4
Phylogenetic Analysis and Visualization

The comparison of sequences in order to infer evolutionary relationships is a fundamental technique of bioinformatics. It has a long history in R, too. There are many packages outside of Bioconductor for evolutionary analysis. In the recipes in this chapter, we will take a good look at how to work with tree formats from a variety of sources. A key focus will be how to manipulate trees to focus on particular parts and work with visualizations based on the new `ggplot`-based tree visualization packages, and the latter's usefulness in terms of viewing and annotating large trees.

The following recipes will be covered in this chapter:

- Reading and writing varied tree formats with ape and treeio
- Visualizing trees of many genes quickly with ggtree
- Quantifying distances between trees with treespace
- Extracting and working with subtrees using ape
- Creating dot plots for alignment visualization
- Reconstructing trees from alignments using phangorn

Technical requirements

The sample data you'll need is available from this book's GitHub repository at `https://github.com/danmaclean/R_Bioinformatics_Cookbook`. If you want to use the code examples as they are written, then you will need to make sure that this data is located in a subdirectory of whatever your working directory is.

Here are the R packages that you'll need. The majority of these will install with `install.packages()`; others are a little more complicated:

- `ape`
- `adegraphics`
- `Bioconductor`:
    - `Biostrings`
    - `ggtree`
    - `treeio`
    - `msa`
- `devtools`
- `dotplot`
- `ggplot2`
- `phangorn`
- `treespace`

`Bioconductor` is huge and has its own installation manager. You can install it with the following code:

```
if (!requireNamespace("BiocManager"))
    install.packages("BiocManager")
BiocManager::install()
```

Further information is available at `https://www.bioconductor.org/install/`.

Normally, in R, a user will load a library and use the functions directly by name. This is great in interactive sessions, but it can cause confusion when many packages are loaded. To clarify which package and function I'm using at a given moment, I will occasionally use the `packageName::functionName()` convention.

Sometimes, in the middle of a recipe, I'll interrupt the code so that you can see some intermediate output or the structure of an object that's important to understand. Whenever that happens, you'll see a code block where each line begins with `##` (double hash) symbols. Consider the following command:

```
letters[1:5]
```

This will give us the following output:

```
## a b c d e
```

Note that the output lines are prefixed with `##`.

Reading and writing varied tree formats with ape and treeio

Phylogenetic analysis is a cornerstone of biology and bioinformatics. The programs are diverse and complex, the computations are long-running, and the datasets are often large. Many programs are standalone and many have proprietary input and output formats. This has created a very complex ecosystem that we must navigate when dealing with phylogenetic data, meaning that, often, the simplest strategy is to use combinations of tools to load, convert, and save the results of analyses in order to be able to use them in different packages. In this recipe, we'll look at dealing with phylogenetic tree data in R. To date, R support for the wide range of tree formats is restricted, but a few core packages have sufficient standardized objects such that workflows can focus on a few types and conversion to those types is streamlined. We'll look at using the `ape` and `treeio` packages to get tree data into and out of R.

Getting ready

For this section, we'll need the tree and phylogenetic information in `datasets/ch4/` from the book's data repository, specifically the `mammal_tree.nwk` and `mammal_tree.nexus` files, which are Newick and Nexus format trees of a mammal phylogeny (you can see brief descriptions of these file types in this book's `Appendix`). We'll need `beast_mcc.tree`, which is a tree file from a run of BEAST, and `RAxML_bipartitionsBranchLabels.H3`, which is an RAxML output file. Both of these files are taken from the extensive data provided with the `treeio` package. We'll require the Bioconductor package, `treeio`, and the `ape` package.

How to do it...

Reading and writing tree formats with `ape` and `treeio` can be executed using the following steps:

1. Load the `ape` library and load in trees:

```
library(ape)
newick <-ape::read.tree(file.path(getwd(), "datasets", "ch4",
"mammal_tree.nwk"))
nexus <-ape::read.nexus(file.path(getwd(), "datasets", "ch4",
"mammal_tree.nexus"))
```

2. Load the `treeio` library and load in BEAST/RAxML output:

```
library(treeio)
beast <- read.beast(file.path(getwd(), "datasets", "ch4",
"beast_mcc.tree"))
raxml <- read.raxml(file.path(getwd(), "datasets", "ch4",
"RAxML_bipartitionsBranchLabels.H3"))
```

3. Check the object types that the different functions return:

```
class(newick)
class(nexus)

class(beast)
class(raxml)
```

4. Convert `tidytree` to `phylo`, and vice versa:

```
beast_phylo <- treeio::as.phylo(beast)
newick_tidytree <- treeio::as.treedata(newick)
```

5. Write output files using the following code:

```
treeio::write.beast(newick_tidytree,file = "mammal_tree.beast")
ape::write.nexus(beast_phylo, file = "beast_mcc.nexus")
```

How it works...

In *Step 1*, we make use of very straightforward loading functions from `ape`—we use the `read.tree()` and `read.nexus()` functions, which can read the generic format trees. In *Step 2*, we repeat this using the specific format functions from `treeio` for BEAST and RaXML output. *Step 3* simply confirms the object types that the function returns; note that `ape` gives `phylo` objects, while `treeio` gives `treedata` objects. The two are interconverted using `as.phylo()` and `as.treedata()` from `treeio` in *Step 4*. By converting in this way, we can get input in many formats into downstream analysis in R. Finally, we write the files in *Step 5*.

See also

The loading functions we used in *Step 2* are just a couple of those available. Refer to the `treeio` package vignettes for a comprehensive list.

Visualizing trees of many genes quickly with ggtree

Once you have computed a tree, the first thing you will want to do with it is take a look. That's possible in many programs, but R has an extremely powerful, flexible, and fast system in the form of the `ggtree` package. In this recipe, we'll learn how to get data into ggtree and re-layout, highlight, and annotate tree images in just a few commands.

Getting ready

You'll need the `ggplot2`, `ggtree`, and `ape` packages. You'll also require the `itol.nwk` file from the `datasets/ch4` folder of this book's repository, which is a Newick tree of 191 species from the *Interactive Tree of Life* online tool's public dataset.

How to do it...

Visualizing trees of many genes quickly with `ggtree` can be executed using the following steps:

1. Load the libraries and get a `phylo` object of the Newick tree:

```
library(ggplot2)
library(ggtree)
itol <-ape::read.tree(file.path(getwd(), "datasets", "ch4",
"itol.nwk"))
```

2. Make a basic tree plot:

```
ggtree(itol)
```

3. Make a circular plot:

```
ggtree(itol, layout = "circular")
```

4. Rotate and invert the tree:

```
ggtree(itol) + coord_flip() + scale_x_reverse()
```

5. Add labels to the tree tips:

```
ggtree(itol) + geom_tiplab( color = "blue", size = 2)
```

6. Make a strip of color to annotate a particular clade:

```
ggtree(itol, layout = "circular") + geom_strip(13,14, color="red",
barsize = 1)
```

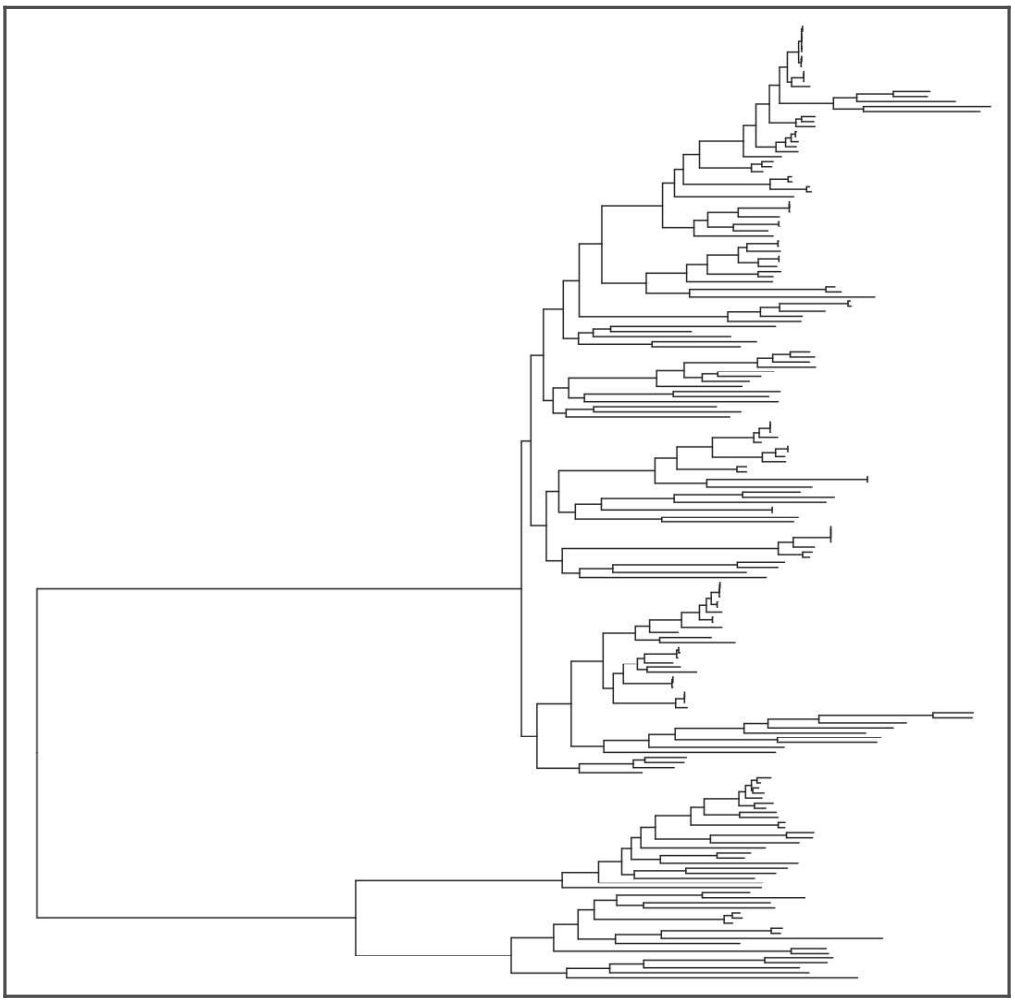
7. Make a blob of color to highlight a particular clade:

```
ggtree(itol, layout = "unrooted") + geom_hilight_encircle(node =
11, fill = "steelblue")
```

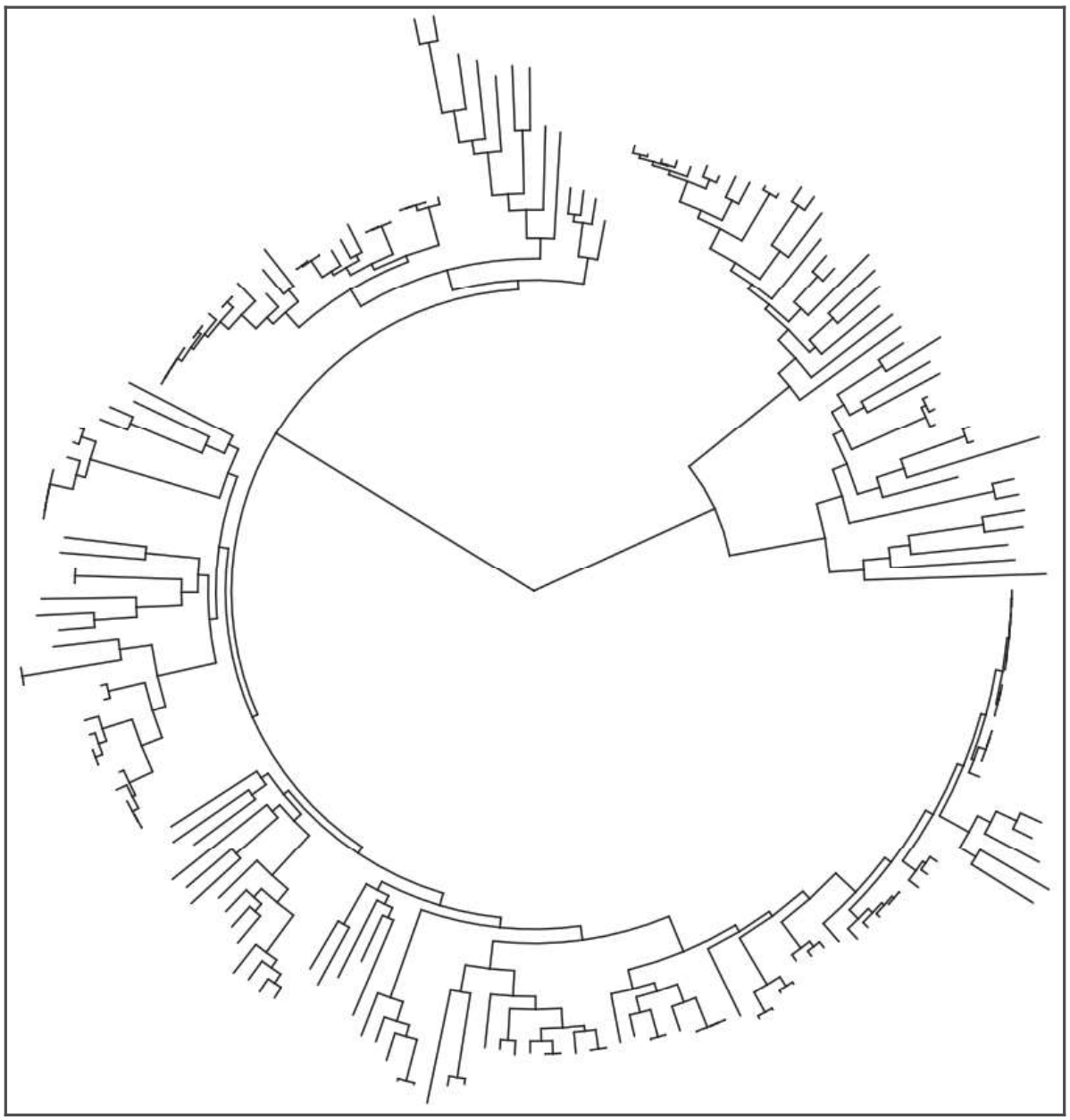
How it works...

This code achieves a lot very quickly. It can do this by virtue of its `ggplot`-like layer syntax. Here's what each step does and its output:

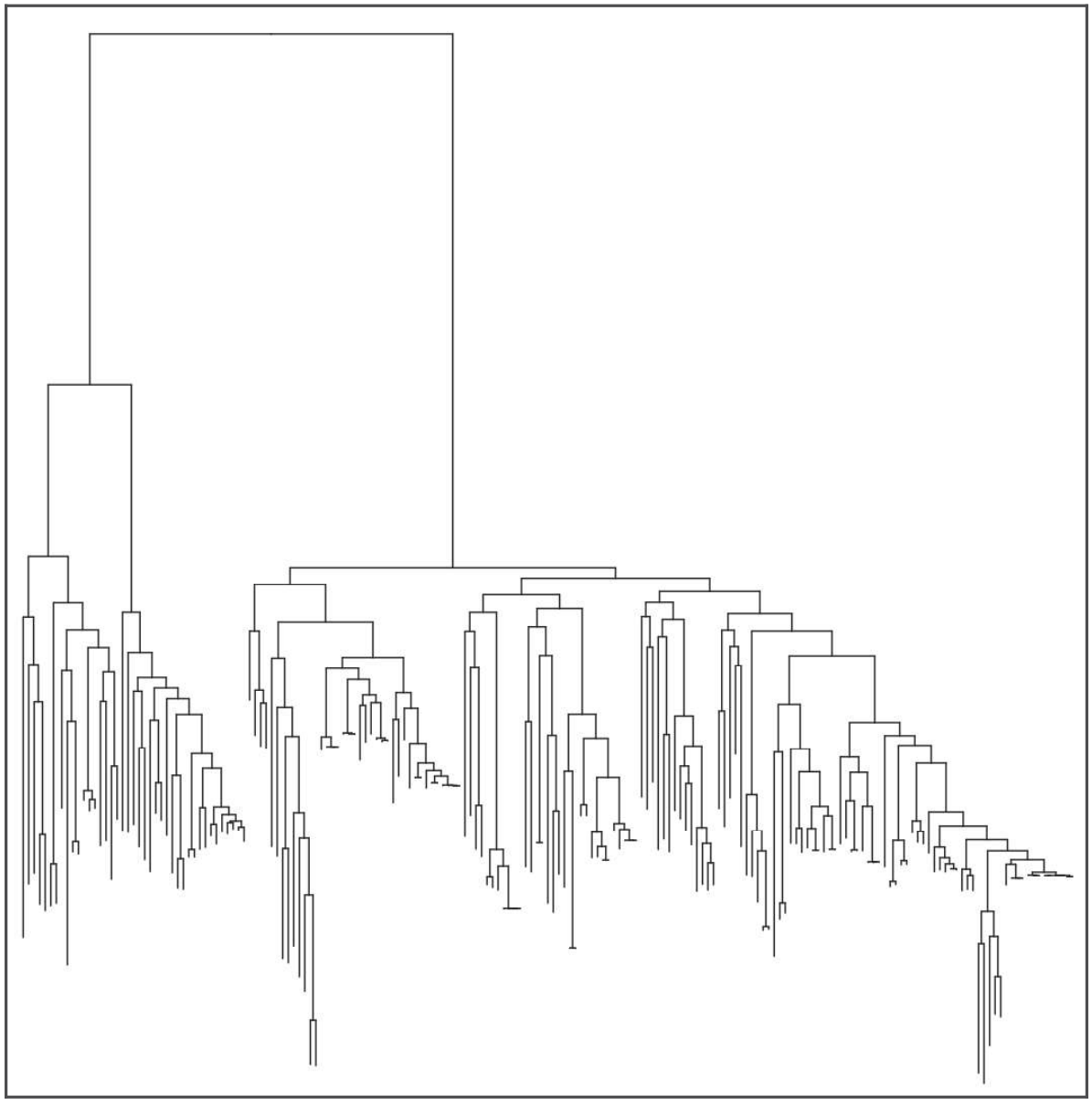
1. Load in a tree from a file. The tree here has 191 tips, so it's quite large. It happens to be in Newick format, so we use the `ape` `read.tree()` function. Note that we don't need to have a `treedata` object for `ggtree` in subsequent steps; the `phylo` object returned from `read.tree()` is perfectly acceptable to `ggtree()`.
2. Create a basic tree with `ggtree()`. This function is a wrapper for a longer `ggplot`-style syntax, specifically, `ggplot(itol) + aes(x,y) + geom_tree() + theme_tree()`. Hence, all the usual `ggplot` functions can be used as extra layers in the plot. The code in this step gives us the following plot:



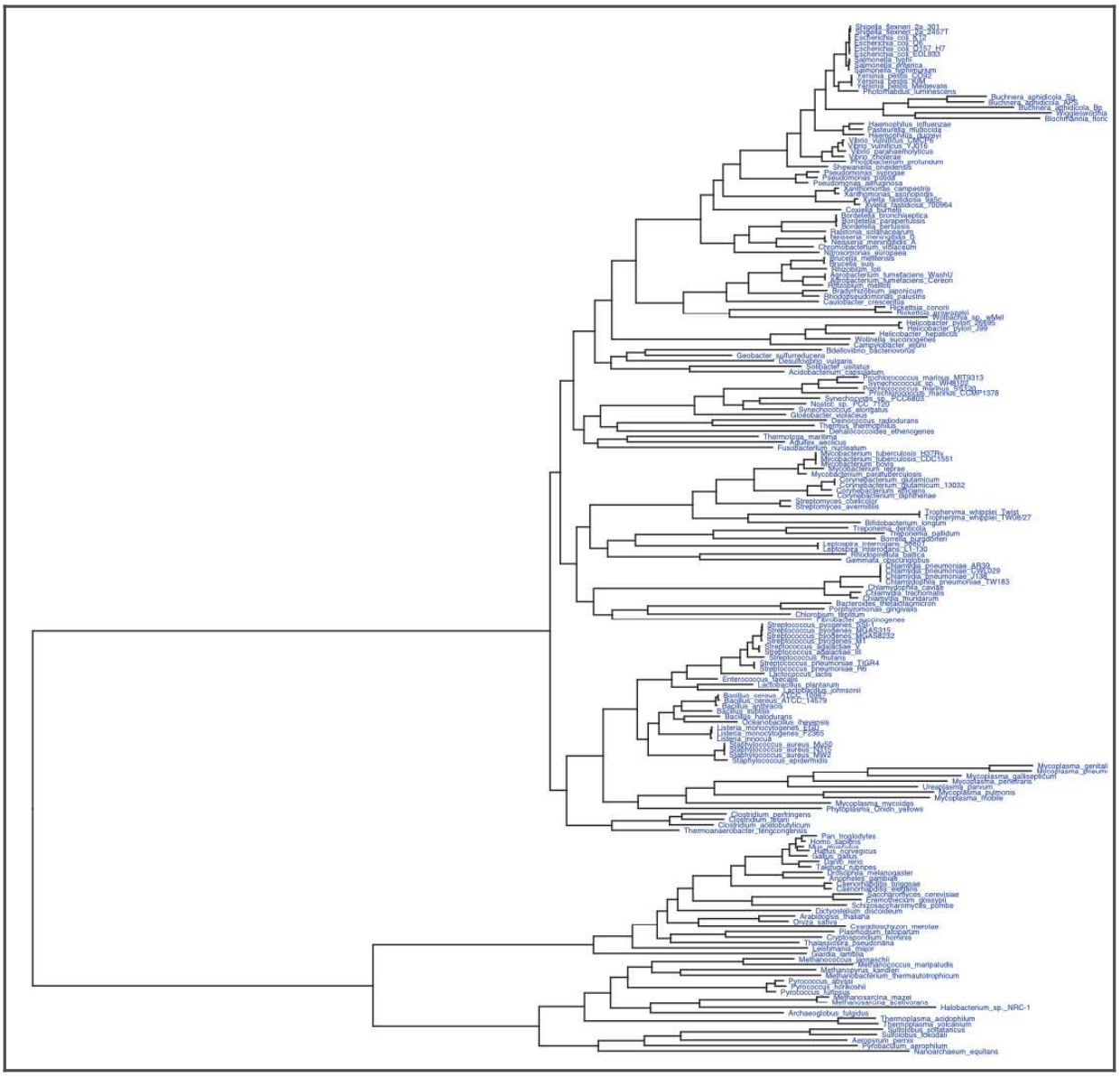
3. Change the layout of the plot. Setting the layout argument to circular gives us a round tree. There are many other tree types available through this argument:



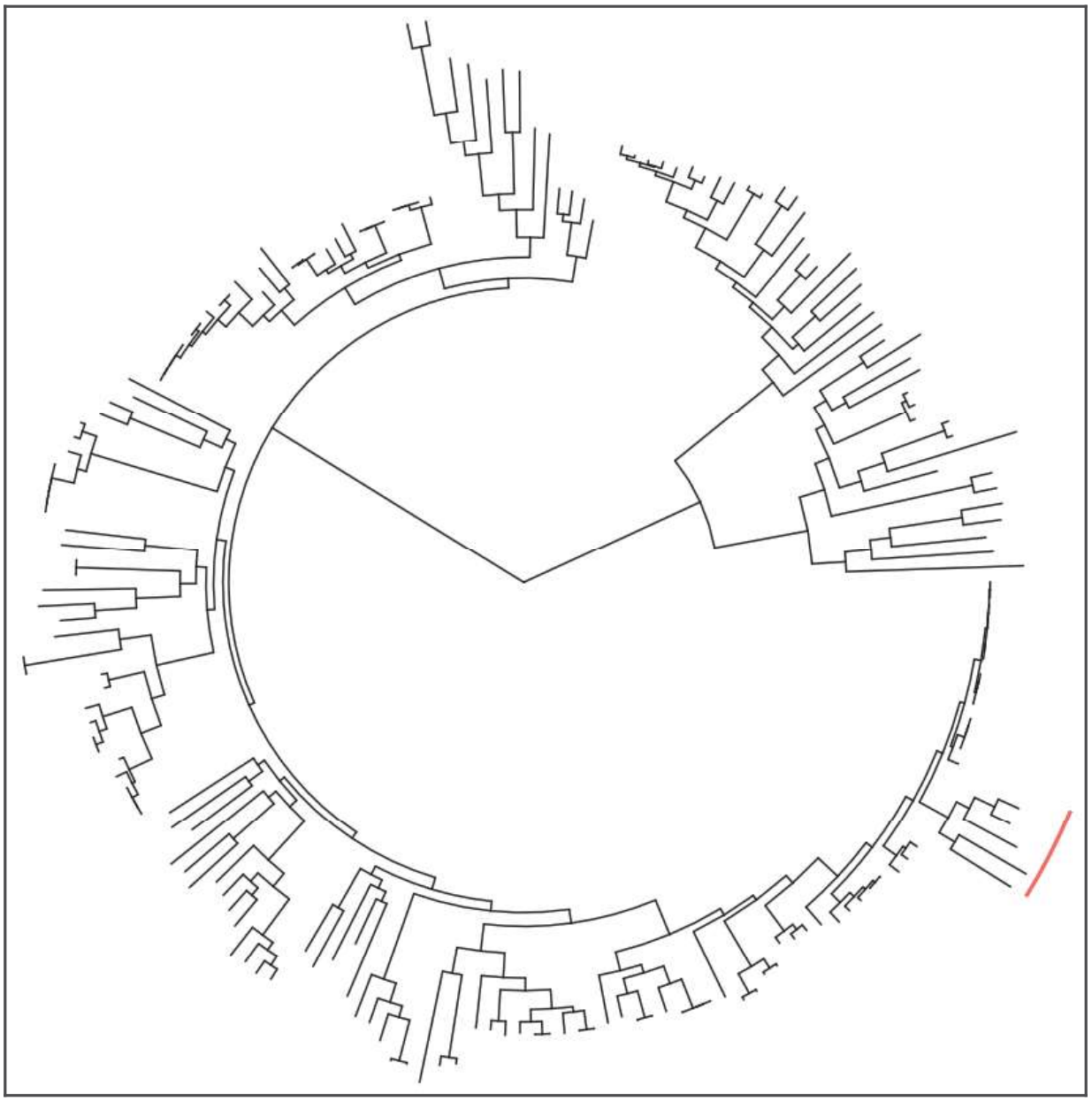
4. We can change the left-right direction of the tree to a top-bottom one using the standard `ggplot` functions, `coord_flip()` and `scale_x_reverse()`, to make the plot look like this:



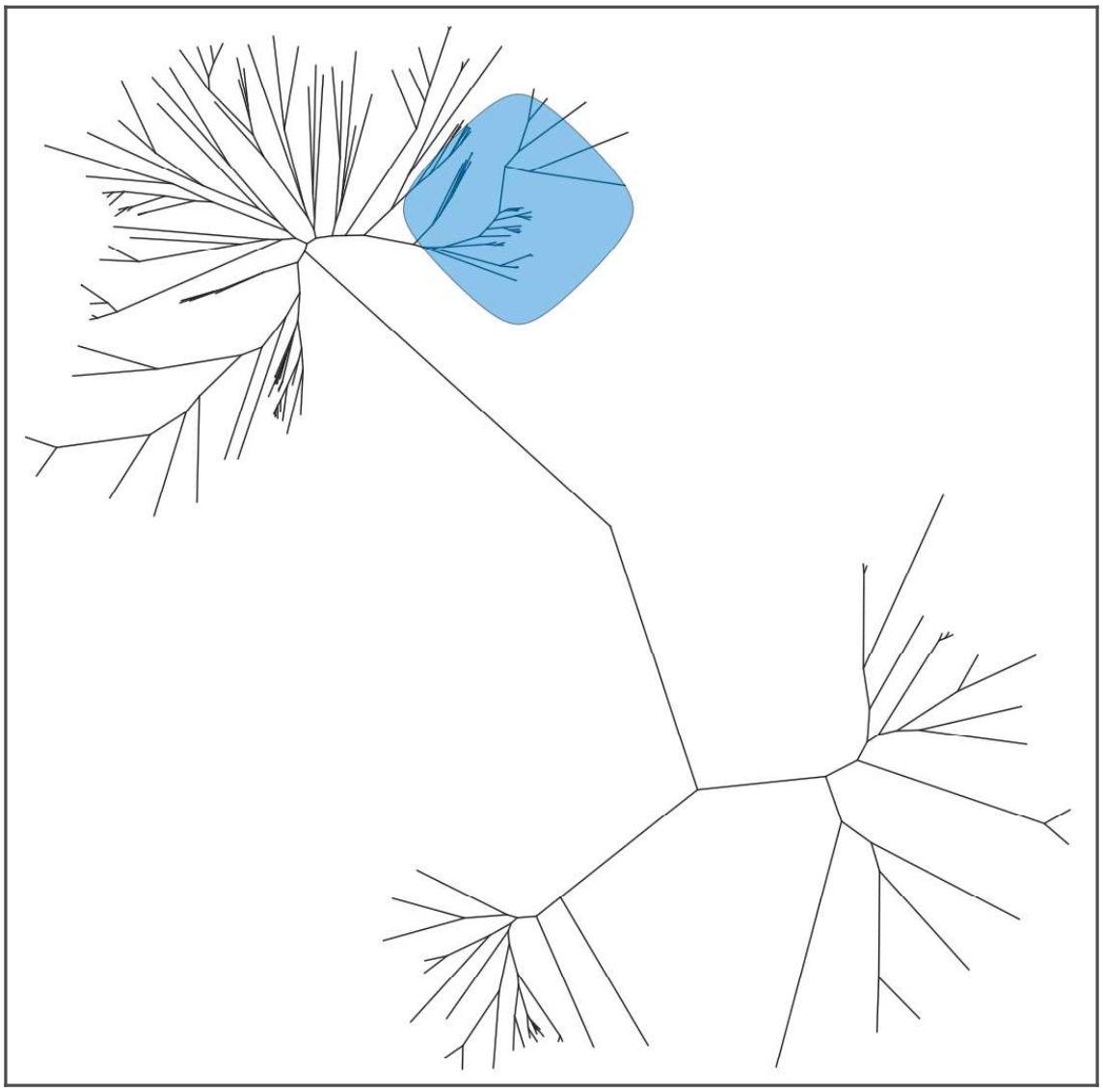
5. We can add names to the end of the tips with `geom_tiplab()`. The size argument sets the text size. This code generates the following output:



6. By adding a `geom_strip()` layer, we can annotate clades in the tree with a block of color. The first argument (`13` in this instance) is the start node in the tree, while the second argument is the end node in the tree for the strip of color. The `barsize` argument sets the width of the color block. The result looks like this:



7. We can highlight clades in unrooted trees with blobs of color using the `geom_hilight_encircle()` geom. We need to pick a value for the `node` argument, which tells `ggtree()` which node to center the color over. The code here provides the following output:



There's more...

Steps 6 and *7* here relied on us knowing which nodes in the tree to manipulate. This isn't always obvious as the nodes are identified by number and not name. We can get at the node number we want if we use the `MRCA()` (**Most Recent Common Ancestor**) function. Simply pass it a vector of node names and it returns the ID of the node that represents the MRCA:

```
MRCA(itol, tip=c("Photorhabdus_luminescens", "Blochmannia_floridanus"))
```

This will give the following output:

```
## 206
```

Quantifying differences between trees with treespace

Comparing trees to differentiate or group them can help researchers to see patterns of evolution. Multiple trees of a single gene tracked across species or strains can reveal differences in how that gene is changing across species. At the core of these approaches are metrics of distances between trees. In this recipe, we'll calculate one such metric to find pairwise differences between 20 different gene trees in 15 different species—hence, 15 different tips with identical names in each tree. Such similarity in trees is usually needed to compare and get distances, and we can't do an analysis like this unless these conditions are met.

Getting ready

For this recipe, we'll use the `treespace` package to compute distances and clusters. We'll use `ape` and `adegraphics` for accessory loading and visualization functions. The input data here will be all 20 files in `datasets/ch4/gene_trees`, each of which is a Newick-format tree for a single gene in each of 15 species.

How to do it...

Quantifying differences between trees with `treespace` can be executed using the following steps:

1. Load the libraries:

```
library(ape)
library(adegraphics)
library(treespace)
```

2. Load all the tree files into a `multiPhylo` object:

```
treefiles <- list.files(file.path(getwd(), "datasets", "ch4",
"gene_trees"), full.names = TRUE)
tree_list <- lapply(treefiles, read.tree)
class(tree_list) <- "multiPhylo"
```

3. Compute the Kendall-Colijn distances:

```
comparisons <- treespace(tree_list, nf = 3)
```

4. Plot pairwise distances:

```
adegraphics::table.image(comparisons$D, nclass=25)
```

5. Plot **principal component analysis** (**PCA**) and clusters:

```
plotGroves(comparisons$pco, lab.show=TRUE, lab.cex=1.5)
groves <- findGroves(comparisons, nclust = 4)
plotGroves(groves)
```

How it works...

The short and sweet code here is really powerful—and gives us a lot of analysis in a few commands.

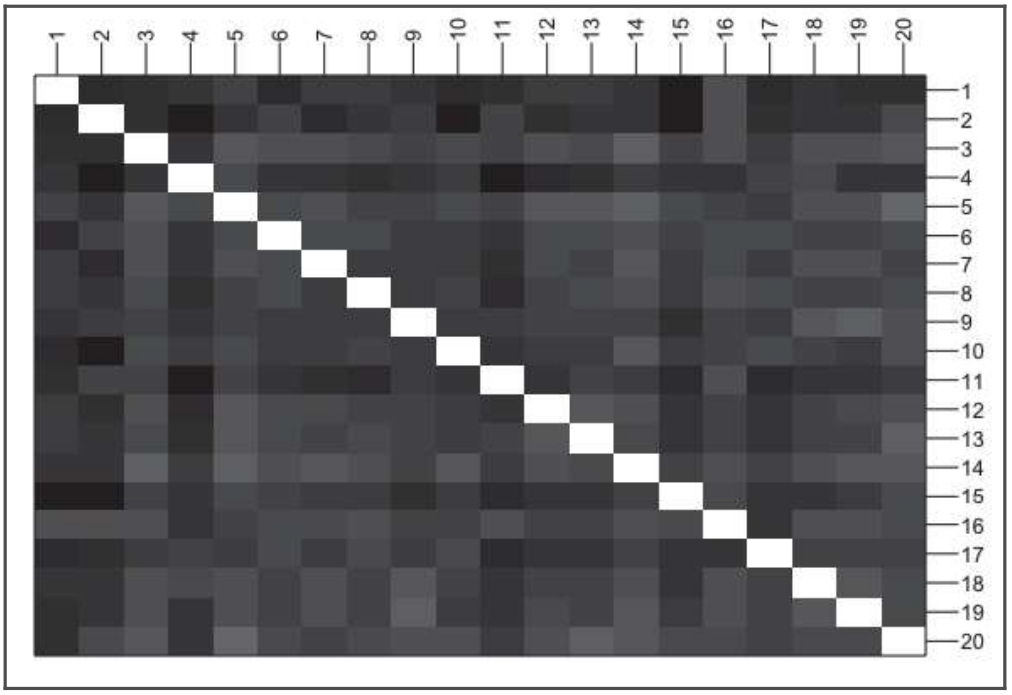
In *Step 1*, initially, we load the libraries we require.

In *Step 2*, after loading the necessary libraries, we make a character vector, `treefiles`, which holds paths to the 20 trees we wish to use. The `list.files()` function that we use takes a filesystem path as its argument and returns the names of files it finds in that path. As `treefiles` is a vector, we can use it as the first argument to `lapply()`.

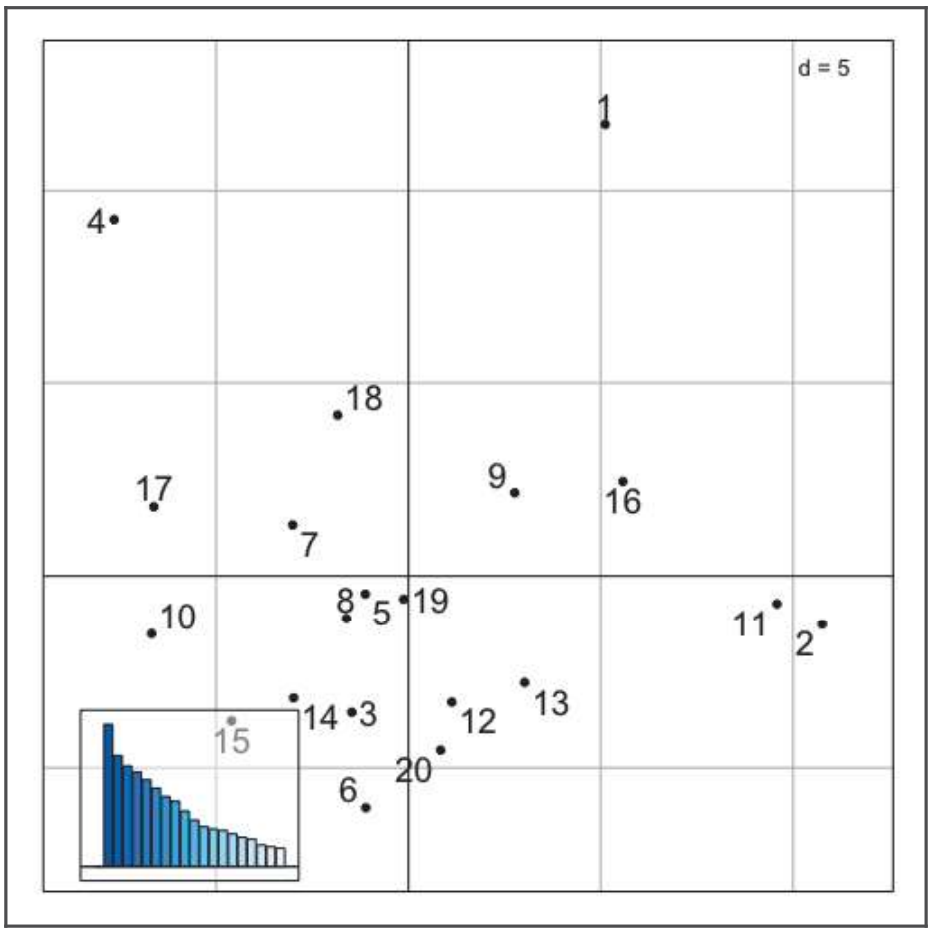
In case you're not familiar with it, `lapply()` is an iterator function that returns an R list (hence, `lapply()`). Simply put, `lapply()` runs the function named in the second argument over the list of things in the first. The current thing is passed as the target function's first argument. So, in *Step 2*, we run the **ape** `read.tree()` function on each file named in `treefiles` and receive a list of `phylo` tree objects in return. The final step is to ensure that the `tree_list` object has the class, `multiPhylo`, so that we satisfy the requirements of the downstream functions. Helpfully, a `multiPhylo` object is a list-like object anyway, so we can get away with adding the `multiPhylo` string to the class attribute with the `class()` function.

In *Step 3*, the `treespace()` function from the package of the same name does an awful lot of analysis. First, it runs pairwise comparisons of all trees in the input, and then it carries out clustering using PCA. These are returned in a list object, with a member *D* containing the pairwise distances for the trees, and `pco` containing the PCA. The default distance metric, the Kendall-Colijn distance, is particularly suitable for rooted gene trees as we have here, though the metric can be changed. The argument `nf` simply tells us how many of the principal components to retain. As our aim is plotting, we won't need more than three.

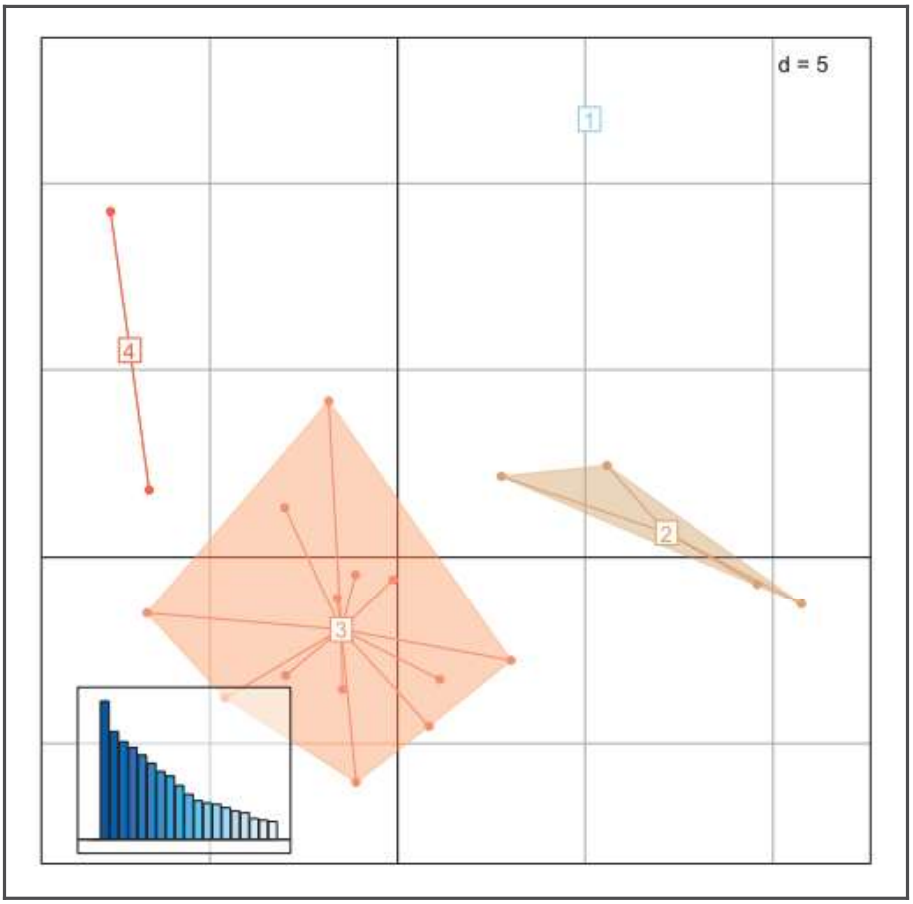
In *Step 4*, we plot the distance matrix in `comparisons$D` using the `table.image()` function in `adegraphics`—a convenient heatmap-style function. The `nclass` argument tells us how many levels of color to use. We get a plot as follows:



In *Step 5*, the `plotGroves()` function plots a `treespace` object directly, so we can see the plot of the PCA:



We can use the `findGroves()` function to group the trees into the number of groups given by the `nclust` argument and re-plot to view that:



There's more...

If you have many trees and the plot is crowded, you can create an interactive plot that can be zoomed and panned using the following code:

```
plotGrovesD3(comparisons$pco, treeNames=paste0("species_", 1:10) )
```

Extracting and working with subtrees using ape

In this short recipe, we'll look at how easy it can be to manipulate trees; specifically, how to pull out a subtree as a new object and how to combine trees into other trees.

Getting ready

We'll need a single example tree; the `mammal_tree.nwk` file in the `datasets/ch4` folder will be fine. All the functions we require can be found in the `ape` package.

How to do it...

Extracting and working with subtrees using `ape` can be executed using the following steps:

1. Load the `ape` library and then load the tree:

```
library(ape)
newick <-read.tree(file.path(getwd(), "datasets", "ch4",
"mammal_tree.nwk"))
```

2. Get a list of all of the subtrees:

```
l <- subtrees(newick)
plot(newick)
plot(l[[4]], sub = "Node 4")
```

3. Extract a specific subtree:

```
small_tree <- extract.clade(newick, 9)
```

4. Combine two trees:

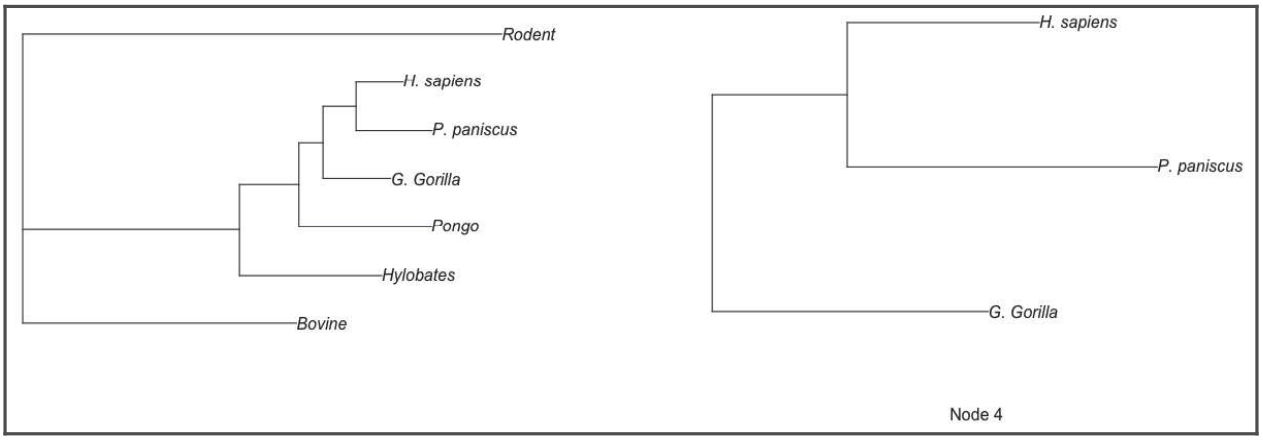
```
new_tree <- bind.tree(newick, small_tree, 3)
plot(new_tree)
```

How it works...

The functions in this recipe are really straightforward, but extremely useful.

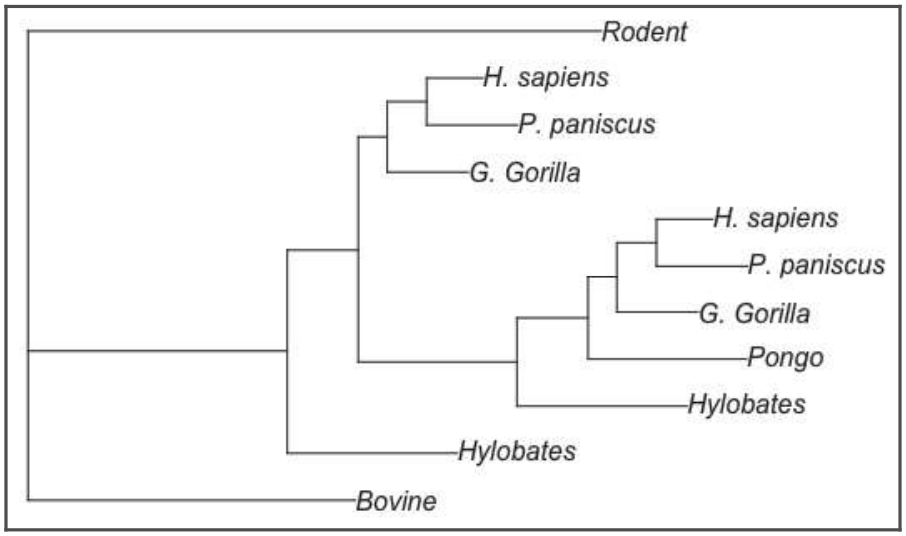
Step 1 is a familiar tree-loading step. We need a phylo object tree to progress.

Step 2 uses the `subtrees()` function, which extracts all non-trivial (greater than one node) subtrees and puts them in a list. The members of the list are numbered according to the node number in the original tree, and each object in the list is a `phylo` object, like the parent. We can inspect the original tree and the subtree at node 4 using the `plot()` function, which generates the following diagram:



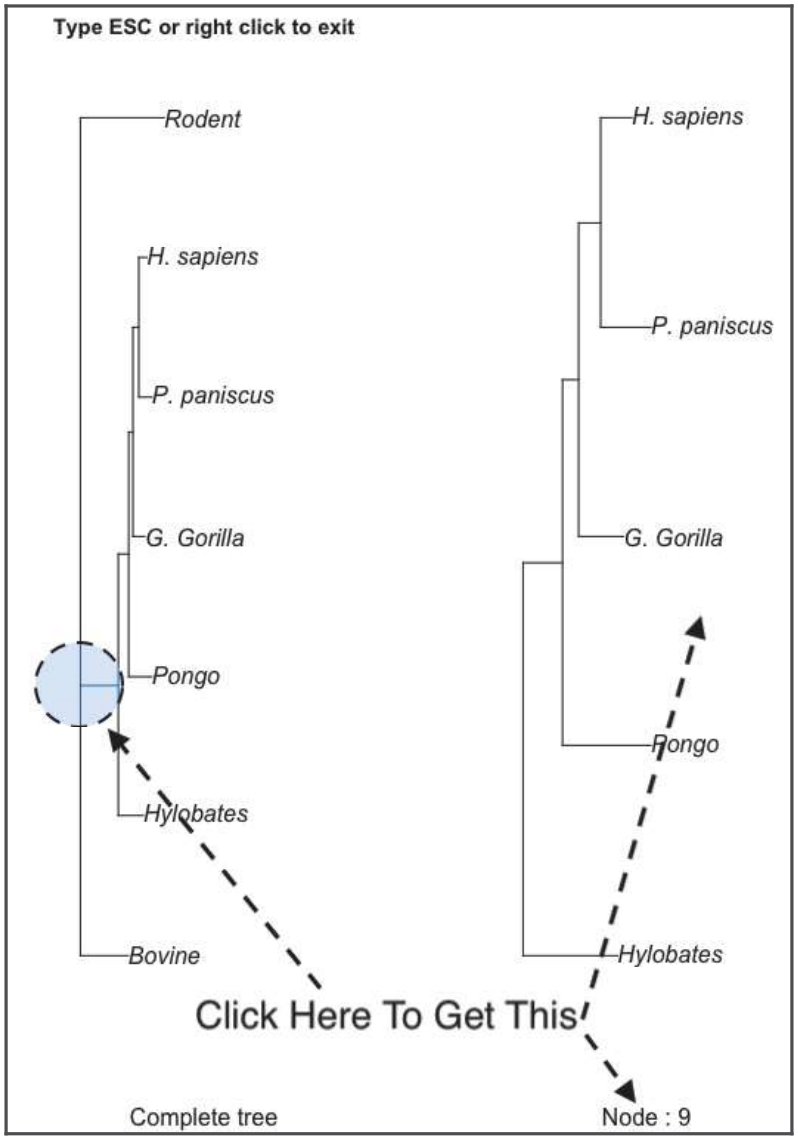
In *Step 3*, we get a single specific subtree using the `extract.clade()` function. The first argument to this function is the tree, while the second is the node that will be extracted. In fact, all nodes downstream of this node are taken and a new `phylo` object is returned.

The last example shows how to use the `bind.tree()` function to combine two `phylo` objects. The first argument is the major tree, which will receive the tree of the second argument. Here, we'll be stitching `small_tree` onto Newick. The third argument is the node in the major tree to which the second tree will be joined. Again, a new `phylo` object is returned. When we plot the new tree, we can see the repeated segment relative to our original tree:



There's more...

A minor problem with the preceding functions is that they expect us to know the node number we want to work with. A simple way to access this is by using the interactive `subtreeplot()` command. The `subtreeplot(newick)` code generates an interactive plot for the tree provided, like the one here. By clicking on particular nodes in the tree, we can get the viewer to render the subtree and print the node ID. We can then use the node ID in the functions:



Creating dot plots for alignment visualization

Dot plots of pairs of aligned sequences are probably the oldest alignment visualization. In these plots, the positions of two sequences are plotted on the *x* axis and *y* axis, and for every coordinate in that space, a point is drawn if the letters (nucleotides or amino acids) correspond at that `(x,y)` coordinate. Since the plot can show regions that match that aren't generally in the same region of the two sequences, this is a good way to visually spot insertions and deletions and structural rearrangements in the two sequences. In this recipe, we'll look at a speedy method for constructing a dot plot using the `dotplot` package and a bit of code for getting a grid plot of all pairwise dot plots for sequences in a file.

Getting ready

We'll need the `datasets/ch4/bhlh.fa` file, which contains three **basic helix-loop-helix** (**bHLH**) transcription factor sequences from pea, soy, and lotus. We'll also need the `dotplot` package, which isn't on CRAN or Bioconductor, so you'll need to install it from GitHub using the `devtools` package. The following code should work:

```
library(devtools)
install_github("evolvedmicrobe/dotplot", build_vignettes = FALSE)
```

How to do it...

Creating dot plots for alignment visualization can be executed using the following steps:

1. Load the libraries and sequences:

```
library(Biostrings)
library(ggplot2)
library(dotplot)
seqs <- readAAStringSet(file.path(getwd(), "datasets", "ch4",
"bhlh.fa"))
```

2. Make a basic dot plot:

```
dotPlotg(as.character(seqs[[1]]), as.character(seqs[[2]] ))
```

3. Change the dot plot and apply the `ggplot2` themes and labels:

```
dotPlotg(as.character(seqs[[1]]), as.character(seqs[[2]] ),
wsize=7, wstep=5, nmatch=4) +
theme_bw() +
labs(x=names(seqs)[1], y=names(seqs)[2] )
```

4. Make a function that will create a dot plot from sequences provided and the sequence index:

```
make_dot_plot <- function(i=1, j=1, seqs = NULL){
 seqi <- as.character(seqs[[i]])
 seqj <- as.character(seqs[[j]])
 namei <- names(seqs)[i]
 namej <- names(seqs)[j]
 return( dotPlotg(seqi, seqj ) + theme_bw() + labs(x=namei,
y=namej) )
}
```

5. Set up data structures to run the function:

```
combinations <- expand.grid(1:length(seqs),1:length(seqs))
plots <- vector("list", nrow(combinations) )
```

6. Run the function on all the possible combinations of pairs of sequences:

```
for (r in 1:nrow(combinations)){
 i <- combinations[r,]$Var1[[1]]
 j <- combinations[r,]$Var2[[1]]
 plots[[r]] <- make_dot_plot(i,j, seqs)
}
```

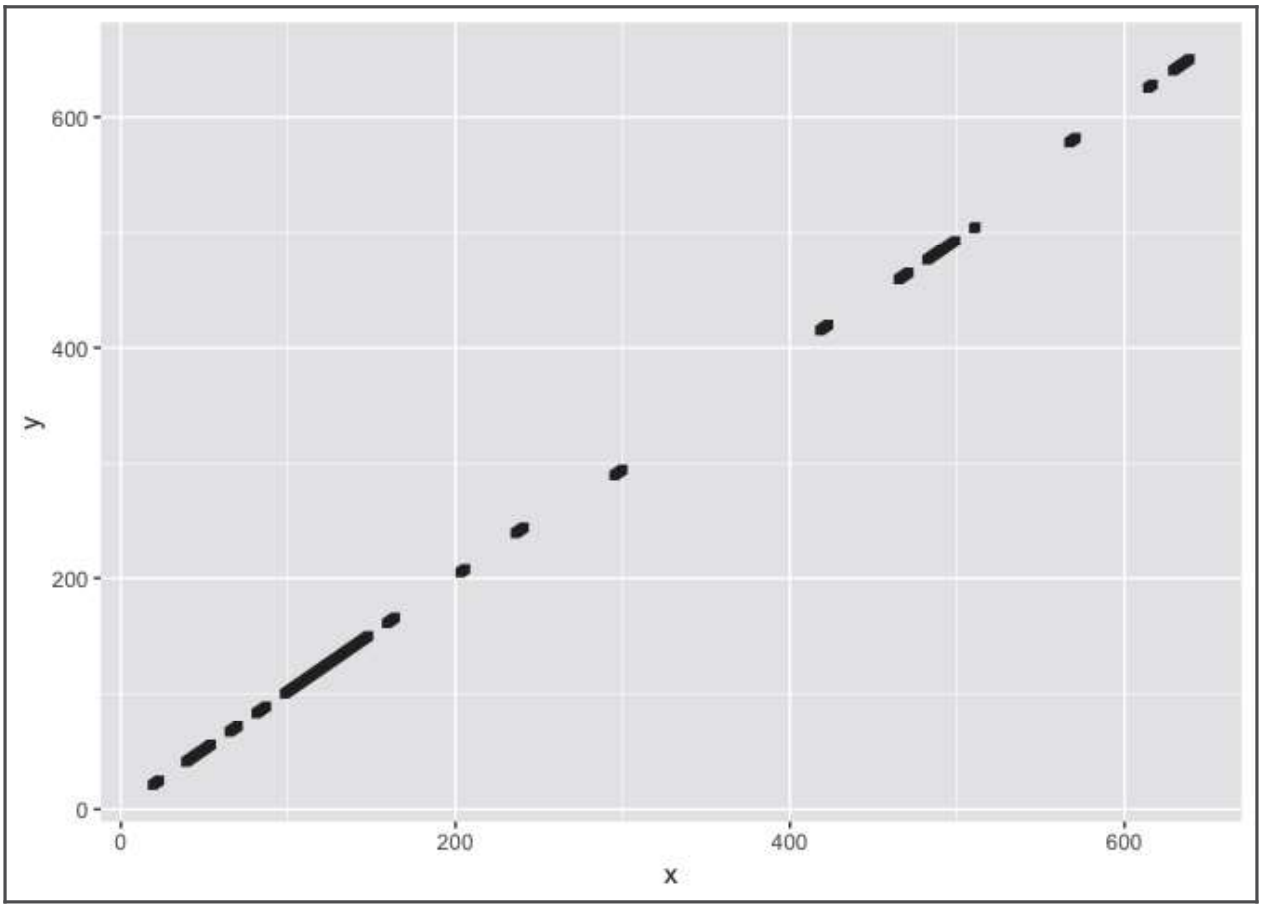
7. Plot the grid of plots:

```
cowplot::plot_grid(plotlist = plots)
```

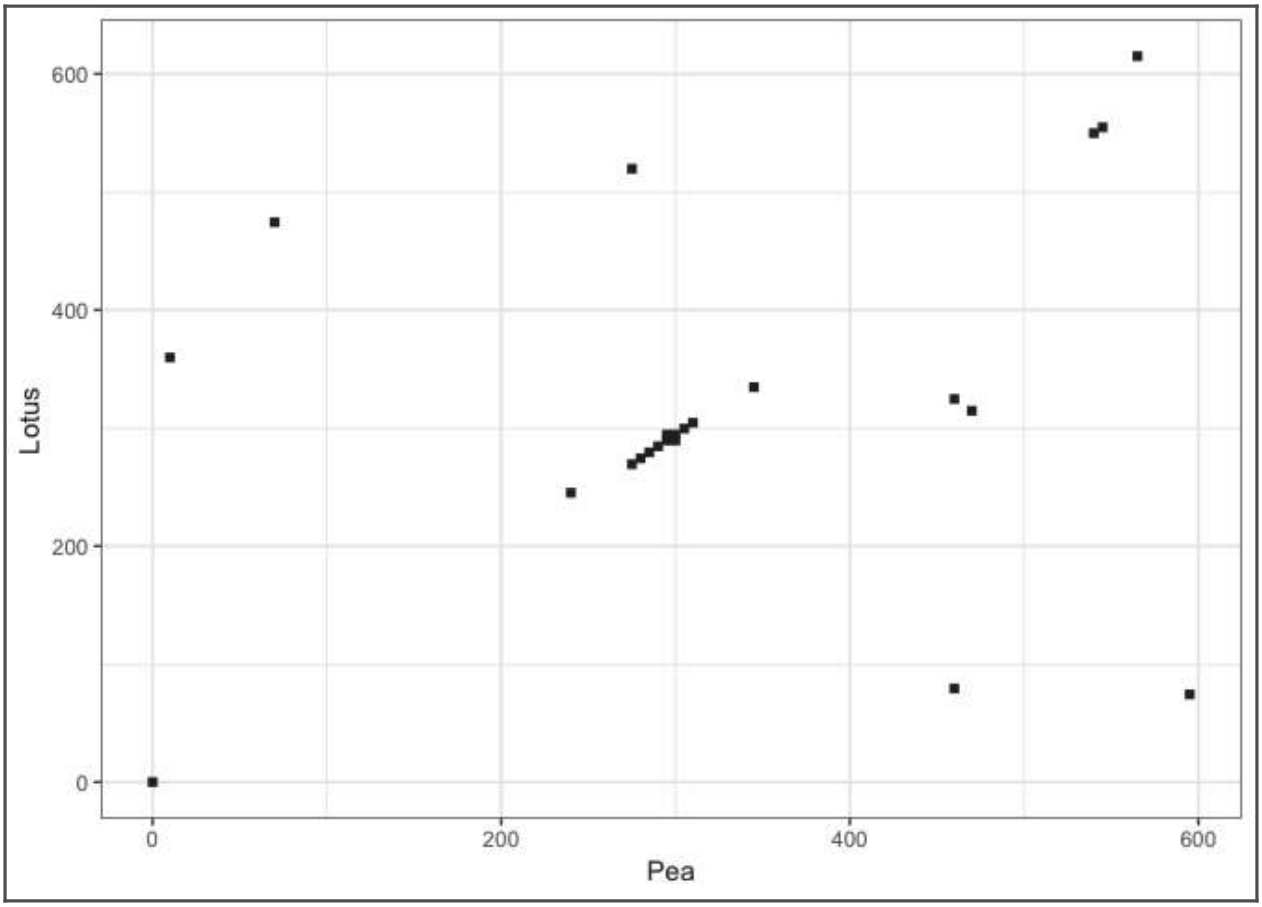
How it works...

The first part of this recipe is pretty familiar. We load in the libraries and use `Biostrings` to load in our protein sequences. Note that our sequences in the `seqs` variable are an instance of the `XStringSet` class.

In *Step 2*, we can create a basic dot plot using the `dotplotg()` function. The arguments are the sequences we want to plot. Note that we can't pass the `XStringSet` objects directly; we need to pass character vectors, so we coerce our sequences into that format with the `as.character()` function. Running this code gives us the following dot plot:



In *Step 3*, we elaborate on the basic dot plot by first changing the way a match is considered. With the `wsize=7` option, we state that we are looking at seven residues at a time (instead of the default of one), the `wstep=5` option tells the plotter to jump five residues each step (instead of one, again), and the `nmatch=4` option tells the plotter to mark a window as matching if four of the residues are identical. We then customize the plot by adding a `ggplot2` theme to it in the usual `ggplot` manner and add axis names with the label function. From this, we get the following dot plot. Note how it is different to the first one:

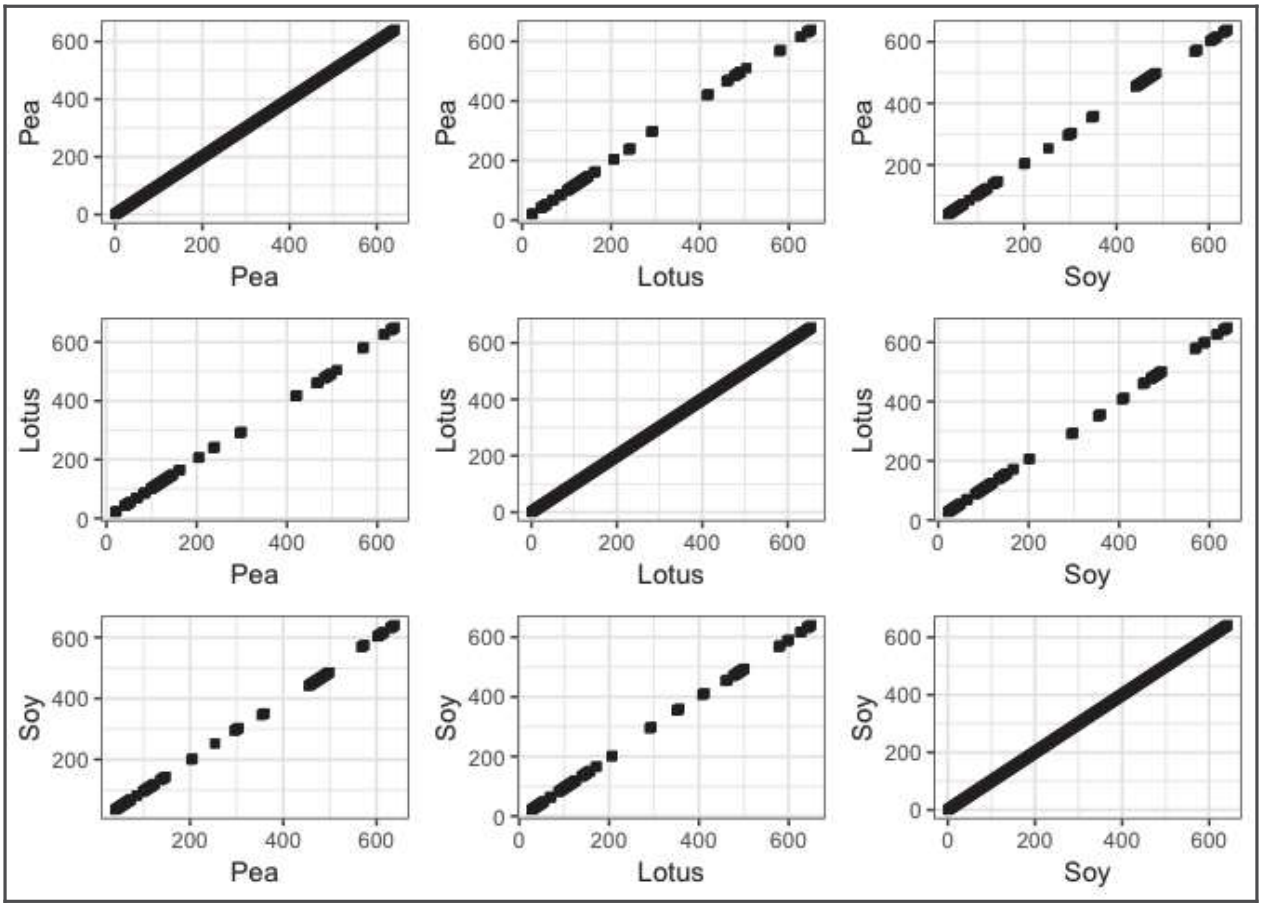


The custom function, `make_dot_plot()`, defined in *Step 4* takes two numbers in variables, `i` and `j`, and an `XStringSet` object in the `seqs` argument. It then converts the i-th and j-th sequence in the `seqs` object to characters and stores those in `seqi` and `seqj` variables. It also extracts the names of those sequences to `namei` and `namej`. Finally, it creates and returns a dot plot using the variables created

To use the function, we need two things; the combinations of sequences to be plotted and a list to hold the results in. In *Step 4*, the `expand.grid()` function is used to create a data frame of all possible combinations of sequences by number, which we store in the `combinations` variable. The `plots` variable, created with the `vector()` function, contains a `list` object with the right number of slots to hold the resultant dot plots.

Step 6 is a loop that iterates over each row of the combination's data frame, extracting the sequence numbers we wish to work with and storing them in the `i` and `j` variables. The `make_dot_plot()` function is then called with `i`, `j`, and `seqs`, and its results stored in the `plots` list we created.

Finally, in *Step 7*, we use the `cowplot` library function, `plot_grid()`, with our list of plots to make a master plot of all possible combinations that looks like this:



Reconstructing trees from alignments using phangorn

So far in this chapter, we've assumed that trees are already available and ready to use. Of course, there are many ways to make a phylogenetic tree and, in this recipe, we'll take a look at some of the different methods available.

Getting ready

For this chapter, we'll use the `datasets/ch4/` file, the `abc.fa` file of yeast ABC transporter sequences, the `Bioconductor Biostrings` package, and the `msa` and `phangorn` packages from CRAN.

How to do it...

Constructing trees using `phangorn` can be executed using the following steps:

1. Load in the libraries and sequences, and make an alignment:

```
library(Biostrings)
library(msa)
library(phangorn)

seqs <- readAAStringSet(file.path(getwd(), "datasets", "ch4",
"abc.fa"))
aln <- msa::msa(seqs, method=c("ClustalOmega"))
```

2. Convert the alignment to the `phyDat` object:

```
aln <- as.phyDat(aln, type = "AA")
```

3. Make UPGMA and neighbor-joining trees from a distance matrix:

```
dist_mat <- dist.ml(aln)

upgma_tree <- upgma(dist_mat)
plot(upgma_tree, main="UPGMA")

nj_tree <- NJ(dist_mat)
plot(nj_tree,"unrooted", main="NJ")
```

4. Calculate the bootstraps and plot:

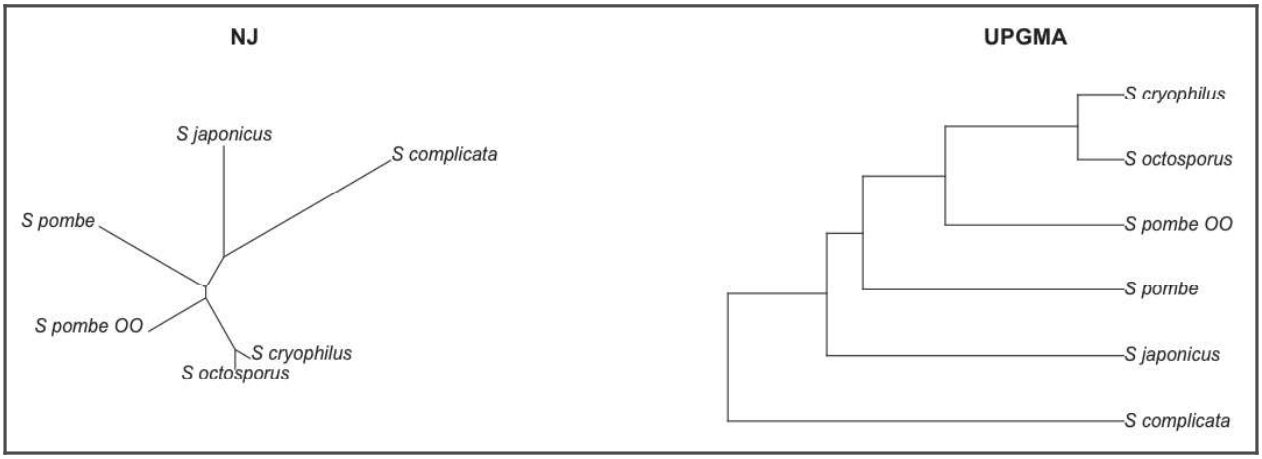
```
bootstraps <- bootstrap.phyDat(aln,FUN=function(x) { NJ(dist.ml(x))
} , bs=100)
plotBS(nj_tree, bootstraps, p = 10)
```

How it works...

The first step carries out a loading and amino acid sequence alignment, as we've seen in an earlier recipe with the `msa` package, returning an `MsaAAMultipleAlignment` object.

The second step uses the `as.phyDat()` function to convert the alignment to a `phyDat` object that can be used by the `phangorn` functions.

In *Step 3*, we actually make trees. Trees are made from a distance matrix, which we can compute with `dist.ml()` and our alignment (this is a maximum-likelihood distance measure; other functions can be used here if needed). The `dist_mat` is passed to the `upgma()` and `NJ()` functions to make UPGMA and neighbor-joining trees, respectively. These return standard `phylo` objects that can be worked with in many other functions. Here, we plot directly:



In the final step, we use the `bootstraps.phyDat()` function to compute bootstrap support for the branches in the tree. The first argument is the `phyDat` object, `aln`, while the second argument, `FUN`, requires a function to calculate trees. Here, we use an anonymous function wrapping the `NJ()` method we used to generate `nj_tree` in the first place. The `bs` argument tells the functions how many bootstraps to compute. Finally, we can plot the resultant bootstraps onto the tree using the `plotBS()` function.