects and native sf objects in R.

As an example of searching for sites using a location, we’ve created a rough representation of Denmark as a polygon. To work with this spatial object in R we also transformed the geoJSON element to an object for the sf package. There are many other tools to work with spatial objects in R. Regardless of how you get the data into R, neotoma2 works with almost all objects in the sf package.

geoJSON <- '{"coordinates": [[  
 [7.92, 55.02],  
 [12.42, 54.42],  
 [12.86, 55.98],  
 [12.41, 56.44],  
 [10.63, 57.96],  
 [7.74, 57.48],  
 [ 7.70, 57.09],  
 [7.92, 55.02]  
 ]],  
 "type": "Polygon"}'  
  
denmark\_sf <- geojsonsf::geojson\_sf(geoJSON)  
  
# Note here we use the `all\_data` flag to capture all the sites within the polygon.  
# We're using `all\_data` here because we know that the site information is relatively small  
# for denmark. If we were working in a new area or with a new search we would limit the  
# search size.  
denmark\_sites <- neotoma2::get\_sites(loc = denmark\_sf, all\_data = TRUE)

You can always simply plot() the sites objects, but you will lose some of the geographic context. The plotLeaflet() function returns a leaflet() map, and allows you to further customize it, or add additional spatial data (like our original bounding polygon, sa\_sf, which works directly with the R leaflet package):

##### Code

Note the use of the %>% pipe here. If you are not familiar with this symbol, check our [“Piping in R” section](#piping-in-r) of the Appendix.

neotoma2::plotLeaflet(denmark\_sites) %>%   
 leaflet::addPolygons(map = .,   
 data = denmark\_sf,   
 color = "green")

##### Result

#### site Object Helpers

If we look at the [data structure diagram](#X295fd52714430d967ee1e3ff2425c0027a1bc33) for the objects in the neotoma2 R package we can see that there are a set of functions that can operate on sites. As we retrieve more information for sites objects, using get\_datasets() or get\_downloads(), we are able to use more of these helper functions.

As it is, we can take advantage of functions like summary() to get a more complete sense of the types of data we have in denmark\_sites. The following code gives the summary table. We do some R magic here to change the way the data is displayed (turning it into a [DT::datatable()](https://rstudio.github.io/DT/) object), but the main piece is the summary() call.

##### Code

# Give information about the sites themselves, site names &cetera.  
neotoma2::summary(denmark\_sites)  
# Give the unique identifiers for sites, collection units and datasets found at those sites.  
neotoma2::getids(denmark\_sites)

##### Result

In this document we list only the first 10 records (there are more, you can use length(datasets(denmark\_sites)) to see how many datasets you’ve got). We can see that there are no chronologies associated with the site objects. This is because, at present, we have not pulled in the dataset information we need. In Neotoma, a chronology is associated with a collection unit (and that metadata is pulled by get\_datasets() or get\_downloads()). All we know from get\_sites() are the kinds of datasets we have and the location of the sites that contain the datasets.

### get\_datasets()

Within Neotoma, collection units and datasets are contained within sites. Similarly, a sites object contains collectionunits which contain datasets. From the table above (Result tab in Section 3.1.3.2) we can see that some of the sites we’ve looked at contain pollen records, some contain geochronologic data and some contain other dataset types. We could write something like this: table(summary(denmark\_sites)$types) to see the different datasettypes and their counts.

With a sites object we can directly call get\_datasets() to pull in more metadata about the datasets. The get\_datasets() method also supports any of the search terms listed above in the [Site Search](#Xb5d6e9ac3a71ce9ee21fb4adf696179f0c676dd) section. At any time we can use datasets() to get more information about any datasets that a sites object may contain. Compare the output of datasets(denmark\_sites) to the output of a similar call using the following:

#### Code

# This may be slow, because there's a lot of sites!  
# denmark\_datasets <- neotoma2::get\_datasets(denmark\_sites, all\_data = TRUE)  
  
denmark\_datasets <- neotoma2::get\_datasets(loc = denmark\_sf, datasettype = "pollen", all\_data = TRUE)  
  
datasets(denmark\_datasets)

#### Result

You can see that this provides information only about the specific dataset, not the site! For a more complete record we can join site information from summary() to dataset information using datasets() using the getids() function which links sites, and all the collection units and datasets they contain.

### filter() Records

If we choose to pull in information about only a single dataset type, or if there is additional filtering we want to do before we download the data, we can use the filter() function. For example, if we only want sedimentary pollen records (as opposed to pollen surface samples), and want records with known chronologies, we can filter by datasettype and by the presence of an age\_range\_young, which would indicate that there is a chronology that defines bounds for ages within the record.

#### Code

denmark\_records <- denmark\_datasets %>%   
 neotoma2::filter(!is.na(age\_range\_young))  
  
neotoma2::summary(denmark\_records)  
  
# We've removed records, so the new object should be shorter than the original.  
length(denmark\_records) < length(denmark\_datasets)

#### Result

We can see now that the data table looks different (comparing it to the [table above](#X39f1e17cfac8df9bf9fdfa2f36b7bbe8bee9f42)), and there are fewer total sites. Again, there is no explicit chronology for these records, we need to pull down the complete download for these records, but we begin to get a sense of what kind of data we have.

### Pulling in sample() data

Because sample data adds a lot of overhead (for this pollen data, the object that includes the dataset with samples is 20 times larger than the dataset alone), we try to call get\_downloads() only after we’ve done our preliminary filtering. After get\_datasets() you have enough information to filter based on location, time bounds and dataset type. When we move to get\_download() we can do more fine-tuned filtering at the analysis unit or taxon level.

The following call can take some time, but we’ve frozen the object as an RDS data file. You can run this command on your own, and let it run for a bit, or you can just load the object in.

## This line is commented out because we've already run it for you.  
## denmark\_dl <- denmark\_records %>% get\_downloads(all\_data = TRUE)  
## saveRDS(denmark\_dl, "data/dkDownload.RDS")  
denmark\_dl <- readRDS("data/dkDownload.RDS")

Once we’ve downloaded, we now have information for each site about all the associated collection units, the datasets, and, for each dataset, all the samples associated with the datasets. To extract samples all downloads we can call:

allSamp <- samples(denmark\_dl)

When we’ve done this, we get a data.frame that is 10055 rows long and 37 columns wide. The reason the table is so wide is that we are returning data in a **long** format. Each row contains all the information you should need to properly interpret it:

## [1] "age" "agetype" "ageolder" "ageyounger"   
## [5] "chronologyid" "chronologyname" "units" "value"   
## [9] "context" "element" "taxonid" "symmetry"   
## [13] "taxongroup" "elementtype" "variablename" "ecologicalgroup"  
## [17] "analysisunitid" "sampleanalyst" "sampleid" "depth"   
## [21] "thickness" "samplename" "datasetid" "database"   
## [25] "datasettype" "age\_range\_old" "age\_range\_young" "datasetnotes"   
## [29] "siteid" "sitename" "lat" "long"   
## [33] "area" "sitenotes" "description" "elev"   
## [37] "collunitid"

For some dataset types or analyses, some of these columns may not be needed, however, for other dataset types they may be critically important. To allow the neotoma2 package to be as useful as possible for the community we’ve included as many as we can.

#### Extracting Taxa

If you want to know what taxa we have in the record you can use the helper function taxa() on the sites object. The taxa() function gives us not only the unique taxa, but two additional columns – sites and samples – that tell us how many sites the taxa appear in, and how many samples the taxa appear in, to help us better understand how common individual taxa are.

##### Code

neotomatx <- neotoma2::taxa(denmark\_dl)

##### Results

#### Understanding Taxonomies in Neotoma

Taxonomies in Neotoma are not as straightforward as we might expect. Taxonomic identification in paleoecology can be complex, impacted by the morphology of the object we are trying to identify, the condition of the palynomorph, the expertise of the analyst, and many other conditions. You can read more about concepts of taxonomy within Neotoma in the Neotoma Manual’s [section on Taxonomic concepts](https://open.neotomadb.org/manual/database-design-concepts.html#taxonomy-and-synonymy).

We use the unique identifiers (*e.g.*, taxonid, siteid, analysisunitid) throughout the package, since they help us to link between records. The taxonid values returned by the taxa() call can be linked to the taxonid column in the samples() table. This allows us to build taxon harmonization tables if we choose to. You may also note that the taxonname is in the field variablename. Individual sample counts are reported in Neotoma as [variables](https://open.neotomadb.org/manual/taxonomy-related-tables-1.html#Variables). A “variable” may be either a species, something like laboratory measurements, or a non-organic proxy, like charcoal or XRF measurements, and includes the units of measurement and the value.

#### Simple Harmonization

Let’s say we want all samples from which *Poaceae* (grass) taxa have been reported to be grouped together into one pseudo-taxon called *Poaceae-undiff*. **NOTE**, this is may not be an ecologically useful grouping, but is used here for illustration.

There are several ways of grouping taxa, either directly by exporting the file and editing each individual cell, or by creating an external “harmonization” table (which we did in the prior neotoma package). First, lets look for how many different ways *Poaceae* appears in these records. We can use the function str\_detect() from the stringr package to look for patterns, and then return either TRUE or FALSE when the string is detected:

## # A tibble: 4 × 11  
## # Groups: units, context, element, taxonid, symmetry, taxongroup,  
## # elementtype, variablename, ecologicalgroup [4]  
## units context element taxonid symmetry taxongroup elementtype variablename  
## <chr> <chr> <chr> <int> <lgl> <chr> <chr> <chr>   
## 1 NISP <NA> pollen 125 NA Vascular plan… pollen Poaceae und…  
## 2 NISP <NA> pollen 417 NA Vascular plan… pollen Poaceae   
## 3 NISP <NA> pollen 2113 NA Vascular plan… pollen Poaceae (>4…  
## 4 NISP <NA> pollen 33543 NA Vascular plan… pollen Poaceae und…  
## # ℹ 3 more variables: ecologicalgroup <chr>, samples <int>, sites <int>

We can harmonize taxon by taxon a number of different ways. One way would be to get every instance of a *Poaceae* taxon and just change them directly. Here we are taking the column variablename from the allSamp object (this is where the count data is). The square brackets are telling us which rows we’re changing, here only rows where we detect "Poaceae" in the variable name. For each of those rows, in that column, we assign the value "Poaceae undiff":

There were originally 4 different taxa identified as being within the genus *Poaceae* (including *Poaceae*., *Poaceae (>40 µm)*, and *Poaceae undiff. (<40 µm)*). The above code reduces them all to a single taxonomic group *Poaceae undiff*.

Note that this changes *Poaceae* in the allSamp object *only*, not in any of the downloaded objects. If we were to call samples() again, the taxonomy would return to its original form.

A second way to harmonize taxa is to use an external table, which is especially useful if we want to have an artifact of our choices. For example, a table of pairs (what we want changed, and the name we want it replaced with) can be generated, and it can include regular expressions (if we choose):

| original | replacement |
| --- | --- |
| Poaceae.\* | Poaceae-undiff |
| Picea.\* | Picea-undiff |
| Plantago.\* | Plantago-undiff |
| Quercus.\* | Quercus-undiff |
| … | … |

We can get the list of original names directly from the taxa() call, applied to a sites object that contains samples, and then export it using write.csv().

taxaplots <- taxa(denmark\_dl)  
# Save the taxon list to file so we can edit it subsequently.  
readr::write\_csv(taxaplots, "data/mytaxontable.csv")

#### Looking at the Taxonomic Structure

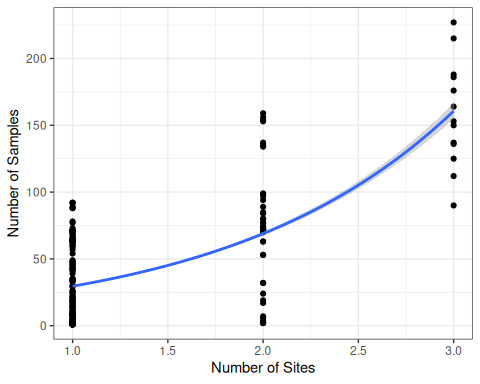
The taxa function returns all our taxonomic information, and it provides some additional information, the columns samples and sites which record the number of samples across all datasets that contain the taxon, and the number of sites with the taxon. The plot below shows the relationship between samples and sites, which we would expect to be somewhat skewed, as it is.

This is effectively a rarefaction curve, the more sites a taxon is found at, the more samples it is found at.

##### Code

taxaplots <- taxa(denmark\_dl)  
ggplot(data = taxaplots, aes(x = sites, y = samples)) +  
 geom\_point() +  
 stat\_smooth(method = 'glm',   
 method.args = list(family = 'poisson')) +  
 xlab("Number of Sites") +  
 ylab("Number of Samples") +  
 theme\_bw()

##### Result



**Figure**. *A plot of the number of sites a taxon appears in, against the number of samples a taxon appears in.*

#### Editing the Taxonomy Table

The plot (above) is mostly for illustration, but we can see, as a sanity check, that the relationship is as we’d expect. Here, each point represents a separate taxon, roughly, so there is a large density of points (taxa) that plot in the lower left section of the figure, and fewer points in the upper right. This means that there are a large number of taxa that are rarely present and then several that are quite common.

Exporting the taxon table to a csv file allows us to edit the table, filtering and selecting taxa based on contextual information, such as the ecologicalgroup or taxongroup to help you out. Once you’ve cleaned up the translation table you can load it in (try to save it under a different file name!), and then apply the transformation:

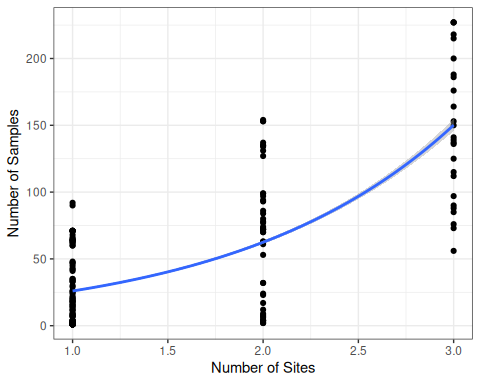
translation <- readr::read\_csv("data/taxontable.csv")

I did a bunch of work here. . . Then we read it in.

You can see we’ve changed some of the taxon names in the taxon table. To replace the names in the samples() output, we’ll join the two tables using an inner\_join() (meaning the variablename must appear in both tables for the result to be included), and then we’re going to select only those elements of the sample tables that are relevant to our later analysis, using the harmonizedname column as our new name for the taxa:

allSamp <- samples(denmark\_dl)  
  
allSamp <- allSamp %>%  
 inner\_join(translation, by = c("variablename" = "variablename")) %>%   
 dplyr::select(!c("variablename")) %>%   
 group\_by(siteid, sitename, harmonizedname,  
 sampleid, units, age,  
 agetype, depth, datasetid,  
 long, lat) %>%  
 summarise(value = sum(value), .groups='keep')

We now have a cleaner set of taxon names compared to the original table, both because of harmonization, and because we cleared out many of the non-**TRSH** (trees and shrubs) taxa from the harmonization table. Plotting the same set of taxa with the new harmonized names results in this plot:



**Figure**. *The same site/sample plot as above, with with the new harmonized taxonomy. Note that the distribution of points along the curve is smoother, as we remove some of the taxonomic issues.*

## Simple Analytics

### Stratigraphic Plotting

To plot at strategraphic diargram we are only interested in one site and in one dataset. By looking at the summary of downloads we can see that Lake Solsø has two collection units that both have a pollen record. Lets look at the SOLSOE81 collection unit, which is the second download. To get the samples from just that one collection unit by specifying that you want only the samples from the second download.

We can use packages like rioja to do stratigraphic plotting for a single record, but first we need to do some different data management. Although we could do harmonization again we’re going to simply take the taxa at a single site and plot them in a stratigraphic diagram. However, if you would like to plot multiple sites and you want them to have harmonized taxa we have provided examples on how to do both.

#### Raw Taxon

# Get a particular site, in this case we are simply subsetting the  
# `denmark\_dl` object to get Lake Solsø:  
plottingSite <- denmark\_dl[[2]]  
  
# Select only pollen measured using NISP and convert to a "wide"  
# table, using proportions. The first column will be "age".  
# This turns our "long" table into a "wide" table:  
counts <- plottingSite %>%  
 samples() %>%  
 toWide(ecologicalgroup = c("TRSH"),  
 unit = c("NISP"),  
 elementtypes = c("pollen"),  
 groupby = "age",  
 operation = "prop") %>%  
 arrange(age)  
  
counts <- counts[, colSums(counts > 0.01, na.rm = TRUE) > 5]

#### With Harmonization

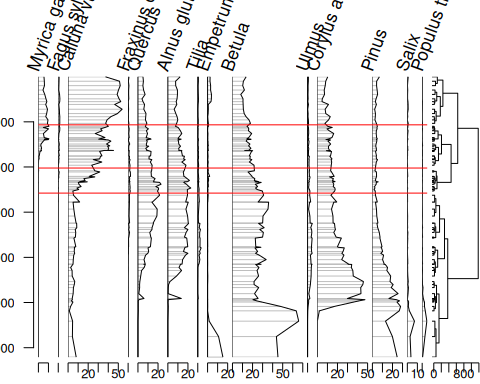
# Get a particular site, in this case we are simply subsetting the  
# `denmark\_dl` object to get Lake Solsø:  
plottingSite <- denmark\_dl[[2]]  
  
# Select only pollen measured using NISP and convert to a "wide"  
# table, using proportions. The first column will be "age".  
# This turns our "long" table into a "wide" table:  
counts\_harmonized <- plottingSite %>%  
 samples() %>%  
 toWide(ecologicalgroup = c("TRSH"),  
 unit = c("NISP"),  
 elementtypes = c("pollen"),  
 groupby = "age",  
 operation = "prop") %>%  
 arrange(age) %>%  
 pivot\_longer(-age) %>%  
 inner\_join(translation, by = c("name" = "variablename")) %>%   
 dplyr::select(!c("name", taxonid)) %>%   
 group\_by(harmonizedname, age) %>%  
 summarise(value = sum(value), .groups='keep')%>%  
 pivot\_wider(names\_from = harmonizedname, values\_from = value)  
  
counts\_harmonized <- counts\_harmonized[, colSums(counts\_harmonized > 0.01, na.rm = TRUE) > 5]

Hopefully the code is pretty straightforward. The toWide() function provides you with significant control over the taxa, units and other elements of your data before you get them into the wide matrix (depth by taxon) that most statistical tools such as the vegan package or rioja use.

To plot the data we can use rioja’s strat.plot(), sorting the taxa using weighted averaging scores (wa.order). I’ve also added a CONISS plot to the edge of the the plot, to show how the new *wide* data frame works with distance metric funcitons.

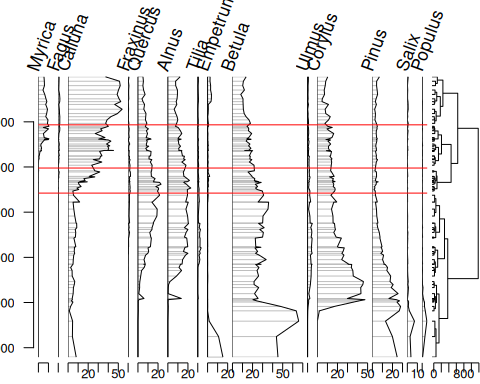
#### Raw Taxon

# Perform constrained clustering:  
clust <- rioja::chclust(dist(sqrt(counts)),  
 method = "coniss")  
  
# Plot the stratigraphic plot, converting proportions to percentages:  
plot <- rioja::strat.plot(counts[,-1] \* 100, yvar = counts$age,  
 title = denmark\_dl[[1]]$sitename,  
 ylabel = "Calibrated Years BP",  
 xlabel = "Pollen (% of Trees and Shrubs)",  
 srt.xlabel = 70,  
 y.rev = TRUE,  
 clust = clust,  
 wa.order = "topleft",  
 scale.percent = TRUE)  
  
rioja::addClustZone(plot, clust, 4, col = "red")



#### With Harmonization

# Perform constrained clustering:  
clust <- rioja::chclust(dist(sqrt(counts\_harmonized)),  
 method = "coniss")  
  
# Plot the stratigraphic plot, converting proportions to percentages:  
plot <- rioja::strat.plot(counts\_harmonized[,-1] \* 100, yvar = counts\_harmonized$age,  
 title = denmark\_dl[[1]]$sitename,  
 ylabel = "Calibrated Years BP",  
 xlabel = "Pollen (% of Trees and Shrubs)",  
 srt.xlabel = 70,  
 y.rev = TRUE,  
 clust = clust,  
 wa.order = "topleft",  
 scale.percent = TRUE)  
  
rioja::addClustZone(plot, clust, 4, col = "red")



## Conclusion

So, we’ve done a lot in this example. We’ve (1) searched for sites using site names and geographic parameters, (2) filtered results using temporal and spatial parameters, (3) obtained sample information for the selected datasets and (4) performed basic analysis including the use of climate data from rasters. Hopefully you can use these examples as templates for your own future work, or as a building block for something new and cool!

## Appendix Sections

### Installing packages on your own

We use several packages in this document, including leaflet, sf and others. We load the packages using the pacman package, which will automatically install the packages if they do not currently exist in your set of packages.

options(warn = -1)  
pacman::p\_load(neotoma2, dplyr, ggplot2, sf, geojsonsf, leaflet, terra, DT, readr, stringr, rioja)

Note that R is sensitive to the order in which packages are loaded. Using neotoma2:: tells R explicitly that you want to use the neotoma2 package to run a particular function. So, for a function like filter(), which exists in other packages such as dplyr, you may see an error that looks like:

Error in UseMethod("filter") :   
 no applicable method for 'filter' applied to an object of class "sites"

In that case it’s likely that the wrong package is trying to run filter(), and so explicitly adding dplyr:: or neotoma2:: in front of the function name (i.e., neotoma2::filter())is good practice.

### Piping in R

Piping is a technique that simplifies the process of chaining multiple operations on a data object. It involves using either of these operators: |> or %>%. |> is a base R operator while %>% comes from the tidyverse ecosystem in R. In neotoma2 we use %>%.

The pipe operator works as a real-life pipe, which carries water from one location to another. In programming, the output of the function on the left-hand side of the pipe is taken as the initial argument for the function on the right-hand side of the pipe. It helps by making code easier to write and read. Additionally, it reduces the number of intermediate objects created during data processing, which can make code more memory-efficient and faster.

Without using pipes you can use the neotoma2 R package to retrieve a site and then plot it by doing:

# Retrieve the site  
plot\_site <- neotoma2::get\_sites(sitename = "%ø%")  
# Plot the site  
neotoma2::plotLeaflet(object = plot\_site)

This would create a variable plot\_site that we will not need any more, but it was necessary so that we could pass it to the plotLeaflet function.

With the pipe (%>%) we do not need to create the variable, we can just rewrite our code. Notice that plotLeaflet() doesn’t need the object argument because the response of get\_sites(sitename = "%ø%") gets passed directly into the function.

#### 2.2.3.1. Code

# get\_sites and pipe. The `object` parameter for plotLeaflet will be the  
# result of the `get\_sites()` function.  
get\_sites(sitename = "%ø%") %>%  
 plotLeaflet()

#### 2.2.3.2. Result