Développement d'une librairie pour les Modèles de Markov Cachés en mémoire linéaire

Jérôme Hellinckx

Université Libre de Bruxelles, Belgique
jerhelli@ulb.ac.be

Abstract

Un des problèmes récurrent auquel l'utilisation extensive des Modèles de Markov Cachés fait face est une demande conséquente d'espace mémoire qui est directement dépendante de la taille des données d'entrée. C'est dans ce cadre que des algorithmes en mémoire linéaire furent proposés ces dernières années. Dans cet article, nous définissions brièvement les HMMs et leurs algorithmes ainsi que leurs équivalents en mémoire linéaire décrits par Churbanov and Winters-Hilt (2008); Lam and Meyer (2010); Miklós and Meyer (2005). Nous proposons également une librairie C++ permettant de construire et d'employer de tels modèles respectant la contrainte sur la mémoire et nous démontrons son fonctionnement.

Introduction

Les Modèles de Markov Cachés (HMMs) sont des modèles probabilistes servant d'outil de modélisation et dont l'utilisation se retrouve dans des domaines nombreux et variés allant de la reconnaissance vocale à la bioinformatique pour l'analyse de séquences biologiques (Yoon, 2009). Avoir recours aux algorithmes conventionnels permettant de résoudre les problèmes relatifs aux HMMs (ces problèmes sont décrits plus loin) n'est malheureusement pas toujours une solution acceptable étant donné que ces algorithmes procèdent à une consommation de la mémoire très copieuse pour de longues séquences (Churbanov and Winters-Hilt, 2008). Dans l'optique de pallier à cela, des propositions d'algorithmes moins gourmands en mémoire ont vu le jour parmi la littérature scientifique. Mentionnons par exemple Miklós and Meyer (2005) qui ouvre le chemin vers une implémentation en mémoire linéaire de l'algorithme d'entrainement Baum-Welch ou encore Churbanov and Winters-Hilt (2008) qui montre comment implémenter le fameux problème de décodage en respectant mieux l'allocation de la mémoire.

Cet article a pour objectif d'une part de récapituler le travail existant traitant de l'utilisation plus efficiente de la mémoire par les algorithmes des HMMs et d'autre part de proposer une librairie C++ implémentant ces travaux. Pour réaliser cela, nous commençons par une rapide rappel sur les HMMs et leurs notations en poursuivant par la description des problèmes typiques qu'ils rencontrent. Nous introduisons alors l'équivalent des résolutions de ces problèmes en mémoire linéaire en expliquant en parallèle les choix faits par la librairie permettant leur implémentation. Nous finissions en dévoilant les résultats obtenus en comparant avec les résultats des travaux dont nous nous sommes inspirés.

Modèles de Markov Cachés

Définition

Un Modèle de Markov Caché, dénoté HMM dans cet article, est un modèle probabiliste qui peut être utilisé pour décrire l'évolution d'événements observables qui sont issus de facteurs internes non-observables. Les événements observables sont appelés *symboles* et les facteurs invisibles *états* (Yoon, 2009). Un HMM peut alors être décrit comme une machine à état fini stochastique pour laquelle chaque transition entre états cachés se termine par une émission de symbole, où la transition ainsi que l'émission dépendent de distributions de probabilités (Churbanov and Winters-Hilt, 2008). Nous introduisons maintenant les notations utilisées dans cet article pour les paramètres décrivant l'implémentation conventionnelle d'un HMM (Churbanov and Winters-Hilt, 2008) :

- Un ensemble d'états $S = \{S_1 \dots S_N\}$ avec q_t l'état visité à l'instant t ;
- Un ensemble de symboles $\mathcal{O} = \{o_1 \dots o_M\}$, appelé alphabet, avec o_t le symbole émit à l'instant t ;
- Un ensemble de fonctions de densité de probabilité $B = \{b_1(o) \dots b_N(o)\}$, décrivant les probabilités d'émission $b_j(o_t)$ qui est la probabilité que l'état visité à l'instant t, $q_t = S_j$ émette le symbole o pour $1 \leq j \leq N$, $o_t \in \mathcal{O}$;
- Une matrice de probabilités de transition entre états $A = \{a_{i,j}\}$ pour $1 \leq i, j \leq N$, où $a_{i,j} = P(q_{t+1} = S_j \mid q_t = S_i)$;
- Un vecteur de distributions initiales d'états $\Pi = \{\pi_1, \dots, \pi_N\}$.

L'ensemble de ces paramètres, dénoté Θ, définit entièrement un HMM. Notons aussi $O = \{o_1 \dots o_T\}$, une séquence de

symboles avec $o_t \in \mathcal{O}$ pour $1 \leq t \leq T$.

Les trois problèmes basiques d'un HMM

Afin de rendre plus aisées les explications sur les méthodes employées par la librairie pour proposer des HMMs en mémoire linéaire, nous expliquons ci-dessous les trois problèmes basiques des HMMs. Nous partirons alors des notions et formules introduites pour éclairer notre implémentation en mémoire linéaire. Cette section fait donc principalement office de rappel et n'a que très peu d'utilité pour un lecteur bio-informaticien aguerri.

Problème d'évaluation Soit un HMM paramétrisé par Θ, quelle est la probabilité d'observer la séquence O, $P(O \mid \Theta)$? Il existe deux algorithmes de programmation dynamique permettant de calculer efficacement cette probabilité. D'une part le *forward algorithm* qui définit une *forward variable* $\alpha_t(i) = P(o_1 \dots o_t \mid q_t = S_i, \Theta)$ déterminant donc la probabilité d'observer $o_1 \dots o_t$ et d'arriver à $q_t = S_i$. Cette variable est obtenue par la formule récursive

$$\alpha_t(i) = \begin{cases} \pi_i b_i(o_1), & t = 1 \\ \left[\sum_{j=1}^{N} \alpha_{t-1}(j) a_{j,i}\right] b_i(o_t), & t = 2 \dots T \end{cases} \tag{1}$$

D'autre part le *backward algorithm*, algorithme analogue mais fonctionnant dans le sens inverse, définit une *backward variable* $\beta_t(i) = P(o_{t+1} \dots o_T \mid q_t = S_i, \Theta)$ qui détermine donc la probabilité d'observer $o_t \dots o_T$ après être arrivé à $q_t = S_i$. Elle se calcule aussi de manière récursive

$$\beta_t(i) = \begin{cases} 1, & t = T \\ \sum_{j=1}^{N} a_{i,j} b_j(o_{t+1}) \beta_{t+1}(j), & t = T-1 \dots 1 \end{cases} \tag{2}$$

Nous trouvons alors la probabilité d'observation de la séquence O, $P(O \mid \Theta)$, en additionnant les $\alpha_T(i), \forall i \in S$ pour le *forward algorithm*. Pour le *backward algorithm*, il faut encore considérer la probabilité initiale $\pi(i)$ et la probabilité d'émission de o_1, il faut donc effectuer la somme de $\pi(i) b_i(o_1) \beta_1(i), \forall i \in S$.

Problème de décodage Soit une séquence de symboles O et un HMM paramétrisé par Θ, quel est le chemin d'états permettant de maximiser la probabilité d'observation de O, $Q^* = \max_Q P(Q \mid O, \Theta)$? Intuitivement, nous cherchons donc la séquence d'états optimale, celle qui expliquerait *au mieux* les symboles observés. Introduisons le *Viterbi algorithm*, permettant de trouver un tel chemin (souvent dénoté chemin Viterbi). Cet algorithme définit deux variables, $\phi_t(i)$ qui est le score maximum le long d'un chemin d'états $q_1 \dots q_{t-1}$ arrivant à l'état i en q_t et émettant les symboles $o_1 \dots o_t$, et $\psi(i)$, qui est l'état j en $t-1$ pour lequel $\phi_{t-1}(j) a_{j,i}$ est maximal, soit formellement

$$\begin{aligned} \phi_t(i) &= \begin{cases} \pi_i b_i(o_1), & t = 1 \\ \max\limits_{1 \leq j \leq N} \left[\phi_{t-1}(j) a_{j,i}\right] b_i(o_t) & t = 2 \dots T \end{cases} \\ \psi_t(i) &= \begin{cases} 0, & t = 1 \\ \operatorname{argmax}\limits_{1 \leq j \leq N} \left[\phi_{t-1}(j) a_{j,i}\right] & t = 2 \dots T \end{cases} \end{aligned} \tag{3}$$

La terminaison trouve alors q_T^*, le dernier état du chemin optimal, en prenant le maximum des $\phi_T(i)$. Pour trouver les autres états $q_1^* \dots q_{T-1}^*$ afin d'obtenir Q^*, il suffit d'effectuer un retour en arrière sur ψ, où $q_t^* = \psi_{t+1}(q_{t+1}^*)$.

Problème d'entrainement Soit un ensemble de séquences de symboles observés $X = \{O_1 \dots O_K\}$, comment choisir la paramétrisation $\Theta = (S, B, A, \pi)$ d'un HMM représentant X ? Il existe actuellement une large littérature pour estimer les paramètres de Modèles de Markov Cachés et il serait inutile d'en faire la liste exhaustive. Nous nous limitons donc aux algorithmes pour lesquels une approche en mémoire linéaire a déjà été décrite, étant donné qu'une telle approche est celle adoptée par la librairie proposée dans cet article.

Tous les algorithmes d'entrainement que nous développons ici sont des algorithmes itératifs, en ce qu'ils réestiment l'ensemble de paramètres Θ du HMM à chaque itération n en utilisant le Θ obtenu pendant l'itération $n-1$ où la réestimation pour chaque itération n dépend de l'algorithme d'entrainement spécifique utilisé et de l'ensemble de séquences d'entrainement X. Notons que le nombre d'itérations dépend d'une valeur ϵ (donnée au préalable) qui indique l'*amélioration* minimale de la probabilité d'observation de X à chaque réestimation de Θ, les itérations cessent donc dès que l'amélioration est $\leq \epsilon$ (ou qu'un nombre maximum d'itérations est atteint).

Entraînement Viterbi Le premier algorithme d'entrainement que nous introduisons est l'entrainement Viterbi, qui, à chaque itération n, calcule le chemin d'états optimal (voir ci-dessus) pour chaque $O \in X$. L'algorithme compte alors les émissions $b_i(o_t)$ et les transitions $a_{i,j}$ dans ces chemins optimaux et en dérive la réestimation de Θ en normalisant les comptes (Lam and Meyer, 2010). Soit $A_{i,j}^n$ le nombre de transitions de i à j et $B_i^n(o)$ le nombre d'émissions de o par i dans les chemins optimaux $Q_1^* \dots Q_K^*$ pour l'itération n :

$$\begin{aligned} a_{i,j}^{n+1} &= \frac{A_{i,j}^n}{\sum_{k=1}^{N} A_{i,k}^n} \\ b_i^{n+1}(o) &= \frac{B_i^n(o)}{\sum_{o' \in \mathcal{O}} B_i^n(o')} \end{aligned} \tag{4}$$

Le même raisonnement est applicable au paramètre π. Remarquons que l'implémentation *normale* de l'entrainement

Viterbi requiert de garder en mémoire le retour en arrière ψ afin d'effectuer les comptes pour chaque séquence d'entrainement $O \in X$.

Entraînement Baum-Welch Alors que l'entrainement Viterbi ne considère qu'un seul chemin d'états pour une séquence d'entrainement $O \in X$ donnée, l'entrainement Baum-Welch se sert de tous les chemins possibles pour calculer la nouvelle valeur des paramètres. L'algorithme de Baum-Welch typique se sert ainsi des algorithmes *forward* et *backward* afin de tenir compte de tous les chemins. Soit $A^n_{i,j}(O)$, définie par Miklós and Meyer (2005) comme la somme pondérée des probabilités des chemins d'états de la séquence O contenant au moins une transition de i à j sachant que la pondération de chacun de ces chemins est le nombre de fois que cette transition se produit le long de ce chemin et $B^n_i(o, O)$ la somme analogue pour les émissions, soit

$$\begin{aligned} A^n_{i,j}(O) &= \sum_{t=1}^{T} \alpha^n_t(i) a^n_{i,j} b^n_j(o_{t+1}) \beta^n_{t+1}(j) \\ B^n_i(o, O) &= \sum_{t=1}^{T} \delta_{o_t,o} \alpha^n_t(i) b^n_i(o_t) \end{aligned} \tag{5}$$

où $\delta_{o_t,o} = 1$ si $o = o_t$, 0 sinon. Les nouvelles valeurs de Θ sont alors obtenues formellement par

$$\begin{aligned} a^{n+1}_{i,j} &= \frac{\sum_{O \in X} A^n_{i,j}(O)/P(O)}{\sum_{k=1}^{N} \sum_{O \in X} A^n_{i,k}(O)/P(O)} \\ b^{n+1}_i(o) &= \frac{\sum_{O \in X} B^n_i(o, O)/P(O)}{\sum_{o' \in \mathcal{O}} \sum_{O \in X} B^n_i(o', O)/P(O)} \end{aligned} \tag{6}$$

avec $P(O)$ la probabilité d'observation de la séquence O en n. Ajoutons qu'une implémentation typique de cet algorithme requiert de garder en mémoire les matrices *forward* et *backward* de chaque séquence $O \in X$ pour calculer les $A^n_{i,j}(O)$ et $B^n_i(o, O)$.

Méthodes utilisées pour implémenter les HMMs en mémoire linéaire

Le problème qui se pose est donc de fournir une librairie permettant de construire des HMMs en fournissant les résolutions typiques aux problèmes basiques cités ci-dessus, tout en respectant la contrainte de la mémoire linéaire. Nous séparons les explications pour chaque problème en commençant à chaque fois par une approche théorique avant de développer brièvement un paragraphe sur l'implémentation en `C++` proposée par la librairie présentée.

Avant de parcourir chaque problème, il s'avère pertinent de d'abord énoncer la stratégie générale adoptée par la librairie. L'approche adoptée est de respecter le design pattern *Strategy* pour les algorithmes des HMMs. Dès lors les objets HMMs (instances de la classe `HiddenMarkovModel`) proposés ne définissent pas eux-mêmes ces algorithmes, mais utilisent plutôt un de leurs attributs qui pointe vers l'algorithme qui a été donné à sa construction. Ce découplage permet donc de fournir lors de la construction n'importe quel algorithme pour n'importe quel problème. Ainsi, pour entrainer un objet HMM avec l'algorithme Baum-Welch linéaire, nous lui donnons la classe qui implémente cet algorithme lors de la construction et l'appel à la méthode d'entrainement du HMM délèguera l'entrainement à ladite classe. Remarquons qu'il est en conséquence très facile d'ajouter des algorithmes d'entrainement différents puisqu'il suffit d'ajouter une classe implémentant le nouvel algorithme désiré et de donner une instance de cette classe à l'objet HMM visé. Notons aussi que tous ces algorithmes se situent dans le fichier `hmm_algorithms.cpp`.

Algorithmes *forward* et *backward*

Théorie Un bref coup d'oeil aux équations 1 et 2 permet d'observer qu'à chaque récursion t, le calcul des variables *forward* α_t et *backward* β_t ne nécessite que de connaitre les valeurs de α_{t-1} et β_{t+1} obtenues lors de la récursion précédente (Miklós and Meyer, 2005). Cette considération permet donc que l'utilisation de la mémoire ne dépende pas de la taille T de la séquence O analysée et ne s'accompagne pas d'une perte de performance pour le problème d'évaluation et le problème de décodage.

Implémentation Au lieu de stocker les variables *forward* et *backward* dans des matrices $M \times T$, les algorithmes *forward* et *backward* que nous avons implémentés ne gardent simultanément en mémoire que deux vecteurs de taille M (par exemple l'algorithme *forward* utilise les vecteurs `alpha_prev_t` et `alpha_t` dans sa méthode `forward_step` appelée à chaque itération).

Décodage Viterbi

Théorie L'observation permettant de respecter la contrainte mémoire pour les algorithmes *forward* et *backward* est également valable pour la variable ϕ de l'algorithme de Viterbi (Churbanov and Winters-Hilt, 2008). En effet, l'équation 3 montre que le calcul de ϕ_t requiert de connaitre uniquement ϕ_{t-1}.

De plus, Churbanov and Winters-Hilt (2008) décrit une méthode permettant de ne pas avoir recourt à une matrice $M \times T$ pour les valeurs de retour en arrière ψ. En effet, l'article avance que l'utilisation d'une liste chaînée permet de réduire l'espace mémoire alloué pour ψ en faisant l'observation que les chemins de retour en arrière convergent vers le chemin d'états optimal, et reviennent donc ensemble, en arrière, aux valeurs ψ_1. Dès lors, l'utilisation d'une liste

chainée permet de se *débarrasser* des chemins d'états non-optimaux.

Implémentation La difficulté dans l'implémentation du décodage Viterbi en mémoire linéaire proposé par Churbanov and Winters-Hilt (2008) réside dans la nécessité de libérer la mémoire utilisée par les chemins d'états non-optimaux. Cette difficulté est toutefois levée si l'implémentation est faites dans un langage utilisant un *garbage collector*, Churbanov and Winters-Hilt (2008) par exemple utilise `Java`.

Le même article propose en parallèle une solution pour une implémentation en `C++` : l'emploi de pointeurs partagés (`shared_ptr`). De là, introduisons la structure de donnée `Traceback` utilisée dans notre libraire, cette structure n'étant autre que l'implémentation de ψ. `Traceback` possède deux vecteurs de pointeurs partagés de taille M, soit un *courant* et un *précédent*. À chaque itération de la procédure de décodage, le ψ (voir l'équation 3) obtenu pour chaque état i détermine vers quel pointeur partagé du vecteur *précédent* va pointer le pointeur partagé i du vecteur *courant*. Ainsi, à la fin de l'itération t, tous les pointeurs du vecteur *courant* pointent vers un des pointeurs du vecteur *précédent*. Et donc, comme expliqué dans la section théorique, tous les pointeurs du vecteur *précédent* qui ne sont pas référencé par au moins un des pointeurs de *courant* se trouvent sur un chemin non-optimal, nous pouvons en conséquence nous en débarrasser. Pour ce, avant de passer à la récursion suivante, nous copions le vecteur *courant* dans le vecteur *précédent* (méthode `next_column`), les pointeurs non référencés ont leur compte de référence mis à 0 (le vecteur *précédent* maintenait ce compte à 1) et sont donc supprimés.

Entraînement Viterbi

Théorie Étant donné que l'entrainement Viterbi utilise la procédure de décodage de Viterbi décrite dans l'équation 3, nous pouvons à nouveau appliquer l'observation faites précédemment : le calcul des valeurs de ϕ peut être poursuivit en ne retenant que les valeurs obtenues lors de la récursion précédente.

Établissons encore une autre observation faites par Lam and Meyer (2010) : pour obtenir les valeurs $A^n_{i,j}$ et $B^n_i(o)$ il est uniquement nécessaire de connaitre le *nombre de fois* que chaque transition et émission a été utilisée dans le chemin optimal pour chaque séquence d'entrainement $O \in X$ mais pas *où* dans la matrice Viterbi chaque transition et émission a été utilisée.

En s'aidant des deux observations précédentes, Lam and Meyer (2010) décrit un algorithme d'entrainement Viterbi qui calcule les comptes $A^n_{i,j}$ et $B^n_i(o)$ en linéarisant l'utilisation de la mémoire par rapport aux longueurs des séquences d'entrainement. Pour ce, deux nouvelles notations sont introduites par l'article :

- $A_{i,j}(t, m)$: le nombre de fois que la transition de i à j est utilisée dans le chemin d'états optimal finissant dans l'état m à la position t de la séquence ;
- $B_i(o, t, m)$: le nombre de fois que l'état i émet le symbole o dans le chemin d'états optimal finissant dans l'état m à la position t de la séquence.

L'algorithme d'entrainement Viterbi effectue donc une récursion similaire au décodage Viterbi, en ce qu'il calcule ϕ_t à chaque t. Lam and Meyer (2010) montre ensuite le calcul des $A_{i,j}(t, m)$ et $B_i(o, t, m)$ pour la récursion t

$$\begin{aligned} A_{i,j}(t, m) &= A_{i,j}(t-1, l) + \delta_{l,i}\delta_{m,j} \\ B_i(o, t, m) &= B_i(o, t-1, l) + \delta_{m,i}\delta_{o,o_t} \end{aligned} \tag{7}$$

avec $\delta i, j = 1$ si $i = j$, 0 sinon et $l = \psi(m)$ (voir équation 3).

Puisque $A_{i,j}(t, m)$ détient uniquement les comptes du nombre de transitions de i à j arrivant en m, il est évident qu'il faudra avoir M comptes pour chaque transition. Soit $\mathcal{T}$ l'ensemble des transitions entrainées (transitions *libres*) et $\mathcal{E}$ l'ensemble des émissions entrainées (émissions *libres*), le nombre de comptes à garder en mémoire à chaque itération est donc de $|S \times \mathcal{T}| + |S \times \mathcal{E}|$.

Implémentation Nous représentons les comptes $A_{i,j}(t, m)$ et $B_i(o, t, m)$ via des instances de `TransitionScore` et `EmissionScore`. Ces objets utilisent des matrices de dimensions $|S| \times |\mathcal{T}|$ et $|S| \times |\mathcal{E}|$ pour stocker les comptes des paramètres libres de l'HMM (ces transitions et émissions libres sont données par le modèle entrainé). De plus, afin de garder une trace des retours en arrière au temps t, l'algorithme utilise un objet `Traceback` qu'il réinitialise (`reset`) avant le passage au temps suivant (puisqu'il n'est pas nécessaire de connaitre de retours en arrières autre que ceux au temps t). Ainsi, pour mettre à jour les comptes arrivants à l'état m, l'implémentation se sert du `Traceback` en partant de m pour incrémenter les comptes selon l'équation 7.

Entraînement Baum-Welch

Théorie Rappelons que l'algorithme Baum-Welch définit $A_{i,j}(O)$ et $B_i(o, O)$ qui sont des concepts assez proches des comptes utilisés par l'entrainement Viterbi, sauf qu'au lieu de considérer uniquement le chemin optimal ce sont tous les chemins qui sont pris. Le développement effectué dans la section précédente est en conséquence également applicable à l'entrainement Baum-Welch. En effet, Miklós and Meyer (2005) définit les $A_{i,j}(t, m)$ pour Baum-Welch comme les $A_{i,j}(O)$ sauf qu'au lieu de considérer l'entièreté de la séquence O, $A_{i,j}(t, m)$ ne prend en compte que les

chemins d'états finissant dans l'état m à la position t de la séquence ($B_i(o, t, m)$ est obtenu analogiquement pour les émissions). De plus, Churbanov and Winters-Hilt (2008) utilise les variables $backward$ pour permettre d'obtenir les probabilités de tous les chemins, en ne gardant en mémoire que les valeurs β_{t+1}.

Intuitivement, l'obtention des scores de la récursion suivante pour tous les $1 \leq m \leq N$ requiert donc *d'appliquer l'algorithme backward* aux scores $A_{i,j}(t, m)$ en substituant les variables *backward* par les scores $A_{i,j}(t+1, n)$ pour $1 \leq n \leq N$ puisque pour obtenir le score des transitions arrivant en m il faut considérer les scores des transitions de i à j de tous les états calculés pendant la récursion précédente. Ensuite, à ce score est ajouté la probabilité d'avoir la transition de i à j à la position $t+1$ de la séquence si $i = m$ en utilisant $\beta_{t+1}(j)$ (voir Churbanov and Winters-Hilt (2008) pour une description formelle).

Implémentation Nous avons mentionné dans le paragraphe précédent la similarité de l'adaptation en mémoire linéaire pour les algorithmes d'entrainement Viterbi et Baum-Welch, en ce que les $A_{i,j}(t, m)$ et $B_i(o, t, m)$ utilisés par chaque algorithme sont des concepts tout à fait analogues. C'est pourquoi l'implémentation de l'algorithme Baum-Welch fournie dans la librairie utilise les mêmes structures de données `TransitionScore` et `EmissionScore` que pour l'implémentation de l'algorithme Viterbi.

Modifications apportées par l'ajout d'états silencieux

Prendre en compte les états silencieux lors de l'implémentation d'un HMM a requit une attention particulière. Nous décrivons dans cette section les quelques modifications supplémentaires qu'ils ont apportés aux algorithmes.

Forward, *backward* et Viterbi avec états silencieux

Durbin et al. (1998) explique, bien que de façon non-exhaustive, comment intégrer des états silencieux aux algorithmes *forward*, *backward* et Viterbi. Étant donné que le traitement est similaire aux trois algorithmes, nous choisissons de décrire les changements apportés à l'algorithme *forward* en détails et de donner une adaptation de ces changements aux autres algorithmes.

Rappelons tout d'abord que la variable *forward* $\alpha_t(i)$ détermine la probabilité d'observer $o_1 \dots o_t$ et d'arriver à $q_t = S_i$ et est calculée récursivement via l'équation 1. Remarquons d'entrée que cette récusion n'est pas valable avec des états silencieux étant donné que si l'état i est silencieux alors $b_i(o_t)$ renvoie 0, garder l'équation 1 en l'état ignorerait donc complètement les états silencieux lors du calcul des valeurs de α. Comment donc ajuster l'équation 1 afin de ne pas ignorer les états silencieux ? Il faut d'abord réaliser que si i n'est *pas* silencieux, $\alpha_t(i)$ *consomme* le symbole t de la séquence O. *A contrario* avec i un état silencieux, $\alpha_t(i)$ ne *consomme pas* le symbole t. Dès lors, les calculs des $\alpha_t(i)$ des états silencieux nécessitent les $\alpha_t(j)$, donc les variables *forward* de la récursion actuelle, afin de *consommer* le symbole t !

Ainsi, le calcul $\alpha_t(i)$ avec i silencieux nécessite d'avoir déjà trouvé les valeurs $\alpha_t(j)$ des états non-silencieux d'une part mais aussi des états silencieux qui précèdent i d'autre part puisque i ne *consomme* pas le symbole t et sa valeur dépend donc aussi de tous les *chemins silencieux* de la récursion t ! La récursion formelle qui en découle est décrite par Durbin et al. (1998) avec $t = 2 \dots T$ et S_{silent} l'ensemble des états $i \in S$ tel que i est silencieux

$$\alpha_t(i) = \begin{cases} \left[\sum_{j=1}^{N} \alpha_{t-1}(j) a_{j,i}\right] b_i(o_t), & i \in S \setminus S_{silent} \\ \sum_{j=1}^{N} \alpha_t(j) a_{j,i}, & i \in S_{silent} \end{cases} \tag{8}$$

Nous ajoutons que lors des calculs des variables $forward$ initiales, il convient également d'appliquer le raisonnement développé dans le paragraphe précédent en ce que $\alpha_1(i)$ pour i non-silencieux peut être non-nul même si $\pi_i = 0$ s'il existe un *chemin silencieux* entre l'état initial et i.

Au niveau de l'implémentation, afin de parcourir les états silencieux dans l'ordre et donc respecter le raisonnement ci-dessus, nous effectuons un tri topologique sur le sous-graphe constitué uniquement d'états silencieux. Ce tri est alors placé à la fin du vecteur d'états pour s'assurer qu'en parcourant ce vecteur dans l'ordre, les $\alpha_t(j)$ des états j non-silencieux soient déjà obtenus lors du calcul des variables *forward* des états silencieux.

Durbin et al. (1998) explique comment facilement étendre ceci aux algorithmes *backward* et Viterbi. La modification nécessaire pour le premier algorithme est assez intuitive car elle réside dans le parcourt des états silencieux dans l'ordre topologique inverse puisque le *backward* algorithm s'exécute *en arrière*. Pour le second, l'ajustement théorique est encore plus trivial puisqu'il est uniquement nécessaire de remplacer la somme par le maximum. Toutefois, la mise en pratique est plus délicate au niveau du retour en arrière étant donné que l'ajout de chemins silencieux fait en sorte que l'état précédent pointé par ϕ pour les états silencieux n'est pas obligatoirement un état en $t-1$!

Entrainement Viterbi et Baum-Welch avec états silencieux

Les articles utilisés pour implémenter les algorithmes des HMMs en mémoire linéaire ne développent pas le cas particulier des états silencieux. L'ajout d'un traitement pour les

états silencieux dans les algorithmes en mémoire linéaire Viterbi et Baum-Welch fournis avec la librairie est donc issu d'un bref travail de réflexion que nous introduisons ici, bien qu'il soit directement inspiré du travail dans Durbin et al. (1998) expliqué dans la section précédente.

Viterbi Lors du calcul des comptes $A_{i,j}(t, m)$, nous considérons pour l'instant n'avoir qu'une seule valeur de retour en arrière pour l'itération t. Or, comme démontré précédemment, le nombre de retours en arrière entre t et $t-1$ dépend de la longueur du chemin d'états silencieux. Ainsi, au lieu d'incrémenter le compte $A_{i,j}(t-1, l)$ si $l = i$ et $m = j$, il faudra incrémenter le compte si *un des états le long du chemin d'états silencieux vaut* i.

Baum-Welch L'ajustement de l'algorithme Baum-Welch en mémoire linéaire pour les états silencieux semble *a priori* plus complexe. Cependant, en constatant assez intuitivement que la récursion des scores $A_{i,j}(t, m)$ est tout à fait équivalente à la récursion des variables *backward* en substituant $\beta_j(o_{t+1})$ par le score $A_{i,j}(t+1, n)$, il vient qu'il suffit d'appliquer les mêmes changements apportés dans l'algorithme *backward* décrits ci-dessus aux calculs des scores $A_{i,j}(t, m)$.

Résultats

Afin de s'assurer que les valeurs calculées par les algorithmes implémentés soient correctes, tous ces algorithmes sont testés de façon extensive dans un fichier de test `hmm_tests.cpp`. Les valeurs sur lesquels nous exécutons les assertions sont issues d'une part des outils HMM de MATLAB et d'autre part d'une libraire python open source trouvable à l'adresse `https://github.com/jmschrei/pomegranate`.

Vérifions maintenant que les algorithmes d'entrainement implémentés s'exécutent bien en mémoire linéaire par rapport à la taille de la séquence. Pour cela, nous générons un HMM avec 20 états et 200 paramètres libres à entrainer (le nombre de paramètres à entrainer reste bien évidemment constant pour ces tests étant donné que varier leur nombre ferait varier l'espace nécessaire pour stocker leur scores $A_{i,j}(t, m)$ et $B_i(o, t, m)$). La FIGURE 1 illustre les résultats obtenus, sachant que le HMM utilisé est le même pour les deux algorithmes d'entrainement testés (Viterbi et Baum-Welch). Nous observons donc que la mémoire est utilisée linéairement par rapport à la taille des séquences, la contrainte sur la mémoire que nous nous sommes fixée est ainsi respectée. La FIGURE 1 est bornée supérieurement par une taille de séquence à 500 pour l'HMM considéré étant donné que toute valeur au-delà de cette limite ne changeait pas la mémoire utilisée par les deux algorithmes, prouvant encore que ces algorithmes sont bien en mémoire linéaire.

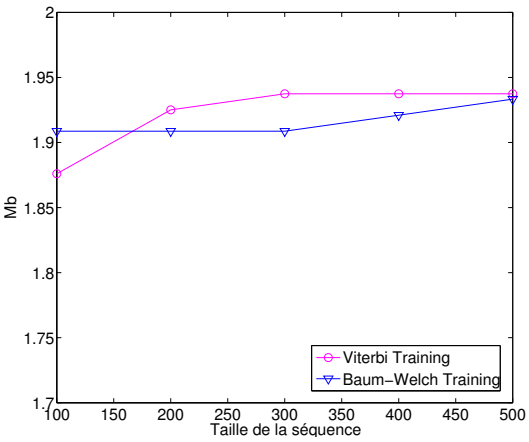

FIGURE 1 – Utilisation de la mémoire

Les deux algorithmes allouent une quantité de mémoire proche l'une de l'autre ce qui est aisément explicable par le fait qu'ils utilisent un même nombre de scores qui dépend du nombre de paramètres entrainés et du nombre d'états de l'HMM, or, ces nombres sont égaux ici.

Au niveau des performances, Churbanov and Winters-Hilt (2008) montrait déjà que la durée des itérations de l'algorithme Baum-Welch en mémoire linéaire grandissait exponentiellement avec le nombre d'états compris dans l'HMM. Nous obtenons les mêmes résultats avec la FIGURE 3. La FIGURE 2 renseigne sur la durée d'une itération pour l'algorithme d'entrainement Viterbi. Lam and Meyer (2010) démontrait le gain en temps d'exécution en utilisant cet algorithme au lieu de l'algorithme Baum-Welch. Nous remarquons un comportement similaire en comparant la FIGURE 2 avec la FIGURE 3.

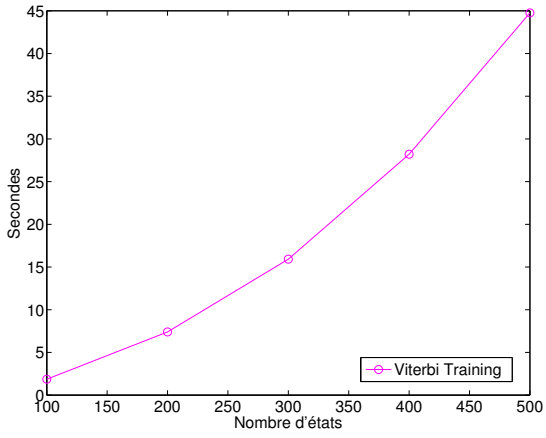

FIGURE 2 – Durée d'une itération d'entrainement

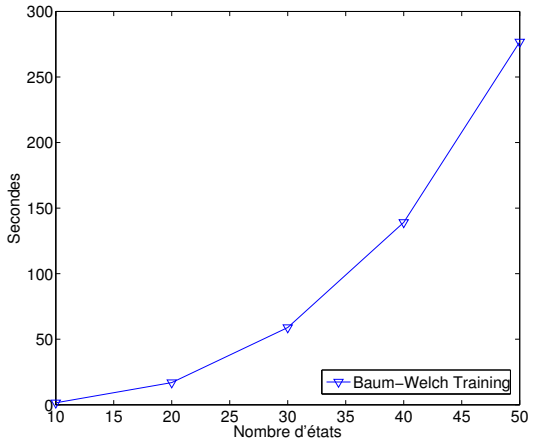

FIGURE 3 – Durée d'une itération d'entrainement

Conclusion et perspectives

Nous avons présenté dans cet article une implémentation possible pour les algorithmes des HMMs en mémoire linéaire décrits par Miklós and Meyer (2005); Churbanov and Winters-Hilt (2008); Lam and Meyer (2010). Cette implémentation est visitable à l'adresse `https://github.com/jhellinckx/limehmm`.

En plus d'une implémentation en mémoire linéaire, nous nous sommes également efforcés de permettre l'emploi d'états silencieux dans ces algorithmes, ce qui a rendu la tâche bien plus difficile étant donné que les articles introduisant ces algorithmes ne donnaient pas d'indication quant à la manière d'adapter lesdits algorithmes aux états silencieux. Nous nous sommes donc inspiré du travail fait par Durbin et al. (1998) pour ajuster l'implémentation. Bien qu'étant non-exhaustives, les propositions qui y sont décrites ainsi qu'une bonne compréhension des modifications sémantiques apportées par les états silencieux nous ont permis de les intégrer à la librairie.

Enfin signalons que la librairie accompagnant cet article ne se targue pas de proposer tous les outils nécessaires à l'utilisation de HMMs. En effet, une multitude d'améliorations peut y être apporté. Citons notamment l'ajout d'un support pour autoriser des distributions continues pour les états ou encore l'ajout de l'algorithme EM stochastique en mémoire linéaire pour entrainer l'HMM (décrit par Lam and Meyer (2010)).

References

Churbanov, A. and Winters-Hilt, S. (2008). Implementing em and viterbi algorithms for hidden markov model in linear memory. *BMC Bioinformatics*, 9 :224.

Durbin, R., Eddy, S., Krogh, A., and Mitchison, G. (1998). *Biological sequence analysis*. Cambridge University Press.

Lam, T. Y. and Meyer, I. (2010). Efficient algorithms for training the parameters of hidden markov models using stochastic expectation maximization (em) training and viterbi training. *Algorithms for Molecular Biology*, 5 :38.

Miklós, I. and Meyer, I. (2005). A linear algorithm for baum-welch training. *BMC Bioinformatics*, 6 :231.

Yoon, B.-J. (2009). Hidden markov models and their applications in biological sequence analysis. *Current Genomics*, 10 :402–415.