

Débora Yoshihara Caldeira Brandt



Aspirante a mestranda no departamento de Genética e Biologia Evolutiva do IB-USP, com o prof. Diogo Meyer.

Meus Exercícios

Linque para a página com os meus exercícios resolvidos: [exec](#)

Proposta de Trabalho Final

Plano B

(Obs: troquei a ordem do plano B e plano A pois achei que assim ficaria mais fácil de entender a ideia. O plano A na verdade é uma extensão do plano B, que não sei se chegarei a alcançar)

Proposta: Escrever uma função que extraia de um alinhamento de sequências de vários alelos de duas espécies as posições polimórficas e divergentes e as bases presentes nessas posições em cada espécie. Além das localizações dos polimorfismos e divergências no alinhamento, a função retornará também as posições desses sítios com coordenadas cromossômicas.

A função:

Entrada:

- Alinhamento de alelos de duas espécies em formato phylip.

(Os alinhamentos devem entrar já “limpos”: sem gaps, e todos com o mesmo comprimento.)

(Formato phylip: cada linha é composta por [nome do alelo em 8 caracteres] [2 espaços] [sequencia do alelo].)

- Coordenada cromossômica do primeiro nucleotídeo do alinhamento.

Passos:

- 1) Ler o arquivo de entrada transformando-o num data frame no qual a primeira coluna tem o nome dos alelos e as colunas seguintes têm cada uma um dos nucleotídeos da sequência. ¹⁾

- 2) Criar nesse data frame uma coluna da classe fator, que terá como níveis as espécies às quais as sequências pertencem. ²⁾

(Como os nomes dos alelos, no meu caso, seguem uma lógica do tipo [nome da sp abreviado]+[número do alelo], eu poderia definir os níveis do fator como o primeiro caracter do nome do alelo (ex: se forem alelos de humanos e gorilas, os níveis seriam “h” e “g”).) ³⁾

3) Tratar os nucleotídeos também como fatores (com 4 níveis: A, T, C, G). ⁴⁾ 4) Usando indexação, verificar as bases de cada sítio (cada coluna da tabela). Colocar a primeira base num vetor, e cada base diferente dela vai sendo adicionada nesse vetor, junto com o nome do fator (espécie) associado a ela (como label desse elemento do vetor, talvez). ⁵⁾

5) Posições polimórficas serão aquelas que tem mais de um tipo de base na 1a espécie, mas são fixas na 2a espécie. ⁶⁾

6) Posições divergentes serão aquelas que têm apenas um tipo de base na 1a espécie, e apenas um tipo de base na 2a, porém diferente da presente na 1a.

7) Com as posições polimórficas e divergentes do alinhamento, usar a coordenada cromossômica de início do alinhamento para determinar as coordenadas dos sítios de interesse (isso seria apenas uma adição simples...)

Saída:

Uma tabela ⁷⁾ estilo summary contendo:

- Posições polimórficas no alinhamento
- Posições equivalentes às polimórficas com coordenadas genômicas (referentes a posição dada pelo usuário)
- Bases existentes nas posições polimórficas
- Divergências entre as sequências das duas espécies: posição, posição genômica e nucleotídeos

Plano A

Adicionar a execução do teste de McDonald-Kreitman à função acima.

Brevíssimo contexto:

O teste de McDonald-Kreitman é um teste de neutralidade que considera a frequência de polimorfismos sinônimos ou não-sinônimos (P_s ou P_n) e divergências sinônimas ou não-sinônimas (D_s ou D_n). Sob neutralidade, espera-se que a razão P_n/P_s seja semelhante a D_n/D_s . Já sob seleção positiva, espera-se uma razão D_n/D_s maior, enquanto que sob seleção balanceadora espera-se que a razão P_n/P_s seja aumentada em relação a D_n/D_s .

Essa parte da função a tornaria muitíssimo mais interessante, porém uma dificuldade será dividir os polimorfismos e divergências nas categorias sinônimos e não-sinônimos. Precisei informar à função uma tabela de código genético e a leitura do sequenciamento deverá ser feita por códons, para identificar o tipo de mudança (sinônima ou não sinônima).

Outras idéias...

Flexibilizar o tipo de arquivo de entrada:

- a função poderia ter um argumento no qual o usuário informaria o tipo de arquivo - fasta, phylip, etc
- e a parte de leitura do arquivo em data frame poderia converter qualquer um deles ao formato de

data frame que o resto da função usa

-verificar funções já existentes para alinhamentos e fazer com que a função aceite alinhamentos com gaps, por exemplo

Comentários

Olá Debora,

A descrição da sua propstas está muito boa! Acho que a lógica é mesmo o Plano B antes do A. Acho que deve portanto, ficar no plano para a tarefa da disciplina e avançar ao B se houver tempo. Tenho poucas sugestões que coloquei ao longo do texto nos passos da proposta. Dê uma olhada nos comentários que aparecem como números na página html! É só colocar o cursor em cima do números.

— [Alexandre Adalardo de Oliveira](#) 2012/04/04 15:46

Página de ajuda

mk.test package:BIE5782 R Documentation

Teste de McDonald-Kreitman

Description:

Calcula o índice de neutralidade (NI) e a direção da seleção (DoS) a partir de um alinhamento de alelos de duas espécies.

Usage:

```
mk.test(phy, frameshift=1)
```

Arguments:

phy: Arquivo PHYLIP. Alinhamento dos alelos de duas espécies.

frameshift: numérico. Quadro de leitura do alinhamento (1, 2 ou 3).

Details:

O alinhamento não deve ter gaps ou inserções.

A espécie de interesse (aquela para a qual é calculada a quantidade de polimorfismos) é a que aparece primeiro no alinhamento.

Os nomes dos alelos devem indicar na primeira letra qual a espécie.

Value:

Lista contendo:

Posicoes polimorficas : Posições no alinhamento que são polimórficas na espécie 1.

Bases polimorficas : Bases que existem nas posições polimórficas da espécie 1.

Posicoes divergentes : Posições no alinhamento que têm substituições fixas nas duas espécies.

Bases divergentes : Bases que existem nas posições divergentes acima. A primeira base é a da espécie 2, e a 2a é a da espécie 1.

Contagens: Número de polimorfismos não-sinônimos (Pn) e sinônimos (Ps), e de divergências não-sinônimas (Dn) e sinônimas (Ds).

NI : Índice de Neutralidade como proposto por McDonald & Kreitman (1991)

DoS : Direção da Seleção, como proposta por Stoletzki & Eyre-Walker (2011)

Warning:

A função retorna apenas os valores dos estimadores de neutralidade NI e DoS, mas não informa se eles são significativamente maiores ou menores que 1, no caso de NI ou de 0, no caso de DoS, indicando neutralidade.

Valores de NI menores que 1 e de DoS positivos indicariam seleção positiva.

Valores de NI maiores que 1 e de DoS negativos indicariam segregação de mutações fracamente deletérias.

Author(s):

Débora Y. C. Brandt

deboraycb@gmail.com

References:

Stoletzki, N., & Eyre-Walker, A. (2011). Estimation of the neutrality index. *Molecular biology and evolution*, 28(1), 63-70.

doi:10.1093/molbev/msq249~

McDonald, J. H., & Kreitman, M. (1991). Adaptive protein evolution at the Adh locus in *Drosophila*. *Nature*, 351.~

Examples:

Baixar o arquivo de exemplo de alinhamento na seção abaixo.

O alinhamento tem 32 bases, ou seja, 10 códons + 2 bases.

```
mk.test("alinhamentoteste.phy", frameshift=1)
# Usando frameshift=1 a última posição polimórfica (32) é descartada, pois
como só temos duas bases desse códon, não é possível determinar se o
polimorfismo é sinônimo ou não.
```

```
mk.test("alinhamentoteste.phy", frameshift=3)
# Nesse caso, o polimorfismo da posição 2 é descartado. Observar que a
ausência de divergência sinônima leva NI a 0.
```

Exemplo de alinhamento

[alinhamentoteste.txt](#)

Obs: O wiki não permite que eu suba arquivos no formato PHYLIP, então coloquei em .txt

Código da Função

```
mk.test<-function(phy, frameshift=1){
  ### Argumento frameshift ###
  if(frameshift!=1 && frameshift!=2 && frameshift!=3){
    stop("argument frameshift must be 1, 2 or 3\n\n")
  }
  ### Arrumando alinhamento ###
  alin<-read.table("alinhamentoteste.phy", skip=1, as.is=TRUE)
  size<-readLines("alinhamentoteste.phy", 1)
  size<-as.numeric(unlist(strsplit(size, " ")))
  nseq<-size[1]
  nnuc<-size[2]
  nuclist<-strsplit(alin[,2], NULL)
  nucchar<-data.frame()
  for(s in 1:nseq){
    for(n in 1:nnuc){
      nucchar[s,n]<-unlist(nuclist)[n+nnuc*(s-1)]
    }
  }
  sp.list<-strsplit(alin[,1], NULL)
  for(w in nseq:1){
    if(sp.list[[w]][1]!=sp.list[[w-1]][1]){
      sp1<-w-1
      sp2<-nseq-sp1
      break
    }
  }
  nucs1<-nucchar[1:sp1,]
  nucs2<-nucchar[sp1+1:sp2,]
  ### POLIMORFISMOS ###
  ## Posicoes polimorficas na especie 2:
  polim2<-vector()
```

```
for(k in 2:sp2){
  diferencas<-which(nucs2[1,]!=nucs2[k,])
  posicoesnovas<-setdiff(diferencas,polim2)
  polim2<-c(polim2,posicoesnovas)
}
## Posicoes polimorficas na especie 1:
polim1<-vector()
for(l in 2:sp1){
  pos.pol<-which(nucs1[1,]!=nucs1[l,])
  pos.pol.novas<-setdiff(pos.pol,polim1)
  polim1<-c(polim1,pos.pol.novas)
}
polim<-setdiff(polim1, polim2)
## Bases polimorficas
nuc.pol.list<-list()
for(posicao in polim){
  nuc.pol.list[[posicao]]<-nucs1[1,posicao]
  for(alelo in 2:sp1){
    if(all(nuc.pol.list[[posicao]]!=nucs1[alelo,posicao])){
      nuc.pol.list[[posicao]]<-
c(nuc.pol.list[[posicao]],nucs1[alelo,posicao])
    }
  }
}
## transformando em vetor para a saída:
nuc.pol<-vector()
for(o in polim){
  nuc.pol[which(polim==o)]<-paste(nuc.pol.list[[o]], collapse="")
}
#### DIVERGENCIA ####
## Posicoes monomorficas na especie 1:
fix1<-setdiff(1:nnuc, polim1)
## Posicoes monomorficas na especie 1 e 2 ao mesmo tempo:
fix<-setdiff(fix1, polim2)
## Posicoes monomorficas e divergentes:
div<-vector()
for(n in fix){
  if(nucs1[1,n]!=nucs2[1,n]){
    div<-c(div, n)
  }
}
## Bases divergentes
nuc.div<-vector()
for(q in div){
  nuc.div[which(div==q)]<-paste(nucs2[1,q], nucs1[1,q], sep="")
}
saida1<-list(polim,nuc.pol, div, nuc.div)
names(saida1)<-c("Posicoes polimorficas", "Bases polimorficas", "Posicoes
divergentes", "Bases divergentes")
#####
```

```

#### MK      ###
####  Tabela de código genético  ###
codons<-c("TTT", "TTC", "TTA", "TTG", "CTT", "CTC", "CTA", "CTG", "ATT",
"ATC", "ATA", "ATG", "GTT", "GTC", "GTA", "GTG", "TGT", "TGC", "TGA", "TGG",
"CGT", "CGC", "CGA", "CGG", "AGT", "AGC", "AGA", "AGG", "GGT", "GGC", "GGA",
"GGG", "TAT", "TAC", "TAA", "TAG", "CAT", "CAC", "CAA", "CAG", "AAT", "AAC",
"AAA", "AAG", "GAT", "GAC", "GAA", "GAG", "TCT", "TCC", "TCA", "TCG", "CCT",
"CCC", "CCA", "CCG", "ACT", "ACC", "ACA", "ACG", "GCT", "GCC", "GCA", "GCG")
aa<-c("Phe", "Phe", "Leu", "Leu", "Leu", "Leu", "Leu", "Leu", "Leu", "Ile", "Ile",
"Ile", "Met", "Val", "Val", "Val", "Val", "Cys", "Cys", "STOP", "Trp",
"Arg", "Arg", "Arg", "Arg", "Ser", "Ser", "Arg", "Arg", "Gly", "Gly",
"Gly", "Gly", "Tyr", "Tyr", "STOP", "STOP", "His", "His", "Gln", "Gln",
"Asn", "Asn", "Lys", "Lys", "Asp", "Asp", "Glu", "Glu", "Ser", "Ser",
"Ser", "Ser", "Pro", "Pro", "Pro", "Pro", "Thr", "Thr", "Thr", "Thr",
"Ala", "Ala", "Ala", "Ala" )
codon.table<-data.frame(cbind(codons, aa),stringsAsFactors=FALSE)
####  Lista de codons polimorficos  ###
cod.pol.list<-list()
for (i in polim){
  if(frameshift==1){
    ii<-i
  }
  if(frameshift==2){
    ii<-i-1
  }
  if(frameshift==3){
    ii<-i-2
  }
  if(ii%%3==1){
    # esse é o 1o do códon. juntar os 2 próximos
    cod.pol.list[[i]]<-paste(nucs1[1,i], nucs1[1,i+1], nucs1[1,i+2],
sep="")
    for(j in 2:length(nuc.pol.list[[i]])){
      # esse for corre o vetor dos nucs alternativos daquela posicao
      cod.pol.list[[i]][j]<-paste(nuc.pol.list[[i]][j], nucs1[1,i+1],
nucs1[1,i+2], sep="")
    }
  }
  if(ii%%3==2){
    # esse é o 2o do códon. juntar um antes e um depois
    cod.pol.list[[i]]<-paste(nucs1[1,i-1], nucs1[1,i], nucs1[1,i+1],
sep="")
    for(j in 2:length(nuc.pol.list[[i]])){
      # esse for corre o vetor dos nucs alternativos daquela posicao
      cod.pol.list[[i]][j]<-paste(nucs1[1,i-1],nuc.pol.list[[i]][j],
nucs1[1,i+1], sep="")
    }
  }
  if(ii%%3==0){
    # esse é o 3o do códon. juntar os 2 anteriores
    cod.pol.list[[i]]<-paste(nucs1[1,i-2], nucs1[1,i-1], nucs1[1,i],

```

```
sep="")
  for(j in 2:length(nuc.pol.list[[i]])){
    # esse for corre o vetor dos nucs alternativos daquela posicao
    cod.pol.list[[i]][j]<-paste(nucs1[1,i-2], nucs1[1,i-1],
nuc.pol.list[[i]][j], sep="")
  }
}
}
### Contando polimorfismos sinonimos ou nao sinonimos ###
Pn=0
Ps=0
for(i in polim){
  for(j in 2:length(nuc.pol.list[[i]])){
    if(cod.pol.list[[i]][j]%in%codon.table$codons==FALSE){
      cat("Ignorando polimorfismo em códon(s) truncado(s):",
cod.pol.list[[i]][j], "na posicao", i, "\n\n")
    }
    else {
      if (codon.table[codon.table$codons==cod.pol.list[[i]][1],2] !=
codon.table[codon.table$codons==cod.pol.list[[i]][j],2]){
        Pn<-Pn+1
      }
      if(codon.table[codon.table$codons==cod.pol.list[[i]][1],2] ==
codon.table[codon.table$codons==cod.pol.list[[i]][j],2]){
        Ps<-Ps+1
      }
    }
  }
}

### Lista de codons divergentes ###
cod.div.list<-list()
for (i in div){
  if(frameshift==1){
    ii<-i
  }
  if(frameshift==2){
    ii<-i-1
  }
  if(frameshift==3){
    ii<-i-2
  }
  if(ii%%3==1){
    # esse é o 1o do códon. juntar os 2 próximos
    cod.div.list[[i]]<-paste(nucs2[1,i], nucs2[1,i+1], nucs2[1,i+2],
sep="")
    cod.div.list[[i]][2]<-paste(nucs1[1,i], nucs1[1,i+1], nucs1[1,i+2],
sep="")
  }
  if(ii%%3==2){
```



```

    # esse é o 2o do códon. juntar um antes e um depois
    cod.div.list[[i]]<-paste(nucs2[1,i-1], nucs2[1,i], nucs2[1,i+1],
sep="")
    cod.div.list[[i]][2]<-paste(nucs1[1,i-1],nucs1[1,i], nucs1[1,i+1],
sep="")
  }
  if(ii%%3==0){
    # esse é o 3o do códon. juntar os 2 anteriores
    cod.div.list[[i]]<-paste(nucs2[1,i-2], nucs2[1,i-1], nucs2[1,i],
sep="")
    cod.div.list[[i]][2]<-paste(nucs1[1,i-2], nucs1[1,i-1], nucs1[1,i],
sep="")
  }
}
### Contando codons divergentes sinonimos e nao sinonimos ###
Dn=0
Ds=0
for(i in div){
  for(j in 1:2){
    if(cod.div.list[[i]][j]%in%codon.table$codons==FALSE){
      cat("Ignorando divergência em códon(s) truncado(s):",
cod.div.list[[i]][j], "na posicao", i, "\n\n")
    }
  }
  if (codon.table[codon.table$codons==cod.div.list[[i]][1],2] !=
codon.table[codon.table$codons==cod.div.list[[i]][2],2]){
    Dn<-Dn+1
  }
  if(codon.table[codon.table$codons==cod.div.list[[i]][1],2] ==
codon.table[codon.table$codons==cod.div.list[[i]][2],2]){
    Ds<-Ds+1
  }
}
}

saida2<-list()
saida2[[1]]<-paste("Pn =",Pn, " Ps =",Ps," Dn =", Dn," Ds =", Ds)
names(saida2)<- "Contagens"
### Índices
NI.MK<-(Pn/Ps)/(Dn/Ds)
DoS<-(Dn/(Dn+Ds))-(Pn/(Pn+Ps))
saida3<-list(NI.MK, DoS)
names(saida3)<-c("NI", "DoS")
saida<-c(saida1, saida2, saida3)
return(saida)
}

```

Arquivo da Função

[mk.test.r](#)

1)

talvez haja necessidade de usar `strsplit()` aqui! e um `unlist()`

2)

não há necessidade de juntar colunas em data frame dentro da função trate tudo com vetor a menos que a entrada já seja o data frame

3)

pelo que me lembre o formato phyllip tem o nome da espécie nas 10 primeiras posições das linhas. Use a função `strsplit()` para separar esses primeiros 10 caracteres. O resultado é uma lista e deve ser `unlist()` para ser uma vetor de dados ou classes como descreve em dois, depois use o `strsplit` novamente para a linha de caracteres restantes que corresponde a sequencia!!

4)

não há necessidade de tratá-los como fator! Como caracteres vai funcionar bem.

5)

pode fazer um teste lógico direto do vetor de caracteres tipo `sp1==sp2`, com isso ela comparará todas as bases da sequencia entre as espécies e retornará falso na posição em que as bases são diferentes, outra função importante é a `which()` que retorna a posição de verdadeiros de um teste lógico...

6)

ops! terão mais de uma sequencia para cada espécie?!Complica mas não muito, pense em uma matriz de caracteres para cada espécie e será possível saber onde há diferenças apenas com o testes lógicos “==”

7)

acho que deve fazer a saída usando o `list`, sendo o elementos na ordem que colocou um vetor, um data frame, um vetor? pode juntar com `l?`, outro data frame

From:

<http://ecor.ib.usp.br/> - **ecoR**

Permanent link:

http://ecor.ib.usp.br/doku.php?id=05_curso_antigo:alunos2012:alunos:trabalho_final:deboraycb:start



Last update: **2020/08/12 06:04**