Inference in Hidden Markov Models

```
library(knitr)
library(dbplyr)
library(ggplot2)
set.seed(42)
```

On rappelle le modèle des chaînes de markov cachés (HMM) on a des observations $x_1, \dots, x_n$ et une suite de variable latentes $z_1, \dots, z_n$ (états cachés) qui prennent des valeurs discrètes dans $\{1, \dots, K\}$. La loi jointe d'un modèle HMM s'écrit

$$p(x_1, \dots, x_n, z_1, \dots, z_n) = p(z_1) \prod_{i=2}^{n} p(z_i \mid z_{i-1}) \prod_{i=1}^{n} p(x_i \mid z_i) \tag{1}$$

La séquence des variables cachées $z_1, \dots, z_n$ est supposée suivre une *chaîne de markov* (d'où la terminologie HMM) avec:

1. Une loi initiale $z_1 \sim \pi$ telle que $\sum_k \pi_k = 1$.
2. Une matrice stochastique de transition $A \in \mathcal{M}_{K \times K}$, *i.e.* $p(z_{i+1} = l \mid z_i = k) = A_{kl}$.

Les *lois d'émissions des observations* $\Psi_i(k) = p(x_i \mid z_i = k)$ sont paramétriques, avec des paramètres θ, et peuvent être continues ou discrètes

- **discrète** $p(x_i = j \mid z_i = k) = B_{kj}$, $\theta = B$
- **continues**, par exemple en Gaussien $p(x_i \mid z_i = k) = \mathcal{N}(x_i \mid \mu_k, \sigma_k^2)$, $\theta = (\mu_k, \sigma_k^2)_k$

On notera donc un modèle HMM par le triplet (π, A, θ) qui le définit entièrement. On s'intéresse à trois problèmes fondamenteux liés aux HMMs:

1. Le calcul de la vraisemblance des observations $p(x_1, \dots, x_n \mid \pi, A, \theta)$.
2. Le calcul du maximum à posteriori ou *décodage* : $\text{argmax}_{z_1, \dots, z_n} p(z_1, \dots, zn \mid x_{1:n}; \pi, A, \theta)$.
3. L'estimation des paramètres par algorithme EM.

Les paramètres π, A et θ du modèles sont connus

C'est un cas idéalisé, mais c'est une première étape intéressante en soit, et nécessaire avant de parler d'estimation. On suppose qu'on observe une séquence $x_{1:n}$ du HMM.

Simulation d'un HMM

Coder une fonction `SimuHMMdisc(n, param)`qui simule une séquence (x_i, z_i) de taille n issue d'un HMM avec loi d'émission discrète paramétrée par une matrice B.

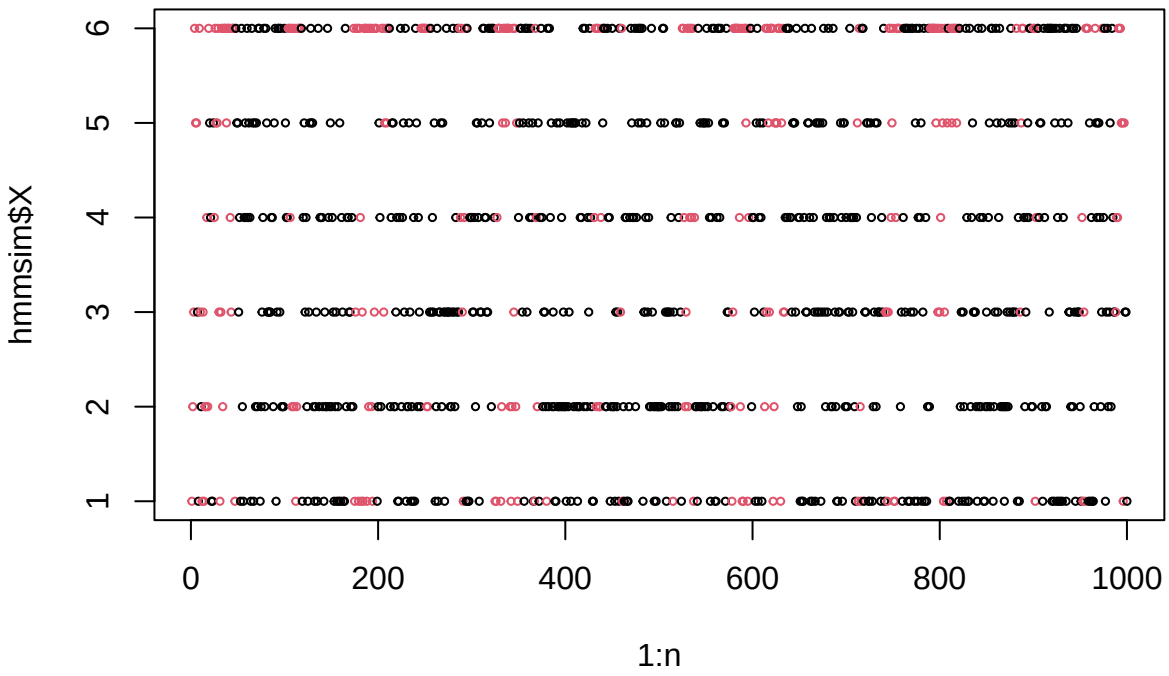
La tester avec le modèle suivant et faire un plot avec i en abscisse et x_i en ordonnée avec la couleur du point données par z_i.

```
n = 1e3
# Chaîne de Markov sur les états cachés
A <- matrix(c(0.95,0.1,0.05,0.9),2,2)

# Loi d'émission
```

```r
B <- t(matrix(c(rep(1/6,6),rep(1/10,5),5/10),6,2))

param = list()
param$pi = c()
param$A = A
param$pi = Re(eigen(t(param$A))$vector[,1]) # take the real-part
param$pi = param$pi / sum(param$pi)

param$B = B
```

```r
SimuHMMdisc = function(n, param) {

  Z<-rep(0,n) # hidden states
  X<-rep(0,n) # emission (obs.)
  K<-length(param$pi) # nb of hidden states
  J<-ncol(param$B)    # nb of modalities
  Z[1]<-sample(1:K,prob = param$pi,size = 1,replace=TRUE)
  X[1]<-sample(1:J,prob=param$B[Z[1],],size=1)
  for (i in 2:n){
    Z[i]<- sample(1:K,prob=param$A[Z[i-1],],size=1)
    X[i]<- sample(1:J,prob=param$B[Z[i],],size=1)
  }

  return(list(X=X,Z=Z))
}

hmmsim = SimuHMMdisc(n, param)
plot(x=1:n, y=hmmsim$X, col=hmmsim$Z, cex=.5, alpha=.5)
```

```
## Warning in plot.window(...): "alpha" n'est pas un paramètre graphique

## Warning in plot.xy(xy, type, ...): "alpha" n'est pas un paramètre graphique

## Warning in axis(side = side, at = at, labels = labels, ...): "alpha" n'est pas
## un paramètre graphique

## Warning in axis(side = side, at = at, labels = labels, ...): "alpha" n'est pas
## un paramètre graphique

## Warning in box(...): "alpha" n'est pas un paramètre graphique

## Warning in title(...): "alpha" n'est pas un paramètre graphique
```



Calcul de la vraisemblance : l'algorithme forward

On souhaite calculer $p(x_1, \ldots, x_n \mid \pi, A, \theta)$. Comme vu en cours, on peut la réécrire comme une marginalisation de la loi jointe sur les variables cachées (non-observées). On utilise ensuite la forme particulière de la loi jointe dans un modèle HMM données dans la première équation de ce document.

$$p(x_{1:n}) = \sum_{z_{1:n}} p(x_{1:n}, z_{1:n}), \tag{2}$$

$$= \sum_{z_1} \cdots \sum_{z_n} p(z_1) \prod_{i=2}^{n} p(z_i \mid z_{i-1}) \prod_{i=1}^{n} p(x_i \mid z_i) \tag{3}$$

Cette méthode "brute" nécessite de sommer sur n variables latentes pouvant prendre K valeurs chacunes, occasionant K^n opérations ! Cela deviendra rapidement infaisable pour des valeurs "raisonnables" de K et n.

Décomposition en sous-problèmes

On peut réécrire la vraisemblance comme la marginalisation sur la dernière variable z_n

$$p(x_{1:n}) = \sum_{k=1}^{K} p(x_{1:n}, z_n = k) = \sum_{k=1}^{K} \alpha_n(k) \tag{4}$$

On pose plus généralement pour une observation $i = 1, \ldots, n$ et un état $k = 1, \ldots, K$ le message:

$$\alpha_i(k) = p(x_{1:i}, z_i = k) \tag{5}$$

L'intérêt de cette réécriture est la récursion suivante qui lie $\alpha_i(k)$ à α_{i-1} :

$$\begin{aligned}
\alpha_i(k) &= p(x_{1:i}, z_i = k), && (6)\\
&= \sum_{l=1}^{K} p(x_{1:i}, z_{i-1} = l, z_i = k), && (7)\\
&= \sum_{l=1}^{K} p(x_{1:i-1}, x_i, z_{i-1} = l, z_i = k), && (8)\\
&= \sum_{l=1}^{K} p(x_i, z_i = k \mid x_{1:i-1}, z_{i-1} = l) p(x_{1:i-1}, z_{i-1} = l), && (9)\\
&= \sum_{l=1}^{K} p(x_i \mid z_i = k, x_{1:i-1}, z_{i-1} = l) p(z_i = k \mid x_{1:i-1}, z_{i-1} = l) p(x_{1:i-1}, z_{i-1} = l), && (10)\\
&= \sum_{l=1}^{K} p(x_i \mid z_i = k) p(z_i = k \mid z_{i-1} = l) p(x_{1:i-1}, z_{i-1} = l), && \text{(HMM model)}\\
& && (11)\\
&= \sum_{l=1}^{K} \Psi_i(k) A_{lk} \alpha_{i-1}(l), && (12)\\
& && (13)
\end{aligned}$$

Note: Les calculs jusqu'à l'avant dernière ligne sont toujours valables, sans aucune hypothèse sur le modèle. A l'avant dernière ligne on utilise le fait que dans un HMM, connaitre (conditionner par rapport à) z_{i-1} rend les x_i et z_i indépendant du passé.

Note2: On initialise la récursion avec $\alpha_1(l) = p(x_1, z_1 = l) = \pi_l \Psi_i(l), \forall l$.

En pratique, les $\alpha_i(k)$ sont des probas qui peuvent devenir très petites et on privilégie plutôt un codage en log, la récursion s'écrit alors :

$$\log \alpha_i(k) = \log \left(\sum_{l=1}^{K} \exp\{\log A_{lk} + \log \Psi_i(k) + \log \alpha_{i-1}(l)\} \right). \tag{14}$$

Un aparté: le "log-sum-exp" trick **R** n'a pas une précision de calcul infinie, par exemple

```
cat("En R, exp(-1000) = ", exp(-1000), '\n')
```

```
## En R, exp(-1000) =  0
```

```
cat("Le test d'égalité exp(-1000) == 0 renvoie", exp(-1000) == 0, '\n')
```

```
## Le test d'égalité exp(-1000) == 0 renvoie TRUE
```

```
cat("Le passage au log renvoie donc log(exp(-1000)) =", log(exp(-1000)), '\n')
```

```
## Le passage au log renvoie donc log(exp(-1000)) = -Inf
```

Pourtant, on sait que $\log(e^{-1000}) = -1000$!

Dans la récursion ci-dessus, on va calculer à chaque itération des quantités de type `log(sum(exp(a)))`. Si à l'itération $i-1$, les $\log \alpha_{i-1}(k)$ sont très négatifs (*i.e* les $\alpha_{i-1}(k)$ sont petits), alors à l'itération i il y a un risque de propagation des *underflow* numériques. En effet, on va calculer les $\sum_k \exp(\log \alpha_{i-1}(k) + y_{ik})$ où y_{ik} est lui-même négatif (log de probas). Chaque terme de la somme peut finir par dépasser la résolution numérique de **R**. Alors on aura la somme qui vaudra 0 et le log de la somme qui vaudra `-Inf`. Cela se propagera ensuite à toute les itération suivantes avec des `log(sum(exp(-Inf, ..., -Inf))) = -Inf`.

Il existe une façon de se prémunir de cette propagation. L'idée est d'utiliser l'identité suivant qui est vraie pour tout $m \in \mathbb{R}$

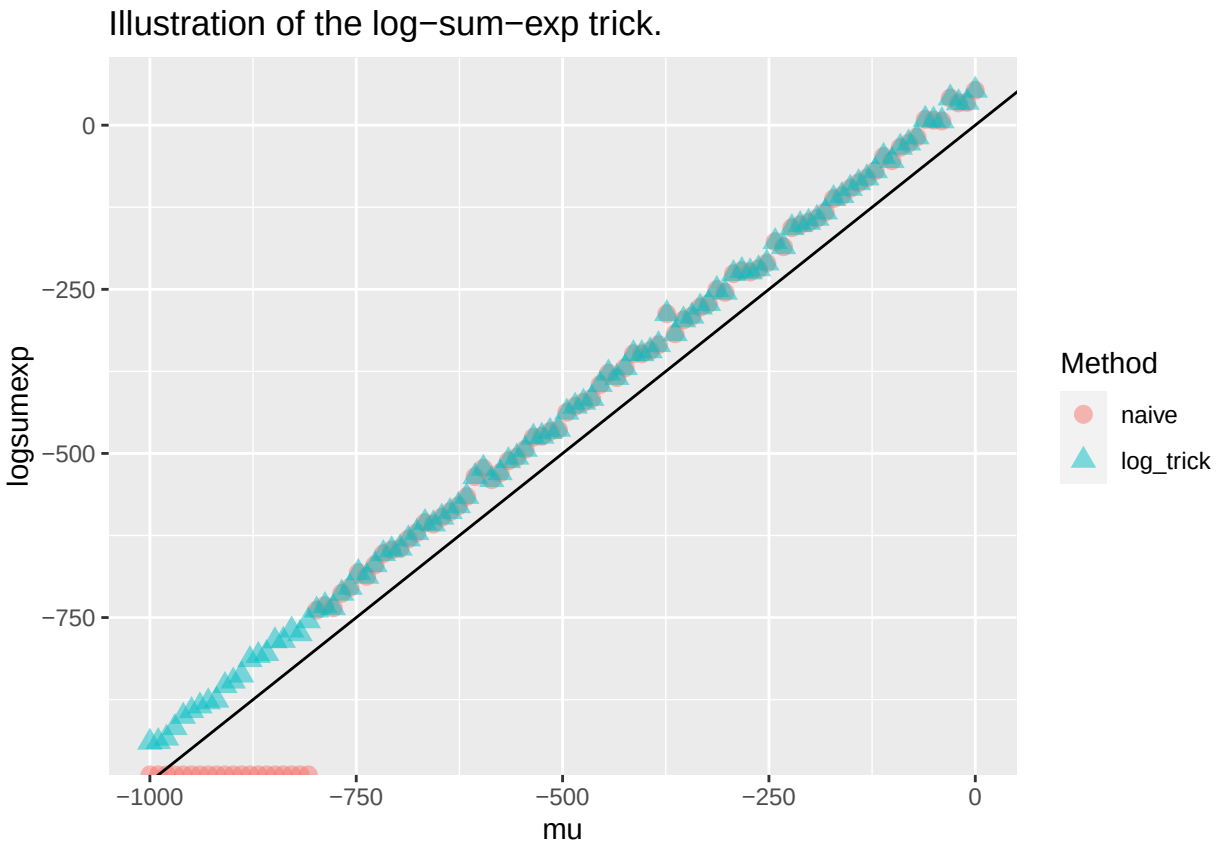
$$\log(\sum_k e^{a_k}) = m + \log(\sum_k e^{a_k - m}),$$

avec $m := \max_k a_k$. De cette manière, on s'assure de toujours avoir un $e^0 = 1$ quelque par dans la somme à l'intérieur du log.

```
logsumexp <- function(logx) {
  # compute \log(\sum exp(logx)) by rescaling it by m = \max(logx)
  # indeed : \log(\sum exp(logx)) = m + \log(\sum exp(logx - m))
  # This ensures an exp(0) somewhere in the sum
  m = max(logx)
  return(m + log(sum(exp(logx - m))))
}
```

Pour illustrer l'intérêt de cette méthode, on peut générer des points $a_1, \ldots, a_K$ selon une distribution $\mathcal{N}(\mu_e, 20)$, avec $\mu_1 = 0 < \ldots < \mu_E = -1000$. Au bout d'un certain seuil, les e^{a_k} vont underflow et l'intérêt du log-sum-exp trick apparait.

```
K = 200
nexp = 100 # taille de la sequence des $\mu_e$
mus = seq(0, -1000, length.out=nexp)
res = data.frame(naive=NA, log_trick=NA, mu = mus)
for (i in 1:length(mus)){
  a = rnorm(K, mean = mus[i], sd = 20)
  res$naive[i] = log(sum(exp(a)))
  res$log_trick[i] = logsumexp(a)
}

library(reshape2)
df = reshape2::melt(res, id='mu', variable.name = "Method", value.name = "logsumexp")
gg = ggplot(df) +
    geom_point(aes(x=mu, y=logsumexp, color=Method, shape=Method), size=3, alpha=0.5) +
    geom_abline(slope=1, intercept = 0) + # add y=x line
    ggtitle("Illustration of the log-sum-exp trick.")
print(gg)
```



Sur le graphique, on voit clairement que la méthode log trick donne toujours un nombre flottant tandis que la méthode naïve underflow un peu après $\mu_e < -750$.

```
tail(res) %>% kable()
```

	naive	log_trick	mu
95	-Inf	-891.9045	-949.4949
96	-Inf	-900.6153	-959.5960
97	-Inf	-917.8089	-969.6970
98	-Inf	-933.8207	-979.7980
99	-Inf	-939.0995	-989.8990
100	-Inf	-940.0257	-1000.0000

Bonus (en exercice) On remarque également que le résultat semble toujours au dessus de la droite $y = x$. Montrer que c'est vrai en espérance, *i.e.* $\mathbb{E}_{\mathbf{a}}\left[\log(\sum_k \exp(a_k))\right] > \mu_e$. (Une inégalité de Jensen pourrait s'avérer utile ici).

Questions Dans le cas d'une loi d'émission discrète avec les paramètres `param` défini au debut. Coder

1. une fonction `forward(x, i, param)` qui prend en entrée les observations x, un temps i et les paramètres π, A, θ, et calcule le vecteur des $\log \alpha_i$.

```
forward <- function(x, i, param) {

  K = length(param$pi)
  epsilon = 0 # 1e-300
  # init recursion, we add small epsilon to B to avoid
```

```
  # log(0)
  logalpha = log(param$pi) + log(param$B[,x[1]] + epsilon)

  # begin
  for (j in 2:i) {
    temp = logalpha
    # we could also use an apply(..., 1, logsumexp) on a well
    # chosen matrix.
    for (k in 1:K) {
      logalpha[k] = log(param$B[k,x[j]] + epsilon) +
        logsumexp(temp + log(param$A[,k]))
      # NEVER DO THIS
      # logalpha[k] = log(sum(exp(temp +
      #                       log(param$B[k,x[j]] + epsilon) +
      #                       log(param$A[,k])
      #                     )))
    }
  }
  return(logalpha)
}
```

L'illustrer dans le cadre suivant. Que remarquez vous entre les valeurs de $\log \alpha_{10}$ et le vrai z_{10} (accessible ici puisqu'on a simulé) ?

```
logalpha_10 = forward(hmmsim$X, i=10, param)
print(logalpha_10)
## [1] -18.53122 -19.77959
print(hmmsim$Z[10])
## [1] 2
```

2. Une fonction `HMMloglik(x, param)` qui calcule la log-vraisemblance de `x` sous un modèle HMM discret avec paramètres `param`. On rappelle :

$$\log p(x_{1:n}) = \log \sum_{k=1}^{K} p(x_{1:n}, z_n = k) = \log \sum_{k=1}^{K} \exp(\log \alpha_n(k))$$

```
HMMloglik = function(x, param) {
  n = length(x)
  # again, we use the logsumexp trick to compute
  # the log-likelihood
  return(logsumexp(forward(x, n, param)))
}
HMMloglik(hmmsim$X, param)
```

```
## [1] -1756.867
```

Decoding : the Viterbi algorithm

Note de cours

Le décodage consiste à trouver la séquence de variables cachées $\hat{z}_{1:n}$ la plus probable étant donnés les paramètres et les observations $x_{1:n}$.

$$\hat{z}_{1:n} = \arg\max_{z_{1:n}} \log p(z_{1:n} \mid x_{1:n}\,; \theta, \pi, A)$$

C'est donc du maximum a posteriori ! Comme l'espace de recherche des $z_{1:n}$ est discret, une énumération serait possible mais trop coûteuse : K^n possibilités ! Encore une fois, une décomposition en sous problème simple va s'avérer utile. L'algorithme de Vitberbi permet de résoudre le problème de décodage à l'aide de manière séquentielle. Si on connaissait la meilleure séquence $\hat{z}_{1:(n-1)}$ on pourrait facilement résoudre $\arg\max_{z_n} p(\hat{z}_{1:(n-1)}, z_n, x_{1:n} | \theta, \pi, A)$.

Définissons $V_i(k) = \max_{z_{1:(i-1)}} p(x_{1:i}, z_i = k, z_{1:(i-1)} \mid \theta, \pi, A)$, alors on peut montrer que V_i vérifie la récursion suivante :

$$\begin{aligned} V_1(k) &= \pi_1 \Psi_i(k), \\ V_i(k) &= \Psi_i(k) \cdot \max_{l=1,\dots,K} V_{i-1}(l) A_{lk}, \end{aligned} \tag{15}$$

On calcule les $V_i(k)$ pour tout les $i = 1, \dots, n$, et on trouve $\hat{z}_n$ grace à $\hat{z}_n = \arg\max_k V_n(k)$. On peut ensuite reconstruire la séquence $\hat{z}_{n-1}$ jusqu'à $\hat{z}_1$ de proche en proche. Pour retrouver $\hat{z}_{i-1}$ on prend l'état qui a donné la transition $\hat{z}_{i-1} \to \hat{z}_i$ la plus probable (conditionnellement aux observations $x_{1:n}$). C'est-à-dire l'argmax sur les transitions possibles : $S_i(k) = \arg\max_{l=1,\dots,K} V_{i-1}(l) A_{lk}$

$$\hat{z}_{i-1} := S_i(\hat{z}_i)$$

Note : ici, l'opérateur max joue un rôle analogue à l'opérateur $\sum$ pour la marginalisation sur les variables z. En réalité, l'algorithme de Viterbi est un cas particulier de l'algorithme "max-product" pour calculer les modes qui est exacte pour les arbres. Plus d'information ici par exemple

Questions

1. Montrer que le problème de maximisation de Viterbi peut se réécrire en fonction de la loi jointe

$$\arg\max_{z_{1:n}} \log p(z_{1:n} \mid x_{1:n}; \theta, \pi, A) = \arg\max_{z_{1:n}} \log p(z_{1:n}, x_{1:n} \mid \theta, \pi, A)$$

2. Montrer que dans un modèle HMM on a bien l'identité :

$$\max_{z_{1:(i-1)}} p(x_{1:i}, z_i = k, z_{1:(i-1)} \mid \theta, \pi, A) = \Psi_i(k) \cdot \max_{l=1,\dots,K} V_{i-1}(l) A_{lk}$$

3. Programmer une fonction `Viterbi(x, param)` qui renvoie l'estimateur du MAP $\hat{z}_{1:n}$.

Astuce: le log est une fonction croissante donc $\log(\max f(x)) = \max \log(f(x))$, on peut (et il faut) donc coder les quantités précédentes en log-space.

```
Viterbi<-function(x, param){
  epsilon<-1e-6
  K<-nrow(param$A)
  n<-length(x)
  S<-matrix(0,K,n)
  logV<-matrix(-Inf,K,n)
  Zest<-rep(0,n)
  for (k in 1:K){
    logV[k,1]<-log(param$B[k, x[1]]+epsilon)+log(param$pi[k])
  }
  # Forward
  for (t in (2:n))
    for (k in (1:K)){
      logV[k,t]=max(logV[,t-1]+log(param$A[,k])+log(param$B[k, x[t]]))
      S[k,t-1]=which.max(logV[,t-1]+log(param$A[,k])+log(param$B[k, x[t]]))
    }
```

```
  # Back-tracking
  Zest[n]<-which.max(logV[,n])
  for (t in (n-1):1)
    Zest[t]<-S[Zest[t+1],t]

  return(Zest)
}
```

4. Tester votre fonction sur le code suivant

```
z_hat = Viterbi(hmmsim$X, param)
n_wrong = sum(z_hat != hmmsim$Z)
cat('Proportion of wrong states : ', n_wrong / length(hmmsim$X))
```

```
## Proportion of wrong states :  0.204
```

Estimation : l'algorithme de Baum-Welch (EM pour les HMMs)

Jusqu'ici on a supposé les paramètres $\eta = (\pi, A, \theta)$ connus, ce qui n'est pas très réaliste en pratique. Dans cette dernière partie on s'intéresse à l'estimation des paramètres du modèle à l'aide des observations $x_1, \dots, x_n$.

Etant donné que les HMMs sont des modèles à variables latentes, on peut utiliser l'algorithme EM, qui s'appelle historiquement l'algorithme de *Baum-Welch*.

- Initialisation de $\eta^{(0)} = (\pi^{(0)}, A^{(0)}, \theta^{(0)})$
- Pour $t = 0, \dots, T$ itérer entre
  - **E-step** (lissage ou *smoothing*) Calcul de borne inférieure

$$\begin{aligned} Q_t(\eta) &= E_{z_{1:n} \sim p(\cdot \mid x_{1:n}, \eta^{(t)})}[\log p(x, z \mid \eta)], \\ &= \sum_k \tau_1(k) \log \pi_k + \sum_{i=1}^{n-1} \sum_{k,l=1}^{K} \xi_{i,i+1}(k,l) \log A_{k,l} + \sum_{i=1}^{n} \tau_i(k) \log \Psi_i(k) \end{aligned} \tag{16}$$

  - Cela revient à calculer les quantités
    1. $\tau_i(k) = p(z_i = k \mid x_{1:n}; \eta^{(t)})$,
    2. $\xi_{i,i+1}(k,l) = p(z_i = k, z_{i+1} = l \mid x_{1:n}; \eta^{(t)})$

    à l'aide de l'algorithme forward-backward.
  - **M-step** Maximisation de la borne en η

$$\eta^{(t+1)} := \arg\max_{\pi, A, \theta} Q_t(\pi, A, \theta)$$

    Cette étape dépend de la loi des émissions !
  - STOP si $l(\eta^{(t+1)}) - l(\eta^{(t)}) < \epsilon$

Algorithme forward-backward pour le lissage (E-step)

L'algorithme forward-backward permet le calcul des lois marginales à posteriori des $z_i \mid x_{1:n}$ et des jointes $(z_i, z_{i+1}) \mid x_{1:n}$. Il procède en deux phases qui consistent à parcourir la chaîne dans le sens temporel (*forward*), puis dans le sens inverse (*backward*).

L'idée principale est que l'on peut séparer la chaîne d'un HMM en 2 parties (passé & futur) conditionellement à z_i :

$$p(z_i = k \mid x_{1:n}) \propto p(z_i = k, x_{1:i})\, p(x_{(i+1):n} \mid z_i = k) = \frac{\alpha_i(k)\beta_i(k)}{p(x_{1:n})}.$$

On définit donc les 2 quantités

- **Forward** $\alpha_i(k) = p(z_i = k, z_{1:i}) = \Psi_i(k) \sum_l A_{lk} \alpha_{i-1}(l)$ (cf. première partie)

- **Backward** $\beta_i(k) := p(x_{(i+1):n} \mid z_i = k) = \sum_{l=1}^{K} A_{kl}\Psi_{i+1}(l)\beta_{i+1}(l)$ (cf. slides de cours) avec l'initialisation $\beta_n(k) = 1, \forall k$.

On pourra calculer les $\tau_i(k)$ et $\xi_{i,i+1}(k,l)$ grâce aux formules

$$\begin{aligned} \tau_i(k) &\propto \alpha_i(k)\beta_i(k), \\ \xi_{i,i+1}(k,l) &\propto \alpha_i(k)\Psi_{i+1}\beta_{i+1}(l)A_{kl}. \end{aligned} \tag{17}$$

Note On va encore coder en log-space et pouvoir utiliser le log-sum-exp pour assurer la stabilité numérique au fil de la récursion !

Questions

0. **(facultatif)** Vérifier la formule de récursion pour $\beta_i(k)$ et les formule liant τ_i et $\xi_{i,i+1}$ à α et β (cf. slides).

Nous avons déjà implémenté la partie forward de l'algorithme qui calcule les $\log \alpha_i(k)$ pour un i donné. On voit qu'on aura maintenant besoin des $\log \alpha_i$ pour tout i.

1. Modifier `forward(x, i, param)` en une fonction `forward_estep(x, param)` qui retourne la matrice $K \times n$ des $(\log \alpha_i(k))_{k,i}$.

```
forward_estep <- function(x, param) {

  n = length(x)
  K = length(param$pi)
  epsilon = 0 # 1e-300
  logalpha = matrix(0, K, n)
  # init recursion, we add small epsilon to B to avoid
  # log(0)
  logalpha[,1] = log(param$pi) + log(param$B[,x[1]] + epsilon)

  # begin
  for (j in 2:n) {
    # we could also use an apply(..., 1, logsumexp) on a well
    # chosen matrix.
    for (k in 1:K) {
      logalpha[k, j] = log(param$B[k,x[j]] + epsilon) +
        logsumexp(logalpha[,j-1] + log(param$A[,k]))
    }
  }
  return(logalpha)
}
lalpha = forward_estep(hmmsim$X, param)
```

2. Implémenter `backward(x, param)` qui renvoie la matrice $K \times n$ des $(\log \beta_i(k))_{k,i}$.

```
backward <- function(x, param) {

  n = length(x)
  K = length(param$pi)
  epsilon = 0 # 1e-300
  logbeta = matrix(0, K, n)
  # init recursion log(beta_n) = log(1) = 0
  logbeta[,n] = 0

  # begin
  for (j in (n-1):1) {
```

```
    # we could also use an apply(..., 1, logsumexp) on a well
    # chosen matrix.
    for (k in 1:K) {
      logbeta[k, j] = logsumexp(
          log(param$B[,x[j+1]] + epsilon) +
          logbeta[,j+1] +
          log(param$A[k,])
        )
    }
  }
  return(logbeta)
}

lbeta = backward(hmmsim$X, param)
```

3. Implémenter une fonction `estep(x, param)` qui fait le lissage et renvoie les $\tau_i(k)$ (matrice $K \times V$ et $\xi_{i,i+1}(k,l)$ (tenseur $K \times K \times (n-1)$). Renvoyer la log-vraisemblance du modèle également.

```
estep = function(x, param) {

  n = length(x)
  K = length(param$pi)
  epsilon = 0 # 1e-300

  tau = matrix(0, K, n)
  xi = array(0, c(K, K, n-1))

  lalpha = forward_estep(x, param)
  lbeta = backward(x, param)

  # compute the likelihood (for free since the logalpha are
  # already computed)
  loglik = logsumexp(lalpha[,n])

  for (i in 1:n) {

    # normalize tau in logspace, the denominator is always
    # the same : p(x_{1:n}), i.e. loglik.
    ltau = lalpha[,i] + lbeta[, i]
    #sanity check
    if(abs(logsumexp(ltau) - loglik) > 1e-8) stop('Pb in ltau')

    ltau = ltau - loglik
    tau[, i] = exp(ltau)

    #
    if (i < n) {
      lxi = matrix(0, K, K)
      for(k in 1:K) {
        for (l in 1:K) {
          lxi[k, l] =  lalpha[k,i] +
            log(param$B[l, x[i+1]] + epsilon) +
            lbeta[l, i+1] +
            log(param$A[k,l] + epsilon)
```

```
        }
      }
      # sanity check
      if(abs(logsumexp(c(lxi)) - loglik) > 1e-8) stop('Pb in lxi')
      # normalize in log-space with loglik
      lxi = lxi - loglik
      xi[,,i] = exp(lxi)
    }
  }



  return(list(tau=tau, xi=xi, loglik=loglik))
}
smooth_post = estep(hmmsim$X, param)
stopifnot(smooth_post$loglik==HMMloglik(hmmsim$X, param))
```

4. Programmer une fonction `mstep(x, smoothed_post)` qui calcule $\eta^{(t+1)}$ dans un HMM à émission discrète.

```
mstep = function(x, smoothed_post) {

  J = length(unique(x))
  K = nrow(smoothed_post$tau)

  param_est = list()
  param_est$pi = smoothed_post$tau[,1]
  param_est$A = apply(smoothed_post$xi, c(1,2), sum)
  param_est$A = param_est$A / rowSums(param_est$A)
  # sanity check for normalization of A
  if(! all(rowSums(param_est$A) - rep(1, K) < 1e-8)){
    stop("Problem in normalization of A.")
  }

  symbols = sort(unique(x))
  param_est$B = matrix(0, K, J)
  for (k in 1:K) {
    nk =  sum(smoothed_post$tau[k,])
    for (j in 1:J) {
      inds = which(x == symbols[j])
      param_est$B[k,j] = sum(smoothed_post$tau[k, inds])  / nk
    }
  }
  # sanity check for normalization of B
  if(! all(rowSums(param_est$B) - rep(1, K) < 1e-8)){
    stop("Problem in normalization of B.")
  }

  return(param_est)
}

eta = mstep(hmmsim$X, smooth_post)
rowSums(eta$A)
```

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

```
rowSums(eta$B)
```

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

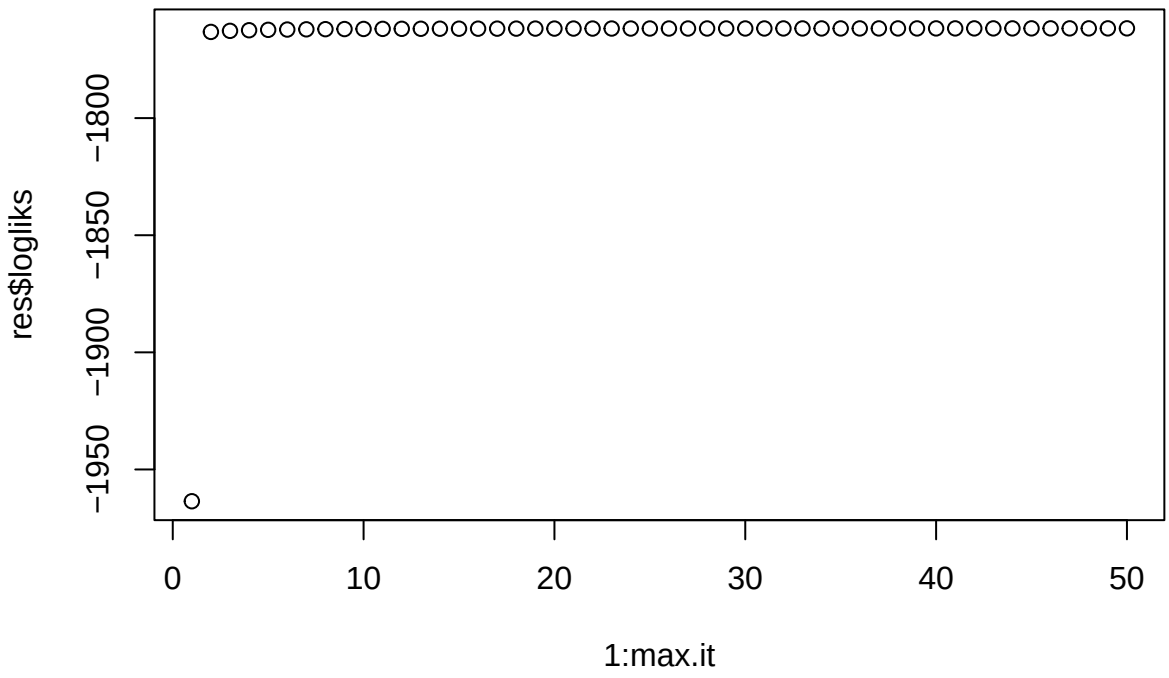
5. Programmer une fonction `baum_welch(x, init_param, max.it, atol)` qui retourne un estimateur $\hat{\eta}$ et le vecteur stockant l'évolution de la log-vraisemblance dans l'algorithme.

```
baum_welch = function(x, init_param, max.it, atol) {
  logliks = rep(NA, max.it)
  param = init_param
  old_loglik = -Inf

  for(t in 1:max.it) {
    #E-step : smoothing
    smooth_post = estep(x, param)
    logliks[t] = smooth_post$loglik

    cat("------- Iteration", t - 1, " - loglik =", logliks[t], '\n')
    if (logliks[t] < old_loglik) stop("Decreasing log-likelihood !")

    if(abs(logliks[t] - old_loglik) < atol) break

    #M-step : update
    param = mstep(x, smoothed_post = smooth_post)
    old_loglik = logliks[t]
  }
  return(list(param=param, logliks=logliks))
}
```

La tester avec le code ci-dessous et vérifier qu'elle est bien monotone croissante. Essayer l'initialisation commentée (petite perturbation des paramètres). Essayez votre propre initialisation. Que constatez vous sur l'impact de l'initialisation ?

```
# random init (bad idea)
init_param = list()
K = length(param$pi)
J = length(unique(hmmsim$X))
init_param$pi = rep(1/K, K)
init_param$A = abs(matrix(rnorm(n = K^2, sd = 3), K, K))
init_param$A = init_param$A / rowSums(init_param$A)
init_param$B = abs(matrix(rnorm(n = K * J, sd = 3), K, J))
init_param$B = init_param$B / rowSums(init_param$B)

# small perturbation from initial param
# init_param = param
# # add small noise to the true parameter
# init_param$A = init_param$A +
#   matrix(rnorm(prod(dim(init_param$A)), 0, 5e-3),
#          nrow(init_param$A),
#          ncol(init_param$A))
# init_param$A = abs(init_param$A)
# init_param$A = init_param$A / rowSums(init_param$A)

max.it=50
```

```
res = baum_welch(hmmsim$X, init_param, max.it = max.it, atol=1e-6)
## ------- Iteration 0  - loglik = -1963.566
## ------- Iteration 1  - loglik = -1763.209
## ------- Iteration 2  - loglik = -1762.76
## ------- Iteration 3  - loglik = -1762.496
## ------- Iteration 4  - loglik = -1762.323
## ------- Iteration 5  - loglik = -1762.201
## ------- Iteration 6  - loglik = -1762.111
## ------- Iteration 7  - loglik = -1762.043
## ------- Iteration 8  - loglik = -1761.991
## ------- Iteration 9  - loglik = -1761.95
## ------- Iteration 10  - loglik = -1761.917
## ------- Iteration 11  - loglik = -1761.89
## ------- Iteration 12  - loglik = -1761.868
## ------- Iteration 13  - loglik = -1761.85
## ------- Iteration 14  - loglik = -1761.835
## ------- Iteration 15  - loglik = -1761.823
## ------- Iteration 16  - loglik = -1761.812
## ------- Iteration 17  - loglik = -1761.802
## ------- Iteration 18  - loglik = -1761.794
## ------- Iteration 19  - loglik = -1761.787
## ------- Iteration 20  - loglik = -1761.78
## ------- Iteration 21  - loglik = -1761.775
## ------- Iteration 22  - loglik = -1761.769
## ------- Iteration 23  - loglik = -1761.765
## ------- Iteration 24  - loglik = -1761.76
## ------- Iteration 25  - loglik = -1761.756
## ------- Iteration 26  - loglik = -1761.752
## ------- Iteration 27  - loglik = -1761.749
## ------- Iteration 28  - loglik = -1761.745
## ------- Iteration 29  - loglik = -1761.742
## ------- Iteration 30  - loglik = -1761.739
## ------- Iteration 31  - loglik = -1761.736
## ------- Iteration 32  - loglik = -1761.733
## ------- Iteration 33  - loglik = -1761.73
## ------- Iteration 34  - loglik = -1761.727
## ------- Iteration 35  - loglik = -1761.725
## ------- Iteration 36  - loglik = -1761.722
## ------- Iteration 37  - loglik = -1761.719
## ------- Iteration 38  - loglik = -1761.717
## ------- Iteration 39  - loglik = -1761.714
## ------- Iteration 40  - loglik = -1761.712
## ------- Iteration 41  - loglik = -1761.709
## ------- Iteration 42  - loglik = -1761.707
## ------- Iteration 43  - loglik = -1761.704
## ------- Iteration 44  - loglik = -1761.702
## ------- Iteration 45  - loglik = -1761.7
## ------- Iteration 46  - loglik = -1761.697
## ------- Iteration 47  - loglik = -1761.695
## ------- Iteration 48  - loglik = -1761.692
## ------- Iteration 49  - loglik = -1761.69
plot(1:max.it, res$logliks)
```




```
print(sum(abs(param$A - res$param$A)))
## [1] 1.387464
param$A - res$param$A
##            [,1]       [,2]
## [1,]  0.5754177 -0.5754177
## [2,] -0.1183142  0.1183142
init_param$A
##           [,1]      [,2]
## [1,] 0.4365755 0.5634245
## [2,] 0.1499384 0.8500616
res$param$A
##           [,1]      [,2]
## [1,] 0.3745823 0.6254177
## [2,] 0.2183142 0.7816858
```

Pour aller plus loin

Refaire le TP avec des lois d'émissions Gaussiennes (variance $\sigma^2 = 1$ connu et fixée, on n'estime que les moyennes). Que faudra-t-il modifier finalement ?