Rapport de stage

Master 1 ANDROIDE : Esquerre-Pourtère Arthur

Apprentissage dans les réseaux de Markov : étude de l'existant et implémentation dans la librairie pyAgrum

---

Juin-Août 2020

Tuteur de stage : Monsieur Pierre-Henri Wuillemin

Organisme d'accueil : LIP6/CNRS

Enseignant référent : Monsieur Thibaut Lust

Établissement : Sorbonne Université (UPMC)

Résumé

La représentation de probabilités peut être faite sous la forme d'un tableau où les lignes correspondent à toutes les combinaisons possibles des variables, et où on trouverait sur la colonne de droite les probabilités associées à ces combinaisons. Malheureusement, dans le cas où nous avons beaucoup de variables aléatoires, ou bien si elles peuvent prendre beaucoup de valeurs différentes, le nombre de lignes dans notre tableau risque de devenir très grand. Ainsi, on peut utiliser des modèles graphiques, pouvant être représentés par des graphes dans lesquels chaque nœud représente une variable aléatoire et chaque arête représente une dépendance entre ces variables. Les modèles graphiques permettent en conséquence de factoriser les probabilités et ainsi, de simplifier le problème.

La librairie pyAgrum [1] est une librairie python conçue pour l'utilisation de modèles graphiques et notamment les réseaux bayésiens. Cette librairie est en fait un wrapper pour la librairie aGrUM codée, elle, en C++. Cette librairie permet aussi depuis quelques mois de construire des réseaux de Markov. Toutefois, contrairement aux réseaux bayésiens, il n'y a pas actuellement dans cette librairie d'algorithme implémenté permettant d'apprendre la structure d'un réseau de Markov, ni ses paramètres.

Le but de ce stage est donc de réaliser un état de l'art en étudiant les algorithmes d'apprentissage de réseaux de Markov existants puis de choisir et implémenter certains de ces algorithmes afin qu'ils soient, plus tard, ajoutés dans la librairie pyAgrum. Il est important de préciser que le sujet principal du stage est bien la recherche d'algorithmes et la réalisation d'une démonstration de faisabilité, d'où l'utilisation du langage python, facile d'utilisation. L'implémentation finale dans la librairie ne fait pas partie du sujet de ce stage et devra donc être réalisée dans un travail futur, en utilisant le langage C++, plus efficace, et nécessitera un travail d'optimisation des algorithmes pour être le plus performant possible.

Dans l'idéal, on souhaiterait avoir, à la fin du stage, au moins un algorithme d'apprentissage des paramètres ainsi qu'au moins un algorithme d'apprentissage de la structure.

Sommaire

Introduction

Dans le cadre de ma première année de master d'informatique, spécialité ANDROIDE, à Sorbonne Université, j'ai eu l'occasion d'effectuer un stage de plus de deux mois pendant l'été, afin de mettre en application les connaissances acquises pendant mes études.

J'ai donc été accueilli par l'équipe DECISION du laboratoire LIP6 afin de travailler avec un enseignant chercheur et de découvrir plus en profondeur le domaine de la recherche, qui m'intéresse beaucoup. L'enseignant chercheur qui m'a encadré, et qui a donc été mon tuteur de stage est Monsieur Pierre-Henri WUILLEMIN.

De plus, pendant ce stage, mon enseignant référent à Sorbonne Université a été Monsieur Thibaut LUST.

Dans le cadre de la pandémie de Covid-19 et des mesures sanitaires qui y sont associées, le travail a été effectué non pas dans les locaux du LIP6, mais exclusivement en télétravail.

L'encadrement du stage a donc eu lieu sous la forme d'appels téléphoniques hebdomadaires avec le tuteur de stage ainsi que par l'utilisation de messageries instantanées.

Le sujet exact du stage, qui a été choisi avec M.WUILLEMIN, est le suivant : *Apprentissage dans les réseaux de Markov : étude de l'existant et implémentation dans la librairie pyAgrum.*

Le cahier des charges de ce stage a été le suivant :

- État de l'art : apprentissage dans les modèles graphiques non orientés
- Discussions et, si besoin, sélection d'un ou de plusieurs algorithmes
- Implémentation d'algorithme(s) d'apprentissage paramétrique
- Implémentation d'algorithme(s) d'apprentissage structurel
- Réalisation des tests et validations

Le calendrier originel du stage prévoyait d'effectuer les tâches dans l'ordre donné ci-dessus, en particulier, une semaine était prévue pour me former aux modèles graphiques en général ainsi que pour m'habituer à l'utilisation de la librairie pyAgrum. Il était prévu que la deuxième semaine du stage soit consacrée à la réalisation d'un état de l'art de l'apprentissage dans les réseaux de Markov. La suite du stage a consisté, quant à elle, à implémenter les algorithmes que nous avions choisis, enfin, il était prévu de consacrer les dernières semaines au test des algorithmes implémentés.

Il était compliqué de faire un cahier des charges ainsi qu'un calendrier prévisionnel très précis dès le début du stage, étant donné que la plupart du travail réalisé et le temps nécessaire pour l'effectuer dépendaient beaucoup de ce qui serait trouvé en effectuant l'état de l'art.

Les objectifs du cahier des charges ont été réalisés en intégralité, ainsi, après plusieurs semaines de recherches afin de me former à l'utilisation des modèles graphiques et à faire l'état de l'art de l'apprentissage dans les modèles graphiques non orientés, plusieurs algorithmes ont été choisis, implémentés puis testés. En particulier, un algorithme d'apprentissage des paramètres et deux algorithmes d'apprentissage de la structure ont été implémentés.

De plus, un autre travail, non précisé clairement au départ dans le cahier des charges, mais qui était nécessaire pour tester les algorithmes, a été effectué avant l'implémentation de ces derniers. Ce travail consistait à réfléchir à des méthodes permettant de créer des bases de données d'apprentissage puis à les implémenter.

État de l'art

Les modèles graphiques sont des modèles probabilistes qui peuvent être représentés par des graphes, orientés ou non, et qui permettent d'exprimer des dépendances conditionnelles entre des variables aléatoires.

Parmi ces modèles graphiques, on retrouve les réseaux bayésiens, qui sont des modèles graphiques pouvant être représentés par des graphes orientés sans circuit et les réseaux de Markov, qui sont représentés par des graphes non orientés. Nous allons par la suite nous intéresser principalement aux réseaux de Markov. Toutefois, dans le cadre de ce travail sur les modèles graphiques, un document sur les réseaux bayésiens a aussi été créé, vous le trouverez en annexe. De même un document, réalisé lors de ce stage, sur l'utilisation de la librairie pyAgrum est aussi en annexe.

2.1 Définition et propriétés des réseaux de Markov

Comme expliqué ci-dessus, les réseaux de Markov ou champs aléatoires de Markov, sont aussi des modèles graphiques, ils sont cependant non orientés.

Les nœuds représentent des variables aléatoires et les arêtes des dépendances entre ces variables. Mais contrairement aux réseaux bayésiens, les nœuds n'ont pas de parents ni d'enfants, ils sont juste voisins s'ils sont reliés par une arête. Voir figure [2.1].

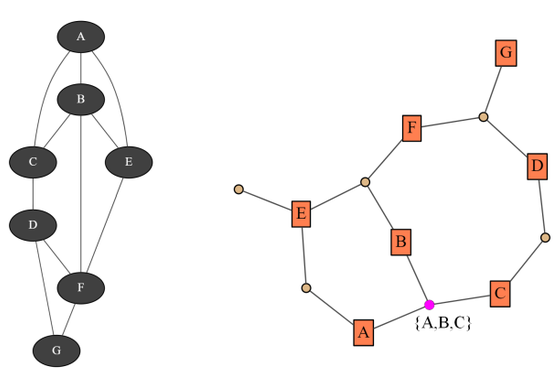


Figure 2.1: Un exemple de réseau de Markov ainsi que sa représentation sous la forme d'un $factor\_graph$

De plus, contrairement aux réseaux bayésiens, les nœuds n'ont pas chacun une table de probabilités conditionnelles. En effet, les nœuds forment des cliques (i.e. un sous-ensemble de sommets dont tous les sommets sont adjacents deux à deux), à chacune de ces cliques est associée une table de potentiels, qu'on appelle souvent $factor\_potentials$ ou $clique\_potentials$. Dans la suite de ce rapport, les valeurs dans ces tables seront appelées valeurs de facteur ou valeurs de potentiel. Il est important de noter qu'il s'agit du potentiel entre plusieurs variables de prendre certaines valeurs et qu'il ne s'agit pas d'une probabilité. Ces valeurs peuvent donc être supérieures à 1 et ne doivent bien sûr pas nécessairement sommer à 1. Ces valeurs de potentiels permettent donc de factoriser la loi jointe qu'on peut retrouver ainsi :

$$P(X_1, ..., X_n) = \frac{1}{Z}\tilde{P}(X_1, ..., X_n)$$

Avec :

$$\tilde{P}(X_1, ..., X_n) = \phi_1(D_1) * \phi_2(D_2) * ... * \phi_m(D_m) = \prod_d \phi_d(D_d)$$

Et Z qui permet de normaliser :

$$Z = \sum_{X_1,...,X_n} \tilde{P}(X_1, ..., X_n)$$

Avec $\phi_1(D_1), ..., \phi_K(D_K)$ les facteurs de notre graphe.

		A	
C	B	0	1
0	0	0.6461	0.2636
	1	0.3277	0.7873
1	0	0.7475	0.4664
	1	0.1612	0.0727

Figure 2.2: Table représentant les valeurs des potentiels pour la clique A,B,C (en rose sur la figure [2.1])

Pour déterminer si un nœud X est indépendant d'un nœud Y sachant un ensemble de nœuds S dans un réseau de Markov, la procédure est assez simple intuitivement, il suffit de regarder s'il existe un chemin dans le graphe entre X et Y ne passant par aucun nœud de S. De plus un nœud X ayant pour couverture de Markov un ensemble de nœuds MB, est indépendant de n'importe quel nœud $Y \notin MB$ sachant MB.

Comme pour les réseaux bayésiens, on trouve le concept de couverture de Markov dans les réseaux de Markov. La différence est que, cette fois, la couverture de Markov d'un nœud correspond à l'ensemble de ses nœuds voisins. Voir figure [2.3]. On peut la définir ainsi : $MB(X) = min_T(X \perp\!\!\!\perp G \setminus T|T)$ avec X un nœud quelconque et G l'ensemble des nœuds du graphe sauf X.

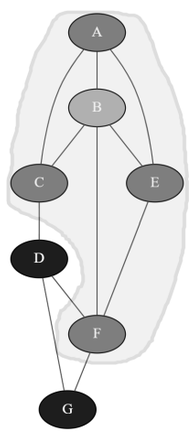


Figure 2.3: Couverture de Markov dans un réseau de Markov, pour le nœud B

Conditional random field [2]

Les conditional random fields (CRF ou Champ aléatoire conditionnel en français), sont une variante des réseaux de Markov, particulièrement utilisée dans le domaine du machine learning, notamment pour le TAL ou encore pour la vision par ordinateur.

Contrairement aux réseaux de Markov, dans le cas des CRF on définit un ensemble de variables (i.e. un ensemble de nœuds) observées, souvent nommé X. Les variables qui ne font pas partie de cet ensemble sont regroupées dans un autre ensemble Y. Il s'agit des variables que l'on va chercher à prédire. En effet, dans le cas des CRF, on cherche à représenter la probabilité conditionnelle $\Pr(Y \mid X)$.

On s'intéresse donc uniquement aux probabilités conditionnelles des variables non observées sachant les données observées. On ne représente donc pas les dépendances des données observées.

Les CRF sont donc particulièrement utiles lorsque l'on cherche à prédire toujours les mêmes variables en ayant toujours les mêmes données, comme c'est souvent le cas en machine learning. C'est donc ce qui les différencie des réseaux de Markov qui, eux, ne font pas de différences entre variables observées et non observées et pour lesquels on peut donc faire de l'inférence pour toutes les variables et qui sont donc plus flexibles... mais où l'on doit en contrepartie avoir les probabilités conditionnelles pour toutes les variables.

2.2 Apprentissage dans les réseaux de Markov

On distingue deux types principaux de méthodes qu'on utilise pour l'apprentissage dans les réseaux de Markov : les méthodes à base de contraintes, souvent appelées **constraint-based approaches** ainsi que les méthodes basées sur un score, appelées **score-based approaches**.

Constraint-based approaches [3]

Dans le cas des méthodes à base de contraintes, on recherche un graphe qui satisfait les hypothèses d'indépendance que l'on observe en étudiant les données. Ce type de méthode est bien plus simple à mettre en place que dans les réseaux bayésiens car les indépendances associées à la séparation dans un réseau de Markov sont bien plus simples que les indépendances associées à la d-séparation dans les réseaux bayésiens.

Toutefois, ce type d'approche a aussi des problèmes, ces méthodes sont sensibles au bruit et risquent donc de faire de mauvaises hypothèses d'indépendance (le bruit dans les données ou une quantité faible implique un manque de robustesse des tests statistiques utilisés).

De plus, ce type d'approche donne une structure mais pas les paramètres, donc il faut compléter cela par une méthode d'estimation des paramètres. On a donc besoin de la log-vraisemblance, ou bien d'un autre score, ce qui pose des problèmes, comme expliqué ci-dessous.

Pour la plupart de ces algorithmes, on doit faire des hypothèses : qu'il existe un réseau de Markov qui est une perfect map (i.e. un réseau qui capture parfaitement toutes les indépendances) pour notre distribution de probabilités, que notre réseau a un nombre maximum d'arête par sommet (pour rendre le calcul plus rapide/faisable)...si ces hypothèses ne sont pas respectées, on risque de trouver une mauvaise structure. De plus en pratique, il n'existe pas toujours de perfect map qui soit un graphe compact.

Les méthodes types constraint-based approaches peuvent parfois être un bon aperçu de la structure du graphe et donc un point de départ pour l'utilisation des méthodes de type score-based approaches.

Score-based approaches [3] [4]

Pour les méthodes du type score-based approaches, on définit une fonction objectif (par exemple la log-vraisemblance) que l'on cherche à optimiser. Dans le cas de l'utilisation de la log-vraisemblance, il n'y a pas d'expression de forme fermée pour maximiser la log-vraisemblance mais la fonction est concave donc on peut faire une méthode itérative comme une descente de gradient.

L'un des avantages de ce type d'approche est que cela permet d'obtenir une structure mais aussi les paramètres... mais c'est aussi un inconvénient car l'estimation de ces paramètres qui se fait à chaque itération est très coûteuse.

Scores alternatifs [3]

Le problème si l'on prend la log-vraisemblance comme fonction objectif est que l'on a besoin de faire de l'inférence à chaque itération, ce qui prend beaucoup de temps de calcul. On utilise donc souvent des méthodes d'inférence approximative, moins coûteuses, ou bien on prend une fonction objectif à optimiser qui n'est pas la log-vraisemblance.

Utilisation d'inférence approximative pour la log-vraisemblance

Étant donné que faire de l'inférence exacte pour calculer le gradient peut être très coûteux, on utilise souvent de l'inférence approximative, qui est bien plus rapide. Des algorithmes tels que le Gibbs sampling ou encore le loopy belief propagation sont souvent utilisés.

Toutefois en utilisant de tels algorithmes, on risque d'avoir une erreur dans l'estimation du gradient ou bien encore un gradient qui oscille, ce qui peut poser problème pour la convergence de notre algorithme.

Il existe malgré tout des méthodes qui permettent de réduire ces problèmes.

Pseudo-vraisemblance

Le calcul de la pseudo-vraisemblance se fait ainsi : pour une instanciation X de toutes les variables, on fait le produit pour chaque variable x_j de cette instanciation de sa probabilité étant donné les valeurs de toutes les autres variables. Toutefois, puisque nous sommes dans le cadre des réseaux de Markov, il est inutile de donner toutes les valeurs de l'instanciation, en effet il suffit de donner les valeurs des variables qui font parti de la couverture de Markov de la variable x_j. On obtient la formule suivante :

$$PL(X) = \prod_{j=1}^{n} \Pr(x_j \mid MB(x_j))$$

Avec $MB(x_j)$, la couverture de Markov de la variable x_j.

L'intérêt de faire cela est qu'on retire l'aspect exponentiel du calcul, ce qui le rend bien plus facile à faire.

De plus, en pratique on utilise la pseudo-log-vraisemblance plutôt que d'utiliser directement la pseudo-vraisemblance car cela permet de transformer le produit en somme, ce qui a beaucoup d'avantages d'un point de vue computationnel.

Voici la formule de calcul de la pseudo-log-vraisemblance sur un jeu de données :

$$\ell_{\mathrm{PL}}(\boldsymbol{\theta} : \mathcal{D}) = \frac{1}{M} \sum_{m} \sum_{j} \ln P(x_j[m] \mid \boldsymbol{x}_{-j}[m], \boldsymbol{\theta}).$$

Figure 2.4: Formule pseudo-vraisemblance [3]

La fonction de la pseudo-vraisemblance est concave, donc la descente de gradient permettra d'obtenir l'optimum.

En théorie, vraisemblance et pseudo-vraisemblance ont le même optimum mais en pratique, le manque de données ou bien un modèle pas suffisamment expressif peuvent faire que ces deux approches auront des résultats différents.

Si on observe beaucoup de variables et qu'on ne fait des requêtes que sur quelques variables, la pseudo-vraisemblance est plus adaptée. À l'inverse, si on fait une requête sur un grand nombre de variables, la vraisemblance est plus adaptée, mais elle sera plus lente.

Contrastive divergence

Quand on augmente la vraisemblance, cela revient en fait à l'augmenter sur les instances de notre jeu de données et donc à réduire la vraisemblance des autres données. À partir des paramètres actuels de notre réseau de Markov, on obtient des échantillons. La contrastive divergence consiste ensuite à augmenter la différence de probabilité entre les instances de notre jeu de données et les instances qu'on a obtenues en échantillonnant. On fait cela en boucle jusqu'à converger. L'utilisation d'une méthode comme le Gibbs sampling permet d'accélérer l'échantillonnage.

Maximum-margin training

Le méthode du maximum-margin training est utilisable uniquement dans le cas des conditional random fields. Dans un conditional random field, on cherche à prédire Y sachant X. Autrement dit, si dans notre jeu de données on a la paire (y[m],x[m]), on voudrait que notre modèle donne la plus haute probabilité à y[m]. On cherche donc à ce que la probabilité de y[m] sachant x[m] soit supérieure à la probabilité de y sachant x[m] pour tout $y \neq y[m]$. On cherche donc à augmenter la différence entre la log-probabilité de y[m] sachant x[m] et le maximum (en fonction de y) de la log-probabilité de y sachant x[m]. Cette différence est appelée margin, et on cherche donc à la maximiser.

Log linear model [3] [4]

On peut choisir de représenter notre factor-graph sous la forme d'un log-linear model. Avec D un sous-ensemble de variables et $\phi(D)$ un facteur ayant pour scope D. On réécrit un facteur $\phi(D)$, ainsi :

$$\phi(D) = \exp(-\epsilon(D)) \text{ avec } \epsilon(D) = -\ln(\phi(D))$$

Tout les réseaux de Markov utilisant des facteurs strictement positifs peuvent être représentés par cette forme logarithmique.

On définit une feature $f(D)$ comme étant une fonction allant des valeurs de D vers l'ensemble des réels $\mathbb{R}$. En pratique, dans le cas de données discrètes, on utilise notamment des features qui sont des ensembles de tests $X_i = x_i$, avec X_i une variable et x_i une valeur de cette variable. On peut par exemple donner la valeur 1 si le test est positif et zéro sinon.

Une feature est en fait un factor, qui peut être négatif.

On peut donc définir notre log-linear model à partir d'un réseau de Markov H ainsi :

- Un ensemble de features $f_1(D_1)....,f_k(D_k)$, avec D_i un sous-ensemble de noeuds de H
- Un ensemble de poids $w_1....,w_k$

$\Pr(X_1,...,X_n) = \frac{1}{Z}\exp\left(-\sum_j w_j f_j(D_j)\right)$

Plusieurs features peuvent avoir le même scope et ainsi, il est possible de représenter un ensemble de tables de potentiels.

Il existe plusieurs algorithmes de type **score-based approaches** qui utilisent ces log linear model, comme on peut le voir ci-dessous.

Algorithmes d'apprentissage de structure

Dans les prochains paragraphes, nous allons présenter plusieurs algorithmes, issus d'articles scientifiques.

Learning Markov Network Structure with Decision Trees [5]

Dans cet algorithme, on utilise les log linear model pour représenter un réseau de Markov sous forme de feature et de poids.

Pour chaque variable X_i , on apprend un arbre de décision pour représenter la probabilité conditionnelle de X_i étant donné toutes les autres variables. Chaque nœud intérieur teste la valeur d'une variable et chaque arc sortant correspond à une valeur de cette variable. Chaque feuille représente la probabilité conditionnelle de X_i étant donné les valeurs attribuées aux variables dans les nœuds ancêtres.

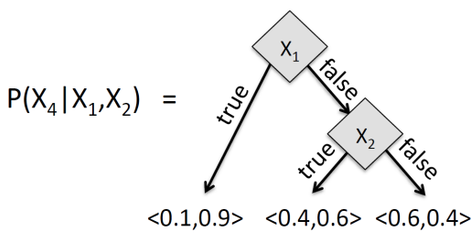


Figure 2.5: Exemple d'un arbre de décision utilisé pour cet algorithme, tiré de [5]

On sélectionne la variable de chaque nœud intérieur de manière à maximiser la log-vraisemblance conditionnelle.

Ces arbres de décisions peuvent ensuite être transformés en features. Une fois que l'on a appris ces features, il reste à apprendre les poids correspondant en maximisant la log-vraisemblance ou bien la pseudo-vraisemblance.

Comparé à d'autres algorithmes d'apprentissage dans les réseaux de Markov, celui-ci donne des résultats similaires mais son principal avantage est qu'il est très rapide.

D'autres algorithmes se basant sur celui expliqué précédemment existent, notamment en le combinant avec la L1-regularisation. Voir [6].

Efficient Structure Learning of Markov Networks using L1-Regularization [7]

Cet algorithme utilise le gradient pour choisir les features à ajouter : le but est d'ajouter en premier les features les plus pertinentes, afin non pas d'améliorer le résultat, mais d'améliorer significativement le temps de calcul. En pratique toutefois, cela permet aussi

d'obtenir de meilleurs résultats. On peut aussi décider de ne pas uniquement choisir la feature sur la valeur de son gradient mais aussi sur le gain à plus long terme que l'on estime que l'on va gagner en ajoutant cette feature.

La L1-Regularization permet de ne choisir que les features réellement significatives en mettant beaucoup de poids à 0. Alors qu'une L2-Regularization pénalise beaucoup les poids élevés mais peu les poids peu élevés, on obtient alors beaucoup de features inutiles associées à des poids peu élevés, ce qui complexifie les calculs. La L1-Regularization permet aussi de réduire le problème d'overfitting.

Il est nécessaire de faire cette régularisation ou bien d'empêcher d'une manière ou d'une autre d'avoir un modèle trop complexe sans quoi notre algorithme va faire de l'overfitting et rajouter des arêtes entre la plupart des nœuds, cela nuira à la performance du modèle (sur les données autres que celles du jeu de test) et en plus, le fait d'avoir un système trop complexe le rendra beaucoup plus lent pour faire les calculs.

Efficient Markov Network Structure Discovery Using Independence Tests [8]

On se base sur plusieurs hypothèses pour utiliser ces algorithmes, en effet, la distribution doit être graph-isomorph et on fait l'hypothèse qu'il existe un oracle idéal qui peut répondre aux requêtes d'indépendance statistique. En pratique, un tel oracle n'existe pas. Mais on peut en implémenter une approximation en faisait des tests d'indépendance statistique sur le jeu de données, en particulier en utilisant le test du chi-2. On obtient ainsi une p-value qu'on compare avec un seuil $1 - \alpha$, on considère en général $\alpha = 0.05$.

Le but est de faire des tests d'indépendance pour réduire l'ensemble des structures possibles jusqu'à ce qu'il n'en reste qu'une.

L'algorithme GSMN est une adaptation pour les réseaux de Markov de l'algorithme GS, utilisé dans les réseaux bayésiens. Cet algorithme consiste en 2 phases : la phase grow et la phase shrink. Pendant la phase grow, une variable Y est ajoutée à la couverture de Markov de X si et seulement si $(X \not\perp\!\!\!\perp Y \mid MB(X))$. Dans la phase grow, les variables peuvent être ajoutées dans un ordre quelconque. On risque d'avoir des faux positifs : des variables qui ne devraient pas appartenir à la couverture de Markov. On fait donc une phase shrink en testant si $(X \perp\!\!\!\perp Y \mid MB(X) - Y)$ si le test est positif, on retire Y de la couverture de Markov.

Toute combinaison de variables doit avoir une probabilité strictement positive.

L'algorithme GSIMN est une amélioration de l'algorithme GSMN du point de vue du temps de calcul. En plus de faire des test d'indépendance, cet algorithme utilise le *Triangle Theorem* (voir [2.6]), basé sur un ensemble d'axiomes (voir [2.7]) pour déduire des relations d'indépendances à partir de celles que l'on connaît déjà. De plus, les tests d'indépendance sont faits dans un ordre rigide, censé permettre de faire en priorité les tests les plus pertinents. Cela permet donc d'augmenter significativement la vitesse de calcul.

$$(X \not\perp\!\!\!\perp W \mid \mathbf{Z}_1) \wedge (W \not\perp\!\!\!\perp Y \mid \mathbf{Z}_2) \Longrightarrow (X \not\perp\!\!\!\perp Y \mid \mathbf{Z}_1 \cap \mathbf{Z}_2)$$
$$(X \perp\!\!\!\perp W \mid \mathbf{Z}_1) \wedge (W \not\perp\!\!\!\perp Y \mid \mathbf{Z}_1 \cup \mathbf{Z}_2) \Longrightarrow (X \perp\!\!\!\perp Y \mid \mathbf{Z}_1)$$

Figure 2.6: Triangle Theorem

$$\begin{array}{rrcl}
\textbf{Symmetry} & (\mathbf{X} \perp\!\!\!\perp \mathbf{Y} \mid \mathbf{Z}) & \iff & (\mathbf{Y} \perp\!\!\!\perp \mathbf{X} \mid \mathbf{Z}) \\
\textbf{Composition/} & (\mathbf{X} \perp\!\!\!\perp \mathbf{Y} \cup \mathbf{W} \mid \mathbf{Z}) & \iff & (\mathbf{X} \perp\!\!\!\perp \mathbf{Y} \mid \mathbf{Z}) \wedge (\mathbf{X} \perp\!\!\!\perp \mathbf{W} \mid \mathbf{Z}) \\
\textbf{Decomposition} & & & \\
\textbf{Intersection} & (\mathbf{X} \perp\!\!\!\perp \mathbf{Y} \mid \mathbf{Z} \cup \mathbf{W}) \wedge (\mathbf{X} \perp\!\!\!\perp \mathbf{W} \mid \mathbf{Z} \cup \mathbf{Y}) & \Longrightarrow & (\mathbf{X} \perp\!\!\!\perp \mathbf{Y} \cup \mathbf{W} \mid \mathbf{Z}) \\
\textbf{Strong Union} & (\mathbf{X} \perp\!\!\!\perp \mathbf{Y} \mid \mathbf{Z}) & \Longrightarrow & (\mathbf{X} \perp\!\!\!\perp \mathbf{Y} \mid \mathbf{Z} \cup \mathbf{W}) \\
\textbf{Transitivity} & (\mathbf{X} \perp\!\!\!\perp \mathbf{Y} \mid \mathbf{Z}) & \Longrightarrow & (\mathbf{X} \perp\!\!\!\perp \gamma \mid \mathbf{Z}) \vee (\gamma \perp\!\!\!\perp \mathbf{Y} \mid \mathbf{Z})
\end{array} \tag{3.1}$$

Figure 2.7: Axiomes

Learning Markov Network Structure using Few Independence Tests [9]

Cet article présente l'algorithme DGSIMN (Dynamic Grow-Shrink Inference-based Markov network learning algorithm), une amélioration de l'algorithme GSIMN. L'algorithme DGSIMN est plus rapide en termes de temps de calcul que l'algorithme GSIMN, tout en offrant des résultats similaires.

Tout comme l'algorithme GSIMN, on utilise des tests de type chi2 et des axiomes pour faire des tests d'indépendance.

Toutefois, contrairement à GSIMN qui fait ces tests dans un ordre rigide, l'algorithme DGSIMN, quant à lui, choisit dynamiquement et à la façon d'un algorithme gourmand le meilleur test à effectuer.

En faisant cela, on obtient parfois des résultats du type $(X \not\perp\!\!\!\perp Y \mid S)$ avec $S \neq D_x$. D_x étant la liste des variables qu'on a déjà trouvées dépendantes de X, qu'on représente par une colonne de noms de sommets. Si $D_x \nsubseteq S$, on ne peut pas ajouter Y à D_x. Mais plutôt que de ne pas prendre en compte le résultat du test, on va l'utiliser pour créer une deuxième colonne de noms de sommets, D_x va donc ainsi devenir un ensemble de colonnes. Le meilleur test à effectuer à chaque fois correspondra donc à celui qui permet de faire augmenter le plus possible la taille de ces colonnes.

Implémentation et résultats

3.1 Choix des algorithmes à implémenter

Après réflexions et discussions avec le tuteur de stage, il a été décidé d'utiliser un algorithme de type constraint-based pour la structure et donc d'apprendre séparément les paramètres.

L'apprentissage des paramètres se fera donc via la maximisation de la pseudo-log-vraisemblance, qui permet d'effectuer les calculs beaucoup plus rapidement que la log-vraisemblance. Pour cela nous utiliserons une méthode itérative, l'algorithme du gradient.

Concernant l'apprentissage de la structure, il a été décidé de se baser sur les algorithmes GS/GSMN/GSMNStar/GSIMN/DGSIMN [8] [9] présentés précédemment. Il n'est pas prévu d'implémenter tout ces algorithmes, mais ces algorithmes ayant tous la même base de fonctionnement, en se complexifiant à chaque fois, le choix exact des algorithmes à implémenter se fera en fonction des résultats obtenus et du déroulement du stage.

3.2 Création d'une base de données d'apprentissage

Afin de tester les algorithmes que nous allons implémenter, il est nécessaire d'avoir une base de données contenant des échantillons que l'on pourra utiliser pour apprendre les réseaux de Markov représentant les distributions de probabilités à l'origine de ces échantillons.

Tout d'abord, on doit s'assurer que ces distributions de probabilités sont bien graph-isomorph et donc représentables par un réseau de Markov. Afin de s'assurer de cette propriété, il a été décidé d'obtenir nos échantillons directement depuis des réseaux de Markov. De plus, cela nous aidera pour avoir des métriques pour comparer nos algorithmes : on pourra comparer le graphe appris avec le graphe ayant réellement servi à échantillonner.

La première étape consiste donc à générer des réseaux de Markov aléatoirement. L'algorithme fonctionne ainsi :

- on choisit le nombre de nœuds du graphe (i.e. le nombre de variables aléatoires), sa densité (qui permet d'influer sur le nombre d'arêtes), ainsi que la taille du domaine des variables aléatoires ;
- création aléatoire d'un arbre reliant entre tous les nœuds représentant les variables aléatoires, en utilisant l'objet UndiGraph de la librairie pyAgrum ;
- ajout aléatoire d'arêtes à ce graphe, jusqu'à obtenir la densité souhaitée ;
- transformation du graphe en réseau de Markov, les facteurs correspondent aux cliques maximales de ce graphe, les cliques sont trouvées en utilisant l'algorithme de Bron–Kerbosch [10] [11].

Le nombre d'arêtes final souhaité dans un graphe, contenant N nœuds, ainsi généré peut-être calculé ainsi, avec C la densité du graphe : $\lfloor (N-1) \times (C+1) \rfloor$.

L'indice de densité correspond donc au nombre d'arêtes présent dans l'arbre (N-1), multiplié par 1 plus cet indice. Ainsi, si on génère un graphe de 9 nœuds avec une densité de 0.25, notre arbre aura 8 arêtes, puis on ajoutera $0.25 \times 8 = 2$ arêtes. Le graphe aura donc $8 + 2 = 10$ arêtes à la fin.

Comme montré dans la formule, on arrondit le résultat du calcul au nombre inférieur, puisque le nombre d'arêtes doit être un entier. Pour cette raison, plusieurs valeurs différentes, mais proches, de densité peuvent donner le même nombre d'arêtes au final.

Après avoir créé le réseau de Markov, il est nécessaire de faire de l'échantillonnage pour créer la base de données. Pour cela, on utilise donc une méthode de type Monte-Carlo : pour chaque échantillon, on échantillonne une variable en utilisant de l'inférence puis une deuxième en faisant une inférence, sachant la valeur de la première variable, et ainsi de suite jusqu'à avoir échantillonné toutes les variables.

En pratique, l'algorithme implémenté est un peu plus malin, ceci afin d'éviter de faire trop d'inférences, méthode qui est assez coûteuse. Les variables sont toujours échantillonnées

dans le même ordre, et l'inférence se fait sachant uniquement l'ensemble minimal nécessaire pour faire l'inférence des variables déjà échantillonnées. Par exemple, si on échantillonne D sachant A, B et C et que D est indépendant de A sachant B et C (pas de chemin possible entre A et D ne passant ni par B ni par C), alors on peut ignorer la valeur de A et faire l'inférence uniquement en fonction des valeurs de B et C. Ainsi, une fois l'inférence effectuée, celle-ci est stockée en mémoire cache et pourra être réutilisée, pour un prochain échantillon ayant tiré les mêmes valeurs dans l'ensemble minimal nécessaire pour faire l'inférence.

Toutefois, cette méthode, qui consiste à stocker en cache les inférences déjà effectuées fonctionne bien en pratique mais ne serait pas adaptée à certaines situations, en particulier du big-data, qui risquerait de poser des problèmes dans la mémoire de l'ordinateur en remplissant le cache.

3.3 Apprentissage des paramètres

3.3.1 Implémentation

Bien qu'ayant trouvé des documents scientifiques [3] [4] sur l'utilisation de l'algorithme du gradient pour optimiser la pseudo-log-vraisemblance dans un réseau de Markov, tout ces documents utilisaient un log-linear modèle. Étant donné qu'il a été décidé ne pas se restreindre à ce cadre, nous avons décidé de nous inspirer de ces documents pour dériver directement par nous-mêmes la formule de la pseudo-log-vraisemblance afin de faire cet algorithme. Cette dérivation se trouve en annexe.

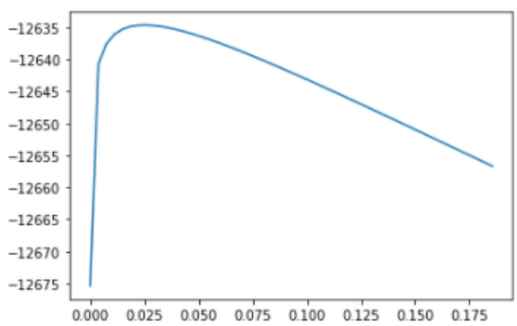


Figure 3.1: Évolution de la pseudo-log-vraisemblance d'un ensemble d'échantillons en fonction de la valeur d'un facteur (dont la valeur optimale vaut environ 0.033)

Au début, nous avions utilisé la formule habituelle du pas du gradient à savoir :
$x_{t+1} = x_t + \alpha \times \nabla x_t$
Avec x la valeur d'un facteur que l'on optimise, α le taux d'apprentissage et ∇x_t la valeur du gradient. Cela a posé des problèmes, et après analyse, nous avons vu que cela était dû à la forme que prend la fonction de la pseudo-log-vraisemblance en fonction d'une valeur d'un facteur, lorsque la valeur optimale est très petite. Comme on peut le voir dans la figure [3.1], la dérivée est très grande lorsqu'on se rapproche de 0 (la pseudo-log-vraisemblance tend vers moins l'infini), alors que celle-ci est plus faible lorsqu'on prend des grandes valeurs. Cela avait pour conséquence que notre algorithme faisait parfois des "bonds" beaucoup trop grands et avait ainsi beaucoup de mal à converger.

Pour régler ce problème, nous avons essayé d'utiliser un pas fixe, qui était négatif ou positif en fonction du signe du gradient, malheureusement, soit le pas était trop petit et rendait l'algorithme trop lent à converger, soit il était trop grand et empêchait l'algorithme de trouver une valeur proche de l'optimal. Il a donc été décidé d'utiliser un pas multiplicatif strictement positif que l'on appellera "step", autrement dit on utilise la formule suivante :
$x_{t+1} = x_t \times (1 + step)$ si ∇x_t est positif,
$x_{t+1} = x_t \times \frac{1}{(1+step)}$ si ∇x_t est négatif.
En pratique, cette méthode a très bien fonctionné, de plus, afin de pouvoir se rapprocher le plus possible de l'optimum, il est nécessaire de pouvoir rendre ce pas plus petit lorsqu'on est proche de cet optimum. Ainsi, nous utilisons un pas adaptatif, cette adaptation est déterminée par un paramètre "stepDiscount" compris entre 0 et 1, 0 et 1 non inclus. Lorsque l'algorithme repère qu'il a fait un pas trop grand (cela se traduit par une augmentation du gradient, étant donné que la fonction de la pseudo-log-vraisemblance est concave, on le voit par ailleurs sur la figure [3.1]), il met à jour

la valeur de “step” ainsi $step = step \times stepDiscount$. De plus, si l'on souhaite ne pas utiliser de pas adaptatif, il suffit de fixer la valeur de stepDiscount à 1.

Il est important de préciser que le choix de l'utilisation de la pseudo-log-vraisemblance implique certaines contraintes. En effet, ce score à optimiser se base sur plusieurs hypothèses. Premièrement, aucun facteur ne doit avoir une valeur égale à 0, toutes les valeurs doivent être strictement positives. Cette contrainte est importante et peut parfois empêcher de représenter vraiment une distribution de probabilité. Deuxièmement, la pseudo-log-vraisemblance a le même maximum que la log-vraisemblance uniquement pour une base de données suffisamment grande pour représenter la distribution de probabilité. Si l'une de ces hypothèse n'est pas respectée, l'algorithme peut potentiellement converger vers une mauvaise solution, voire ne pas converger du tout.

Concernant le point de départ du gradient, nous avons essayé d'utiliser les probabilités jointes pour les valeurs de chaque facteur. Bien que ce point de départ soit plus proche des valeurs optimales qu'un point de départ aléatoire (nous avons observé cela grâce à la divergence de KL, expliquée ci-dessous), ce point de départ rendait l'algorithme beaucoup plus lent à converger. Nous n'avons pas trouvé la raison exacte pour laquelle l'algorithme était plus lent à converger. Répondre à cette question et trouver un meilleur point de départ pourrait faire l'objet d'un travail futur et pourrait potentiellement accélérer grandement l'algorithme. En effet, cela n'ayant pas fonctionné, nous avons utilisé un point de départ aléatoire, pour lequel chaque valeur de chaque facteur est un nombre aléatoire strictement positif.

3.3.2 Tests

Métriques

Afin d'évaluer la performance des algorithmes, des métriques sont nécessaires, pour cela nous allons utiliser des métriques permettant de comparer directement des distributions de probabilités. Nous allons donc calculer la table représentant la distribution de probabilités du graphe original avec celle représentant la distribution de probabilité du graphe appris. En particulier, nous allons utiliser la divergence de Kullback-Leibler [12], que nous appellerons KL, ainsi que la différence maximale entre deux probabilités de cette table.

Toutefois, la taille des tables de ces distributions de probabilités augmentant de manière exponentielle en fonction de la dimensionalité, ces métriques ne pourront être utilisées que sur des petits réseaux de Markov dont les variables peuvent prendre peu de valeurs différentes.

Nous allons aussi nous intéresser au temps de calcul pour mesurer l'efficacité de l'algorithme en fonction des différentes situations. Ces temps de calculs ont été mesurés sur un ordinateur ayant le processeur suivant : “Intel Core i7-9750H CPU @ 2.60GHz” .

Résultats des tests et analyse

L'algorithme d'apprentissage des paramètres étant relativement lent, les tests suivants ont été effectués sur des graphes de petites tailles, c'est à dire des graphes de 8 nœuds. De plus, les métriques utilisées n'étant pas normalisées, elles ne permettent pas de comparer les résultats obtenus sur des graphes ayant un nombre de nœuds différent.

Nous avons donc créé aléatoirement des graphes de 8 nœuds, avec des valeurs de densité de 0 à 0.5 avec 100 graphes pour chaque valeur de densité. Pour chaque graphe, nous avons obtenu 1000 échantillons.

Les tests suivants ont été faits en prenant une valeur de 0.15 pour le paramètre “step” et une valeur de 0.9 pour “stepDiscount”, en apprenant grâce aux échantillons que nous avons obtenus.

Le réseau de Markov qui nous servira de point de départ pour les tests suivants correspond au réseau original avec les mêmes arêtes et facteurs entre ces nœuds, mais dont les valeurs de ces facteurs ont été effacées puis remplacées par d'autres valeurs aléatoires.

Dans les graphes [3.2], nous comparons le résultat obtenu après 15 itérations de l'algorithme du gradient avec le réseau de Markov pour lequel on aurait choisi des valeurs aléatoires pour chaque valeur de chaque facteur. Ceci permet d'avoir une expérience témoin pour comparer l'efficacité de notre algorithme, en effet ce graphe ayant des valeurs aléatoires correspond en fait au graphe que l'on utilise comme point de départ de notre gradient, comme expliqué ci-dessus, on mesure ainsi l'amélioration qu'a permis de faire notre algorithme.

Il est aussi important de noter que le choix des 15 itérations est arbitraire et a été choisi pour des raisons de temps de calculs. On pourrait choisir de continuer à itérer pour continuer à converger et améliorer le résultat.

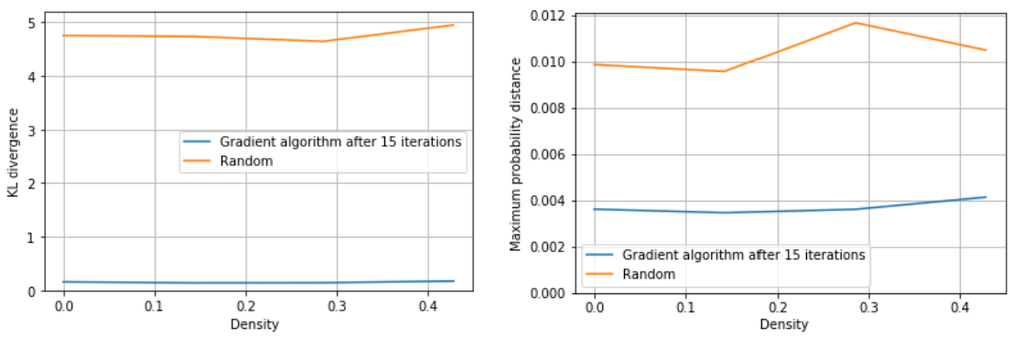


Figure 3.2: Évolution de la divergence KL et de la différence maximale de probabilité en fonction de la densité des graphes générés aléatoirement, les valeurs affichées correspondent aux moyennes obtenues sur 100 graphes de taille 8.

Les figures [3.2] nous montrent très nettement que notre algorithme a permis d'améliorer de beaucoup les paramètres de notre réseau de Markov selon nos métriques et confirment ainsi l'efficacité de nos algorithmes.

On notera également grâce aux graphes [3.2] que la densité du graphe semble ne pas avoir d'influence sur la qualité du résultat final.

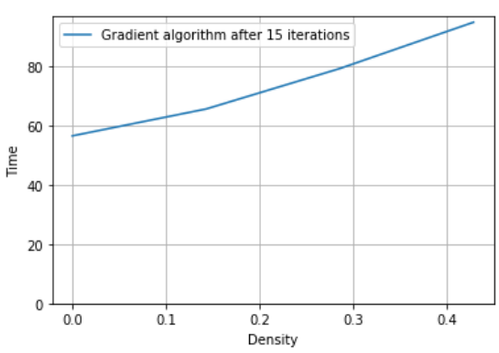


Figure 3.3: Évolution du temps de calcul en fonction de la densité des graphes générés aléatoirement, les valeurs affichées correspondent aux moyennes obtenus sur 100 graphes de taille 8.

Comme on le voit dans la figure [3.3] l'algorithme devient plus long quand la densité du graphe augmente, c'est logique puisque augmenter la densité a tendance à créer des facteurs de plus grandes tailles ayant de plus en plus de valeurs à optimiser.

3.4 Apprentissage de la structure

Comme expliqué ci-dessus, nous avions décidé pour l'apprentissage de la structure, de nous intéresser aux algorithmes présentés dans cet article : [8]. Les algorithmes qui ont été implémentés et sont donc expliqués ci-dessous sont les algorithmes GSMN et GSMNStar. Toutefois, bien que nous ayons prévu au départ de nous intéresser à un troisième algorithme, l'algorithme GSIMN, nous avons au final décidé ne pas le faire. En effet, après une lecture plus approfondie de l'article le présentant, nous avons trouvé plusieurs erreurs dedans, ce qui posait des problèmes pour l'implémenter. Toutefois, l'implémentation a été réalisée en majeure partie et pourrait potentiellement être utilisée dans le futur si ces problèmes étaient résolus.

3.4.1 Implémentation

Nous avons commencé par implémenter l'algorithme GSMN présenté précédemment, son pseudo-code est trouvable dans l'article [8]. Cet algorithme est donc une version adaptée pour les réseaux de Markov de l'algorithme GS, adapté aux réseaux bayésiens.

Son implémentation a été relativement facile. Après cela, l'algorithme GSMNStar a été implémenté, celui-ci fonctionne comme l'algorithme GSMN, en utilisant un ordre particulier pour les tests d'indépendance, et en évitant la répétition de tests en théorie inutiles, permettant ainsi un calcul plus rapide. Son pseudo code se trouve aussi dans l'article [8]. Une fois de plus, cet algorithme n'a pas posé de problème particulier.

Pour les tests d'indépendance, nous avons utilisé le chi2, qui est déjà implémenté dans pyAgrum, comme présenté dans l'article [8]. Mais on pourrait choisir d'en utiliser d'autres, comme le test G2, lui aussi implémenté dans pyAgrum.

De plus, pour la valeur de alpha, qui permet de déterminer le seuil de fiabilité, au lieu de la valeur de 0.05 utilisée dans l'article, il a été choisi de prendre une valeur très petite de 0.0001, qui s'est montrée plus performante dans les tests que nous avons réalisés. Cette valeur peut toutefois être ajustée, par exemple dans le cas où le coût d'un faux négatif et le coût d'un faux positif seraient différents.

Toutefois, un problème pour ces algorithmes a été détecté, en effet, les tests d'indépendance statistique, comme le chi2 ou le G2, peuvent poser problème car le calcul devient trop lourd lorsque le nombre de variables connues est trop grand. L'algorithme peut donc potentiellement ne pas fonctionner dans certaines situations, en particulier dans le cas où le graphe original a un facteur contenant beaucoup de variables, surtout si chaque variable peut prendre de nombreuses valeurs.

3.4.2 Tests

Métriques

Des métriques sont nécessaires pour évaluer la performance des algorithmes précédemment cités.

Nous allons donc nous intéresser au nombre de faux positifs dans le graphe appris comparé au graphe original, c'est à dire le nombre d'arêtes présentes dans le graphe appris mais non présentes dans le graphe original. Inversement, nous allons nous intéresser au nombre de faux négatifs, c'est-à-dire les arêtes présentes dans le graphe original mais pas dans le graphe appris.

De plus, nous allons aussi observer la distance de Hamming normalisée [8], qui correspond au nombre d'erreurs (faux positifs et faux négatifs), divisé par le nombre d'arêtes possibles dans le graphe. Cette normalisation permet de pouvoir comparer efficacement, même pour un nombre de nœuds différent. Cette distance sera appelée par la suite *NHD*.

Enfin, comme pour l'apprentissage de paramètres, nous allons nous intéresser au temps de calcul des algorithmes.

Résultats des tests et analyse

Pour effectuer les tests des algorithmes d'apprentissage de structure, étant donné que ces algorithmes sont rapides, nous avons pu utiliser des graphes bien plus grands, avec de plus grands nombres d'échantillons.

Nous avons donc créé des graphes de 10 à 70 nœuds, avec des valeurs de densité de 0 à 0.5 avec 100 graphes pour chaque couple de valeur de nœuds et de valeur de densité. Pour chaque graphe, nous avons obtenu 10 000 échantillons, que nous allons utiliser pour faire l'apprentissage.

On peut aussi noter que, pour de petits graphes, la densité peut ne pas faire varier le nombre d'arêtes, comme expliqué précédemment. Par exemple pour un graphe de taille 10, une densité de 0 est en fait pareille qu'une densité de 0.05, par conséquent, les tests expliqués ci-dessous ont utilisé les mêmes graphes pour les densités 0 et 0.05 pour les graphes de taille 10. Nous avons fait cela chaque fois que le problème se présentait.

Le point de départ utilisé pour ces algorithmes était un réseau de Markov n'ayant aucune arête, ces algorithmes vont donc choisir les meilleures arêtes à rajouter.

Le graphe [3.4] nous montre que l'algorithme GSMN est bien plus performant que l'algorithme GSMNStar, quelle que soit la taille du graphe, faisant en moyenne environ deux fois moins d'erreurs que GSMNStar. De plus, l'erreur selon la métrique semble diminuer lorsque la taille du graphe augmente.

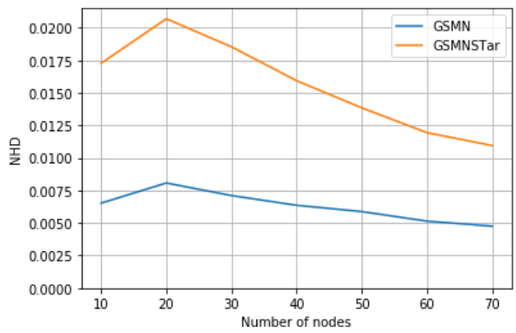


Figure 3.4: Évolution de la NHD en fonction du nombre de nœuds des graphes générés aléatoirement, les valeurs affichées correspondent aux moyennes pour tous les graphes générés avec ce nombre de nœuds, quelle que soit la densité.

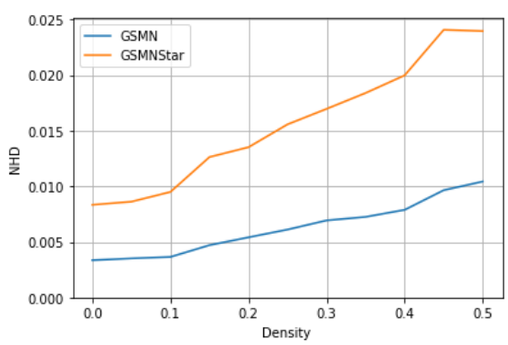


Figure 3.5: Évolution de la NHD en fonction de la densité des graphes générés aléatoirement, les valeurs affichées correspondent aux moyennes pour tous les graphes générés avec cette densité, quel que soit leur nombre de nœuds.

Le graphe [3.5] nous montre aussi que l'algorithme GSMN est bien plus performant que l'algorithme GSMNStar, quelle que soit la densité du graphe, faisant environ deux fois moins d'erreurs. On remarque que les deux algorithmes font beaucoup plus d'erreurs lorsque le graphe devient dense que lorsque le graphe est proche d'un arbre.

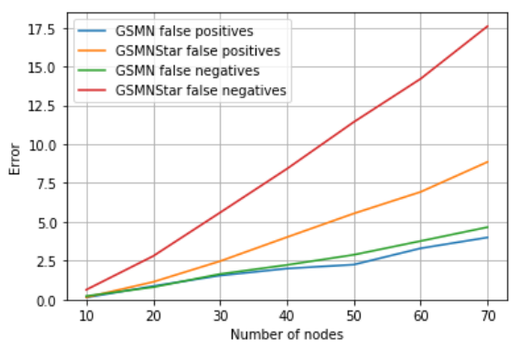


Figure 3.6: Évolution du nombre de faux positifs et de faux négatifs en fonction du nombre de nœuds des graphes générés aléatoirement, les valeurs affichées correspondent aux moyennes pour tous les graphes générés avec ce nombre de nœuds, quelle que soit la densité.

Comme on le voit dans le graphe [3.6], le nombre brut d'erreurs augmente avec le nombre de nœuds, ce qui est logique car cette métrique n'est pas normalisée et que cela augmente le nombre d'arêtes possibles et donc le risque de se tromper. On observe que dans notre expérience, on obtient plus de faux négatifs que de faux positifs, mais comme expliqué ci-dessus, on peut modifier cela en changeant le seuil de fiabilité.

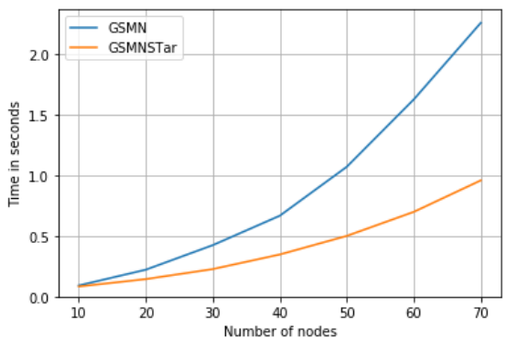


Figure 3.7: Évolution du temps de calcul en fonction du nombre de nœuds des graphes générés aléatoirement, les valeurs affichées correspondent aux moyennes pour tous les graphes générés avec ce nombre de nœuds, quelle que soit la densité.

Le graphe [3.7] nous montre que l'algorithme GSMNStar est beaucoup plus rapide que l'algorithme GSMN, ce qui est logique étant donné qu'il a été conçu pour cela et qu'il permet de faire moins de tests statistiques, qui sont relativement longs à effectuer.

L'algorithme GSMNStar, bien qu'il soit plus rapide que GSMN fait plus d'erreurs que l'algorithme GSMNStar, en effet même si ces algorithmes renvoient toujours le graphe optimal lorsqu'on utilise un oracle parfait pour les tests statistiques, ce type d'oracle n'est pas utilisable en pratique, ce pourquoi on utilise des tests statistiques de type chi2. Le nombre d'erreurs plus grand de l'algorithme GSMNStar s'explique par le fait que celui-ci évite de faire certains tests qui sont en théorie inutiles (dans le cas d'un oracle parfait), seulement étant donné que l'on utilise des tests statistiques, qui peuvent se tromper, certains tests considérés comme inutiles, ne le sont pas en réalité. Par exemple, si l'algorithme GSMNStar a trouvé que A est indépendant de B sachant la couverture de Markov de A, B devrait en théorie être indépendant de A sachant sa couverture de Markov, et l'algorithme ne fera pas ce dernier test. L'algorithme GSMN, lui, fera ce test quand même et rajoutera une arête entre A et B si au moins l'un des deux tests précédents trouve une dépendance. Étant donné la manière dont fonctionne ce test d'indépendance, il se trompe très peu lorsqu'il trouve une dépendance, par contre, il a un fort risque de se tromper en trouvant une indépendance, il est donc pertinent de faire les deux tests précédents plutôt qu'un seul d'entre eux.

3.5 Apprentissage de la structure puis des paramètres

Nous avons ensuite repris la même configuration que pour l'apprentissage de paramètres, mais cette fois nous comparons le résultat obtenu en prenant comme point de départ la structure du graphe original et celui obtenu en prenant comme point de départ la structure apprise via la base de données. En effet, même pour apprendre les paramètres, nous ne connaissons pas forcément la structure originale et devons donc l'apprendre.

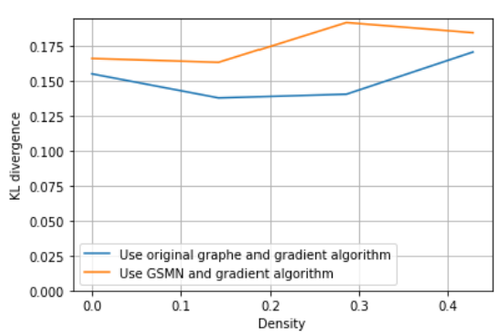


Figure 3.8: Évolution de la divergence KL en fonction de la densité des graphes générés aléatoirement, les valeurs affichées sont les moyennes obtenues sur 100 graphes de taille 8.

Comme on pouvait s'y attendre, on obtient de meilleurs résultats en utilisant le graphe de départ, toutefois cette différence n'est pas très grande et utiliser une structure apprise permet d'obtenir des résultats relativement proches.

Conclusion et perspectives

Ce stage a été très instructif et nous a permis à mon tuteur de stage et à moi-même d'en apprendre plus sur les réseaux de Markov et leur apprentissage.

De plus, ayant au final un algorithme d'apprentissage de paramètres et même deux algorithmes d'apprentissage structurel, les principaux objectifs du projet ont été atteints.

Toutefois, les algorithmes présentés ci-dessus montrent pour la plupart des limitations, par exemple, l'algorithme de descente de gradient ne supporte pas les facteurs ayant une valeur nulle, ce problème pourrait peut-être être réglé en utilisant un autre score à optimiser que la pseudo-log-vraisemblance. D'une manière générale, il serait intéressant d'étudier des solutions nous permettant de ne plus avoir les limitations des algorithmes qui ont été implémentés.

De plus, de futurs travaux seront nécessaires afin de réaliser l'implémentation finale dans la librairie pyAgrum. En effet, comme expliqué auparavant, cette librairie est en fait un wrapper pour la librairie aGrUM codée, elle, en C++. Le code réalisé en python dans ce stage ne peut donc pas être incorporé directement dans cette librairie, il sera nécessaire avant de l'implémenter en C++ dans aGrUM. Il pourrait aussi être pertinent d'effectuer un travail d'optimisation, par exemple en utilisant les meilleures structures de données à chaque fois, afin de rendre ces algorithmes plus performants.

D'autres algorithmes d'apprentissage des réseaux de Markov pourraient aussi être étudiés plus en profondeur et potentiellement implémentés dans pyAgrum, en particulier, les algorithmes de type score-based approaches, qui ont été présentés dans l'état de l'art mais n'ont pas été retenus pour l'implémentation.

Pour l'optimisation, nous avons utilisé une méthode itérative qui dans notre situation a été assez lente : l'algorithme du gradient. Étant donné la forme concave [3.1] de la fonction que l'on optimise, on pourrait utiliser d'autres méthodes itératives, comme la méthode de Newton, qui utilise la dérivée seconde. La formule de la dérivée seconde est présente en annexe. Ce type de méthode pourrait potentiellement être plus rapide, ainsi il serait possible d'effectuer l'apprentissage des paramètres de réseaux de Markov plus grands.

Enfin, les algorithmes étudiés pendant ce stage couvrent les réseaux de Markov en général mais il pourrait être intéressant dans le futur de développer des algorithmes spécifiques aux conditional random fields.

Remerciements

Je tiens à remercier toutes les personnes qui ont contribué directement ou indirectement à la réalisation de ce projet, qui m'a permis de découvrir plus en profondeur le monde de la recherche et m'a aussi conforté dans mes projets d'avenir.

Je souhaite remercier en particulier M.WUILLEMIN, mon tuteur de stage au LIP6, avec qui j'ai beaucoup apprécié travailler et qui m'a beaucoup aidé en me donnant des pistes de réflexion ainsi que des conseils pour résoudre les problèmes auxquels j'ai été confronté. Il a toujours été très présent pour répondre aux questions que je lui ai posées.

Je remercie aussi M.LUST, qui a été mon enseignant référent à Sorbonne Université pendant ce stage.

Je remercie également toute l'équipe enseignante du master informatique de Sorbonne Université, qui m'a permis d'obtenir les connaissances théoriques et pratiques nécessaires à la réalisation de ce type de projet.

Références bibliographiques

[1] C. Gonzales, L. Torti, and P.-H. Wuillemin, "aGrUM: a Graphical Universal Model framework", in *International Conference on Industrial Engineering, Other Applications of Applied Intelligent Systems*, ser. Proceedings of the 30th International Conference on Industrial Engineering, Other Applications of Applied Intelligent Systems, Arras, France, Jun. 2017. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01509651.

[2] H. Larochelle. (2020). Hugo larochelle - YouTube, [Online]. Available: https://www.youtube.com/channel/UCiDouKcxRmAdc5OeZdiRwAg (visited on 07/07/2020).

[3] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009, 1268 pp., Google-Books-ID: 7dzpHCHzNQ4C, ISBN: 978-0-262-01319-2.

[4] D. Koller. (2020). Probabilistic graphical models, Coursera, [Online]. Available: https://www.coursera.org/specializations/probabilistic-graphical-models? (visited on 07/03/2020).

[5] D. Lowd and J. Davis, "Learning markov network structure with decision trees", in *2010 IEEE International Conference on Data Mining*, 2010, pp. 334–343. DOI: 10.1109/ICDM.2010.128.

[6] ——, "Improving markov network structure learning using decision trees", *J. Mach. Learn. Res.*, vol. 15, pp. 501–532, 2014.

[7] S.-i. Lee, V. Ganapathi, and D. Koller, "Efficient structure learning of markov networks using l_1-regularization", in *Advances in Neural Information Processing Systems 19*, B. Schölkopf, J. C. Platt, and T. Hoffman, Eds., MIT Press, 2007, pp. 817–824.

[8] F. Bromberg, D. Margaritis, and V. Honavar, "Efficient markov network structure discovery using independence tests", *The journal of artificial intelligence research*, vol. 35, no. 1, pp. 449–484, 2009, ISSN: 1076-9757.

[9] P. Gandhi, F. Bromberg, and D. Margaritis, "Learning markov network structure using few independence tests", in *Proceedings of the 2008 SIAM International Conference on Data Mining*, Society for Industrial and Applied Mathematics, 2008, pp. 680–691. DOI: 10.1137/1.9781611972788.62. [Online]. Available: https://epubs.siam.org/doi/10.1137/1.9781611972788.62.

[10] *Bron–kerbosch algorithm*, in *Wikipedia*, Page Version ID: 966624522, Jul. 8, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Bron%E2%80%93Kerbosch_algorithm&oldid=966624522 (visited on 08/25/2020).

[11] D. Eppstein, M. Löffler, and D. Strash, "Listing all maximal cliques in sparse graphs in near-optimal time", *arXiv:1006.5440 [cs]*, 2010. arXiv: 1006.5440. [Online]. Available: http://arxiv.org/abs/1006.5440 (visited on 07/24/2020).

[12] *Kullback–leibler divergence*, in *Wikipedia*, Page Version ID: 973519700, Aug. 17, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Kullback%E2%80%93Leibler_divergence&oldid=973519700 (visited on 08/17/2020).

[13] I. Ben-Gal, "Bayesian networks", in *Encyclopedia of Statistics in Quality and Reliability*, American Cancer Society, 2008, ISBN: 978-0-470-06157-2. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470061572.eqr089 (visited on 06/23/2020).

Annexes

Dérivée de la pseudo-log-vraisemblance

On cherche à dériver la formule suivante en fonction de chaque valeur du facteur ϕ_d (on utilise $\phi_d(D_d)$ pour signifier qu'on s'intéresse à une valeur en particulier, D_d représente donc une instanciation de certaines variables) :

$$\sum_m \sum_j \ln(P(x_j, m \mid x\_j, m))$$

$$\frac{\partial}{\partial \phi_d(D_d)} \sum_m \sum_j \ln(P(x_j, m \mid x\_j, m)) = \sum_m \sum_j \frac{\partial}{\partial \phi_d(D_d)} \ln(P(x_j, m \mid x\_j, m))$$

La dérivée d'une somme étant la somme des dérivés, nous allons pouvoir calculer séparement chaque $\frac{\partial}{\partial \phi_d} \ln(P(x_j, m \mid x\_j, m)$ pour toutes les valeurs de m et j et les additioner ensuite.

Pseudo vraisemblance :

$$P(x_j \mid x\_j) = P(x_j \mid MB(x_j)) = \frac{P(x_j, MB(x_j))}{P(MB(x_j))} \tag{4.1}$$

$$= \frac{\tilde{P}(x_j, MB(x_j))}{\tilde{P}(MB(x_j))} = \frac{\tilde{P}(x_j, MB(x_j))}{\sum_{x'_j} \tilde{P}(x'_j, MB(x_j))} \tag{4.2}$$

On a donc cette formule pour la log-pseudo-vraisemblance :

$$\ln(P(x_j \mid x\_j)) = \ln(\tilde{P}(x_j, MB(x_j))) - \ln \sum_{x'_j} \tilde{P}(x'_j, MB(x_j)))$$

Avec :

$$\tilde{P}(X_1, ..., X_n) = \phi_1(D_1) * \phi_2(D_2) * ... * \phi_m(D_m) = \prod_d \phi_d(D_d)$$

On obtient donc :

$$\ln(P(x_j \mid x\_j)) \tag{4.3}$$

$$= \sum_{i:Scope[\phi_i] \ni X_j} (\ln(\phi_i(x_j, MB(x_j)))) - \ln(\sum_{x'_j} \prod_{i:Scope[\phi_i] \ni X_j} \phi_i((x'_j, MB(x_j))) \tag{4.4}$$

On dérive la log-pseudo-vraisemblance :

$$\frac{\partial}{\partial \phi_d(D_d)} \ln(P(x_j \mid x\_j)) \tag{4.5}$$

$$= 0 \text{ si } Scope[\phi_d] \not\ni X_j \text{ ou si } (x\_j) \neq D_d \tag{4.6}$$

" si $(x\_j) \neq D_d$" signifie : si les valeurs des variables définies dans D_d ne sont pas toutes égales aux valeurs des variables correspondantes dans (x_j).

Sinon :

$$= \left\{ \begin{array}{ll} \frac{1}{\phi_d(x_j, MB(x_j))} & \text{si } (x_j, x\_j) = D_d \\ 0 & \text{sinon.} \end{array} \right\} - \frac{\prod_{i \neq d, i:Scope[\phi_i] \ni X_j} \phi_i(x_j^D, MB(x_j))}{\sum_{x'_j} \prod_{i:Scope[\phi_i] \ni X_j} \phi_i(x'_j, MB(x_j))} \tag{4.7}$$

Avec x_j^D la valeur de la variable x_j dans D.

On peut aussi calculer la dérivée seconde, on obtient :

$$\left\{ \begin{array}{ll} -\frac{1}{\phi_d(x_j, MB(x_j))^2} & \text{si } (x_j, x\_j) = D_d \\ 0 & \text{sinon.} \end{array} \right\} + \frac{\prod_{i \neq d, i:Scope[\phi_i] \ni X_j} \phi_i(x_j^D, MB(x_j))^2}{(\sum_{x'_j} \prod_{i:Scope[\phi_i] \ni X_j} \phi_i(x'_j, MB(x_j)))^2} \tag{4.8}$$

Les réseaux bayésiens

Définition

Comme expliqué ci-dessus, les réseaux bayésiens sont des modèles graphiques orientés. Autrement dit, ce sont des graphes orientés acycliques, dans lesquels chaque nœud représente une variable aléatoire et chaque arc une dépendance entre ces variables. Plus simplement, cela signifie que la variable correspondant au nœud dont part l'arc, qu'on appelle le parent, a une influence sur la variable correspondant au nœud où arrive cet arc, qu'on appelle l'enfant. Il est donc important que le graphe ne soit pas acyclique, sinon cela implique qu'une variable a une influence (direct ou indirecte) sur elle-même.

La représentation sous forme de graphe permet donc de factoriser des probabilités conditionnelles. On pourrait représenter les probabilités conditionnelles sous forme de tableau où chaque ligne correspond à toutes les combinaisons possibles de ces variables, et sur le colonne de droite, on trouverait la probabilité associée à cette combinaison. Malheureusement, dans le cas où nous avons beaucoup de variables aléatoires, ou bien si elles peuvent prendre beaucoup de valeurs différentes, le nombre de lignes dans notre tableau risque de devenir trop grand, car c'est de nature combinatoire. Les réseaux bayésiens permettent donc souvent de résoudre ce problème en factorisant.

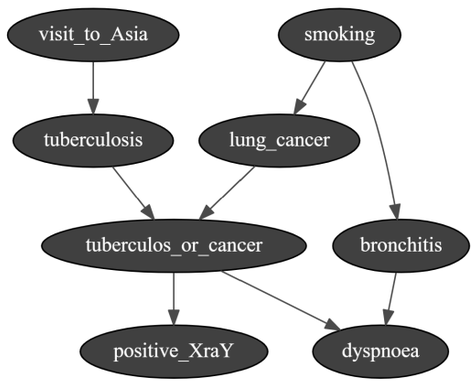


Figure 4.1: Un exemple de réseau bayésien

La figure [4.1] représente un exemple de réseau bayésien. Comme on peut le voir, les nœuds peuvent avoir plusieurs parents et/ou plusieurs enfants. Le nœud *lung_cancer* est donc influencé par le nœud *smoking*, de plus le nœud *lung_cancer* influence le nœud *tuberculos_or_cancer*. De plus, on observe que la variable *tuberculos_or_cancer* a seulement pour parent *lung_cancer* et *tuberculosis*, cela signifie que *tuberculos_or_cancer* est indépendant de *visit_to_Asia* et *smoking* étant donné *lung_cancer* et *tuberculosis*.

Plus généralement, pour savoir si une variable X est indépendante d'une autre variable Y sachant un ensemble de variable S, les réseaux bayésiens utilisent le principe de la D-séparation, qui est beaucoup plus compliqué que la séparation dans les réseaux de Markov.

Chaque nœud a donc une distribution de probabilités conditionnelles en fonction de ses parents. La taille de la table de probabilités conditionnelles d'un nœud dépend donc du nombre de parents qu'il a. Si le nœud n'a pas de parent, alors sa distribution de probabilités est dite *inconditionnelle* ou *apriori*.

	lung_cancer	
smoking	0	1
0	0.1000	0.9000
1	0.0100	0.9900

		dyspnoea	
bronchitis	tuberculos_or_cancer	0	1
0	0	0.9000	0.1000
	1	0.8000	0.2000
1	0	0.7000	0.3000
	1	0.1000	0.9000

Figure 4.2: Tables de probabilités conditionnelles

De plus, dans les réseaux bayésiens, on retrouve le concept de couverture de Markov. Dans un réseau bayésien, la couverture de Markov d'un nœud correspond à ses parents, ses enfants, ainsi que les parents de ses enfants, comme on peut le voir sur la figure [4.3]. Cela signifie que le nœud *lung_cancer* est indépendant de tout les autres nœuds du graphe, étant donné ceux qui sont dans sa couverture de Markov. On peut écrire cela ainsi :

$\Pr(lung\_cancer \mid MB(lung\_cancer), B) = \Pr(lung\_cancer \mid MB(lung\_cancer))$

Avec $MB(lung\_cancer)$ la couverture de Markov du nœud *lung_cancer* et B n'importe quel nœud différent de *lung_cancer*.

On peut généraliser cela pour n'importe quel nœud A :

$\Pr(A \mid MB(A), B) = \Pr(A \mid MB(A))$

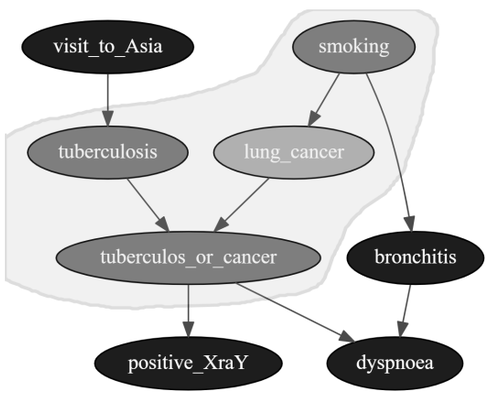


Figure 4.3: Couverture de Markov dans un réseau bayésien, pour le nœud *lung_cancer*

Inférence

On peut utiliser les réseaux bayésiens pour faire de l'inférence, c'est d'ailleurs l'utilisation principale que l'on a de ces réseaux. C'est-à-dire que l'on peut utiliser ce réseau pour calculer des probabilités étant donné des observations (i.e. des assignations de valeurs à certaines variables).

On peut utiliser 2 méthodes pour faire ces inférences :

- L'inférence exacte : calcul exact par énumération, c'est la méthode qui donne le meilleur résultat, mais le temps de calcul peut vite exploser si le nombre de variables est grand ou bien si le domaine de définition des variables est grand car la complexité est exponentielle, en effet, ce problème est NP-complet. (Pour certains types de graphes, il existe toutefois des algorithmes efficaces qui permettent de réduire le nombre de calculs à effectuer.)
- L'inférence approximative : utilisation de méthodes d'échantillonage de type Monte-Carlo pour estimer les probabilités recherchées. Ces méthodes donnent bien sûr des résultats moins précis que l'inférence exacte, mais elles sont bien plus efficaces en termes de temps de calcul. En pratique, c'est celle que l'on doit utiliser lorsqu'on manipule des gros graphes.

Correspondances entre réseaux de Markov et réseaux bayésiens

Les réseaux bayésiens peuvent être transformés en réseaux de Markov et inversement. En effet, les classes de problèmes que ces modèles peuvent représenter ne sont pas disjointes car certains problèmes peuvent être représentés par ces deux types de modèles.

Toutefois, il faut faire attention car ces deux types de modèles graphiques ne représentent pas tout à fait la même chose. Certains problèmes ne peuvent être représentés que par des réseaux de Markov, c'est le cas lorsqu'il y a des dépendances cycliques. Et inversement, certains ne peuvent être représentés par un réseau de Markov mais peuvent l'être par un réseau bayésien. C'est le cas notamment lorsqu'il y a des dépendances induites.

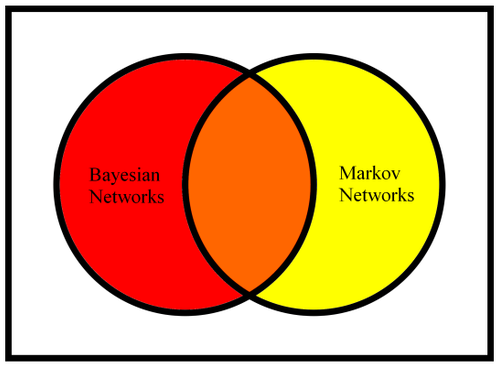


Figure 4.4: Classes des problèmes représentés par les réseaux bayésiens et ceux représentés par les réseaux de Markov.

Utilisation de pyAgrum

La librairie pyAgrum est une librairie python conçue pour l'utilisation de modèles graphiques et notamment les réseaux bayésiens mais aussi les arbres de décisions ou encore les réseaux de Markov. Cette librairie est en fait un wrapper pour la librairie aGrUM codée, elle, en C++.

Pour commencer, il faut importer la librairie pyAgrum ainsi que le module de notebook de pyAgrum qui est très pratique si on travaille sur un notebook :

```
import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
```

Réseaux bayésiens dans pyAgrum

Création et manipulations de base

On peut utiliser pyAgrum pour créer un réseau bayésien rapidement en spécifiant sa structure, par exemple avec le code suivant :

```
bn=gum.fastBN("A->B<-C;B->D")
```

La syntaxe $A->B$ signifie que l'on crée un arc partant de A et allant vers B.

On peut ensuite choisir de rajouter un arc de A vers D avec le code suivant :

```
bn.addArc("A","D")
```

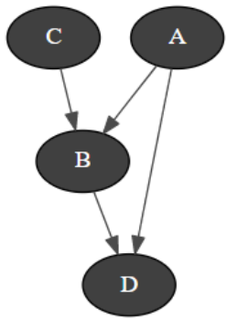


Figure 4.5: Un réseau bayésien créé avec pyAgrum

On obtient donc au final le graphe de la figure [4.5].

De plus, on peut modifier les probabilités conditionnelles d'un nœud en fonction de ses parents avec le code suivant:

```
bn.cpt("B")[{'A': 0, 'C': 0}] = [1, 0]
bn.cpt("B")[{'A': 0, 'C': 1}] = [0.1, 0.9]
bn.cpt("B")[{'A': 1, 'C': 0}] = [0.1, 0.9]
bn.cpt("B")[{'A': 1, 'C': 1}] = [0.01, 0.99]
```

		B	
C	A	0	1
0	0	1.0000	0.0000
	1	0.1000	0.9000
1	0	0.1000	0.9000
	1	0.0100	0.9900

Figure 4.6: Table de probabilités conditionnelles créée avec pyAgrum

On obtient ainsi la table de probabilités conditionnelles de la figure [4.6].

De nombreuses autres manipulations sont possibles avec pyAgrum, des tutoriels sous forme de notebook expliquent cela sur le site de pyAgrum.

Inférence

Comme expliqué précédemment, il existe deux types principaux de méthodes qui permettent de faire de l'inférence dans les réseaux bayésiens, l'inférence exacte et l'inférence approximative. Avec pyAgrum on peut utiliser ces deux types de méthodes.

Premièrement on peut faire de l'inférence exacte ainsi :

```
bn=gum.loadBN(os.path.join("res","asia.bif"))
ie=gum.LazyPropagation(bn)
ie.setEvidence({'smoking':0})
ie.makeInference()
```

Dans le code ci-dessus, on a fixé une évidence *hard* pour la variable *smoking*. Mais on pourrait aussi choisir de fixer une évidence *soft*. Une évidence *soft* correspond à une situation où l'on a une connaissance, incertaine, sur la valeur prise par une variable. Dans pyAgrum, si on veut dire que si la personne fume on a seulement 30 % de chance de le savoir et si elle ne fume pas, on a 80 % de chance de le savoir, cela se fait ainsi :

```
ie.setEvidence({'smoking':[0.3,0.8]})
```

On peut ensuite afficher le résultat ainsi et obtenir le résultat que l'on peut voir sur la figure [4.7] :

```
gnb.sideBySide(gnb.getInference(bn,evs={'smoking':0}),
               ie.posterior("positive_XraY"))
```

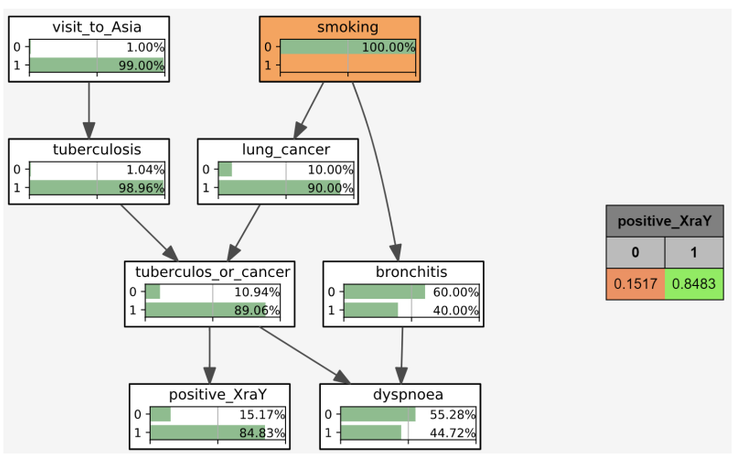


Figure 4.7: Différentes manières d'afficher les inférences

Deuxièmement, on peut choisir d'utiliser les méthodes d'inférence approximative. Et notamment les méthodes d'échantillonnages, parmi lesquelles on peut utiliser entres autres le *Weighted sampling* ou encore le *Gibbs sampling*.

Voici un exemple d'utilisation du *Gibbs sampling* :

```
bn=gum.loadBN(os.path.join("res","asia.bif"))
unsharpen(bn)
ie2=gum.GibbsSampling(bn)
ie2.setEpsilon(1e-2)
gnb.showInference(bn,engine=ie2,evs={'smoking':0})
```

Et on obtient ainsi les inférences que l'on peut voir sur la figure [4.8].

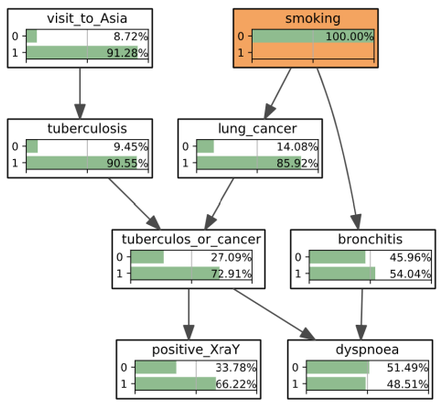


Figure 4.8: Inférences obtenues via la méthode d'échantillonage *Gibbs sampling*

On peut ensuite choisir de comparer les inférences obtenues par l'inférence approximative avec celles obtenues par la méthode d'inférence exacte, comme on peut le voir sur la figure [4.9].

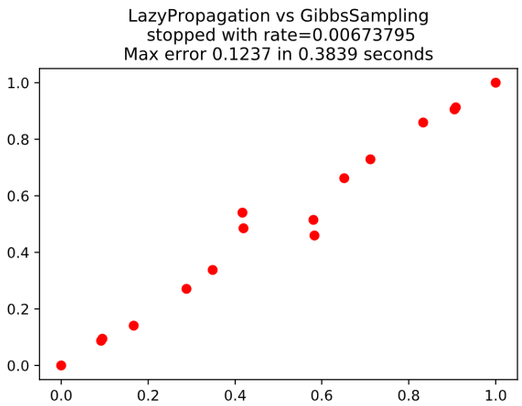


Figure 4.9: Comparaison des inférences obtenues par la méthode exacte et la méthode approximative

Apprentissage

Il est aussi possible d'utiliser pyAgrum pour faire de l'apprentissage dans les réseaux bayésiens. En particulier, on peut faire de l'apprentissage de paramètres avec le code suivant :

```
learner=gum.BNLearner(os.path.join("out","sample_asia.csv"),bn)
learner.setInitialDAG(bn.dag())
bn2=learner.learnParameters()
```

On passe le réseau *bn* en paramètre dans l'initialisation et dans la méthode $setInitialDAG$ afin d'avoir les variables du réseau ainsi que la structure qu'on l'on utilise pour apprendre ces paramètres.

On peut aussi faire l'apprentissage de la structure de ces réseaux, actuellement on peut utiliser 3 algorithmes différents dans pyAgrum pour faire cela : $GreedyHillClimbing$, $LocalSearchWithTabuList$ et $K2$.

Par exemple, l'algorithme $GreedyHillClimbing$ est un algorithme glouton itératif, où, à partir d'une structure initiale, on crée plusieurs graphes enfants en inversant, supprimant ou en ajoutant des arcs. Parmi les graphes ainsi créés, on prend le meilleur et on recommence en itérant et en prenant à chaque fois le meilleur graphe de la génération précédente et en créant les enfants à partir de ce graphe. Cet algorithme s'arrête une fois qu'il a trouvé un minimum local.

Dans pyAgrum, on utilise cet algorithme avec le code suivant :

```
learner=gum.BNLearner(os.path.join("out","sample_asia.csv"),bn)
learner.useGreedyHillClimbing()
bn2=learner.learnBN()
```

Réseaux de Markov dans pyAgrum

Création et manipulations de base

La librairie pyAgrum permet aussi de manipuler les réseaux de Markov.

On peut créer un réseau de Markov en lui spécifiant sa structure. Pour cela on utilise la syntaxe suivante : "$a-b-c$" signifie que l'on crée une clique contenant les sommets a, b et c. Cette clique aura donc des facteurs de potentiel. ";" permet de séparer les cliques. On peut donc utiliser cette méthode :

```
mn=gum.MarkovNet.fastPrototype("a-b-c;c-e-f;c-f-d-g")
```

De même avec cette autre méthode :

```
mn=gum.fastMN("a-b-c;c-e-f;c-f-d-g")
```

On peut voir à quoi ressemble un réseau ainsi créé sur la figure [4.10].

Le réseau de Markov ainsi créé aura des facteurs dont les valeurs sont aléatoires, on peut choisir de regénérer aléatoirement ces facteurs ainsi :

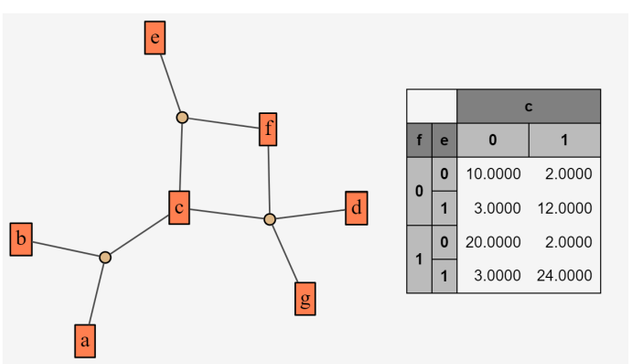
```
mn.generateFactors()
```

Ou bien, on peut choisir d'assigner à notre réseau des valeurs de facteurs à certaines cliques avec par exemple le code suivant :

```
mn.factor({'e','f','c'})[{}]=[[[10,2],[3,12]],[[20,2],[3,24]]]
```

On peut ensuite afficher les facteurs ainsi : (voir figure [4.10])

```
mn.factor({'c','e','f'})
```



		c	
f	e	0	1
0	0	10.0000	2.0000
	1	3.0000	12.0000
1	0	20.0000	2.0000
	1	3.0000	24.0000

Figure 4.10: Réseau de Markov créé avec $fastMN$ et assignation de facteurs à la clique c,e,f

Il est aussi possible de créer un réseau de Markov depuis un réseau bayésien déjà existant.

On peut le voir dans la figure [4.11]. Pour faire cela il suffit de charger un réseau bayésien puis d'utiliser la méthode permettant d'obtenir le réseau de Markov correspondant en utilisant le code ci-dessous :

```
bn=gum.loadBN(os.path.join("res","asia.bif"))
mn=gum.MarkovNet.fromBN(bn)
```

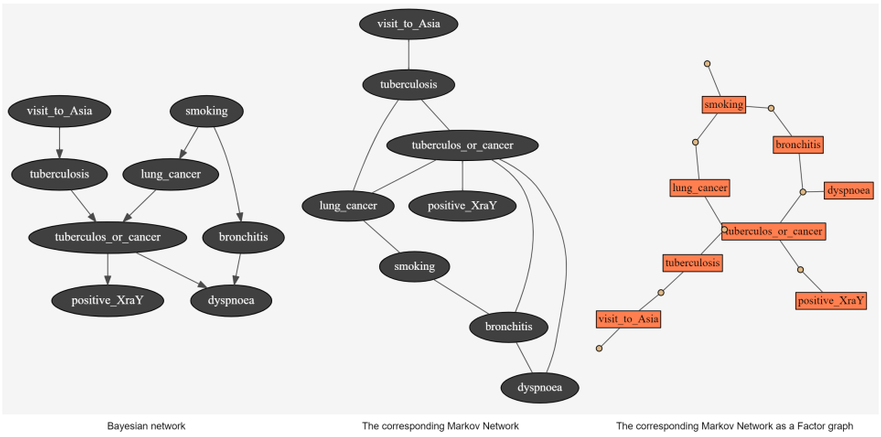


Figure 4.11: Un réseau de Markov créé depuis un réseau bayésien

De plus, comme montré précédemment, en particulier sur la figure [4.11], lorsqu'on représente un réseau de Markov, on peut le représenter sous la forme d'un graphe habituel où l'on affiche les arêtes entre les nœuds ou bien sous la forme d'un graphe où l'on montre les facteurs de chaque clique et où tous les sommets d'une clique sont reliés au facteur correspondant. Cette seconde représentation a un avantage, en effet la visualisation sous forme d'un graphe habituel ne permet pas de lire la factorisation de notre réseau, différents réseaux de Markov ayant des factorisations différentes peuvent correspondre au même graphe, une fois représenté sous cette forme. La représentation sous forme de $factor\_graph$ permet de résoudre ce problème.

On peut afficher facilement ces deux représentations d'un réseau de Markov mn grâce au code suivant :

```
gnb.sideBySide(gnb.getMN(mn,view="graph"),
               gnb.getMN(mn,view="factorgraph"))
```

Inférence

La librairie pyAgrum permet aussi de faire de l'inférence dans les réseaux de Markov. Pour cela on utilise l'algorithme de Shafer-Shenoy. Il s'agit d'un algorithme qui permet de faire de l'inférence exacte, cet algorithme n'est donc pas adapté en pratique si l'on souhaite l'utiliser sur de grands graphes.

Comme le montre le code ci-dessous il faut utiliser la classe $ShaferShenoyMNInference$. Ensuite on peut choisir de définir des évidences, $hard$ ou $soft$.

```
iemn=gum.ShaferShenoyMNInference(mn)
iemn.setEvidence({"a":1,"f":[0.4,0.8]})
iemn.makeInference()
```

Comme pour les réseaux bayésiens, on peut ensuite visualiser ces inférences sous forme de graphe ou bien uniquement pour les variables qui nous intéressent avec le code ci-dessous : (voir figure [4.12])

```
gnb.sideBySide(gnb.getInference(mn,evs={"a":1,"f":[0.4,0.8]}),
               iemn.posterior("b"))
```

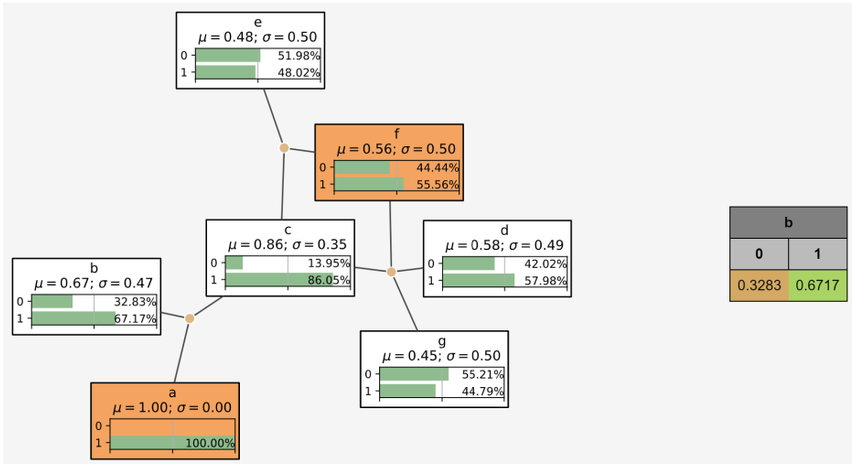


Figure 4.12: Différentes manières d'afficher les inférences