Growing Phylogenetic Trees with Treeline

Erik S. Wright

February 18, 2025

Contents

1 Introduction

This document describes how to grow phylogenetic trees using the `Treeline` function in the DECIPHER package. `Treeline` takes as input a set of aligned nucleotide or amino acid sequences and returns a phylogenetic tree (i.e., *dendrogram* object) as output. This vignette focuses on optimizing balanced minimum evolution (ME), maximum likelihood (ML), and maximum parsimony (MP) phylogenetic trees starting from sequences.

Why is the function called `Treeline`? The goal of `Treeline` is to find the best tree according to an optimality criterion. There are often many trees near the optimum. Therefore, `Treeline` seeks to find a tree as close as possible to the treeline, analogous to how trees cannot grow above the treeline on a mountain.

Why use `Treeline` versus other programs? The `Treeline` function is designed to return an excellent phylogenetic tree with minimal user intervention. Many tree building programs have a large set of complex options for niche applications. In contrast, `Treeline` simply builds a great tree by default. `Treeline`'s unified optimization strategy also makes it easy to compare trees optimized under different optimality criteria. This vignette is intended to get you started and introduce additional options/functions that might be useful.

Treeline uses multi-start optimization followed by hill-climbing to find the highest trees on the optimality landscape. Since Treeline is a stochastic optimizer, it optimizes many trees to prevent chance from influencing the final result. With any luck it'll find the treeline!

2 Performance Considerations

Finding an optimal tree is no easy feat. `Treeline` systematically optimizes many candidate trees before returning the best one. This takes time, but there are things you can do to make it go faster.

- Only use the sequences you need: `Treeline`'s optimization runtime scales approximately quadratically with the number of sequences. Hence, limiting the number of sequences is a worthwhile consideration. In particular, always eliminate redundant sequences, as shown in the example below.
- Compile with OpenMP support: Significant speed-ups can be achieved with multi-threading using OpenMP, particularly for ML and MP *method*s. See the "Getting Started DECIPHERing" vignette for how to enable OpenMP on your computer. Then you will only need to set the argument `processors=NULL` and `Treeline` will use all available processors.
- Compile for SIMD support: `Treeline` is configured to make use of SIMD operations, which are available on most processors. The easiest way to enable SIMD is to add a line with "CFLAGS += -O3 -march=native" to your ∼/.R/Makevars text file. Then, after recompiling, there should be a speed-up on systems with SIMD support. Note that enabling SIMD makes the compiled code non-portable, so the code always needs to be compiled on the hardware being used.
- Set a timeout: The *maxTime* argument specifies the (approximate) maximum number of hours you are willing to let `Treeline` run. If you are concerned about the code running for too long then simply set this argument.
- Limit iterations: `Treeline` will converge after *minIterations* when the score is expected to change less than *tolerance* per iteration, unless *maxIterations* is met before convergence. A reasonable way to converge early is to set *minIterations* to a lower value (e.g., `20`). There is evidence supporting the notion that exhaustive searching is unlikely to result in a significantly more correct tree [8], even as the score continues to improve.
- For ML, choose a model: Automatic model selection is a useful feature, but frequently this time-consuming step can be skipped. For many nucleotide sequences the `"GTR+G4"` model will be automatically selected. Typical amino acid sequences will tend to pick the `"LG+G4"` or `"WAG+G4"` models, unless the sequences are from a particular origin (e.g., mitochondria). Pre-selecting a subset of the available `MODELS` and supplying this as the *model* argument can save time.

Accuracy is another performance consideration. `Treeline` is a stochastic optimizer, so it will continue searching the space of possible trees until convergence. It is possible to find the best tree on the first iteration, but most of the time additional iterations will yield a better scoring tree. If you are feeling unlucky, you can simply increase the number of iterations to ensure a good (scoring) tree is found. Increasing *minIterations* (e.g., to `100`) will largely remove luck from the equation. There is a decreasing marginal return to more iterations, and it's probably not worth searching (almost) endlessly for a slightly better tree. `Treeline`'s default settings are designed to balance runtime versus the reward of better scoring trees.

3 Preparing the Input Data

`Treeline` takes as input a multiple sequence alignment and/or a distance matrix. All distance-based methods (including ME) only require specification of `myDistMatrix` but will generate a distance matrix using `DistanceMatrix` if `myXStringSet` is provided instead. The character-based methods (i.e., ML and MP) require a multiple sequence alignment and will generate a distance matrix to construct the first candidate tree unless one is provided.

Multiple sequence alignments can be constructed from a set of (unaligned) sequences using `AlignSeqs` or related functions. `Treeline` will optimize trees for amino acid (i.e., `AAStringSet`) or nucleotide (i.e., `DNAStringSet` or `RNAStringSet`) sequences. For coding sequences, it is intuitive to assume that nucleotide data would better resolve close taxa, whereas amino acid data would be preferable to determine the branching order of deep taxa. However, recent work challenges this assumption by showing nucleotide data is adequate for determining distant relationships [7]. A good bet is to use nucleotide sequences with the `"ME"` *method*, possibly specifying a model (e.g., `"F81+F"` that corrects for multiple substitutions per site.

Here, we are going to use a set of sequences that is included with DECIPHER. These sequences are from the internal transcribed spacer (ITS) between the 16S and 23S ribosomal RNA genes in several *Streptomyces* species. To avoid letting the result come down to good old-fashioned luck, it is always best to compare multiple trees optimized for different objectives (ME, ML, and MP) and alternative models of evolution. `Treeline` is designed to facilitate this type of comparison, ideally across multiple loci.

```
> library(DECIPHER)
> # specify the path to your sequence file:
> fas <- "<<path to FASTA file>>"
> # OR find the example sequence file used in this tutorial:
> fas <- system.file("extdata", "Streptomyces_ITS_aligned.fas", package="DECIPHER")
> seqs <- readDNAStringSet(fas) # use readAAStringSet for amino acid sequences
> seqs # the aligned sequences
DNAStringSet object of length 88:
     width seq                                                names
 [1]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC supercont3.1 of S...
 [2]   627 NNNNCACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC supercont3.1 of S...
 [3]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC supercont1.1 of S...
 [4]   627 CGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC supercont1.1 of S...
 [5]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC supercont1.1 of S...
 ...   ... ...
[84]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC gi|297189896|ref|...
[85]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC gi|224581106|ref|...
[86]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC gi|224581106|ref|...
[87]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC gi|224581106|ref|...
[88]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC gi|224581108|ref|...
```

Many of these sequences are redundant or from the same genome. We can de-replicate the sequences to accelerate tree building and simplify analyses:

```
> seqs <- unique(seqs) # remove duplicated sequences
> ns <- gsub("^.*Streptomyces( subsp\\. | sp\\. | | sp_)([^ ]+).*$",
         "\\2",
         names(seqs))
> names(seqs) <- ns # name by species (or any other preferred names)
> seqs <- seqs[!duplicated(ns)] # remove redundant sequences from the same species
> seqs
DNAStringSet object of length 19:
     width seq                                                names
```

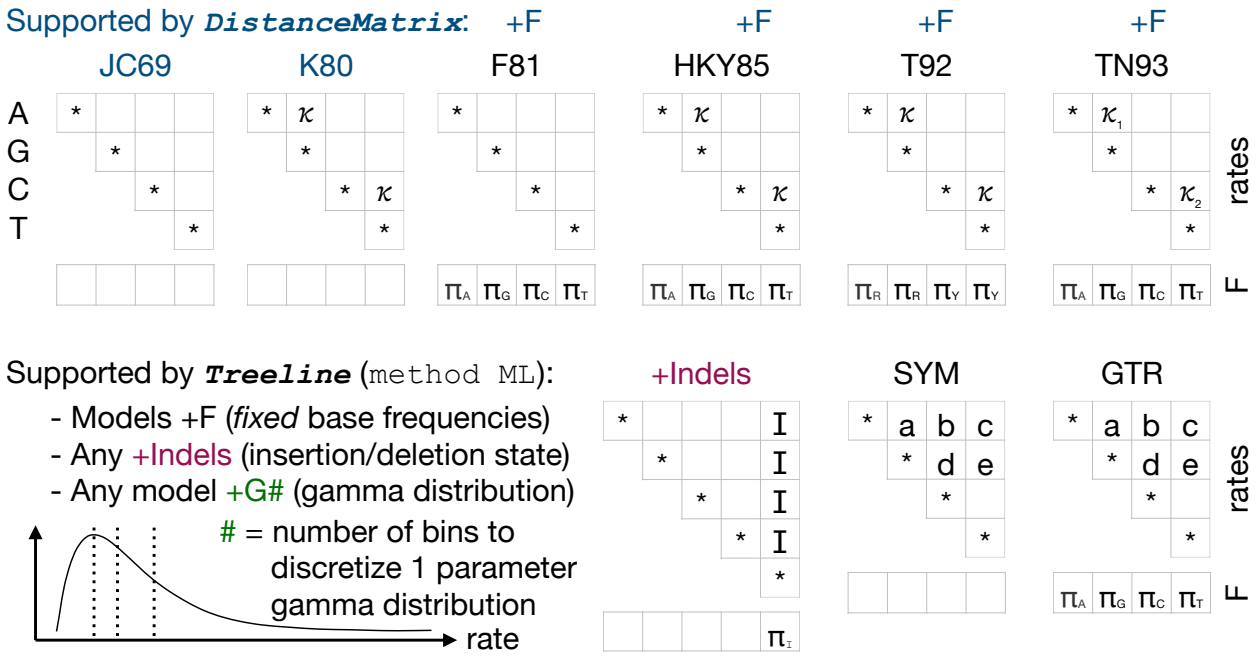

Figure 1: Free rates and frequencies in nucleotide models.

```
 [1]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC albus
 [2]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC clavuligerus
 [3]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC ghanaensis
 [4]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC griseoflavus
 [5]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC lividans
 ...   ... ...
[15]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC cattleya
[16]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC bingchenggensis
[17]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC avermitilis
[18]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC C
[19]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC Tu6071
```

4 Choosing a Method and Model of Evolution

Before choosing a `model` of evolution, it is necessary to choose a *method* for optimizing the tree. The default *method* is `"ME"` because it is fast and performs best on empirical datasets [4, 10]. The ME *method* accepts *myDistMatrix* as input, or *myXStringSet* can be given with or without a *model* to use with `DistanceMatrix` for building a distance matrix. For maximum likelihood, set *method* to `"ML"`, which requires a *model* of sequence evolution. For maximum parsimony, set *method* to `"MP"` and (optionally) specify a *costMatrix*.

`Treeline` supports many `MODELS` of evolution. In many cases, these `MODELS` can be extended by appending the model with `"+F"`, `"+G#"`, or `"+Indels"`. Here is the list of built-in `MODELS`:

```
> MODELS
$Nucleotide
[1] "JC69"   "K80"    "F81"    "HKY85" "T92"    "TN93"   "SYM"    "GTR"
```

```
$Protein
 [1] "AB"            "BLOSUM62"      "cpREV"         "cpREV64"
 [5] "Dayhoff"       "DCMut-Dayhoff" "DCMut-JTT"     "DEN"
 [9] "FLAVI"         "FLU"           "gcpREV"        "HIVb"
[13] "HIVw"          "JTT"           "LG"            "MtArt"
[17] "mtDeu"         "mtInv"         "mtMam"         "mtMet"
[21] "mtOrt"         "mtREV"         "mtVer"         "MtZoa"
[25] "PMB"           "Q.bird"        "Q.insect"      "Q.LG"
[29] "Q.mammal"      "Q.pfam"        "Q.plant"       "Q.yeast"
[33] "rtREV"         "stmtREV"       "VT"            "WAG"
[37] "WAGstar"
```

The nucleotide models each have different numbers of free parameters (Fig. 1). The `MODELS` with few free parameters are supported by `DistanceMatrix` and, therefore, *method* `"ME"`. This is because distance for few-parameter models can be analytically estimated from the sequences with relatively little error. High-parameter models, such as `"GTR"`, must be optimized and are only supported by `Treeline` *method* `"ML"`. All base built-in amino acid `MODELS` have no free parameters and are supported by `DistanceMatrix` and `Treeline`. See `?MODELS` for more information.

4.1 Minimum Evolution

Empirical benchmarks suggest ME with Hamming distance results in the most accurate trees, at least for alignments of single protein domains. Therefore, this is the default configuration when *myXStringSet* is supplied without `myDistMatrix`, which returns branch lengths in units of differences per site. If you would prefer to have branch lengths in units of substitutions per site, it is possible correct for multiple substitutions (e.g., *A* to *G* back to *A*) by setting *model* to any of the `MODELS` of evolution supported by `DistanceMatrix` (e.g., `"JC"` or `"F81+F"` for nucleotides, and `"WAG"` or `"WAG+F"` for amino acids). See Figure 1 for a list of models supported by `DistanceMatrix`. When *method* is `"ME"`, maximum control is gained by supplying *myDistMatrix*, which can be calculated with `DistanceMatrix` beforehand.

For example, a standard *model* to select for nucleotide sequences would be `"TN93+F"` and for amino acid sequences would be `"WAG"`. These models return trees with branch lengths in units of substitutions per site.

4.2 Maximum Likelihood

For ML trees, `Treeline` will automatically select an appropriate *model* according to Akaike information criterion (by default). It is possible to choose specific model(s) (e.g., `model="GTR+G4"`) to limit the possible selections and test your luck with fewer options. There is evidence that the choice of nucleotide model does not substantially alter tree accuracy [1,5,9], and picking the most complex model every time is a reasonable decision. All *model*s can be used with fixed (empirical) letter frequencies (i.e., by appending with `+F`) and/or gamma rate variation across sites (e.g., `+G4`). Note `Treeline` supports two discretizations of the gamma distribution: the default of equal binning, or the Laguerre quadrature if *quadrature* is set to `TRUE`. The former will give likelihoods comparable with other programs, but the latter is more accurate at representing the gamma distribution with limited bins.

For example, a standard *model* to select for nucleotide sequences would be `"GTR+G4+F"` and for amino acid sequences would be `"WAG+G4"`, with *quadrature* set to `TRUE` in both cases. These models return trees with branch lengths in units of substitutions per site.

4.3 Maximum Parsimony

For MP trees, the best results are typically obtained by providing a *costMatrix* rather than relying on the default binary costs. The choice of *costMatrix* is up to you, and several rational options are provided in the examples section of

the Treeline manual page (see ?Treeline). A more systematic approach to deriving a substitution matrix is provided as an example below.

4.4 Treatment of gaps

The standard models of evolution described above all ignore gap ("-" and ".") characters representing insertions or deletions (indels). But you're in luck — Treeline has the ability to incorporate gaps into all *method*s. For ME trees, DistanceMatrix allows gaps to be penalized in Hamming distance or added to any distance corrected from multiple substitutions per site. You can either specify a model with "+Indels" in Treeline, or supply *myDistMatrix* after setting *penalizeGapLetterMatches* to TRUE or NA (see ?DistanceMatrix). For ML trees, gaps can be added to any model of evolution as an additional state by specifying a model "+Indels", which adds two free parameters (Fig. 1). Incorporating gaps results in branch lengths in units of *changes* per site, since both substitutions and indels contribute to distance. For MP trees, gaps can be added as a character to the *costMatrix*. As luck would have it, incorporating gaps tends to result in *slightly* better trees on empirical datasets, although the average improvement is typically very small.

5 Minimum Evolution Phylogenetic Trees

Now, it's time to try our luck at finding the most likely tree. We will use the default settings, which returns a minimum evolution tree based on a Hamming distance matrix. Simply specify a *model* to correct for multiple substitutions (e.g., "TN93+F" or "WAG").

Since Treeline is a stochastic optimizer, it is critical to always set the random number seed for reproducibility. This will result in the same sequence of random numbers every time and, therefore, reproductibility. You can pick any lucky number, and if you ever wonder how much you pushed your luck, you can try running again from a different random number seed to see how much the result came down to luck of the draw. Note that setting a time limit, as done below with *maxTime*, negates the purpose of setting a seed – never set a time limit if reproducibility is desired or you'll have no such luck.

```
> set.seed(123) # set the random number seed
> treeME <- Treeline(seqs, verbose=FALSE, processors=1)
> set.seed(NULL) # reset the seed
```

Treeline returns an object of class *dendrogram* that stores the tree in a nested list structure. We can take an initial look at the tree and its attributes.

```
> treeME
'dendrogram' with 2 branches and 19 members total, at height 0.1533688
> attributes(treeME)
$members
[1] 19

$height
[1] 0.1533688

$class
[1] "dendrogram"

$method
[1] "ME"

$score
```

```
[1] 1.152586

$midpoint
[1] 11.03906
> str(treeME, max.level=4)
--[dendrogram w/ 2 branches and 19 members at h = 0.153]
  |--[dendrogram w/ 2 branches and 18 members at h = 0.11]
  |  |--leaf "cattleya" (h= 0.0376  )
  |  `--[dendrogram w/ 2 branches and 17 members at h = 0.0931]
  |     |--[dendrogram w/ 2 branches and 7 members at h = 0.087]
  |     |  |--[dendrogram w/ 2 branches and 5 members at h = 0.0787] ..
  |     |  `--[dendrogram w/ 2 branches and 2 members at h = 0.0686] ..
  |     `--[dendrogram w/ 2 branches and 10 members at h = 0.0849]
  |        |--[dendrogram w/ 2 branches and 2 members at h = 0.0779] ..
  |        `--[dendrogram w/ 2 branches and 8 members at h = 0.0764] ..
  `--leaf "AA4" (h= 2.78e-17  )
```

6 Maximum Likelihood Phylogenetic Trees

For the next example, we will grow a maximum likelihood phylogenetic tree, which is the most computationally demanding optimization objective that is supported. We will set a stringent time limit (0.01 hours) to make this example faster, although longer time limits (e.g., 24 hours) are advised because setting very short time limits leaves the result partly up to luck.

```
> set.seed(123) # set the random number seed
> tree <- Treeline(seqs,
         method="ML",
         model="GTR+G4",
         maxTime=0.01,
         verbose=FALSE,
         processors=1)
> set.seed(NULL) # reset the seed
> plot(tree)
```

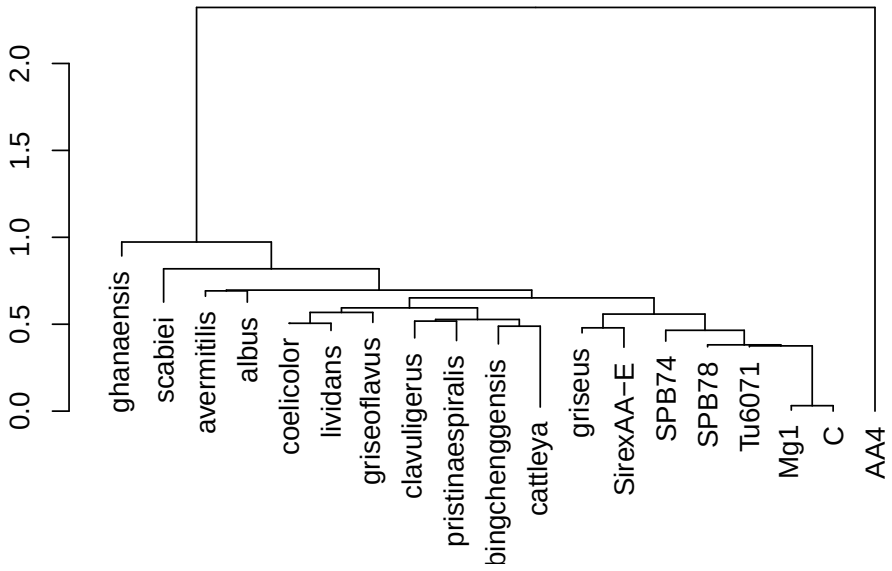

Figure 2: ML tree showing the relationships between *Streptomyces* species.

6.1 Plotting Branch Support Values

Maybe it was just beginner's luck, but we already have a reasonable looking starting tree! `Treeline` automatically returns a variety of information about the tree that can be accessed with the `attributes` and `attr` functions:

```
> attr(tree, "members") # number of leaves below this (root) node
[1] 19
> attr(tree, "height") # height of the node (in this case, the midpoint root)
[1] 2.32241
> attr(tree, "score") # best score (in this case, the -LnL)
[1] 4362.241
> attr(tree, "model") # either the specified or automatically select transition model
[1] "GTR+G4"
> attr(tree, "parameters") # the free model parameters (or NA if unoptimized)
    FreqA     FreqC     FreqG     FreqT     FreqI       A/G       C/T       A/C
0.1766174 0.2431846 0.3454590        NA        NA 3.2884697 2.9229539 0.6987279
      A/T       C/G    Indels     alpha
1.0916873 0.5799400        NA 0.1910815
> attr(tree, "midpoint") # center of the edge (for plotting)
[1] 9.893555
```

The tree is (virtually) rooted at its midpoint by default. For maximum likelihood trees, all internal nodes include aBayes branch support values [2]. These are given as probabilities that can be used in plotting on top of each edge. We can also italicize the leaf labels (species names) and add a scale bar.

```
> plot(dendrapply(tree,
        function(x) {
                s <- attr(x, "probability") # choose "probability" (aBayes)
                if (!is.null(s) && !is.na(s)) {
                        s <- formatC(as.numeric(s), digits=2, format="f")
                        attr(x, "edgetext") <- paste(s, "\n")
                }
                attr(x, "edgePar") <- list(p.col=NA, p.border=NA, t.col="#CC55AA", t.ce
                if (is.leaf(x))
                        attr(x, "nodePar") <- list(lab.font=3, pch=NA)
                x
        }),
        horiz=TRUE,
        yaxt='n')
> # add a scale bar (placed manually)
> arrows(2, 0, 2.4, 0, code=3, angle=90, len=0.05, xpd=TRUE)
> text(2.2, 0, "0.4 subs./site", pos=3, xpd=TRUE)
```

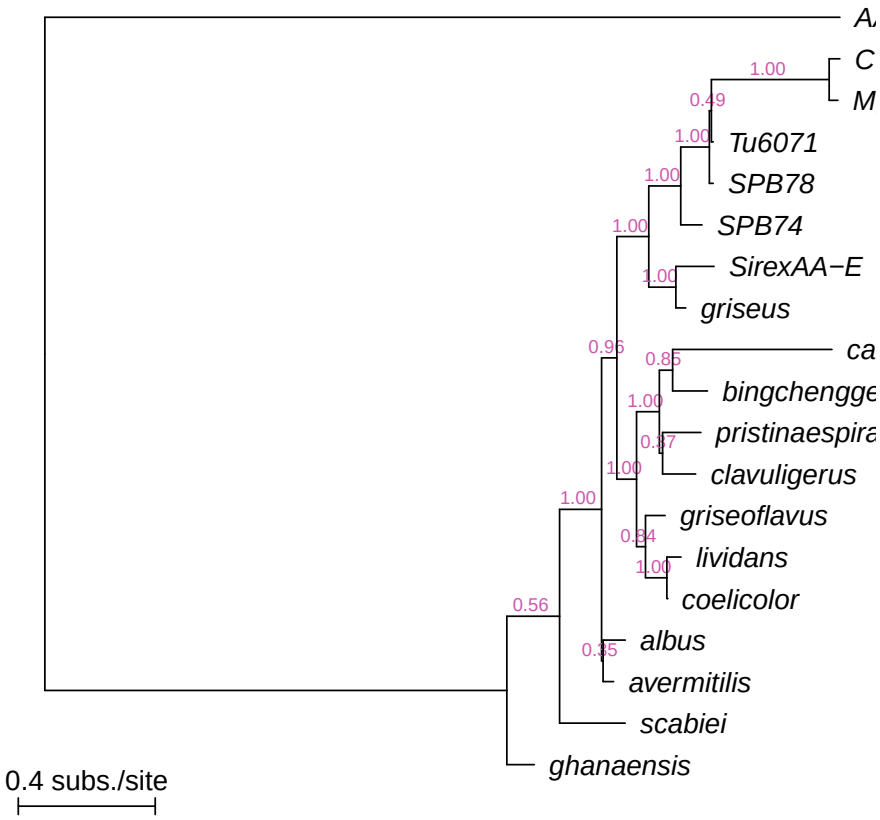

Figure 3: ML tree with aBayes probabilities at each internal node.

7 Maximum Parsimony Phylogenetic Trees

While ME and ML trees are based on models of evolution, MP relies on a cost matrix giving the penalty for switching characters along a branch. The default *costMatrix* is binary, which is biologically implausible and may invite bad luck. Hence, we will construct a binary tree and use the result to infer a more appropriate *costMatrix*.

```
> set.seed(123) # set the random number seed
> tree_UniformCosts <- Treeline(seqs,
          method="MP",
          reconstruct=TRUE,
          verbose=FALSE,
          processors=1)
> set.seed(NULL) # reset the seed
```

Since we set *reconstruct* to `TRUE`, `Treeline` output the state transition matrix as an attribute of the tree. We will use this to make our own luck by deriving a more biologically plausible *costMatrix*. It is apparent that transitions are more frequent than transversions and, therefore, are presumably less costly.

```
> mat <- attr(tree_UniformCosts, "transitions")
> mat # count of state transitions
    A  C   G   T
A   0 49 107  55
C  26  0  69 151
G 107 63   0  80
T  43 81  51   0
> mat <- mat + t(mat) # make symmetric
> mat <- mat/(sum(mat)/2) # normalize
> mat <- -log2(mat) # convert to bits
> diag(mat) <- 0 # reset diagonal
> mat # a derived cost matrix
          A        C        G        T
A 0.000000 3.555816 2.043168 3.169925
C 3.555816 0.000000 2.740241 1.926654
G 2.043168 2.740241 0.000000 2.751212
T 3.169925 1.926654 2.751212 0.000000
```

Now we can compare the two trees to see whether specifying a non-uniform cost matrix made a difference. We will highlight different partitions between the trees with dashed edges. Ideally the two tree topologies would be identical, implying the tree is robust to the specification of the cost matrix. The fact that this isn't the case suggests the cost matrix has a substantial influence over the tree, as might be expected. Note the scale of the two trees is different, because branch lengths are in units of average cost (per site) according to each *costMatrix*.

```
> set.seed(123) # set the random number seed
> tree_NonUniformCosts <- Treeline(seqs,
        method="MP",
        costMatrix=mat,
        reconstruct=TRUE,
        verbose=FALSE,
        processors=1)
> set.seed(NULL) # reset the seed
> splits <- function(x) {
        y <- sapply(x, function(x) paste(sort(unlist(x)), collapse=" "))
        if (!is.leaf(x))
                y <- c(y, splits(x[[1]]), splits(x[[2]]))
        y
 }
> splits_UniformCosts <- splits(tree_UniformCosts)
> splits_NonUniformCosts <- splits(tree_NonUniformCosts)
> dashEdges <- function(x, splits) {
        y <- paste(sort(unlist(x)), collapse=" ")
        if (!y %in% splits)
                attr(x, "edgePar") <- list(lty=2)
        x
 }
> layout(matrix(1:2, nrow=1))
> plot(dendrapply(tree_UniformCosts, dashEdges, splits_NonUniformCosts),
        main="MP uniform costs")
> plot(dendrapply(tree_NonUniformCosts, dashEdges, splits_UniformCosts),
        main="MP non-uniform costs")
```

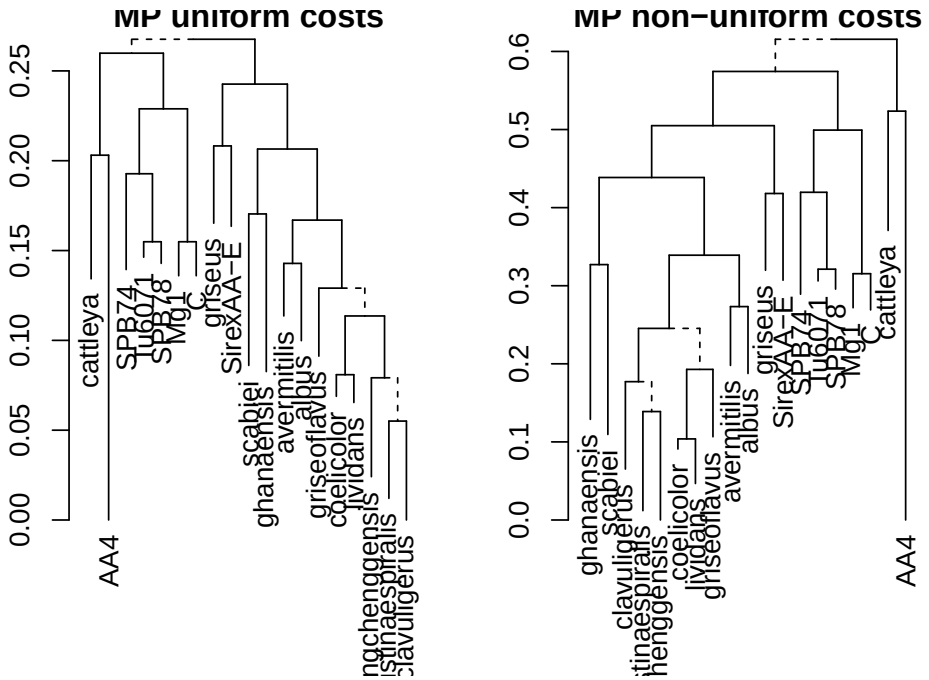

Figure 4: Comparison of MP trees built with different cost matrices.

7.1 Ancestral State Reconstruction

We're in luck —when *reconstruct* is `TRUE`, `Treeline` infers ancestors for each internal node on the tree [6]. These character states can be used by the function `MapCharacters` to determine state transitions along each edge of the tree. This information enables us to plot the total number of substitutions occurring along each edge. The state transitions can be accessed along each edge by querying a new "change" attribute.

```
> new_tree <- MapCharacters(tree_NonUniformCosts, labelEdges=TRUE)
> plot(new_tree, edgePar=list(p.col=NA, p.border=NA, t.col="#55CC99", t.cex=0.7))
> attr(new_tree[[1]], "change") # state changes on first branch left of (virtual) root
 [1] "A168T" "A177G" "A208T" "C269T" "A274G" "A275C" "A308G" "C333G" "A371T"
[10] "A375G" "C386G" "C395T" "A403G" "G405T" "A406G" "C417T" "G432T" "A453G"
[19] "G455T" "G598T"
```

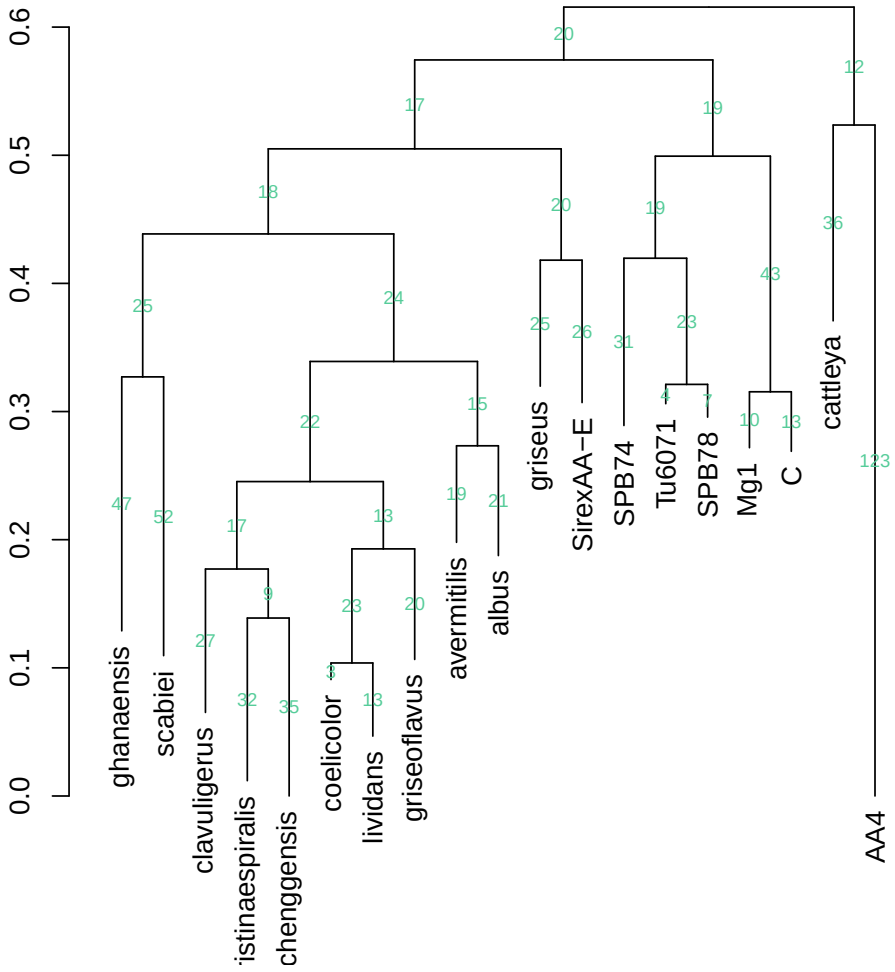

Figure 5: Edges labeled with the number of state transitions.

8 Calculating Bootstrap Support Values

Phylogenetic trees output by `Treeline` contain information in both their topology and branch lengths. The goal of phylogenetics is often to determine the branching order of a set of taxa, but this requires a test for statistical significance. It is usually best to compare trees across different genes, such as how often trees constructed from different genes support the same hypothesis. In the absence of multiple genes, another option is to quantify the amount of support for each branch separating two sets of taxa.

The aBayes probabilities are a good proxy for whether a partition in the tree is correct [3], but they are only available for maximum likelihood trees. For the other trees we need to make our own luck by bootstrapping the alignment. The idea behind bootstrapping is to resample columns (sites) of the alignment with replacement and determine whether each partition was found in the original tree. Repeating this process allows us to measure the level of support for each branch.

```
> reps <- 100 # number of bootstrap replicates
> tree1 <- Treeline(seqs, verbose=FALSE, processors=1)
> partitions <- function(x) {
         if (is.leaf(x))
                 return(NULL)
         x0 <- paste(sort(unlist(x)), collapse=" ")
         x1 <- partitions(x[[1]])
         x2 <- partitions(x[[2]])
         return(list(x0, x1, x2))
 }
> pBar <- txtProgressBar()
> bootstraps <- vector("list", reps)
> for (i in seq_len(reps)) {
         r <- sample(width(seqs)[1], replace=TRUE)
         at <- IRanges(r, width=1)
         seqs2 <- extractAt(seqs, at)
         seqs2 <- lapply(seqs2, unlist)
         seqs2 <- DNAStringSet(seqs2)

         temp <- Treeline(seqs2, verbose=FALSE)
         bootstraps[[i]] <- unlist(partitions(temp))
         setTxtProgressBar(pBar, i/reps)
 }
================================================================================
> close(pBar)
```

Now we can label edges by the percentage of times each partition appeared among the bootstrap replicates.

```
> bootstraps <- table(unlist(bootstraps))
> original <- unlist(partitions(tree1))
> hits <- bootstraps[original]
> names(hits) <- original
> w <- which(is.na(hits))
> if (length(w) > 0)
        hits[w] <- 0
> hits <- round(hits/reps*100)
> labelEdges <- function(x) {
        if (is.null(attributes(x)$leaf)) {
                part <- paste(sort(unlist(x)), collapse=" ")
                attr(x, "edgetext") <- as.character(hits[part])
        }
        return(x)
 }
> tree2 <- dendrapply(tree1, labelEdges)
> attr(tree2, "edgetext") <- NULL # remove text from (virtual) root branch
> plot(tree2, edgePar=list(t.cex=0.5), nodePar=list(lab.cex=0.7, pch=NA))
```

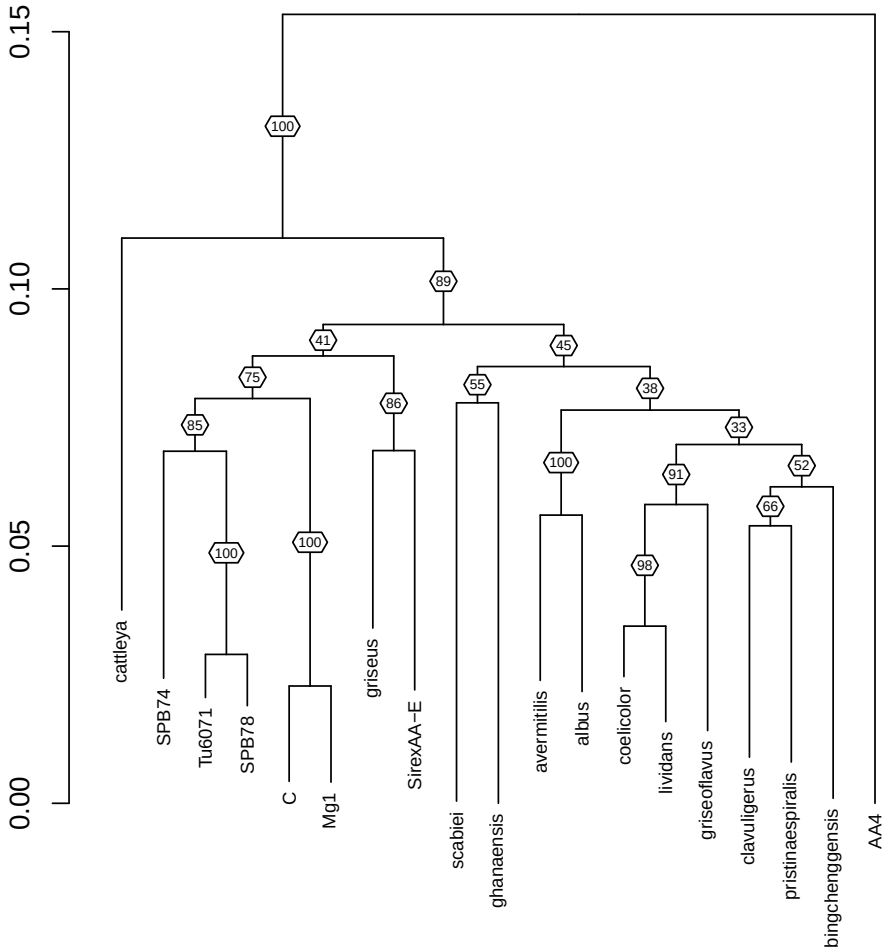

Figure 6: Tree with bootstrap support probabilities at each internal node.

9 More Examples of Manipulating Dendrograms

It is sometimes useful to alter *dendrogram* objects output by `Treeline`. There are three main ways for working with *dendrogram*s: apply a function to each leaf with `rapply`, apply a function to every node with `dendrapply`, or apply your own function recursively. The next examples will illustrate each of these approaches with increasing complexity.

In the first example, we will use `rapply` to query and set attributes of each leaf.

```
> rapply(tree, attr, which="label") # label of each leaf (left to right)
 [1] "ghanaensis"        "scabiei"           "avermitilis"
 [4] "albus"             "coelicolor"        "lividans"
 [7] "griseoflavus"      "clavuligerus"      "pristinaespiralis"
[10] "bingchenggensis"   "cattleya"          "griseus"
[13] "SirexAA-E"         "SPB74"             "SPB78"
[16] "Tu6071"            "Mg1"               "C"
[19] "AA4"
> labels(tree) # alternative
 [1] "ghanaensis"        "scabiei"           "avermitilis"
 [4] "albus"             "coelicolor"        "lividans"
 [7] "griseoflavus"      "clavuligerus"      "pristinaespiralis"
[10] "bingchenggensis"   "cattleya"          "griseus"
[13] "SirexAA-E"         "SPB74"             "SPB78"
[16] "Tu6071"            "Mg1"               "C"
[19] "AA4"
> rapply(tree, attr, which="height") # height of each leaf (left to right)
 [1] 8.926831e-01 6.272020e-01 6.608982e-01 6.270114e-01 5.054356e-01
 [6] 4.640741e-01 5.105017e-01 4.212123e-01 4.060434e-01 3.867419e-01
[11] 2.324664e-02 4.502042e-01 3.676072e-01 4.026977e-01 3.700399e-01
[16] 3.699553e-01 5.897868e-03 0.000000e+00 4.440892e-16
> italicize <- function(x) {
         if(is.leaf(x))
                 attr(x, "label") <- as.expression(substitute(italic(leaf),
                         list(leaf=attr(x, "label"))))
         x
 }
> rapply(tree, italicize, how="replace") # italicize leaf labels
'dendrogram' with 2 branches and 19 members total, at height 2.32241
```

In the second example, we will use `dendrapply` to identify exclusive groups wherein the members of each group are more similar to each other than they are to those outside the group [11].

```
> d <- DistanceMatrix(seqs, correction="F81+F", verbose=FALSE, processors=1)
> exclusive <- function(x) {
         if (!is.leaf(x)) { # leaves are trivially exclusive
                 leaves <- unlist(x)
                 max_dist <- max(d[leaves, leaves]) # max within group
                 if (all(max_dist < d[-leaves, leaves]))
                         attr(x, "edgePar") <- list(col="orange")
         }
         x
 }
> plot(dendrapply(tree, exclusive))
```

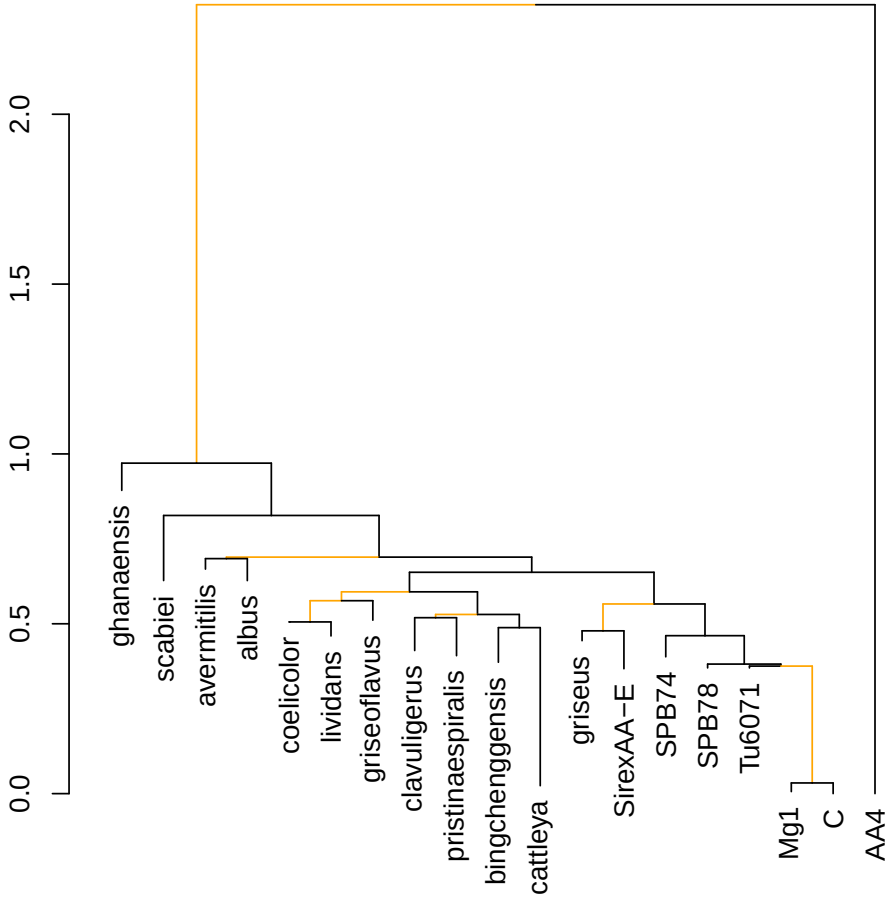

Figure 7: Tree with colored branches above exclusive groups.

In the third example, we will extract the branching order of five species of interest using a recursive function. This might be useful if we wanted to count how many times different topologies occurred among a set of trees. Recursion is the most flexible approach and can be applied with more sophisticated functions to accomplish goals beyond what is possible with `dendrapply`.

```
> Spp <- c("coelicolor", "lividans", "AA4", "Mg1", "scabiei") # species to retain
> extractClade <- function(x) {
         if (is.leaf(x)) {
                 if (sum(Spp %in% labels(x)) > 0L) {
                         labels(x)
                 } else {
                         NULL
                 }
         } else {
                 x <- lapply(x, extractClade)
                 x <- x[lengths(x) > 0]
                 if (length(x) == 1)
                         x <- x[[1]]
                 x
         }
 }
> extractClade(tree)
[[1]]
[[1]][[1]]
[1] "scabiei"

[[1]][[2]]
[[1]][[2]][[1]]
[[1]][[2]][[1]][[1]]
[1] "coelicolor"

[[1]][[2]][[1]][[2]]
[1] "lividans"


[[1]][[2]][[2]]
[1] "Mg1"



[[2]]
[1] "AA4"
```

10 Inspecting the Inputs and Outputs

If you are feeling down on your luck, you might want to double-check the inputs and outputs for any issues. First, we can check for any input sequences with unexpectedly few or many characters by comparing character frequencies across all input sequences. Next, we can look for input sequences that significantly deviate from the expected background frequencies using Pearson's chi-squared test. We can also check for sequences with extreme distances

that might be incorrectly aligned. Outliers in any of these checks may point to spurious sequences that should be double-checked for correctness or completion.

```
> freqs <- alphabetFrequency(seqs, baseOnly=TRUE)
> head(freqs)
       A   C   G   T other
[1,] 110 139 206 136    36
[2,] 101 137 207 123    59
[3,] 107 166 222 108    24
[4,] 112 151 207 124    33
[5,] 116 138 196 114    63
[6,] 112 139 195 115    66
> # summarize the number of non-base characters (gaps/ambiguities)
> summary(freqs) # "other" is non-base characters
       A               C               G               T
 Min.   :101.0   Min.   :132.0   Min.   :192.0   Min.   :108.0
 1st Qu.:110.0   1st Qu.:137.5   1st Qu.:197.0   1st Qu.:114.5
 Median :114.0   Median :139.0   Median :206.0   Median :123.0
 Mean   :113.0   Mean   :141.8   Mean   :204.6   Mean   :121.6
 3rd Qu.:116.5   3rd Qu.:144.0   3rd Qu.:210.5   3rd Qu.:126.5
 Max.   :120.0   Max.   :166.0   Max.   :222.0   Max.   :136.0
     other
 Min.   :24.00
 1st Qu.:32.00
 Median :45.00
 Mean   :45.95
 3rd Qu.:60.00
 Max.   :69.00
> # index of sequence with the most non-base characters
> which.max(freqs[, "other"])
[1] 15
> freqs <- freqs[, DNA_BASES]
> background <- colMeans(freqs)
> background
       A        C        G        T
113.0000 141.7895 204.6316 121.6316
> # look for sequences deviating from background frequencies
> chi2 <- colSums((t(freqs) - background)^2/background)
> pval <- pchisq(chi2, length(background) - 1, lower.tail=FALSE)
> w <- which(pval < 0.05)
> seqs[w] # outlier sequences
DNAStringSet object of length 0
> freqs[w,] # frequencies of outliers
     A C G T
> # get sequence index of any very distant outlier sequences
> D <- DistanceMatrix(seqs, verbose=FALSE, processors=1)
> t <- table(which(D > 0.9, arr.ind=TRUE)) # choose a cutoff
> head(sort(t, decreasing=TRUE)) # index of top outliers, if any
integer(0)
```

It is also possible to check whether the output tree reasonably represents the distances between sequences. For ME trees, the tree should explain greater than `0.9` of the variance in the distance matrix used to construct the tree. We can use Pearson's correlation for trees with branch lengths in different units than the distance matrix (i.e., ML or MP). Lower correlations may result from alignments with sites having different genealogies, such as concatenated alignments or non-orthologous sequences.

```
> P <- Cophenetic(treeME) # patristic distances
> D <- as.dist(D) # conver to 'dist' object
> plot(D, P, xlab="Pairwise distance", ylab="Patristic distance", log="xy")
> abline(a=0, b=1)
> # for ME trees we want explained variance > 0.9
> V <- 1 - sum((P - D)^2)/sum((D - mean(D))^2)
> V # check the input data if V << 1
[1] 0.9441992
> cor(P, D) # should be >> 0
[1] 0.9725586
> cor(log(P), log(D)) # should be >> 0
[1] 0.973351
```

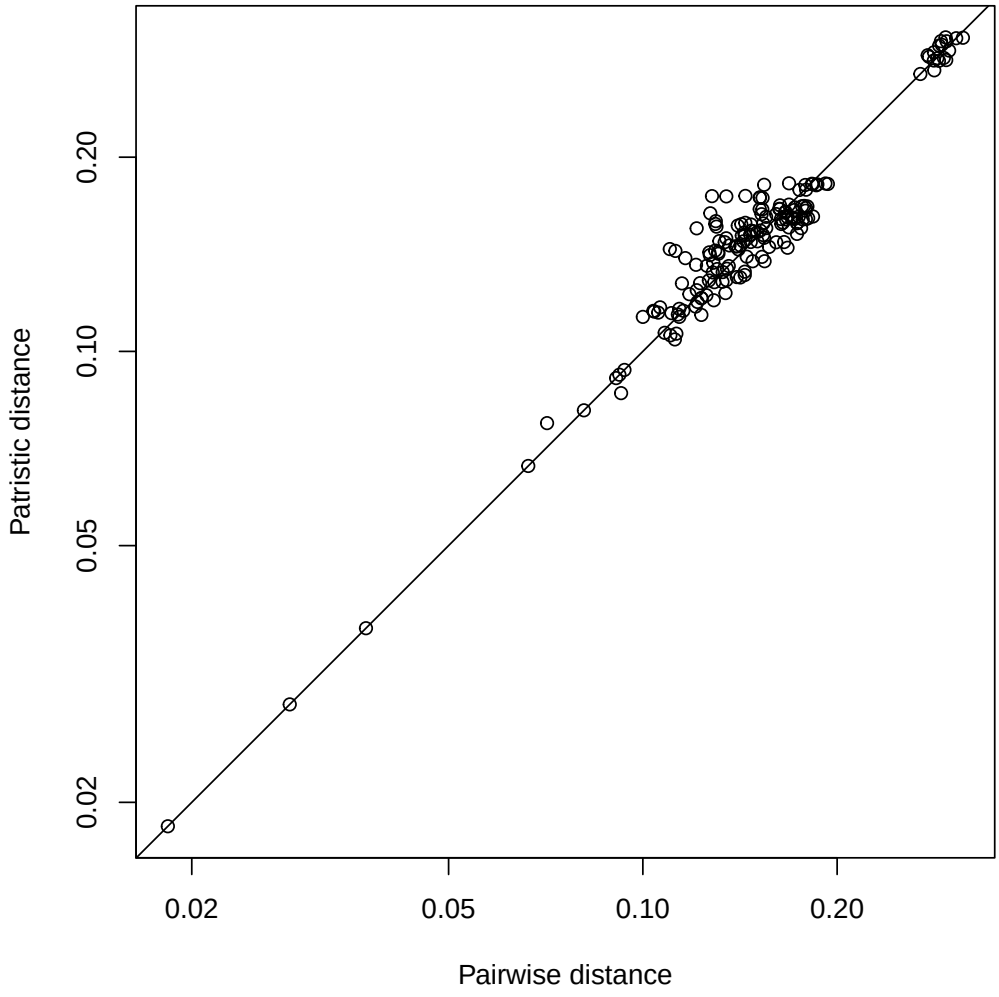

Figure 8: Confirming correlation between input distances and output patristic distances.

11 Exporting the Tree

We've had a run of good luck with this tree, so we'd better save it before our luck runs out! The functions `ReadDendrogram` and `WriteDendrogram` will import and export trees in Newick file format. If we leave the *file* argument blank then it will print the output to the console for our viewing:

```
> WriteDendrogram(tree, file="")
(('ghanaensis':0.08020989,('scabiei':0.1916202,(('avermitilis':0.03090385,'albus':0.0647
```

To keep up our lucky streak, we should probably include any model parameters in the output along with the tree. Luckily, Newick format supports square brackets (i.e., "[]") for comments, which we can append to the end of the file for good luck:

```
> params <- attr(tree, "parameters")
> cat("[", paste(names(params), params, sep="=", collapse=","), "]",
          sep="", append=TRUE, file="")
[FreqA=0.17661737505633,FreqC=0.243184606933302,FreqG=0.345458963969295,FreqT=NA,FreqI=NA
```

12 Session Information

All of the output in this vignette was produced under the following conditions:

- R version 4.4.2 (2024-10-31), `x86_64-pc-linux-gnu`
- Running under: `Ubuntu 24.04.2 LTS`
- Matrix products: default
- BLAS: `/usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3`
- LAPACK: `/usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.26.so` ; LAPACK version3.12.0
- Base packages: base, datasets, grDevices, graphics, methods, stats, stats4, utils
- Other packages: BiocGenerics 0.53.6, Biostrings 2.75.3, DECIPHER 3.3.2, GenomeInfoDb 1.43.4, IRanges 2.41.3, S4Vectors 0.45.4, XVector 0.47.2, generics 0.1.3
- Loaded via a namespace (and not attached): DBI 1.2.3, GenomeInfoDbData 1.2.13, KernSmooth 2.23-26, R6 2.6.1, UCSC.utils 1.3.1, buildtools 1.0.0, compiler 4.4.2, crayon 1.5.3, evaluate 1.0.3, httr 1.4.7, jsonlite 1.8.9, knitr 1.49, maketools 1.3.2, sys 3.4.3, tools 4.4.2, xfun 0.50

References

[1] Abadi, S., Azouri, D., Pupko, T., & Mayrose, I. Model selection may not be a mandatory step for phylogeny reconstruction. *Nat. Comm.*, 10(1).

[2] Anisimova, M., Gil, M., Dufayard, J., Dessimoz, C., & Gascuel, O. Survey of branch support methods demonstrates accuracy, power, and robustness of fast likelihood-based approximation schemes. *Syst. Biol.*, 60(5), 685-699.

[3] Ecker, N., Huchon, D., Mansour, Y., Mayrose, I., & Pupko, T. A machine-learning-based alternative to phylogenetic bootstrap. *Bioinformatics*, 40, i208-i217.

[4] Gonnet, G. Surprising results on phylogenetic tree building methods based on molecular sequences. *BMC Bioinformatics*, 13(148).

[5] Hoff, M., Orf, S., Riehm, B., Darriba, D., & Stamatakis, A. Does the choice of nucleotide substitution models matter topologically? *BMC Bioinformatics*, 17(143).

[6] Joy, J., Liang, R., McCloskey, R., Nguyen, T., & Poon, A. Ancestral Reconstruction. *PLoS Comp. Biol.*, 12(7), e1004763.

[7] Kapli, P., Kotari, I., Telford, M., Goldman, N., & Yang, Z. DNA Sequences Are as Useful as Protein Sequences for Inferring Deep Phylogenies. *Syst. Biol.*, 72(5), 1119-1135.

[8] Kumar, S., Tao, Q., Lamarca, A., & Tamura, K. Computational Reproducibility of Molecular Phylogenies. *Mol. Biol. Evol.*, 40(7).

[9] Ripplinger, J. & Sullivan, J. Does Choice in Model Selection Affect Maximum Likelihood Analysis? *Syst. Biol.*, 57(1), 76-85.

[10] Ripplinger, S., Sigorskikh, A., Efremov, A., Penzar, D., & Karyagina, A. PhyloBench: A Benchmark for Evaluating Phylogenetic Programs. *Mol. Biol. Evol.*, 41(6).

[11] Wright, E. & Baum, D. Exclusivity offers a sound yet practical species criterion for bacteria despite abundant gene flow. *BMC Genomics*, 19(724).