An introduction to phylogenetic comparative methods

chundra

12/28/2021

```
require(expm)
require(phytools)
require(ggtree)
require(rstan)
```

The following notebook demonstrates some fundamentals of phylogenetic inference. A popular family of phylogenetic models assumes that discrete features evolve according to a continuous-time Markov process, in which transitions between feature values take place according to *transition rates*. The backbone of phylogenetic inference is Felsenstein's Pruning Algorithm, which iteratively computes the likelihood of *transition rates* under a tree topology and cross-linguistic data.

To understand how the pruning algorithm works, we first need to take into consideration how continuous-time Markov chains (CTMCs) work. As mentioned above, a CTMC assumes that transitions between states in a system (or values of a feature) take place according to non-negative transition rates. A well-formed CTMC rate matrix might look like the following:

```
Q <- rbind(c(-3,2,1),c(1,-2,1),c(.5,.5,-1))
colnames(Q) = rownames(Q) = c('SVO','SOV','VSO')
```

Critically, values in off-diagonal cells (representing the transition rates between each pair of states) must be greater than zero, and the values in diagonal cells must equal the negative sum of values in the off-diagonal cells of each row.

We can exponentiate the rate matrix to calculate the probability that the system will end up in a particular state after a fixed interval of time, if it started in another state. E.g., if we set the time interval to .1, we can compute the probability that our word order feature will be `SOV` if the value was `SVO` before .1 time intervals elapsed.

```
t <- .1
P <- expm(Q*t)
P
```

```
##            SVO        SOV        VSO
## SVO 0.75068638 0.15867900 0.09063462
## SOV 0.08036633 0.82899905 0.09063462
## VSO 0.04326365 0.04737097 0.90936538
```

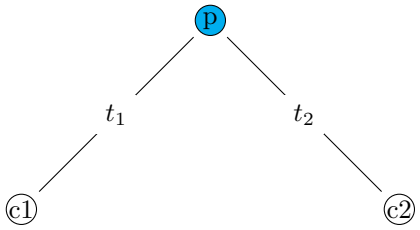
```
P['SVO','SOV']
```

```
## [1] 0.158679
```

It is thus clear that if we assume a single trajectory of change (e.g., a lineage in a phylogenetic tree, which has a length, a parent node, and a child node) it is straightforward to compute $P(\text{child state}|\text{parent state}, \text{branch length}, \text{rates})$. But we usually do not know all of this information a priori. We usually start with data observed among a sample of related languages, as well as a phylogenetic representation of these languages' relationship. We know neither the evolutionary rates nor the states for the internal nodes of the tree. The pruning algorithm computes the likelihood of a set of evolutionary rates for a phylogeny and observed data: that is, the probability of observing the attested data under the rates and phylogeny.

The pruning algorithm is an iterative algorithm that exploits conditional independence among branches of a tree to efficiently carry out computation. We can illustrate how it would work for a very shallow subtree in a

hypothetical tree. Below, we assume that c(hild) 1 and c(hild) 2 represent languages in a tree for which data are attested; t1 and t2 represent the lengths of the branches leading to c1 and c2, respectively.



According to Felsenstein (2004:254), we can compute the $L_p(s)$, the probability of everything that is observed in the subtree of which p is the parent, conditional on node p having the state s, according to the following formula (for simplicity, assume that the feature in question is binary, having the states 0 and 1):

$$L_p(s) = \left(\sum_{i \in \{0,1\}} P(i|s, t_1, \text{rates}) L_{c_1}(i) \right) \left(\sum_{i \in \{0,1\}} P(i|s, t_2, \text{rates}) L_{c_2}(i) \right)$$

We can unpack the left term as follows: $P(i|s, t_1, \text{rates})$ is the probability of ending up in state i if we started in state s and t_1 time intervals (the length of the branch connecting nodes p and $c1$) have elapsed, given some evolutionary rates. We get this probability by exponentiating the rate matrix, as above. $L_{c_1}(i)$ is the likelihood that node c_1 has state i. For tips (i.e., terminal nodes of the tree), which represent languages for which data are attested, this number will be either 0 or 1, as we know (e.g.,) whether prepositions are present or absent in a given language. We can expand the sum fully, since we are working with a binary feature:

$$\left(\sum_{i \in \{0,1\}} P(i|s, t_1, \text{rates}) L_{c_1}(i) \right) = P(0|s, t_1, \text{rates}) L_{c_1}(0) + P(1|s, t_1, \text{rates}) L_{c_1}(1)$$

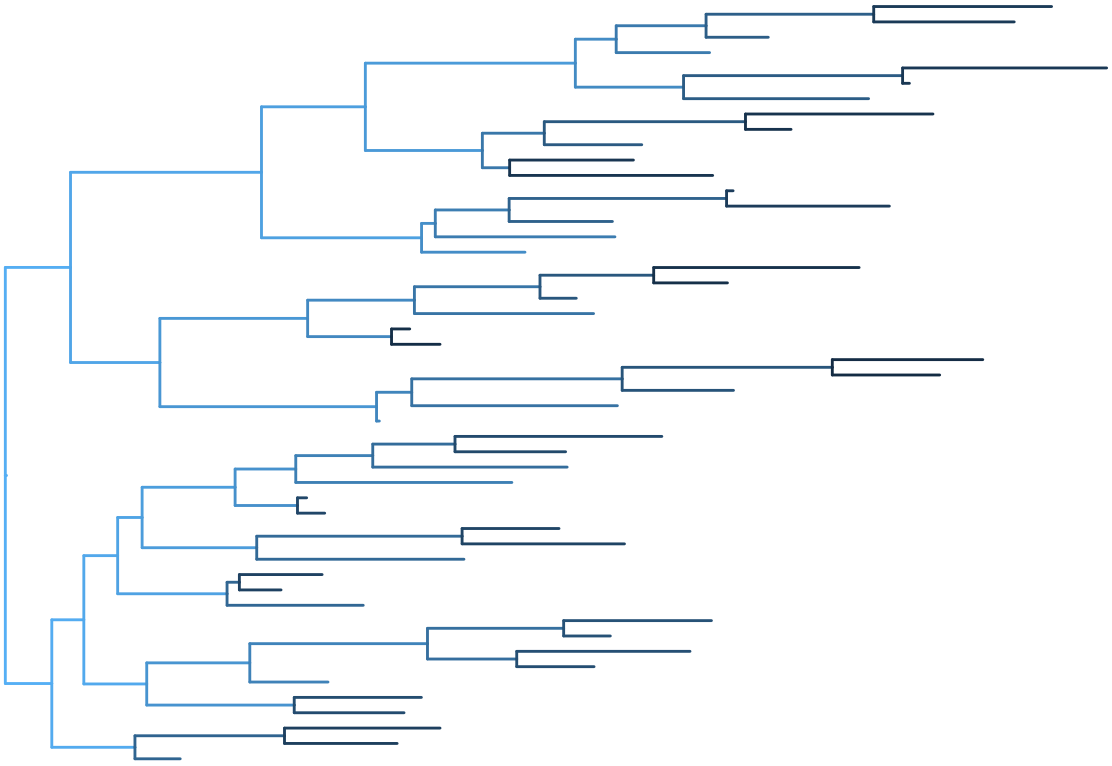
Articulated in words, this is the probability that a state s could yield a value of 0 *or* 1 for node c_1 times the probability that node $c1$ is in either state, given everything that could descend from it. If c_1 is a tip, then $L_{c_1}(0)$ and $L_{c_1}(1)$ will be either 1 or 0, respectively, depending on whether the feature is present or absent. If the feature is present, then the term above works out to the probability of that state s results in the feature being present.

$$P(0|s, t_1, \text{rates}) L_{c_1}(0) + P(1|s, t_1, \text{rates}) L_{c_1}(1) = P(0|s, t_1, \text{rates}) \cdot 0 + P(1|s, t_1, \text{rates}) \cdot 1 = P(1|s, t_1, \text{rates})$$

The right term can be interpreted in the same fashion, and the likelihood of the subtree is the product of these terms: one for each descendant node.

Once we compute the likelihood of the subtree of which node p is a parent, if we have the likelihood of the subtree for which its sibling is present, we can then compute the likelihood of the subtree that begins directly above these nodes. These likelihoods are computed in *post-order traversal*. This means that we do not visit a node in order to compute the likelihood of the subtree descending from it until we have visited all of the nodes that descend from it. The following graphic attempts to demonstrate this traversal: lighter branches have parent nodes that are visited later in the traversal.

```
tree <- rtree(50)
tree <- reorder.phylo(tree,'pruningwise')
nodes <- c(tree$edge[,2],tree$edge[nrow(tree$edge),1])
ggtree(tree,aes(col=order(nodes)/length(nodes)))
```



The final node to be visited is the root, which is the ancestor of the entire tree. The likelihood of the entire tree is the following:

$$L(\text{tree}) = \sum_{s \in \text{states}} P_{\text{root}}(s) L_{\text{root}}(s)$$

$P_{\text{root}}(s)$ is the prior probability of state s at the root of the tree. There are multiple options for this prior. One option is to assume a uniform prior over states (e.g, for D states, each state has a prior probability of $\frac{1}{D}$); an alternative is to use the stationary probability of each state (i.e., the probability of seeing a given state as time approaches infinity). Yet another alternative can be found in FitzJohnn et al 2009.

A practical example

We can analyze the evolutionary dynamics of specific features in Indo-European. We can work with the DiACL Eurasia dataset from Lund University, and a reference tree of Indo-European.

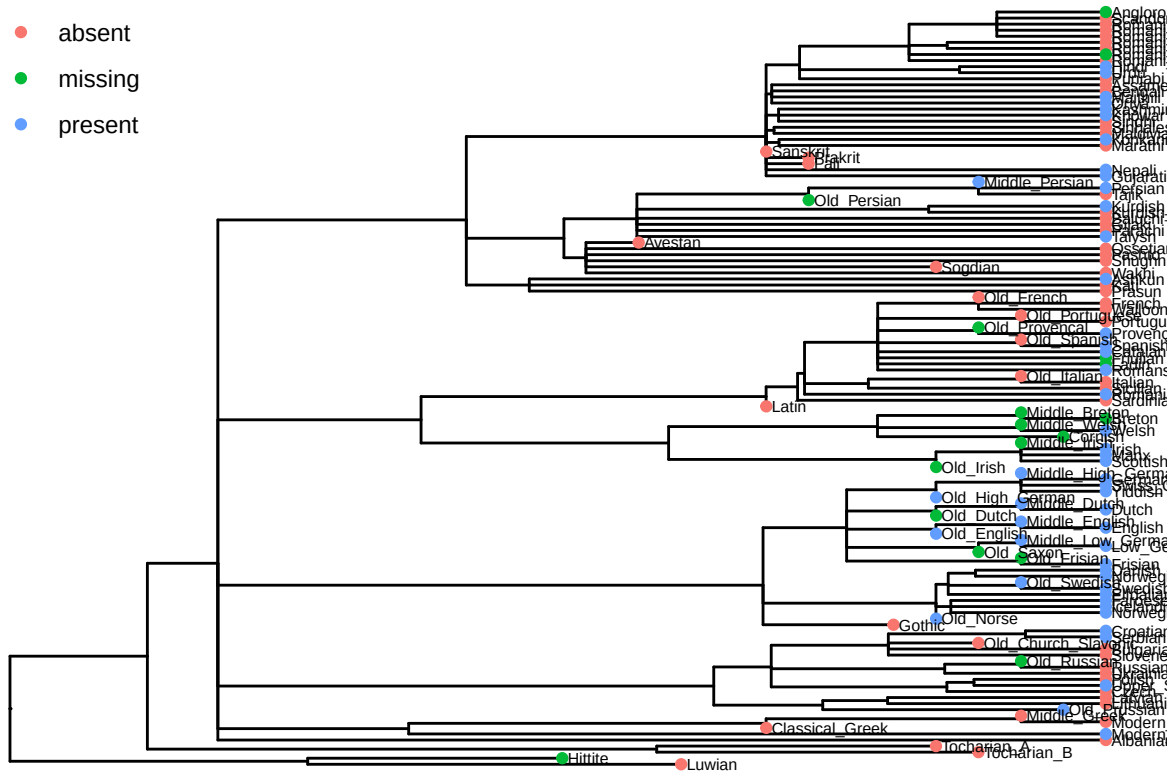
```
# read data
diacl <- read.csv('diacl_binary.txt',sep='\t', row.names = 1)
# read in Indo-European Maximum Clade Credibility tree
tree <- read.tree('IE_MCC.newick')
```

Prior to doing any quantitative work, we need to ensure that the branches of the tree are re-ordered in order to facilitate post-order traversal:

```
tree <- reorder.phylo(tree,'pruningwise')
```

We select a single feature to work with. Here, we visualize the distribution of the feature "Word.order.WH.element.WH.V", which takes the value 1 if wh-words precede the verb in main clauses or 0 otherwise.

```
diacl.for.display <- diacl[tree$tip.label,]
diacl.for.display[diacl.for.display==0] <- 'absent'
diacl.for.display[diacl.for.display==1] <- 'present'
diacl.for.display[is.na(diacl.for.display)] <- 'missing'
diacl.for.display <- data.frame(language=rownames(diacl.for.display),Word.order.WH.element.WH.V=as.facto
#diacl.for.display <- diacl.for.display[,119:120]
ggtree(tree) %<+% diacl.for.display + geom_tippoint(aes(col=Word.order.WH.element.WH.V)) + geom_tiplab(s
```



This is a binary feature. Hence, we can infer a gain rate, according to which the feature arises, and a loss rate, according to which it is lost. In order to infer the posterior distributions of these rates, we need to prepare a few more data objects.

First, we need to take our vector representing feature presence/absence/missing values, and convert it to a two-column matrix indicating (1) the likelihood of absence and (2) likelihood of presence of the feature for each language in the tree. That looks as follows:

```
states <- diacl[,"Word.order.WH.element.WH.V"]
names(states) <- rownames(diacl)
states <- states[tree$tip.label]

#for missing data, set each likelihood to 1
tip.lik <- to.matrix(states,seq=c(0,1))
tip.lik[rowSums(tip.lik)==0,] <- c(1,1)
```

Then, we create data objects representing the parent node of each branch, the child node of each branch, the length of each branch, the total number of nodes in the tree (including tips), and the total number of tips in the tree:

```
parent <- tree$edge[,1]
child <- tree$edge[,2]
b.lens <- tree$edge.length/1000  #scale branch lengths by 1000
N <- length(unique(c(parent,child)))
T <- length(child[which(!child %in% parent)])
```

We can then set up the Stan code that infers the rates. We make use of the `functions` block to predefine some functions in order to not have to write them out multiple times. The first function carries out matrix exponentiation for a two-rate CTMC. The second function carries out the pruning algorithm. All that is then left to do is define the variables and parameters, place priors over the parameters, and then increment the log model probability by the pruning likelihood using `target +=`.

```
model_code = "functions {
  //compute transient probability for continuous-time markov process of character evolution,
  //i.e., p(end state|start state, rates, time)
  real evprob(int x, int y, real z, real a, real b) {
    real p;
    if (y==0) {
      p = b/(a+b);
      if (x==0) {
        p = p + a/(a+b)*exp(-(a+b)*z);
      }
      if (x==1) {
        p = p - b/(a+b)*exp(-(a+b)*z);
      }
    }
    if (y==1) {
      p = a/(a+b);
      if (x==0) {
        p = p - a/(a+b)*exp(-(a+b)*z);
      }
      if (x==1) {
        p = p + b/(a+b)*exp(-(a+b)*z);
      }
    }
    return p;
  }
  real pruning_likelihood(int T, int N, int B, int[,] tiplik, int[] parent, int[] child, real[] brlen, i
  matrix[N,2] lambda;
  for (t in 1:T) {                      //put tip log likelihoods into matrix
    lambda[t,1] = log(tiplik[t,1]);
    lambda[t,2] = log(tiplik[t,2]);
  }
  for (n in (T+1):N) {
    lambda[n,1] = 0;
    lambda[n,2] = 0;
  }
  for (b in 1:B) {                      //for each branch (arranged for post-order traversal)
    lambda[parent[b],1] += log(evprob(0,0,alpha,beta,brlen[b])*exp(lambda[child[b],1])
     + evprob(0,1,alpha,beta,brlen[b])*exp(lambda[child[b],2]));
    lambda[parent[b],2] += log(evprob(1,0,alpha,beta,brlen[b])*exp(lambda[child[b],1])
     + evprob(1,1,alpha,beta,brlen[b])*exp(lambda[child[b],2]));
    }
    return log(.5*exp(lambda[parent[B],1]) + .5*exp(lambda[parent[B],2]));
```

```
  }
}
data {
  int<lower=1> N; //number of tips+internal nodes+root
  int<lower=1> T; //number of tips
  int<lower=1> B; //number of branches
  int<lower=1> child[B];                //child of each branch
  int<lower=1> parent[B];               //parent of each branch
  real<lower=0> brlen[B];                //length of each branch
  int<lower=0,upper=1> tiplik[T,2];     //likelihoods for data at tips in tree
}
parameters {
  real<lower=0> alpha;                            //gain rate
  real<lower=0> beta;                             //loss rate
}
model {
  alpha ~ gamma(1,1);     //any priors defined on [0,inf) are possible
  beta ~ gamma(1,1);
  target += pruning_likelihood(T,N,B,tiplik,parent,child,brlen,alpha,beta);
}"
```

We then set up the data list, fit the model, and visualize the posterior distributions of model parameters:

```
data.list <- list(N=N,
                  T=T,
                  B=length(parent),
                  brlen=b.lens,
                  child=child,
                  parent=parent,
                  tiplik=tip.lik)

fit <- stan(model_code=model_code,data=data.list)
```

```
##
## SAMPLING FOR MODEL '701d6657c8e9d8308670a2f7ce618a80' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 0.000428 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 4.28 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 1: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 1: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 1: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 1: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 1: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 1: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 1: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 1: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 1: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 1.18428 seconds (Warm-up)
```

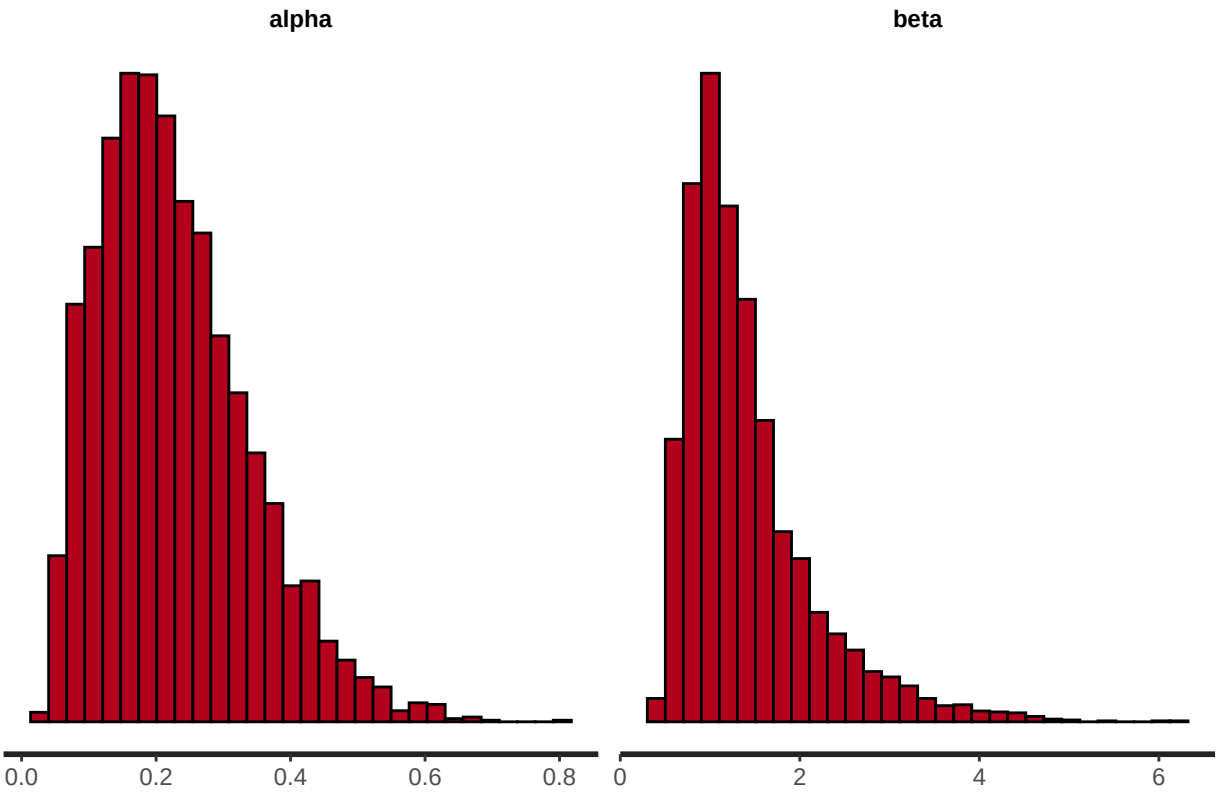
```
## Chain 1:                1.15977 seconds (Sampling)
## Chain 1:                2.34405 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL '701d6657c8e9d8308670a2f7ce618a80' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 0.000151 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 1.51 seconds.
## Chain 2: Adjust your expectations accordingly!
## Chain 2:
## Chain 2:
## Chain 2: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 2: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 2: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 2: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 2: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 2: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 2: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 2: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 2: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 2: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2:
## Chain 2:  Elapsed Time: 1.3099 seconds (Warm-up)
## Chain 2:                1.37574 seconds (Sampling)
## Chain 2:                2.68564 seconds (Total)
## Chain 2:
##
## SAMPLING FOR MODEL '701d6657c8e9d8308670a2f7ce618a80' NOW (CHAIN 3).
## Chain 3:
## Chain 3: Gradient evaluation took 0.000142 seconds
## Chain 3: 1000 transitions using 10 leapfrog steps per transition would take 1.42 seconds.
## Chain 3: Adjust your expectations accordingly!
## Chain 3:
## Chain 3:
## Chain 3: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 3: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 3: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 3: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 3: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 3: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 3: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 3: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 3: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 3: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 3: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 3: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 3:
## Chain 3:  Elapsed Time: 1.49025 seconds (Warm-up)
## Chain 3:                1.22073 seconds (Sampling)
## Chain 3:                2.71098 seconds (Total)
## Chain 3:
##
```

```
## SAMPLING FOR MODEL '701d6657c8e9d8308670a2f7ce618a80' NOW (CHAIN 4).
## Chain 4:
## Chain 4: Gradient evaluation took 0.000198 seconds
## Chain 4: 1000 transitions using 10 leapfrog steps per transition would take 1.98 seconds.
## Chain 4: Adjust your expectations accordingly!
## Chain 4:
## Chain 4:
## Chain 4: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 4: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 4: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 4: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 4: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 4: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 4: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 4: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 4: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 4: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 4: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 4: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 4:
## Chain 4:  Elapsed Time: 1.21562 seconds (Warm-up)
## Chain 4:                1.41273 seconds (Sampling)
## Chain 4:                2.62835 seconds (Total)
## Chain 4:
```

```
print(fit)
```

```
## Inference for Stan model: 701d6657c8e9d8308670a2f7ce618a80.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##         mean se_mean   sd   2.5%    25%    50%    75%  97.5% n_eff Rhat
## alpha   0.23    0.00 0.11   0.07   0.14   0.21   0.29   0.49   935    1
## beta    1.41    0.03 0.72   0.59   0.92   1.21   1.68   3.38   753    1
## lp__  -65.57    0.03 0.95 -68.02 -65.98 -65.29 -64.87 -64.63  1328    1
##
## Samples were drawn using NUTS(diag_e) at Wed Oct 12 14:40:39 2022.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

```
stan_hist(fit)
```

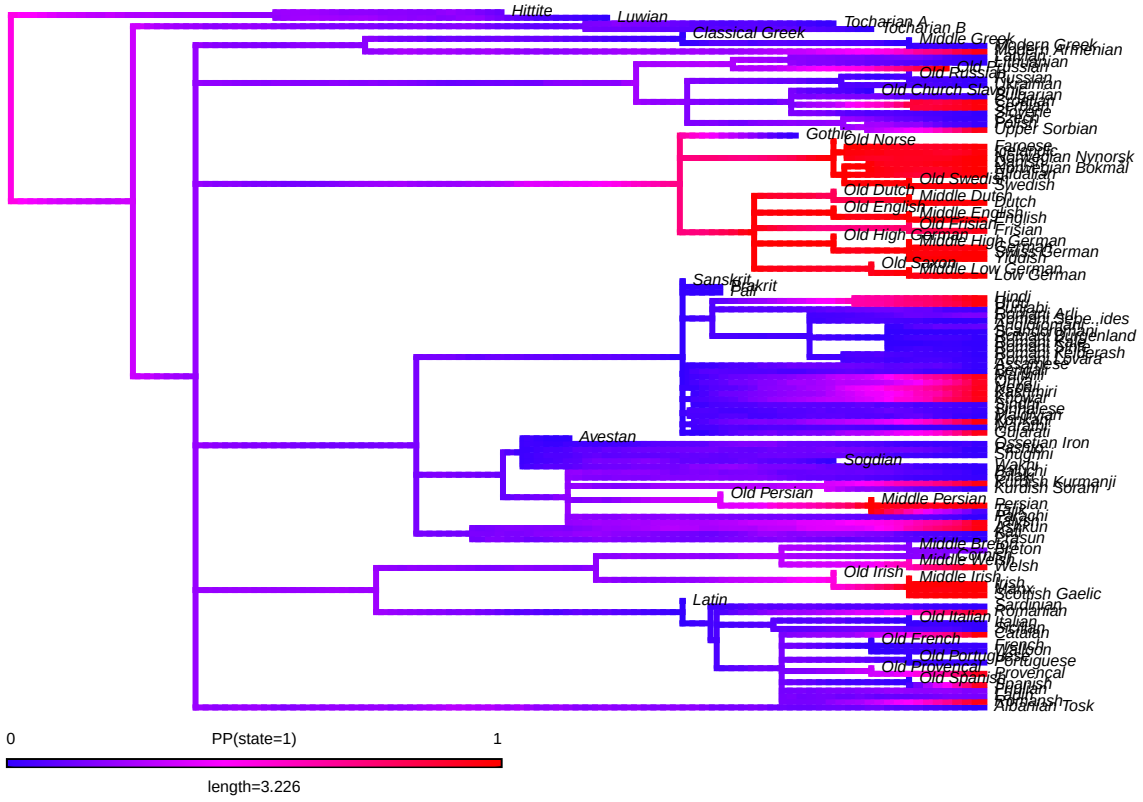


It is clear that the gain rate (alpha) is much lower than the loss rate (beta), indicating that WH-V order is gained infrequently and lost frequently. We can convert the gain and loss rates into **waiting times**. The feature is gained roughly .25 times per millennium and lost roughly 1.5 times per millennium. Thus, if the feature is lost, we can expect to wait on average $1000/.25 = 4000$ years before it is regained, and if it is gained, we can expect to wait on average $1000/1.5 = 666$ years before it is lost again. In short, it seems to be highly dispreferred.

We can explore the distribution of possible character histories by means of **stochastic character mapping**. This approach reconstructs ancestral states for internal nodes of the tree and simulates changes on the basis of posterior rates, usually multiple times, then creates a pleasant color gradient over the tree.

```
tree.scaled <- tree
tree.scaled$edge.length <- tree.scaled$edge.length/1000
mapped.tree.list <- list()
for (t in 1:100) {
  i = sample(1:4000,1)
  alpha_t = extract(fit)$alpha[i]
  beta_t = extract(fit)$beta[i]
  Q <- rbind(c(-alpha_t,alpha_t),c(beta_t,-beta_t))
  rownames(Q) <- colnames(Q) <- c('0','1')
  mapped.tree <- make.simmap(tree.scaled,tip.lik,Q=Q)
  mapped.tree.list[[t]] <- mapped.tree
}

#http://blog.phytools.org/2012/11/issue-with-plotting-output-trees-from.html

applyBranchLengths<-function(tree,edge.length){
  if(class(tree)=="multiPhylo"){
    trees<-lapply(tree,applyBranchLengths,
                  edge.length=edge.length)
```

```
    class(trees)<-"multiPhylo"
    return(trees)
  } else {
    tree$edge.length<-edge.length
    if(!is.null(tree$maps)){
      for(i in 1:nrow(tree$edge)){
        temp<-tree$maps[[i]]/sum(tree$maps[[i]])
        tree$maps[[i]]<-temp*tree$edge.length[i]
      }
    }
    return(tree)
  }
}

h<-sapply(mapped.tree.list,function(x) max(nodeHeights(x)))
newbl<-mapply(applyBranchLengths,mapped.tree.list, lapply(as.list(h),"*",mapped.tree.list[[1]]$edge.leng
class(newbl)<-"multiPhylo"

densityMap(newbl,fsize=.5)
```



According to the visualization, WH-V word order develops relatively late in the history of Indo-European, and is absent for millennia. When it does develop, we see a relatively small number of gains (e.g., it develops once in the Germanic languages), and appears to be lost in some languages as well (e.g., Tajik). This appears to bear out the result found in the rates: WH-V order arises infrequently and is lost with higher frequency than it is gained.