# Mise à jour du système d'initialisation des variables latentes

Vincent KUBICKI

20 septembre 2017

## Table des matières



**Résumé**

Ce document est le quatrième décrivant les évolutions proposées et implémentées dans MixtComp lors du plan de mise à plat effectué durant l'été 2017. Les trois documents précédents présentaient :

1. une feuille de route dans [1]
2. la refonte de la gestion des dégérescences et de l'initialisation des paramètres dans [2]
3. des tests de performances dans [3]

Le présent document décrit la mise en place de façon systématique (pour tous les modèles) d'une méthode d'initialisation des variables latentes s'inspirant de celle qui assurait 100 % de prédictions correctes sur le jeu de données test en prédiction dans [3]. Cette méthode utilise les probabilités observées pour initialiser les appartenances aux classes. Comme l'initialisation est une des partie les plus byzantines de MixtComp à cause des tirages de variables latentes, ce document fourni aussi un récapitulatif de toute la séquence d'initialisation, en pointant les particularités des divers modèles. Il contient aussi une description de modification permettant de mieux gérer les chaînes de Markov internes à certains modèles, ainsi qu'une modification de la mise en cache des probabilités observées qui sont longues à calculer.

# 1 Rappels sur les initialisations

## 1.1 Initialisation des paramètres

L'initialisation des paramètres par sélection d'un individu représentatif a déjà été l'objet des précédents documents cités en introduction. Ce chapitre constitue plutôt un récapitulatif des méthodes d'initialisations (paramètres et données) particulières pour chaque modèle, avant qu'elles aient été modifiées comme présenté dans la section 3.

Pour rappel, en apprentissage comme en prédiction, les données observées (c'est-à-dire fournies par l'utilisateur) sont initialisées lors de l'appel à `MixtureComposer::setDataParam`, qui est appelé directement par l'exécutable, juste après l'appel à `MixtureComposer::createMixtures`.

En apprentissage, les données sont complétées de façon uniforme avec `MixtureComposer::initData`, et les paramètres sont initialisés avec `MixtureComposer::initParam`. Un individu représentatif est sélectionné pour chacune des classes, et les différents modèles utilisent les données complétés pour effectuer l'initialisation de leurs paramètres.

## 1.2 Initialisation des variables latentes

A côté de l'initialisation des paramètres, il y a une autre initialisation qui est celle des variables latentes. Ces variables latentes sont toutes complétées en SEM. Elles sont de nature différentes, avec différentes dépendances les unes aux autres, ce qui complexifie l'initialisation. On distingue trois types de variables latentes :

— les appartenances aux classes : variable $z_i$ pour l'observation $i$, pouvant prendre une valeur de 1 à $K$ le nombre de classes
— les valeurs manquantes ou censurées : par exemple ?, mais aussi les intervalles de type $[a, b]$, etc...
— les variables latentes des modèles : par exemple les appartenances aux sous-régression pour chaque pas de temps dans le modèle de données fonctionnelles

Jusqu'à présent, les appartenances aux classes étaient tirées uniformément (ou en utilisant les proportions). En début de SEM, on ne sait absolument rien sur les paramètres que l'on veut estimer. On était donc obligé de tout initialiser de façon "uniforme". Les $t_{ik}$ étaient initialisés à la valeur $\frac{1}{K}$ et les classes étaient tirées de façon aléatoires. Comme en plus on ne connaissait pas les paramètres avant d'implémenter l'initialisation par représentants de classes, on appelait pour chaque modèle la méthode `IMixture::initData` qui ne requière pas cette information, et qui permet l'initialisation uniforme des variables manquantes et latentes.

Cela signifie que les valeurs observées des données n'étaient jamais utilisées, ce qui pouvait amener sur les cas tests (où on connait les appartenances réelles) à des initialisations très différentes des valeurs réelles, sans que la chaîne SEM n'arrive à corriger cet écart dans le nombre d'itérations imparti. Asymptotiquement on observerait cette correction (ainsi que des dégénérescences...), malheureusement le monde réel n'est pas asymptotique. Cette difficulté à converger s'explique parce que tirer de mauvaises classes amène à tirer de mauvaises valeurs sur les variables latentes, qui amènent des mauvais tirages sur les classes, ...

Dans le Gibbs on connaissait les paramètres et donc on les utilisait pour compléter les individus. Mais on initialisait les appartenances aux classes au hasard. Et utiliser les paramètres correspondant à des classes tirées uniformément n'est au final pas d'une grande utilité à partir du moment où les appartenances sont tirées aléatoirement.

# 2 Particularités des anciennes initialisations de chaque modèle

Voici les particularités de chaque modèle, et les initialisations utilisées avant d'avoir mis en place les séquences d'initialisations unifiées présentées dans la section 5.

## 2.1 Modèles de base

Les modèles de base sont ceux correspondant à des instances du patron de classe `MixtureBridge` (qui hérite bien sûr de `IMixture`). L'initialisation des paramètres est simple pour ces modèles, il s'agit d'utiliser l'individu représentatif pour déterminer le "centre" de la distribution de chaque classe. On utilise ensuite une information annexe, comme le nombre de classes, pour fixer les paramètres gérant la dispersion. A l'issue de l'appel à `IMixture::InitParam`, ces modèles sont prêts à être utilisés.

## 2.2 Modèle de rang

Dans `RankMixture::initParam`, après que les paramètres $\mu$ et $\pi$ aient été affectés pour chaque classe, de multiples échantillonnages étaient effectuées immédiatement, pour que les valeurs manquantes et les ordres de présentations soient tirés suivant la loi paramétrée par l'individu représentatif de chaque classe.

`computeObservedProba` était appelé dans `setDataParam` car le calcul est assez long et nécessite de nombreux tirages. Les résultats étaient ensuite conservés en cache et prêts à être réutilisés à chaque appel de `lnObservedProbability`.

### 2.3 Modèle pour les données ordinales

Pour ce modèle, il faut faire attention à ce qu'aucun individu complété ne soit de probabilité nulle.

Dans `initData`, toutes les valeurs de $z$ [1] sont fixées à 0, pour garantir qu'aucun individu n'aura de probabilité nulle quel que soit les valeurs de paramètres.

Dans `initParam`, une valeur de $\pi$ non nulle est sélectionnée, et de multiples itérations de Gibbs sont effectuées dans `initBOS` afin d'avoir des valeurs non nulles de $z$ lors de la prochaine étape M.

Comme pour les données de rang, `computeObservedProba` était appelé dans `setDataParam` car le calcul est assez long et nécessite de nombreux tirages. Les résultats étaient ensuite conservés en cache et prêt à être réutilisés à chaque appel de `lnObservedProbability`.

### 2.4 Modèle pour les données fonctionnelles

Les quantiles des valeurs des abscisses pour les fonctions sont calculés à la fin de `setDataParam`.

Dans `initData`, les quantiles sont utilisés pour déterminer les domaines de chaque sous-régression.

Dans `initParam`, les quantiles sont déterminés de nouveau classe par classe en se basant sur l'individu représentatif. De cette façon, les appartenances aux sous-regressions sont compatibles avec les paramètres qui ont été estimés pour l'individu représentatif. Ces assignations aux sous-regressions écrasent celles déterminées par `initData` en apprentissage.

## 3 Utilisation des probabilités observées

Pour éviter les problèmes décrits dans la section 1.2 il faudrait effectuer la première complétion avec des individus ayant une probabilité en moyenne plus élevée qu'avec l'initialisation uniforme, que ce soit en début de SEM ou en début de Gibbs.

Les trois types de variables latentes présentées dans la section 1.2 ne sont pas initialisées de la même façon. L'initialisation de l'appartenance aux classes $z$ est de la responsabilité de la classe `MixtureComposer`, alors que les complétions des variables manquantes et des variables latentes sont de la responsabilité des modèles qui héritent de la classe `IMixture`. Ce chapitre se focalise sur l'initialisation des appartenances aux classes.

Il y a deux échantillonneurs de Gibbs dans MixtComp. Le premier est appelé en apprentissage, après l'estimation de paramètres dans le SEM, et le second est appelé en prédiction, qui ne contient par ailleurs qu'un Gibbs. Lors de l'initialisation d'un Gibbs, on dispose de paramètres estimés mais il n'y avait pas de possibilité d'utiliser cette information. En effet, pour tirer $z$, il fallait des individus complétés, et pour les compléter en utilisant les paramètres, il fallait connaître leur classe $z$. C'est pour cette raison que jusqu'à présent on initialisait $z$ de façon aléatoire, car il fallait un point de départ.

1. la variable latente du modèle BOS, et non l'appartenance aux classes

On peut imaginer une méthode alternative qui résout ce paradoxe. On peut calculer les $t_{ik}$ comme on le ferait en EM, où on ne complète pas les individus. Pour ceci, on calcule les probabilités marginalisées sur toutes les variables manquantes ou latentes. De cette façon, on obtient des $t_{ik}$ "observés" qui fournissent la probabilité d'appartenir à chaque classe sachant ce qui a été observé (que ce soit des valeurs, des intervalles, manquant, etc ...). On utilise ainsi plus d'information à notre disposition qu'en tirant les appartenances aux classes de façon aléatoire comme c'était le cas jusqu'à présent.

Une fois que $z$ a été tiré de cette façon, on peut effectuer la complétion de toutes les autres variables latentes en utilisant les modèles. On a au final plus de chance de démarrer le Gibbs dans une zone de vraisemblance complétée importante, et donc d'avoir des imputations de valeurs par les modes (ou médianes suivant les modèles) plus correctes (au sens de la capacité prédictive par exemple).

Cette méthode a été inspirée par des essais concluants sur le modèle fonctionnel, présentés dans [3]. Il s'agit simplement ici de la généraliser à tous les modèles et de la rendre plus robuste. Un des problèmes de la vraisemblance observée est que pour certains modèles (par exemple sur les données de rang) elle est calculée par tirages, et ceci peut amener à avoir des individus dont la probabilité d'appartenir à chacune des classe est nulles. On peut pour les individus dont les $t_{ik}$ observés ne sont pas définis, effectuer une initialisation uniforme ne tenant pas compte des données observées. Ainsi on utilise un maximum d'information observée, et on effectue le tirage uniforme uniquement quand on a pas le choix.

Pour l'implémentation, on utilise la méthode `IMixture::lnObservedLikelihood` qui fournit la vraisemblance marginalisée sur les valeurs censurées et les variables latentes des modèles. Il faut juste ajouter une méthode `MixtureComposer::eStepObserved` permettant de calculer les $t_{ik}$ des données observées là où `MixtureComposer::eStep` permet de calculer les $t_{ik}$ des données complétées. Ensuite `MixtureComposer::sStepNoCheck` serait appelée.

Si on avait uniquement des modèles simples (sans chaîne de Markov interne), on pourrait effectuer l'initialisation du Gibbs (en apprentissage ou en prédiction) de la façon suivante :

1. détermination des paramètres :
   - en apprentissage, dans `MixtureComposer::storeSEMRun` lors de la dernière itération du SEM
   - en prédiction, dans `MixtureComposer::setDataParam`
2. calcul des $t_{ik}$ "observés" et tirage de $z$
3. appel de `IMixture::samplingStepNoCheck`

Les prises en compte des modèles plus complexes sont décrites dans la section 5.

# 4 Autres modifications

Ces modifications ont été mises en place pour rendre les modifications d'état plus explicites et centralisées. Elles permettent aussi de standardiser les comportements des modèles.

## 4.1 Prise en compte des chaînes de Markov

Les chaînes de Markov introduisent des particularités. Par exemple `IMixture::samplingStepNoCheck` correspond à une itération pour une chaîne de Markov. On ne peut

pas appeler cette méthode directement en initialisation, car une chaîne de Markov doit avoir été placée dans un état initial avant la première itération. La meilleure solution est d'appeler systématiquement `MixtureComposer::initData` pour être assuré d'avoir un état initial utilisable pour les chaînes de Markov.

Un autre problème est qu'appeler `IMixture::samplingStepNoCheck` une seule fois après avoir appelé `MixtureComposer::initData` ne va pas permettre de modifier sensiblement les variables latentes, puisqu'on effectue qu'une itération de la chaîne de Markov. Les variables latentes ne sont ainsi pas tirées en suivant la distribution stationnaire de la chaîne. Pour les données ordinales cela conduit à une dégénérescence très probable dès les premières itérations du SEM.

Les initialisations des chaînes de Markov étaient "cachées" dans l'ancienne architecture. Elles relevaient de la responsabilité de chaque modèle, et pour les modèles BOS et ISR ces initialisations étaient effectuées dans `MixtureComposer::initParam` et dans `MixtureComposer::setDataParam`.

On transfère donc l'appel à cette initialisation des modèles vers `MixtureComposer`. On crée une méthode `IMixture::initializeMarkovChain` qui est appelée par `MixtureComposer::initializeMarkovChain`. Cette méthode suppose que les individus ont déjà été complétés de façon uniforme avec `IMixture::initData` et que les paramètres sont connus. Il faut donc modifier les séquences d'initialisation en conséquence, ce qui est l'objet de la section 5.

## 4.2 Calcul de distribution empirique

Pour les modèles ISR et BOS, il faut effectuer des tirages pour déterminer une distribution observée empiriquement. Le calcul direct en serait trop long car il faut sommer sur toutes les combinaisons possibles de valeurs pour les variables latentes. Cette distribution est ensuite utilisée pour déterminer les probabilités observées de chaque individu.

Dans l'ancienne version du code, ce calcul était effectuée au besoin soit dans la méthode `IMixture::storeSEMRun` en apprentissage, soit dans la méthode `IMixture::setDataParam` en prédiction. La responsabilité du déclenchement du calcul appartenait au modèle qui devait connaître l'algorithme englobant. C'est le même problème que celui décrit pour la mise en cache de la vraisemblance observée, décrit dans la section 4.3.

On souhaite rendre l'appel à ce calcul de distribution explicite et non plus implicite. Pour ceci, on ajoute une nouvelle méthode `IMixture::computeObservedProba`. Ensuite les appels à `IMixture::lnObservedProbability` utilisent la distribution obtenue par `IMixture::computeObservedProba`.

## 4.3 Mise en cache de la vraisemblance observée

Les probabilités observées peuvent être longue à calculer, particulièrement dans le cas de modèles contenant des chaînes de Markov, comme les modèles BOS et ISR. Comme elles sont réutilisées à de nombreux endroits, il est nécessaire d'effectuer une mise en cache. Dans l'ancienne version, elle avait lieu :

— en apprentissage, dans `IMixture::storeSEMRun` à la dernière itération car les paramètres venaient d'être estimés

— en prédiction, dans `IMixture::setDataParam`

Elle est ensuite utilisée dans de nombreux calculs :

— `MixtureComposer::eStepObserved` en début de Gibbs, comme décrit dans la section 1.2

— `MixtureComposer::E_kj` pour les visualisations MASSICCC

— `MixtureComposer::Delta` ajouté par Matthieu pour ses visualisations

— `MixtureComposer::lnProbaGivenClass`, fourni les probabilités observées des individus classe par classe, utilisées aussi par Matthieu pour avoir des $t_{ik}$ observés plus précis que ceux obtenus lors de tirage dans le Gibbs.

— `MixtureComposer::lnObservedLikelihood` pour calculer le BIC et l'ICL

Déléguer la mise en cache au sein de chaque modèle complexifie l'architecture. Chaque modèle doit connaître le fonctionnement de l'algorithme englobant depuis lequel il est appelé pour être à même de définir quand mettre en cache et quand utiliser le cache. En résulte un couplage trop important entre les classes `MixtureComposer` et `IMixture`. Une meilleure solution est que ce soit `MixtureComposer` qui gère le cache au lieu des modèles.

Une raison supplémentaire pour extraire la mise en cache au niveau de `MixtureComposer` est que l'on veut en apprentissage faire un calcul de vraisemblance observée dès l'initialisation du SEM, pour le calcul des $t_{ik}$ observés. Utiliser l'ancienne architecture serait trop complexe.

On profite donc de la série de modifications en cours pour implémenter cette mise en cache. Elle est maintenant effectuée dans `MixtureComposer::setObservedProbaCache`. Cette méthode va être appelée directement depuis `SemStrategy` ou `GibbsStrategy` au lieu d'être appelée implicitement. On concentre ainsi les décisions générant des changements d'état en un endroit unique, pour plus de clarté.

On peut noter que la méthode `MixtureComposer::setObservedProbaCache` nécessite un appel préalable à `MixtureComposer::computeObservedProba` (voir section 4.2) pour pouvoir gérer les modèles contenant un Gibbs.

# 5 Séquence finale d'initialisation

La structure gérant le SEM et le Gibbs a été simplifiée, et le code dupliqué dans `SemStrategy` et `GibbStrategy` a été supprimé. Désormais, en apprentissage, `SemStrategy::run` est appelé, suivi immédiatement de `GibbsStrategy::run`, alors qu'en prédiction, seul `GibbStrategy` est appelé. Chacune de ces méthodes effectue des initialisations qui lui sont propres.

Cela signifie notamment qu'en apprentissage (qui enchaîne un SEM et un Gibbs), une initialisation est effectuée en début de Gibbs. C'est normal, car à la fin du SEM une estimation de paramètres est effectuée en prenant le mode (ou la médiane) des valeurs de paramètres échantillonnées pendant le SEM, créant potentiellement une discontinuité dans les chaînes de Markov de certains modèles pouvant attribuer des probas nulles à certains individus.

Une partie de l'initialisation est identique en Gibbs et en SEM. De plus en apprentissage ou en prédiction (où on effectue un Gibbs seul), la lecture des données et des paramètres est effectuée avant toute autre initialisation, dans la méthode `MixtureComposer::setDataParam`.

Une partie de l'initialisation est commune entre le SEM et le Gibbs.

## 5.1 Nouvelle initialisation en SEM

Cette initialisation est effectuée dans la méthode `SemStrategy::run` :

1. appel de `IMixture::initData` pour avoir des individus représentatifs complétés dans `IMixture::initParam`
2. détermination des paramètres dans `IMixture::initParam`
3. initialisation commune, voir section 5.3

## 5.2 Nouvelle initialisation en Gibbs

Cette initialisation est effectuée dans la méthode `GibbsStrategy::run` :

1. appel de `IMixture::initData` pour avoir des individus complétés lors de l'initialisation des chaînes de Markov
2. initialisation commune, voir section 5.3

Les paramètres n'ont pas besoin d'être déterminés dans l'initialisation du Gibbs, puisque soit ils ont été estimés dans le SEM en apprentissage, soit ils ont été fournis par l'utilisateur en prédiction.

## 5.3 Nouvelle initialisation commune SEM et Gibbs

L'initialisation commune est maintenant regroupée dans la méthode `MixtureComposer::initializeLatent` et contient :

1. calcul des distribution observées empiriques avec `MixtureComposer::computeObservedProba`
2. mise en cache des probabilités observées dans `MixtureComposer::setObservedProbaCache`
3. calcul des $t_{ik}$ "observés" en appelant `MixtureComposer::eStepObserved`. Cette fonction est similaire à `eStep` mais utilise les probabilités observées au lieu des probabilités complétées
4. tirage de $z$ avec `MixtureComposer::sStepNoCheck`
5. initialisation des chaînes de Markov avec `IMixture::initializeMarkovChain` en utilisant le modèle
6. tirage des variables latentes et des valeurs manquantes en utilisant le modèle, grâce à `MixtureComposer::samplingStepNoCheck`. Cela permet la complétion en utilisant le modèle de la classe d'appartenance actuelle

Comme on peut le voir, en SEM comme en Gibbs les vraisemblances observées sont calculées dès que possible. En apprentissage cela se fait juste après que les paramètres aient été estimés par le SEM, alors qu'en prédiction cela se fait directement après la lecture des paramètres issus de l'apprentissage.

| | 1 | 2 |
|---|---|---|
| 1 | 0 | 32 |
| 2 | 68 | 0 |

TABLE 1 – Matrice de confusion typique de la nouvelle méthode d'initialisation

# 6 Résultats

On utilise le cas test déjà utilisé dans [3]. La matrice de confusion obtenue en utilisant les modifications présentées dans le présent document est tout le temps parfaite, comme présentée sur la table 1.

Au début du codage de MixtComp, on observait des problèmes d'écart-types surestimés pour des données continues partiellement observée. Pour résumer, des individus étaient assignés à une mauvaise classe, les valeurs manquantes étaient tirées suivant cette nouvelle classe, ce qui faisaient qu'ils restaient assignés à cette mauvaise classe par la suite. Mais ils contribuaient avec leurs valeurs observées à une surestimation d'écart-type, puisqu'ils étaient mal assignés. Ce problème n'apparaissait qu'avec des petits jeux de données et disparaît définitivement avec la méthode d'initialisation basée sur les probabilités observées.

# 7 A faire

— Il faut effectuer des tests plus divers, notamment sur les contrats InriaTech.
— Les données ordinales ont été désactivées car elles sont plus complexes à modifier que les autres modèles. Cependant, au vu des améliorations en prédiction, il est possible que le modèle fonctionne mieux, et donc cela vaut le coût de le réintroduire.
— Le système actuel ne tient pas compte des classes observées, fournies par l'utilisateur dans les cas d'apprentissage supervisés ou semi-supervisés, pour la sélection des représentants. Ca ne change pas fondamentalement le principe de tout ce qui a été effectué, mais il faut intégrer cette information.
— Il faudra répercuter les modifications dans les autres versions de MixtComp, par exemple JsonMixtComp. Comme il s'agit d'interfaces, ce n'est pas très complexe.
— Pour les calculs de distributions empiriques via `IMixture::computeObservedProba`, le nombre de tirage est fixé comme une constante. Il faudrait qu'elle dépende de la complexité du modèle, par exemple le nombre de modalités.
— Il faut déterminer si seules les dégénérescences aboutissant à des probabilités non bornées déclenchent un redémarrage su SEM.
— MASSICCC ne gère peut-être pas le fait qu'un calcul puisse ne pas aboutir (comme on a supprimé les anciennes méthodes de gestion des dégénerescences).

# 8 Conclusion

Les modifications de la séquence d'initialisation permettent de rendre MixtComp plus performant, et augmentent la simplicité et la cohérence de l'architecture.

Les modifications effectuées durant l'été 2017 sont :

— La suppression des deux systèmes de gestion des dégénérescence : les estimateurs biaisés et le Gibbs pour les variables latentes.
— La mise en place d'une méthode par relance en cas de dégénérescence.
— L'initialisation des paramètres utilisant des individus représentatifs, au lieu de tirages de partitions des individus.
— L'utilisation des probabilités observées pour améliorer les initialisation.
— La simplification de la procédure d'initialisation.

# Références


[1] Vincent KUBICKI. Stabilisation de mixtcomp. 07/2017.

[2] Vincent KUBICKI. Avancement stabilisation mixtcomp. 08/2017.

[3] Vincent KUBICKI. Seconde Étape dans les modifications de mixtcomp. 08/2017.