TP : Introduction à la programmation probabiliste avec Infer.Net

I. Table des matières

II. Introduction

A. Introduction à la programmation probabiliste

1. Avant-propos

a) Appréhension de l'incertitude

Les ordinateurs sont rigoureusement logiques mais le monde réel ne l'est pas, un simple fait qui peut poser de vrais défis pour les programmeurs. Par exemple, supposons que vous deviez représenter ce qu'un utilisateur griffonne sur un écran tactile sous forme de mot. Les gens ne sont généralement pas très appliquées dans leur écriture, donc ce gribouillis peut correspondre à "hill", ou est-ce « bull » ou peut-être même « hello » ? L'utilisateur sait ce qu'il a écrit, mais du point de vue de l'application, la valeur correcte est incertaine mais pas complètement : "hull" est plus probable que "gall" et vous pouvez complètement exclure des mots comme « train » ou « hélicoptère ».

Comment représentez-vous une telle incertitude dans un programme ? Des variables conventionnelles telles que bool ou int doivent avoir des valeurs bien définies. Un gribouillage a besoin d'une variable - nommons-la *unGribouillis*: elle représente l'un des mots possibles; vous ne savez pas lequel. Cependant, cette variable doit représenter l'incertitude de manière à intégrer votre compréhension de la probabilité que chacun des mots possibles soit le bon.

b) Variables aléatoires

La programmation probabiliste est conçue pour gérer une telle incertitude. Elle est basée sur les variables aléatoires, qui sont des extensions de types standard pouvant représenter des valeurs incertaines. Chaque variable aléatoire représente un ensemble ou une plage de valeurs possibles, et a une distribution associée qui assigne une probabilité à chaque valeur possible.

La distribution représente quantitativement votre compréhension des valeurs possibles et vous permet d'utiliser l'analyse statistique pour comprendre le comportement de la variable.

c) Modèles probabilistes

Ainsi, *unGribouillis* est maintenant représenté par une variable aléatoire. Comment obtenez-vous la distribution de *unGribouillis* - les probabilités pour chacun de ses mots possibles ? Du point de vue de l'utilisateur, il existe une relation de cause à effet entre un mot et le gribouillis associé. Avec la programmation probabiliste, vous pouvez construire un modèle probabiliste qui définit comment les utilisateurs transforment les mots en gribouillis. Ce modèle reconnaît, entre autres, que le même mot

peut conduire à différents gribouillis et que des mots différents peuvent conduire à des gribouillis similaires et il définit les probabilités associées.

d) Inférence probabiliste

Comment raisonner rétrospectivement d'un gribouillis particulier à l'ensemble des mots possibles et leurs probabilités ? La programmation probabiliste est basée sur une méthodologie statistique connue sous le nom d'inférence bayésienne. Elle permet de raisonner rétrospectivement depuis une observation - le gribouillage – jusqu'à son origine - les mots possibles et leur probabilités – à partir d'un modèle probabiliste.

e) Apprentissage probabiliste

Un modèle a généralement un ensemble de paramètres ajustables qui régissent son comportement. Comment déterminer les paramètres corrects ? Vous pouvez simplement assigner des valeurs basées sur votre compréhension générale de l'écriture manuscrite, ce qui pourrait fonctionner. Cependant, tout le monde écrit un peu différemment, donc les mots possibles associé à *unGribouillis* et leurs probabilités varient quelque peu d'un utilisateur à l'autre. Pour adapter les paramètres au style d'écriture d'un utilisateur, un programme probabiliste peut traiter les paramètres du modèle eux-mêmes comme des variables aléatoires et apprendre les valeurs réelles des paramètres en fonction des entrées de l'utilisateur.

f) Qualité des modèles

Comment savoir que vous ne manquez de rien ? Un modèle plus sophistiqué avec plus de variables fonctionnerait-il encore mieux ? Si vous ajoutez assez de variables à un modèle, vous pouvez adapter presque n'importe quoi. Cependant, vous atteignez généralement un point ou les rendements décroissent ; à un certain point la complexité supplémentaire commence à réduire la qualité du modèle. Vous avez besoin d'un principe de parcimonie ou « rasoir Occam » pour trouver un équilibre entre précision et complexité. Ce « rasoir » fait partie intégrante de l'inférence bayésienne, qui comprend un moyen robuste pour évaluer la qualité du modèle que vous pouvez utiliser pour choisir le meilleur modèle.

2. Fonctionnement de la programmation probabiliste

Le scénario « Cluedo » qui suit constitue un moyen pratique d'illustrer les concepts de base de la programmation probabiliste et définir une terminologie essentielle.

Roman policier

Vous rentrez chez vous après une soirée de whist chez votre voisin, pour découvrir que votre invité a été assassiné. Vous devez découvrir le coupable. Au départ, vous savez que :

• Il y a deux coupables possibles : le majordome et le cuisinier.

• Trois armes de meurtre sont possibles : un couteau de boucher, un pistolet et un tisonnier de cheminée.

Initialement, le coupable et l'arme du crime sont inconnus. Cependant, le raisonnement probabiliste et les observations peuvent aider à identifier le coupable probable, et ce faisant fournir une introduction aux bases de la programmation probabiliste.

a) Suspects et armes du crime : variables aléatoires

Vous avez besoin d'une variable pour représenter le coupable. Un suspect est coupable et l'autre pas, le choix évident est donc une variable pouvant prendre l'une des deux valeurs possibles. Un type booléen peut représenter deux valeurs possibles. Cependant, un type bool ne peut être que vrai ou faux, et vous ne pouvez pas être certain de la valeur réelle de la variable jusqu'à ce que le coupable avoue. Vous pouvez toutefois estimer la probabilité que chaque suspect soit le coupable.

Ce dont vous avez besoin est une variable aléatoire, qui étend essentiellement le domaine standard des types tels que bool ou double pour gérer les valeurs déterministes et incertaines.

- Une variable aléatoire a un ensemble ou une plage de valeurs possibles, qui sont tirées du domaine du type. Par exemple, les valeurs possibles d'une variable aléatoire bool sont true et false. Les valeurs possibles d'une variable aléatoire double sont des nombres réels sur une plage continue telle que [0, 1] ou [-∞, ∞].
- Chaque variable aléatoire a une distribution de probabilité associée qui spécifie la probabilité de chaque valeur possible. Par exemple, une variable aléatoire bool pourrait avoir une probabilité de 70% d'être vraie, et une probabilité de faux de 30%.

Le scénario Cluedo nécessite deux variables aléatoires :

- *leCoupable* : une variable aléatoire avec deux valeurs possibles: *leMajordome* et *leCuisinier*.
- *lArmeDuCrime* : une variable aléatoire avec trois valeurs possibles: *lePistolet*, *leCouteau*, et *leTisonnier*.

b) Le coupable probable : les distributions de probabilité

Même si vous ne pouvez pas dire définitivement qui a commis le meurtre à ce stade, vous pouvez estimer la probabilité que chaque suspect soit coupable. Chaque variable aléatoire est associée à une fonction connue sous le nom de distribution de probabilité - généralement raccourcie à « distribution » - qui assigne une probabilité à chacune des valeurs possibles de la variable.

Les distributions associées à *leCoupable* et *lArmeDuCrime* sont appelées Les distributions discrètes, qui attribuent des probabilités à un ensemble énumérable de valeurs possibles. Comme la valeur réelle doit être l'une des valeurs possibles, les probabilités doivent totaliser à 100%.

c) Mise en route : la distribution a priori

La programmation probabiliste commence par définir une distribution initiale pour chaque variable aléatoire, qui s'appelle une distribution a priori. Une distribution a priori définit votre compréhension d'une variable avant de faire des observations. Pour le coupable :

- Le majordome est un homme remarquable qui a fidèlement servi la famille pendant des années.
- Le cuisinier a été embauchée récemment et des rumeurs parlent d'un passé peu recommandable.

A partir de ces informations, vous estimez la probabilité a priori pour *leCoupable* d'être : *leMajordome* = 20% et *leCuisinier* = 80%.

d) Enquêter : observations et distribution postérieure

Les probabilités a priori sont un point de départ utile, mais vous pouvez améliorer votre compréhension si vous observez la valeur réelle d'une ou de plusieurs des variables aléatoires de votre modèle.

À ce stade, la variable n'est plus incertaine, ce qui vous permet de faire une meilleure estimation des distributions des autres variables.

Par exemple, supposons que vous sachiez que le majordome est plus susceptible d'avoir utilisé le pistolet, et le cuisinier plus susceptible d'avoir utilisé le couteau ou le tisonnier. Après avoir reçu le rapport du médecin légiste, vous pouvez en déduire une nouvelle distribution *leCoupable*, qui intègre l'observation de *lArmeDuCrime* dans la probabilité a priori de *leCoupable* et améliore votre estimation du coupable probable. Si l'arme du crime est le pistolet, *leMajordome* augmente et *leCuisinier* diminue.

Une observation ne supplante pas l'a priori- cette information est toujours valable - mais vous pouvez utiliser les informations supplémentaires pour réviser votre conviction a priori afin qu'elle reflète correctement toutes les données disponibles. La nouvelle distribution est appelée distribution a posteriori, ou juste "postérieur".

En fait, la distinction entre antérieur et postérieur peut parfois être quelque peu floue. Une manière plus générale et utile de regarder les prieurs et les postérieurs est la suivante :

- Un a priori représente votre compréhension du système avant de faire un ensemble particulier d'observations.
- Le postérieur correspondant représente votre compréhension du système après avoir fait les observations.

Supposons que vous apportiez une observation supplémentaire : un nouveau rapport du médecin légiste donne une estimation du moment où le meurtre a été commis. Avant de recevoir ce deuxième rapport, votre compréhension de la situation est représentée par la croyance postérieure que vous aviez après avoir lu le rapport du premier médecin légiste.

Ce postérieur est donc le choix logique pour le nouvel a priori, que vous pouvez ensuite utiliser avec le temps de meurtre observé pour déduire un deuxième postérieur. Ce type d'inférence est de nature incrémentale et est appelé apprentissage en ligne. Alternativement, vous revenez à la croyance a priori et revenez sur toutes les preuves à partir de zéro.

Plus vous avez de preuves, moins votre croyance a priori est importante. Les preuves concluantes doivent prévaloir sur toute croyance a priori, mais si vous n'avez pas beaucoup de preuves, la croyance a priori compte pour beaucoup plus.

e) Quantifier le problème : Distributions conditionnelles, conjointes et marginales

La discussion qui précède présente des arguments généraux sur la manière de déduire le coupable le plus probable, mais est un peu vague sur les chiffres réels. Cette section apporte une dimension quantitative.

(1) Distributions conditionnelles

Pour déduire un postérieur, vous devez d'abord construire un modèle mathématique pour le scénario du meurtre. Vous avez déjà spécifié un a priori pour *leCoupable*. Vous savez aussi que :

- Le majordome garde un vieux pistolet Webley de son temps dans l'armée dans un tiroir verrouillé, mais le cuisinier ne possède pas de pistolet. Le majordome est beaucoup plus susceptible d'avoir utilisé le pistolet.

- Le cuisinier dispose d'un grand nombre de couteaux de boucher tranchants et a interdit au maître d'hôtel à mettre les pieds dans la cuisine. Le majordome est moins susceptible d'avoir utilisé le couteau de boucher.
- Le majordome est beaucoup plus âgé que le cuisinier et devient un peu fragile. Le majordome est moins susceptible d'avoir utilisé une arme physiquement exigeante comme le tisonnier.

Le moyen le plus simple de commencer à construire un modèle est d'estimer la probabilité que les suspects aient utilisé chacune des armes possibles. Il s'agit d'une distribution conditionnelle, la distribution de *lArmeDuCrime*, conditionnée par une valeur particulière du coupable.

Pour simplifier la discussion, nous allons simplement spécifier les deux distributions, comme indiqué dans la table suivante :

	Pistolet	**Couteau**	**Poker**	
cuisinier	5%	65%	30%	**= 100%**
Majordome	80%	dix%	dix%	**= 100%**

Chaque suspect doit utiliser l'une des armes possibles pour que chaque distribution conditionnelle somme à 100%.

(2) Distributions conjointes

Vous pouvez utiliser l'a priori de *leCoupable* avant et les distributions conditionnelles de la précédente section pour construire un modèle pour le Cluedo, connu sous le nom de distribution conjointe. Une distribution conjointe représente les probabilités de toutes les combinaisons de valeurs possibles de la variable aléatoire-le cuisinier avec le couteau, le majordome avec le tisonnier, etc. Elle contient votre compréhension complète de la scène du meurtre avant de faire des observations.

	Pistolet	**Couteau**	**Poker**
cuisinier	4%	52%	24%
Majordome	16%	2%	2%

Chaque ligne contient les probabilités conditionnelles de la section précédente multipliées par les probabilités correspondantes de *leCoupable* a priori, 80% pour le *leCuisinier* et 20% pour *leMajordome*. Parce que la combinaison du coupable et de l'arme doit être l'une des ces six possibilités, elles totalisent 100%.

Vous pouvez créer des distributions conjointes de différentes manières. Tous produisent les mêmes nombres. L'approche utilisée ici est de la cause à l'effet - Le coupable sélectionne l'arme et l'utilise pour commettre le meurtre.

- *leCoupable* est la variable qui nous intéresse - et dont nous voulons déduire le postérieur - donc nous définissons un a priori pour cette variable.

- *lArmeDuCrime* est la variable que nous pouvons observer, nous définissons donc une distribution conditionnelle pour cette variable.

La cause à l'effet est généralement le moyen le plus simple de construire un modèle probabiliste même si nous voulons raisonner dans le sens inverse, par exemple en déduisant le coupable sur la base de la connaissance de l'arme du crime.

(3) Distributions marginales

Vous pouvez utiliser la distribution conjointe pour poser diverses questions. Supposons que nous voulions connaître la probabilité que le pistolet soit l'arme du crime. Vous pouvez calculer cela de la distribution commune en additionnant la probabilité que le cuisinier ait utilisé le pistolet et la probabilité que le majordome ait utilisé le pistolet. Vous pouvez faire le même calcul pour le couteau et le tisonnier. La distribution qui reste après la sommation de toutes sauf une variable dans la distribution conjointe est la distribution marginale de la variable restante ou plus communément juste marginal.

La figure suivante montre le marginal pour le *lArmeDuCrime*. Avant de recevoir le rapport du médecin légiste, l'arme de meurtre la plus probable semble être le couteau ou le tisonnier.

	Pistolet	**Couteau**	**Poker**
cuisinier	4%	52%	24%
Majordome	16%	2%	2%
	= 20%	**= 54%**	**= 26%**

Vous pouvez calculer le marginal pour *leCoupable* de la même manière, ce qui vous donne simplement les probabilités a priori qui ont été spécifiées plus tôt.

	Pistolet	**Couteau**	**Poker**	
cuisinier	4%	52%	24%	**= 80%**
Majordome	16%	2%	2%	**= 20%**

Jusqu'à présent, ces marginaux vous disent essentiellement ce que vous savez déjà. Ils sont plus intéressants après avoir fait une observation et ajouté de nouvelles informations au mixe.

f) Cluedo : Inférer un postérieur

Lorsque le rapport du médecin légiste arrive, vous pouvez utiliser l'observation de l'arme du crime pour déduire un postérieur pour le coupable, qui devrait contenir une estimation améliorée du coupable probable.

Le postérieur est un marginal conditionnel, le marginal pour *leCoupable*, conditionné par l'observation que l'arme du crime est le pistolet. Dans ce cas simple, vous pouvez obtenir le postérieur depuis la table de distribution conjointe.

Le rapport du médecin légiste signifie que vous pouvez éliminer le couteau et le tisonnier de la grille. Les valeurs restantes pour le pistolet indiquent la probabilité que chaque suspect soit coupable. Cependant, ils ne sont pas une distribution valide, les nombres ne totalisent pas 100%.

Pour terminer le calcul, divisez chaque valeur par le marginal pour le pistolet (20%). Ceci renormalise les valeurs et produit la distribution postérieure.

Le résultat n'est pas tout à fait certain, mais ça sent mauvais pour le majordome !

En pratique, les modèles sont généralement beaucoup plus compliqués que celui utilisé dans cet exemple, ce qui rend le calcul des postérieurs plus difficile. Appliquer la programmation probabiliste aux scénarios du « monde réel » nécessite une approche plus sophistiquée à la fois la construction du modèle et pour l'inférence.

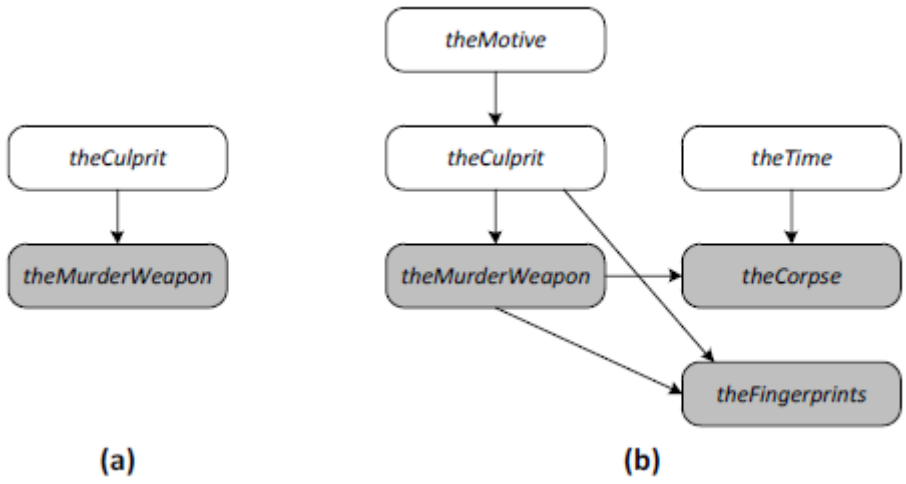
3. Une approche plus sophistiquée

La distribution conjointe dans la section précédente est un modèle très simple, avec seulement deux variables et un petit ensemble de valeurs possibles. Alors que l'observation de l'arme du crime suggère fortement que le majordome est le coupable probable, il y a encore une possibilité que le cuisinier soit coupable. Un modèle plus sophistiqué pouvant gérer un plus grand nombre d'observations pourrait apporter plus de certitude. Cependant, le calcul postérieur pour des modèles sophistiqués est d'autant plus difficile.

a) Des modèles plus sophistiqués

Vous pourriez peut-être gérer les variables supplémentaires en développant la distribution conjointe. Toutefois, pour plus de deux variables - ou pour les variables avec beaucoup de valeurs possibles - les tableaux deviennent rapidement ingérables. De plus, les tableaux sont utiles uniquement pour des distributions discrètes. Si vous voulez définir une distribution pour l'heure du meurtre, elle doit représenter une gamme continue de valeurs, qui ne peuvent pas être représentées par une table.

Une approche plus flexible et plus puissante consiste à créer un modèle conceptuel pour la distribution sous la forme d'un graphique qui représente les relations entre les variables aléatoires du système. La figure suivante montre deux exemples ou le deuxième gère des variables aléatoires supplémentaires.



Le modèle représente les relations entre les variables aléatoires, comme suit :

- Chaque case représente une variable aléatoire.
- Les flèches indiquent les relations de cause à effet entre des variables aléatoires.
- Les cases ombrées indiquent les variables aléatoires observables.
- Les cases non grisées indiquent les variables aléatoires non observables que nous aimerions déduire.

Pour le modèle (a), le modèle est le suivant : le coupable choisit une arme. Vous observez l'arme du crime, et utilisez le modèle et l'observation pour déduire le coupable probable. Bien que le modèle représente un lien de cause à effet - appelé modèle génératif - vous pouvez l'utiliser pour raisonner dans les deux sens. Si vous connaissez le coupable, par exemple, vous pouvez utiliser cette observation pour déduire l'arme de meurtre la plus probable.

Le modèle (b) est un modèle plus sophistiqué intégrant des variables aléatoires supplémentaires - le moment du meurtre, l'état du cadavre, etc. – dont plusieurs sont observables. Vous pouvez utiliser ce modèle combiné à des observations de plusieurs variables aléatoires pour calculer postérieurs, et peut-être soit disculper le majordome soit produire des preuves si accablantes qu'il est obligé d'avouer.

b) Une Inférence plus sophistiquée

Avec Cluedo, vous pouvez utiliser une arithmétique simple pour déduire un postérieur de la table initiale. Cette approche devient plus difficile à mesure que vous ajoutez des variables, surtout si elles ont un grand nombre de valeurs possibles - et ne fonctionne pas du tout pour les variables qui représentent une plage continue de valeurs possibles. Des modèles plus réalistes nécessitent une manière plus sophistiquée d'inférer les postérieurs.

Les modèles conceptuels de la sections précédente 6 sont en réalité une représentation graphique des expressions mathématique de la distribution conjointe. Les programmes probabilistes peuvent utiliser cette expression et les mathématiques de l'inférence bayésienne pour inférer des postérieurs pour des graphiques arbitrairement complexes, y compris des graphiques comportant des variables à distributions continues. Vous pouvez même utiliser la représentation mathématique pour définir directement des modèles qui ne peuvent pas être représentés graphiquement.

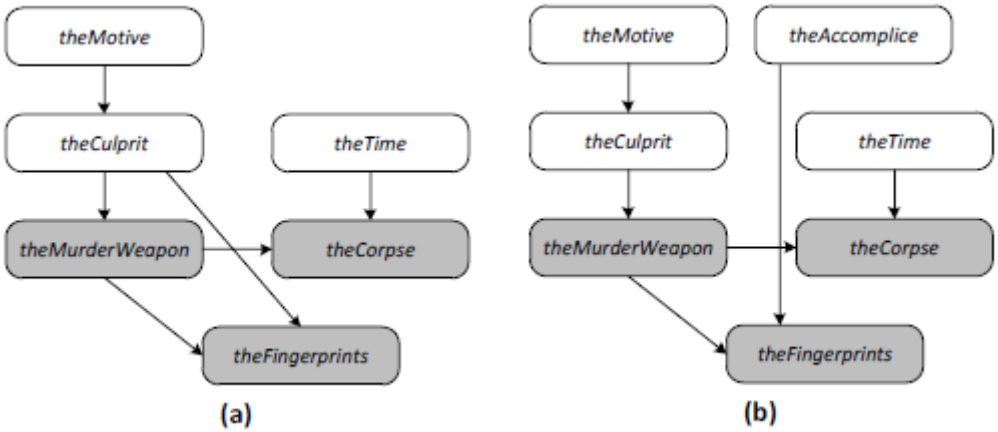
c) Comment choisir le meilleur modèle

Le modèle (a) fournit une estimation du coupable probable, mais probablement pas assez bonne pour le condamner. Un modèle plus complexe et sophistiqué pourrait être plus convaincant. Cependant, ajouter plus de variables ne produit pas nécessairement un meilleur modèle. Il y a généralement un point ou les rendements décroissent, au-delà duquel la complexité n'apporte rien au modèle ou le rend moins bon.

Par exemple, lorsque vous adaptez un polynôme à un ensemble de points de données, vous pouvez toujours obtenir un ajustement exact en ajoutant suffisamment d'éléments au polynôme. Cependant, un polynôme qui s'adapte exactement à chaque point de données, oscille généralement entre chaque point – un phénomène connu sous le nom de surajustement ou surapprentissage - qui pourrait être précis dans un certain sens, mais n'est pas très utile. Un polynôme avec moins d'éléments peut souvent contenir les données presque aussi bien, et fournir un modèle beaucoup plus utile et réaliste.

Ce que vous voulez, c'est un juste milieu : un modèle qui correspond assez bien aux données sans être trop complexe. En bref, vous devez appliquer le rasoir Occam aux modèles possibles, le meilleur modèle est le plus simple, qui adapte correctement les données.

Par exemple, considérons les deux modèles de la figure suivante.



Les deux modèles rendent compte de la présence d'empreintes digitales de différentes manières :

- Avec le modèle (a), le coupable laisse des empreintes digitales sur le lieu du crime.
- Avec le modèle (b), le coupable a pris soin de ne pas laisser d'empreintes digitales, mais avait un complice qui en a laissé.

Vous ne pouvez généralement pas savoir avec certitude quel modèle est le modèle optimal. Cependant, avec la programmation probabiliste, vous pouvez traiter la preuve comme une variable aléatoire et en déduire la distribution de cette variable. La distribution vous donne la probabilité que chaque modèle soit optimal, et vous pouvez utiliser cette information pour choisir le meilleur modèle. Par exemple, si la preuve pour le modèle (b) a une probabilité de seulement 15%, un complice ajoute probablement une complexité inutile au modèle. Vous pouvez donc rester avec l'explication la plus simple : le majordome est coupable.

B. Présentation d'Infer.Net

1. Qu'est-ce que Infer.Net

Infer.NET est un framework permettant d'exécuter l'inférence bayésienne dans des modèles graphiques. Infer.NET fournit les algorithmes de passe-messages et les routines statistiques nécessaires à la réalisation d'inférences pour une grande variété d'applications. Infer.NET diffère du logiciel d'inférence existant de plusieurs manières :

a) Langage de modélisation

Prise en charge des variables univariées et multivariées, continues et discrètes. Les modèles peuvent être construits à partir d'un éventail de facteurs, comme les opérations arithmétiques, l'algèbre linéaire, les contraintes de distance et de positivité, les opérateurs booléens, Dirichlet-Discrete, Gaussian et d'autres. Prise en charge de mixtures hiérarchiques avec des composants hétérogènes.

b) Plusieurs algorithmes d'inférence

Les algorithmes intégrés incluent la propagation d'espérance (EP), la propagation des convictions (un cas particulier de EP), le passage de messages variationnels et l'échantillonnage de Gibbs.

c) Conçu pour l'inférence à grande échelle

Dans la plupart des programmes d'inférence existants, l'inférence est effectuée à l'intérieur du programme et la surcharge d'exécution du programme ralentit l'inférence. Au lieu de cela, Infer.NET compile les modèles dans du code source dédié qui peut être exécuté indépendamment sans surcharge. Il peut également être intégré directement dans l'application et le code source peut être visualisé, inséré, profilé ou modifié selon les besoins, à l'aide d'outils de développement standard.

d) Extensible par l'utilisateur

Des distributions de probabilité, des facteurs, des opérations de message et des algorithmes d'inférence peuvent tous être ajoutés par l'utilisateur. Infer.NET utilise une architecture de plug-in qui le rend ouvert et adaptable.

2. Comment fonctionne Infer.NET

Infer.NET fonctionne en compilant une définition de modèle dans le code source nécessaire pour calculer un ensemble de requêtes d'inférence sur le modèle. Le diagramme ci-dessous résume le processus d'inférence.

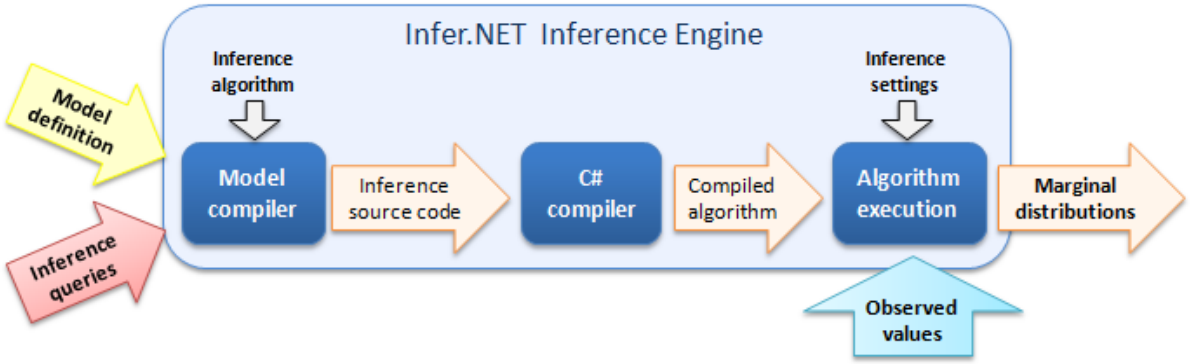


Diagramme de bloc Infer.NET

Les étapes sont :

1. L'utilisateur crée une définition de modèle (à l'aide de l'API de modélisation) et spécifie un ensemble de requêtes d'inférence relatives au modèle.

2. L'utilisateur transmet la définition du modèle et les requêtes d'inférence au compilateur de modèle, qui crée le code source nécessaire à l'exécution de ces requêtes sur le modèle, à l'aide de l'algorithme d'inférence spécifié. Ce code source peut être écrit dans un fichier et utilisé directement si nécessaire.

3. Le code source est compilé pour créer un algorithme compilé. Cela peut être effectué manuellement pour obtenir un contrôle précis de la manière dont l'inférence est effectuée, ou automatiquement exécuté via la méthode Infer.

4. À l'aide d'un ensemble de valeurs observées (telles que des tableaux de données), le moteur d'inférence exécute l'algorithme compilé, en fonction des paramètres spécifiés par l'utilisateur, afin de produire les distributions marginales demandées dans les requêtes. Ceci peut être répété pour différents jeux de valeurs observées sans recompiler l'algorithme.

3. Un exemple simple

Voici un exemple d'utilisation d'Infer.NET pour déterminer la probabilité d'obtenir deux faces lors du lancement de deux pièces non biaisées.

```
Variable<bool> premierePiece = Variable.Bernoulli(0.5);
Variable<bool> deuxiemePiece = Variable.Bernoulli(0.5);
Variable<bool> deuxFaces = (premierePiece & deuxiemePiece);
InferenceEngine engine = new InferenceEngine();
Console.WriteLine("Probabilité d'avoir deux faces: " + engine.Infer(deuxFaces));
```

Le résultat de ce programme est:

```
Probabilité d'avoir deux faces: Bernoulli(0.25)
```

qui donne correctement la probabilité de deux faces à 0,25 ou 1/4.

Ce court exemple contient les trois éléments clés de tout programme Infer.NET :

a) Définition d'un modèle probabiliste

Tous les programmes Infer.NET ont besoin d'un modèle probabiliste pour être définis. Cela se fait dans les trois premières lignes ci-dessus en définissant les variables aléatoires premierePiece et deuxiemPiece et en spécifiant la variable dépendante deuxFaces en fonction de celles-ci.

b) Création d'un moteur d'inférence

Toute inférence est obtenue grâce à l'utilisation d'un moteur d'inférence. Ceci doit être créé et configuré avant toute inférence. La quatrième ligne ci-dessus crée un moteur d'inférence qui utilise l'algorithme d'inférence par défaut (propagation d'espérance).

c) Exécution d'une requête d'inférence

Avec un moteur d'inférence, vous pouvez interroger des distributions marginales sur des variables en utilisant Infer () . Dans la dernière ligne de l'exemple, le moteur est utilisé pour déduire la distribution marginale des deux faces, c'est-à-dire la probabilité que les deux pièces retournent des faces. Le moteur renvoie une distribution Bernouilli qui est ensuite imprimée sur la console.

4. Utilisation d'Infer.Net en Python

Il est possible d'utiliser l'ensemble des fonctionnalités fournies par la bibliothèque Infer.net depuis le Python, par le biais de la bibliothèque Pythonnet, qui constitue le bridge entre la technologie .Net et Python.

Le dépôt suivant illustre l'utilisation d'Infer dans des notebooks Python Jupiter, ainsi que la possibilité de l'intégrer avec d'autres bibliothèques de programmation probabiliste complémentaires en Python, comme PyMC :

https://github.com/Tobimaru/InferNet-pythonnet

III. Mise en œuvre d'Infer.Net dans un exemple détaillé

A. Introduction

1. Contexte

a) Scenario

Les programmes présentés ici sont basés sur le scénario suivant :

- Plusieurs de vos collègues et vous-même rendez-vous à vélo chaque jour au travail.
- Le temps de trajet d'un cycliste varie de jour en jour et sa valeur est donc aléatoire.
- L'incertitude du temps de trajet est représentée par une distribution de probabilité qui définit le temps de trajet moyen et combien il varie.
- L'application apprendra cette distribution à partir de plusieurs temps de trajet observés et utilisera ces connaissances pour faire des prévisions sur les temps de déplacement futurs.

b) Programmes

Les programmes suivants de complexité croissante sont réalisés :

- On apprend la répartition du temps de parcours d'un cycliste unique et on utilise cette information pour prédire les temps de déplacement futurs.
- On conçoit une version restructurée du premier exemple, qui introduit une pratique standard pour la mise en œuvre des modèles qui est utilisé par la suite.
- On permet la possibilité d'un événement inattendu et modélise le temps de trajet comme un mélange de deux distributions.
- On utilise des preuves pour sélectionner le meilleur modèle.

2. Procédure

Cette section décrit brièvement la structure de base et les éléments clés des exemples qui suivent. Les termes et les concepts sont discutés plus en détail au fur et à mesure de l'avancement du document.

a) Installation du package NuGet

Pour les besoins du TP, nous créerons un projet en c# de type Console (.Net 4.5.6 ou .Net Core).

Il nous faudra installer les package NuGet suivants, qui contiennent la librairie Infer.Net :

- Microsoft.ML.Probabilistic
- Microsoft.ML.Probabilistic.Compiler

Dans .Net interactive, les commandes sont les suivantes :

```
// Pour installer Infer.NET
#r "nuget: Microsoft.ML.Probabilistic"
#r "nuget: Microsoft.ML.Probabilistic.Compiler"
```

b) Créer un modèle

Une application Infer.NET est construite autour d'un modèle probabiliste, qui définit les variables aléatoires et comment elles sont liées.

Une variable aléatoire est essentiellement une extension d'un type standard tel que double ou int, qui permet au type d'avoir des valeurs incertaines. Infer.NET représente les variables aléatoires comme instances de la classe Variable <T> , qui se trouve dans l'espace de noms Microsoft.ML.Probabilistic.Models . T est appelé le type de domaine de la variable :

- Les variables aléatoires discrètes ont un ensemble spécifié de valeurs possibles et un type de domaine bool ou int
- Les variables aléatoires continues ont une plage de valeurs possibles et un type de domaine double.

Une variable aléatoire est définie par une distribution de probabilité - généralement abrégée en distribution - qui attribue des probabilités aux valeurs possibles de la variable. Au départ, une variable aléatoire est définie par une distribution a priori, qui représente votre compréhension de la valeur de la variable avant observation.

Remarque : La création d'une variable aléatoire peut impliquer jusqu'à trois étapes distinctes.

Exemple :

1. Déclaration en C# : Variable<bool> x;
2. Définition de C# : x = Variable<bool>.New ();
3. Définition statistique : x = Variable.Random<bool>(someDist);

Dans le langage de l'inférence statistique, "définition" ou "définir" se réfère à la distribution de la variable. Cependant, dans les cas où les étapes 2 et 3 sont des déclarations séparées, nous clarifions la distinction en utilisant « définition statistique » ou « définir statistiquement » pour l'étape 3.

Il existe différentes façons de définir les relations entre les variables aléatoires, qui sont discutés plus tard dans ce document.

c) Observer des variables aléatoires

Vous pouvez effectuer des calculs sur le modèle à ce stade, mais ils ne sont généralement pas très intéressants. Les résultats reflètent simplement les valeurs que vous avez choisies pour les prieurs. Pour apprendre quelque chose de nouveau, vous observez une ou plusieurs variables aléatoires du modèle en attribuer des valeurs à leurs propriétés *ObservedValue*. À ce stade, les variables ne sont plus aléatoires ; ce sont des types standard à valeur fixe.

d) Inférer les postérieurs

Avant de faire des observations, les prieurs du modèle définissent directement ou indirectement une distribution de variable aléatoire particulière. Après avoir fait quelques observations sur une ou plusieurs variables, vous en savez plus sur la valeur de cette variable de sorte qu'elle a une nouvelle distribution appelée la distribution postérieure- couramment abrégé à postérieur. Un postérieur intègre les informations du prieur et des observations, et représente votre connaissance nouvelle et probablement améliorée de la valeur de la variable.

Vous calculez les postérieurs à l'aide d'une instance du moteur d'inférence Infer.NET, qui effectue tout le lourd travail numérique. Le moteur d'inférence est implémenté en tant que Classe **InferenceEngine** dans l'espace de noms **Microsoft.ML.Probabilistic.Models** .

Le moteur d'inférence calcule la distribution postérieure pour une variable aléatoire spécifiée en « résumant » l'effet des autres variables aléatoires du modèle. En général, ce type de distribution s'appelle la distribution marginale de la variable, qui est généralement abrégé en marginal. Lorsque vous interrogez le moteur d'inférence pour le marginal d'une variable aléatoire après avoir observé une ou plusieurs autres variables aléatoires du modèle, le marginal que le moteur retourne est un postérieur - une mise à jour du prieur, conditionnée par les nouvelles informations issues des observations.

e) Utiliser les postérieurs

Vous pouvez utiliser les postérieurs à diverses fins. Une utilisation évidente est de prédire le comportement futur de la variable. Cependant, avec seulement quelques observations, le postérieur pourrait ne pas refléter avec précision le comportement réel de la variable et les prédictions pourraient ne pas être très précises. Vous pouvez améliorer votre compréhension en formulant des observations supplémentaires et incorporer ces informations dans la distribution de la variable, comme suit :

1. Utilisez le postérieur comme nouveau prieur de la variable.
2. Faites quelques observations supplémentaires.
3. Calcule un nouveau postérieur.

Le nouveau postérieur intègre le prieur initial et toutes les observations. Vous pouvez continuer ce processus indéfiniment. Après un nombre suffisant d'observations, le postérieur devrait mieux refléter la valeur réelle de la variable.

f) Et après

Une distribution est contrôlée par un ou plusieurs paramètres. Parfois, vous pouvez faire des estimations a priori raisonnables des valeurs des paramètres, mais elles correspondent rarement exactement aux observations ultérieures. Parfois, vous avez peu ou pas de connaissances préalables. En d'autres termes, les valeurs des paramètres sont incertaines.

Dans une perspective bayésienne, tout, y compris les paramètres de distribution, est incertain et peut être traité comme une variable aléatoire. Vous affectez un prieur à la variable qui reflète votre compréhension initiale - ou votre manque de compréhension – de la valeur du paramètre et utilisez les observations et la programmation probabiliste pour apprendre les distributions de la valeur. Cette approche générale est une forme d'apprentissage de paramètres, utilisé par toutes les procédures pas à pas de cet exemple.

B. Apprentissage de paramètres simple : DureeCycliste1

La section présente les bases de l'utilisation de Infer.NET pour implémenter l'apprentissage d'un paramètre. Il s'agit d'une application simple qui apprend la durée du trajet d'un cycliste individuel basée sur trois jours de temps de parcours observés. Ce processus est parfois appelé « entraînement du modèle ». DureeCycliste1 utilise ensuite le model entraîné pour prédire le temps de trajet de demain.

1. Modèle

Le temps de trajet d'un cycliste varie d'un jour à l'autre, les temps individuels étant répartis autour d'une valeur moyenne. Vous pouvez représenter statistiquement ces temps de trajet sous forme d'échantillons aléatoires d'une distribution qui correspond aux données. DureeCycliste1 utilise une Distribution gaussienne pour représenter la distribution des temps de parcours et caractérise la distribution par les deux paramètres suivants.

• La moyenne de la distribution est la durée moyenne du trajet du cycliste.

• La précision de la distribution détermine la différence de temps de parcours d'un jour à l'autre et reflète des facteurs tels que les variations quotidiennes du trafic.
Il est pratique de penser à la précision en tant que mesure du « bruit » du trafic avec une plus grande précision correspondant à moins de variation au jour le jour.

Remarque : Vous connaissez peut-être mieux l'écart type, σ, en tant que mesure de la largeur d'une distribution gaussienne. Il est souvent plus pratique sur le plan mathématique d'utiliser une des valeurs liées suivantes :

• Variance : σ 2

• Précision : 1 / σ 2

Parce que la moyenne et la précision sont à la fois incertaines et continues, DureeCycliste1 les représente par des variables aléatoires de type double.

• *dureeMoyenne* représente la moyenne, et son incertitude est représentée par un autre Distribution gaussienne.

• *bruitTrafic* représente la précision et son incertitude est représentée par une distribution gamma.

Les distributions gaussienne et gamma et les raisons de leur utilisation sont discutées plus tard.

DureeCycliste1 représente les temps de trajet observés en utilisant trois variables aléatoires :

- *dureeLundi*,
- *dureeMardi*
- *dureeMercredi*

Ces trois variables sont supposées être tirées de la même distribution gaussienne de temps de déplacement.

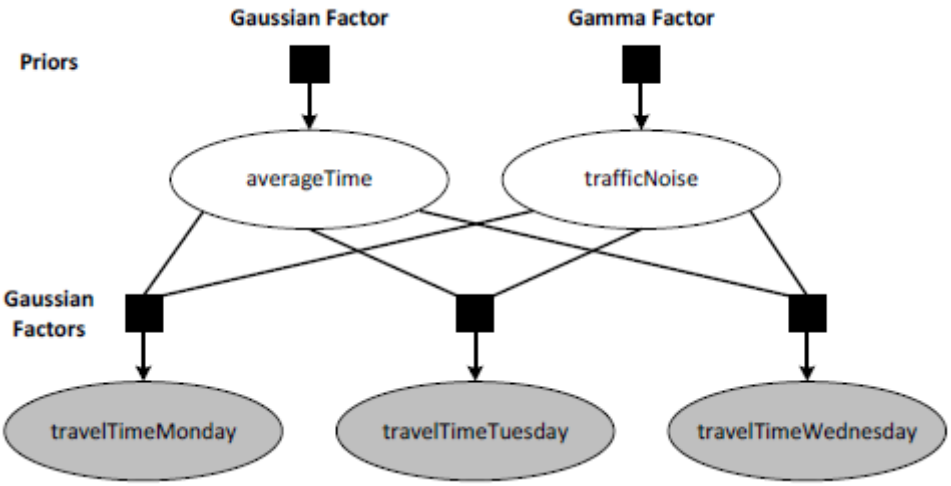
La dernière pièce du puzzle consiste à définir la relation entre *dureeMoyenne*, *bruitTrafic* et les trois variables aléatoires *dureeX*. La Programmation probabiliste utilise généralement un modèle génératif pour définir les relations entre les variables aléatoires. Un modèle génératif décrit comment les données observées sont générées à partir des paramètres de modèle sous-jacents. Bien qu'il soit parfois utile de penser à un modèle génératif comme un ensemble de relations de cause à effet, les modèles génératifs ne sont pas nécessairement causaux.

Pour DureeCycliste1, le processus génératif est le suivant :

1. Echantillonnez les variables aléatoires *dureeMoyenne* et *bruitTrafic* pour générer les valeurs de moyenne et de précision pour ce cycliste.
2. Créez une distribution gaussienne sur les temps de trajet en utilisant la moyenne et la précision de l'étape 1.
3. Echantillonnez la distribution gaussienne de l'étape 2 pour générer la valeur du temps de trajet *dureeLundi*.
4. Répétez l'étape 3 pour *dureeMardi* et *dureeMercredi*.

La figure 1 montre une représentation graphique du modèle DureeCycliste1, appelé graphe de facteurs :

- Les ellipses indiquent des variables aléatoires.
- Les ellipses ombrées indiquent les variables observées.
- Les carrés remplis indiquent les facteurs qui représentent les distributions associées aux variables.
- Les flèches indiquent la direction du processus de génération.



Bien que le processus génératif ait une direction bien définie, l'inférence peut fonctionner dans n'importe quelle direction. Vous observez une ou plusieurs variables aléatoires et le moteur d'inférence utilise les observations et le modèle pour déduire les distributions du reste des variables aléatoires. Dans la figure 1, les données d'entraînement représentées par les trois variables *dureeXYZ* sont observées et le moteur d'inférence utilise les observations et le modèle pour déduire *dureeMoyenne* et *bruitTrafic*. C'est l'opposé de la direction du modèle génératif, donc dans ce cas l'inférence propage l'information en arrière depuis les observations.

Il existe en fait deux versions du modèle DureeCycliste1, qui correspondent à savoir si nous voulons apprendre les paramètres ou prédire l'heure de demain. La figure précédente montre le modèle pour l'entraînement, c'est-à-dire apprendre les paramètres. Le modèle de prédiction a la même structure que le modèle d'entraînement, mais utilise la valeur calculée de *dureeMoyenne* et *bruitTrafic* pour déduire *dureeDemain*. Dans ce cas, le moteur d'inférence propage les informations depuis les paramètres.

Nous allons voir comment ces deux tâches peuvent être implémentées avec un seul modèle, la seule différence étant quelles variables sont inférées. Cependant, pour rendre le code plus explicite quant à ces deux tâches, nous utilisons une variable de prédiction explicite '*dureeDemain*' dans le modèle de prédiction.

2. Application

L'exemple suivant montre la structure du programme DureeCycliste1.

Pour référence ultérieure. Les instructions *using* contiennent les espaces de noms utilisés par la plupart des applications Infer.NET.

```
using System;
using Microsoft.ML.Probabilistic.Distributions;
using Microsoft.ML.Probabilistic.Models;

namespace TP_MSMIN5IN
{
	partial class Program
	{
		static void Main(string[] args)
		{
			DureeCycliste1();
			Console.ReadKey();
		}

		static void DureeCycliste1()
		{

			…
		}
	}
}
```

a) Creation du modèle

La partie initiale de DureeCycliste1 définit le modèle d'entraînement.

(1) Créer dureeMoyenne et bruitTrafic , et spécifier les priorités initiales

La première étape de la phase d'entraînement consiste à créer des variables aléatoires pour représenter *dureeMoyenne* et *bruitTrafic*. Les deux variables ont une plage continue de valeurs possibles, donc elles sont représentées par des variables aléatoires de type double . Avant de lancer l'inférence, vous devez spécifier les prieurs des variables.

Un prieur représente votre meilleure compréhension d'une variable aléatoire avant d'observer une ou plusieurs autres variables aléatoires du modèle. Après avoir fait les observations, vous pouvez utiliser le prieur, le modèle et les observations pour calculer le postérieur de la variable, qui constitue votre compréhension améliorée du comportement de la variable.

Vous pouvez ensuite utiliser le postérieur pour prédire le comportement futur de la variable, ou en tant que prieur pour la prochaine série d'observations, et ainsi de suite.

Un problème évident est de savoir quoi utiliser pour les valeurs de paramètre de prieur initiale. Dans de nombreux cas, vous pouvez supposer que les paramètres de distribution sont basés sur une compréhension du comportement de la variable. Par exemple, un cycliste expérimenté devrait être capable d'estimer combien de temps un voyage particulier devrait prendre et combien les durées varient d'un jour à l'autre. Une telle estimation n'est probablement pas exacte, mais elle sera probablement assez proche et vous spécifiez une valeur de précision qui représente correctement l'incertitude.

Le prieur initial est une hypothèse qui peut paraître un peu floue, mais gardez à l'esprit que toutes les formes d'analyse statistique font des hypothèses à un certain niveau. Avec l'inférence bayésienne, les hypothèses sont explicites, et elles sont traitées de manière raisonnée.

Pour spécifier un prieur initial, définissez la variable aléatoire associée à l'aide de la distribution appropriée avec un ensemble raisonnable de valeurs de paramètres. Le plus simple consiste à utiliser une ou plusieurs des méthodes de création statiques de la classe **Variable**. Si la variable est scalaire, par opposition à un tableau de variables, vous pouvez généralement créer la variable aléatoire et fournir la définition statistique avec un seul appel de méthode.

L'exemple suivant montre comment DureeCycliste1 utilise les méthodes des création statiques pour créer dureeMoyenne et bruitTrafic, et leur leur donner un prieur initial.

```
Variable<double> dureeMoyenne = Variable.GaussianFromMeanAndPrecision(15, 0.01);
Variable<double> bruitTrafic = Variable.GammaFromShapeAndScale(2, 0.5);
```

DureeCycliste1 utilise les valeurs de paramètre suivantes pour la priorité initiale *dureeMoyenne*:

- La moyenne est fixée à 15, selon l'expérience générale du cycliste.
  Un cycliste a généralement une assez bonne idée de la durée d'un parcours. La moyenne spécifiée reflète cette compréhension informée quelque peu incertaine de la variable.
- La précision est définie sur 0.01.
  La moyenne est une supposition éclairée, et la valeur de précision représente à quel point le cycliste croit cette supposition précise. Ici, la petite précision indique une grande incertitude initiale.

Les paramètres *bruitTrafic* sont définis sur des valeurs similaires. Les raisons de choisir ces distributions particulières et la signification du paramètre *bruitTrafic* sont discutées plus tard dans ce document.

En pratique, le choix d'un prieur initial n'est significatif que si vous avez un petit nombre d'observations. Au fur et à mesure que le nombre d'observations augmente, l'influence du prieur initial sur le postérieur diminue. Avec suffisamment d'observations, souvent une nombre relativement faible - la distribution postérieure est effectivement déterminée par les observations. En fait, les applications utilisent souvent simplement un prieur « neutre », tel qu'attribuer des probabilités égales à toutes les valeurs possibles et laisser les observations déterminer la distribution correcte.

(2) Créer et définir les temps de trajet

Chaque valeur observée est représentée par une variable aléatoire. Chaque temps de voyage doit donc être représenté par une variable aléatoire de type **double** – un objet **Variable<double>** qui est régi par une distribution continue. Pour ce scénario, une distribution gaussienne est le choix évident.

La dernière étape de la création du modèle d'entraînement consiste à définir les trois variables à observer :

```
Variable<double> dureeLundi = Variable.GaussianFromMeanAndPrecision(dureeMoyenne,
bruitTrafic);
Variable<double> dureeMardi = Variable.GaussianFromMeanAndPrecision(dureeMoyenne,
bruitTrafic);
Variable<double> dureeMercredi = Variable.GaussianFromMeanAndPrecision(dureeMoyenne,
bruitTrafic);
```

Le fait de définir ces variables en utilisant *dureeMoyenne* et *bruitTrafic* fournit le lien apparent dans le graphe de facteurs

b) Entrainement du modèle

Pour compléter le processus d'entraînement, DureeCycliste1 observe les données d'entraînement et en déduit les postérieurs pour *dureeMoyenne* et *bruitTrafic* . Les postérieurs résument essentiellement les résultats du processus d'entraînement sous une forme utilisable pour des calculs ultérieurs, tels que la prédiction des temps de déplacement futurs, ou en tant que prieur pour un nouvel entraînement.

Vous observez les données en affectant une valeur à la propriété **ObservedValue** de la variable aléatoire. DureeCycliste1 observe les données d'apprentissage de la manière suivante :

```
dureeLundi.ObservedValue = 13;
dureeMardi.ObservedValue = 17;
dureeMercredi.ObservedValue = 16;
```

À ce stade, les valeurs des trois variables de temps de trajet sont fixes et les variables ne sont plus aléatoires.

Tous les détails sont maintenant en place, mais jusqu'à présent, Infer.NET a simplement utilisé le code de modélisation pour construire une représentation interne du modèle. Aucun calcul n'intervient jusqu'à ce que vous interrogiez le moteur d'inférence sur le marginal d'une variable aléatoire.

Pour calculer les postérieurs, DureeCycliste1 crée une instance du moteur d'inférence et l'interroge pour les marginaux de dureeMoyenne et bruitTrafic . Pour interroger sur le marginal d'une variable aléatoire, vous passez la variable à la méthode **InferenceEngine.Infer<T>**, où T est le type de distribution de la variable.

Lorsque vous appelez Infer<T>, le moteur d'inférence:

1. Utilise un compilateur de modèle pour générer du code C #, basé sur le modèle et l'algorithme d'inférence, pour calculer le marginal demandé.
2. Utilise le compilateur .NET C # pour compiler le code généré en un exécutable.
3. Lance l'exécutable pour calculer le marginal demandé.
4. Renvoie le marginal à l'application en tant qu'objet de distribution.

Si vous avez fait des observations avant d'exécuter la requête, le moteur utilise ces observations, le prieur, et le modèle pour déduire un marginal pour la variable aléatoire spécifiée qui est compatible avec les données d'entraînement et le prieur. Les marginaux retournés sont donc postérieurs, par définition.

DureeCycliste1 interroge les marginaux comme suit :

```
InferenceEngine engine = new InferenceEngine();
Gaussian moyennePosterieure = engine.Infer<Gaussian>(dureeMoyenne);
Gamma bruitPosterieur = engine.Infer<Gamma>(bruitTrafic);
```

Console.WriteLine($"Moyenne a posteriori: {moyennePosterieure.ToString()}");
Console.WriteLine($"Bruit traffic a posteriori: {bruitPosterieur.ToString()}");

Dans ce cas, le moteur d'inférence utilise l'algorithme d'inférence par défaut, qui est la propagation d'espérance. Vous pouvez également spécifier d'autres algorithmes, comme indiqué ultérieurement.

Le moteur d'inférence renvoie les postérieurs inférés, *moyennePosterieure* et *bruitPosterieur* et DureeCycliste1 impriment les résultats comme suit :

moyenne posterieure Gaussian(15,33, 1,32)

bruit posterieur Gamma(2,242, 0,2445)[mean=0,5482]

Si vous ne considérez que la moyenne de ces deux distributions postérieures, la moyenne de la durée moyenne du trajet est de 15,33 et la moyenne du bruit de la circulation est de 0,5482. Toutefois la durée moyenne du trajet et le bruit du trafic sont tous deux incertains, comme le montrent les distributions complètes. L'incertitude de ces estimations diminuera avec plus d'observations.

c) Utilisation du modèle pour faire des prédictions

La première étape pour prédire le temps de parcours à partir du modèle formé consiste à définir un modèle de prédiction. DureeCycliste1 prédit un seul temps de trajet, qui peut être représenté par une seule variable aléatoire ordinaire, *dureeDemain*, comme suit :

```
Variable<double> dureeDemain = Variable.GaussianFromMeanAndPrecision(dureeMoyenne,
bruitTrafic);
```

Pour connecter dureeDemain au modèle entraîné, on le définit statistiquement en utilisant une distribution gaussienne avec :

- Le paramètre moyen de la distribution défini sur *dureeMoyenne*.
- Le paramètre de précision de la distribution défini sur *bruitTrafic*.

Le modèle de prédiction est en fait le modèle d'entraînement avec une variable aléatoire supplémentaire.

Prédire le temps de trajet de demain est simple: interrogez le moteur d'inférence pour le marginal de *dureeDemain*.

```
Gaussian distribDemain = engine.Infer<Gaussian>(dureeDemain);
Console.WriteLine($"Prédiction demain {distribDemain.ToString()}, écart type:
{Math.Sqrt(distribDemain.GetVariance())}");

// 4 - Faire d'autres prédictions basées sur les premières
double probMoinsDe18Mn = engine.Infer<Bernoulli>(dureeDemain < 18.0).GetProbTrue();
Console.WriteLine("Probabilité que le trajet prenne moins de 18mn: {0:f2}",
probMoinsDe18Mn);
```

Lorsque vous interrogez le marginal de demain, le moteur d'inférence :

1. Compile le modèle de prédiction.
   L'ajout de dureeDemain modifie le modèle d'entraînement original. Le modèle doit donc être compilé avant d'exécuter les calculs.
2. Utilise les valeurs observées pour *dureeLundi* , *dureeMardi* et *dureeMercredi* pour entraîner le modèle.
   dureeMoyenne et *bruitTrafic* sont maintenant définis par leurs postérieurs.
3. Utilise le modèle formé pour calculer le postérieur de *dureeDemain*.

Cette procédure effectue l'ensemble du calcul depuis le début, y compris les calculs effectués précédemment pour le modèle d'entraînement. Le chapitre suivant introduit un moyen plus efficace de gérer ce processus.

Le code restant montre comment utiliser les méthodes de l'objet gaussien et quelques méthodes annexes pour obtenir divers types d'informations, comme suit :

• Gaussian.GetMean renvoie la moyenne de la distribution.

• Gaussian.GetVariance renvoie la variance de la distribution.

DureeCycliste1 convertit la variance en écart type équivalent, qui est un peu plus facile à visualiser.

Les résultats sont les suivants :

Prédiction demain Gaussian(15,33, 4,613), écart type: 2,14776700894404

La prédiction de *dureeDemain* est une distribution, pas seulement un nombre, vous pouvez donc l'utiliser pour poser des questions relativement sophistiquées. La dernière ligne interroge le modèle pour la probabilité que le voyage de demain prenne moins de 18 minutes.

Les résultats sont les suivants :

Probabilité que le trajet prenne moins de 18mn: 0,89

d) Récapitulatif

Le listing suivant reprend l'ensemble du premier algorithme :

```
using System;
using Microsoft.ML.Probabilistic.Distributions;
using Microsoft.ML.Probabilistic.Models;

namespace TP_MSMIN5IN
{
	partial class Program
	{
		static void Main(string[] args)
		{
			DureeCycliste1();
			Console.ReadKey();
		}

		static void DureeCycliste1()
		{

			//1 - Définition du  modèle
			Variable<double> dureeMoyenne = Variable.GaussianFromMeanAndPreci-
sion(2, 0.01);
			Variable<double> bruitTrafic = Variable.GammaFromShapeAndScale(2,
0.5);
			Variable<double> dureeLundi = Variable.GaussianFromMeanAndPreci-
sion(dureeMoyenne, bruitTrafic);
			Variable<double> dureeMardi = Variable.GaussianFromMeanAndPreci-
sion(dureeMoyenne, bruitTrafic);
			Variable<double> dureeMercredi = Variable.GaussianFromMeanAndPreci-
sion(dureeMoyenne, bruitTrafic);

			// 2 - Observations et entraînement du modèle
			dureeLundi.ObservedValue = 13;
			dureeMardi.ObservedValue = 17;
			dureeMercredi.ObservedValue = 16;
			InferenceEngine engine = new InferenceEngine();
			Gaussian moyennePosterieure = engine.Infer<Gaussian>(dureeMoyenne);
			Gamma bruitPosterieur = engine.Infer<Gamma>(bruitTrafic);
			Console.WriteLine($"moyenne posterieure {moyennePosterieure.ToS-
tring()}");
			Console.WriteLine($"bruit posterieur {bruitPosterieur.ToString()}");

			// 3 - Utilisation du modèle pour faire des prédictions
			Variable<double> dureeDemain = Variable.GaussianFromMeanAndPreci-
sion(dureeMoyenne, bruitTrafic);
			Gaussian distribDemain = engine.Infer<Gaussian>(dureeDemain);
			Console.WriteLine($"Prédiction demain {distribDemain.ToString()},
écart type: {Math.Sqrt(distribDemain.GetVariance())}");

			// 4 - Faire d'autres prédictions basées sur les premières
			double probMoinsDe18Mn = engine.Infer<Bernoulli>(dureeDemain <
18.0).GetProbTrue();
			Console.WriteLine("Probabilité que le trajet prenne moins de 18mn:
{0:f2}", probMoinsDe18Mn);
		}
	}
}
```

C. Restructuration de notre application

DureeCycliste1 a présenté les bases de l'apprentissage des paramètres. Bien que la structure simple de l'application soit utile pour introduire des concepts de base, elle ne peut pas être facilement étendu pour

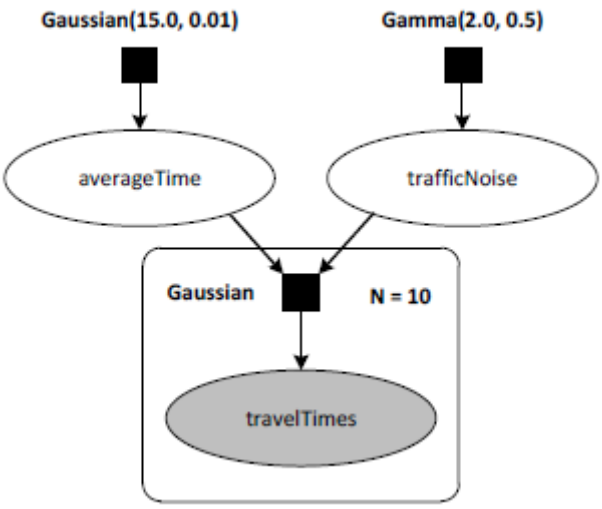
gérer des scénarios plus sophistiqués. En particulier, dans DureeCycliste1 tout mis en œuvre dans une seule méthode, ce qui n'est pas idéal pour des applications de niveau de production.

L'approche privilégiée (utilisée par DureeCycliste2) consiste à encapsuler le code de modélisation dans une classe séparée. Dans certains cas, il est utile de mettre en œuvre des classes distinctes pour les modèles d'entraînement et de prédiction. Dans d'autres, vous pouvez utiliser une seule classe, mais les modèles d'entraînement et de prédiction doivent être des instances de classe distinctes, de sorte que le moteur d'inférence n'a pas à recompiler le modèle à plusieurs reprises.

DureeCycliste2 étend DureeCycliste1 comme suit :

- Il encapsule les modèles d'entraînement et de prédiction dans des classes séparées. DureeCycliste2 utilise ensuite des instances de ces classes pour entraîner le modèle et mettre en œuvre l'apprentissage en ligne.
- Il introduit l'utilisation de tableaux de variables aléatoires pour les variables observées. Les tableaux de variables aléatoires sont beaucoup plus efficaces que d'utiliser un objet **Variable <T>** pour chaque valeur observée.
- Il introduit une manière différente de traiter les a priori, utile pour les modèles d'apprentissage plus sophistiqués. Un exemple simple est donné à la fin de ce chapitre.

La figure suivante montre le graphe de facteurs pour le modèle génératif sous-jacent.



Ce graphique est essentiellement similaire à la figure précédente. Toutefois, DureeCycliste1 a utilisé un Variable <T> pour chaque valeur observée, chacune ayant son propre noeud. Cette approche n'est pas pratique pour plus de quelques valeurs. DureeCycliste2 utilise un tableau de variables aléatoires, TempsDeTrajet, pour représenter les 10 valeurs observées.
Le rectangle autour du nœud TempsDeTrajet est appelé une plaque, et est fondamentalement un moyen compact de répliquer la structure du modèle dans un graphique.

Remarque: par souci de brièveté, cette section ne contient que des extraits édités des parties clés de DureeCycliste2.

1. Classe de base

DureeCycliste2 implémente les modèles d'entraînement et de prédiction en tant que classes séparées. Cependant, les modèles d'entraînement et de prédiction partagent généralement au moins une partie du code. Pour éviter d'implémenter le même code deux fois, les parties communes des modèles DureeCycliste2 sont implémentés dans une classe de base, CyclisteBase. Les deux classes modèles héritent alors de CyclisteBase, et implémentent le code spécifique à l'entraînement ou à la prédiction. La structure de classe est plus compliquée que strictement nécessaire pour DureeCycliste2, mais elle est utile pour les applications plus sophistiquées décrites plus loin.

a) Champs de CyclisteBase

CyclisteBase a la structure de base suivante.

```
public class CyclisteBase
    {

        public InferenceEngine MoteurInference;

        protected Variable<double> Moyenne;
        protected Variable<double> Bruit;

        protected Variable<Gaussian> MoyenneAPriori;
        protected Variable<Gamma> BruitAPriori;

        public virtual void CreationModeleBayesien()
        {
       …
        }

        public virtual void DefinirDistributions(DonneesCycliste distribsApriori)
        {
       …
        }

    }
```

Les champs servent les objectifs suivants :

- *Moyenne* et *Bruit* sont des variables aléatoires ayant le même objectif que dans DureeCycliste1.
- *MoyenneAPriori* et *BruitAPriori* sont des distributions qui représentent les prieurs pour les variables aléatoires.
- *InferenceEngine* représente une instance du moteur d'inférence.

MoyenneAPriori et *BruitAPriori* sont des distributions, mais elles sont typées comme objets Variable <T> sur leurs types de distribution respectifs plutôt que comme types de distribution. Les distributions ne sont pas des variables aléatoires. Cependant, le type variable<T> n'est pas utilisé uniquement pour les variables aléatoires ; il peut également être utilisé pour les valeurs que vous vouloir être en mesure de changer au moment de l'exécution sans nécessiter que le moteur d'inférence recompile le modèle. Utilisation de la variable <T> pour contenir les prieurs de dureeMoyenne et bruitTrafic est plus efficace du point de vue informatique et supporte une approche plus flexible pour la manipulation des prieurs.

La plupart des variables sont protégées car elles ne sont utilisées que par les deux classes dérivées. L'exception est *InferenceEngine*. Par défaut, chaque instance du modèle a sa propre instance du moteur. Cependant, les applications utilisant plusieurs instances du modèle, comme les applications qui gèrent plusieurs cyclistes, pourrait se retrouver avec un grand nombre d'instances de moteur, même si une seule instance est généralement suffisante.

Rendre InferenceEngine public permet à une application parente d'affecter la même instance de moteur d'inférence à chaque modèle et de l'utiliser pour tous les calculs.

CyclisteBase a deux méthodes d'instance. DureeCycliste2 appelle ces méthodes dans l'ordre suivant, après avoir créé une instance des classes d'entraînement ou de prédiction :

1. DureeCycliste2 appelle *CreationModeleBayesien* pour créer le modèle d'entraînement ou de prédiction.
2. DureeCycliste2 appelle DefinirDistributions pour spécifier les prieurs du modèle.

Les modèles d'entraînement et de prédiction implémentent des méthodes supplémentaires pour exécuter divers requêtes, basées sur le modèle et les valeurs observées.

b) Méthode CreationModeleBayesien

DureeCycliste2 appelle *CreationModeleBayesien* pour créer le modèle.
CyclisteBase.CreationModeleBayesien met en œuvre le code de modélisation commun à l'entraînement et à la prédiction comme suit:

```
public virtual void CreationModeleBayesien()
 {
     MoyenneAPriori = Variable.New<Gaussian>();
     BruitAPriori = Variable.New<Gamma>();
     Moyenne = Variable.Random<double, Gaussian>(MoyenneAPriori);
     Bruit = Variable.Random<double, Gamma>(BruitAPriori);
     if (MoteurInference == null)
     {
         MoteurInference = new InferenceEngine(new ExpectationPropagation());
     }
 }
```

Les implémentations des classes EntrainementCycliste et PredictionCycliste surchargent *CreationModeleBayesien* et ajoutent du code en fonction de leur modèle spécifique.

DureeCycliste1 a adopté l'approche la plus simple pour gérer les prieurs: utiliser une méthode de création statique créer et définir la variable aléatoire. Par exemple, DureeCycliste1 attribue un prieur à dureeMoyenne comme suit:

```
Variable<double> dureeMoyenne = Variable.GaussianFromMeanAndPrecision(15, 0.01);
```

DureeCycliste2 utilise une approche plus sophistiquée et flexible pour traiter les prieurs. Les prieurs sont représentés par les objets Variable <T> , qui sont essentiellement des conteneurs pour les distributions.

Par exemple, le prieur de *DureeMoyenne* est représenté par une variable aléatoire *MoyenneAPriori* de type **Variable<Gaussian>**.

CreationModeleBayesien crée MoyenneAPriori à l'aide de la méthode statique **Variable.New<T>**, qui est une méthode de création statique qui crée un nouvel objet Variable <T> avec un type de domaine spécifié mais pas de définition statistique. Cette méthode est utile si vous voulez créer un objet variable et le définir statistiquement plus tard, ou simplement assigner une valeur à la propriété ObservedValue de la variable.

CreationModeleBayesien définit ensuite *Moyenne* de manière statistique en appelant la Méthode statique **Variable.Random**. **Random** est un facteur unaire qui définit une variable aléatoire en utilisant un objet de distribution spécifié ou un objet Variable <T> qui représente un objet de distribution. DureeCycliste2 représente le prieur sous forme d'objet Variable <T> , de sorte qu'il utilise la surcharge **Random<T1, T2>**, où T1 spécifie le type du domaine de la variable DureeMoyenne et T2 spécifie son type de distribution.

Dans cet exemple, Random<T1,T2> crée une variable aléatoire de type double et la définit statistiquement en utilisant la distribution Gaussienne représentée par *MoyenneAPriori*.

BruitTrafic est créé et défini de la même manière.

À ce stade, la distribution a priori réelle n'a pas été spécifiée. Cela peut être fait plus tard, à tout moment avant d'interroger le moteur d'inférence, en affectant une valeur appropriée gaussienne à la propriété **ObservedValue** de la variable MoyenneAPriori. Pour spécifier un autre prieur comme distribution de la variable, affectez simplement l'objet de distribution à ObservedValue .

Cette approche présente deux avantages par rapport à celle utilisée par DureeCycliste1:

- Cela simplifie la mise en œuvre de modèles d'apprentissage avec plusieurs ensembles d'observations.
  Au lieu de créer une nouvelle variable pour chaque ensemble d'observations, DureeCycliste2 utilise *DureeMoyenne* dans l'application et met à jour le prieur en assignant chaque nouveau postérieur à la propriété ObservedValue de la variable MoyenneAPriori.
- Cela améliore les performances.
  Si vous modifiez le modèle, le moteur d'inférence doit le recompiler avant de lancer une requête, ce qui peut prendre beaucoup de temps. Cependant, assigner une nouvelle valeur à une propriété ObservedValue ne modifie pas le modèle, même si la valeur est un type de référence plutôt qu'un type en valeur. Pour toutes les requêtes après la première, le moteur exécute simplement le modèle compilé existant avec de nouvelles valeurs observées.

Enfin, *CreationModeleBayesien* crée une instance du moteur d'inférence, si on ne l'a pas déjà attribué par l'application parente.

c) Méthode DefinirDistributions

CreationModeleBayesien définit le modèle, mais n'affecte aucune valeur aux données du modèle. Après avoir créé le modèle, DureeCycliste2 appelle *DefinirDistributions* pour spécifier les données du modèle. Pour DureeCycliste2, les données de modèle sont constituées des prieurs, mais d'autres applications pourrait utiliser des données différentes. La méthode *DefinirDistributions* de la classe de base attribue des distributions vers les propriétés ObservedValue des variables *MoyenneAPriori* et *BruitAPriori.* La méthode est virtuelle, ce qui permet aux classes dérivées de la surcharger et spécifier différentes données, selon les besoins.

```
public virtual void DefinirDistributions(DonneesCycliste distribsApriori)
{
    MoyenneAPriori.ObservedValue = distribsApriori.DistribMoyenne;
    BruitAPriori.ObservedValue = distribsApriori.DistribBruitTraffic;
}
```

Pour plus de commodité, les données de modèle sont regroupées dans une structure *DonneesCycliste* privée :

```
public struct DonneesCycliste
{
    public Gaussian DistribMoyenne;
    public Gamma DistribBruitTraffic;

    public DonneesCycliste(Gaussian moyenne, Gamma precision)
    {
        DistribMoyenne = moyenne;
        DistribBruitTraffic = precision;
    }
}
```

2. Classe d'entraînement

La classe *EntrainementCycliste* hérite de *CyclisteBase* et met en œuvre le modèle d'entraînement.

L'exemple suivant montre la structure de la classe:

```
public class EntrainementCycliste : CyclisteBase
       {

              protected VariableArray<double> TempsDeTrajet;
              protected Variable<int> NombreDeTrajets;

              public override void CreationModeleBayesien()
              {
                     …
              }

              public DonneesCycliste CalculePosterieurs(double[] donneesObservees)
              {
                     …
              }

       }
```

Les champs représentent les éléments suivants :

- *TempsDeTrajet* représente un tableau de données d'entraînement.
  Le tableau de données est sous la forme d'un objet **VariableArray <T>**, qui est discuté plus tard dans la méthode *CreationModeleBayesien*.
- *NombreDeTrajets* représente la longueur du tableau de données d'apprentissage.
  La raison pour laquelle *NombreDeTrajets* est devenu un objet Variable <T> est expliquée plus loin dans la méthode *CreationModeleBayesien*

Les champs sont protégés plutôt que privés, afin que les classes dérivées puissent y accéder.

EntrainementCycliste a deux méthodes :

- *CreationModeleBayesien* remplace la méthode de base et implémente la méthode spécifique à l'entraînement du modèle.
- *CalculePosterieurs* prend les données d'apprentissage, exécute les requêtes et renvoie les postérieurs.

EntrainementCycliste n'a pas de données de modèle en dehors des prieurs de *DureeMoyenne* et *BruitTrafic*, et donc ne remplace pas *DefinirDistributions*.

a) Méthode CreationModeleBayesien

DureeCycliste2 appelle EntrainementCycliste.CreationModeleBayesien pour créer le modèle d'entraînement qui est implémentée comme suit :

```
public override void CreationModeleBayesien()
{
	base.CreationModeleBayesien();
	NombreDeTrajets = Variable.New<int>();
	Range indiceTrajet = new Range(NombreDeTrajets);
	TempsDeTrajet = Variable.Array<double>(indiceTrajet);
	using (Variable.ForEach(indiceTrajet))
	{
		TempsDeTrajet[indiceTrajet] =
Variable.GaussianFromMeanAndPrecision(Moyenne, Bruit);
	}
}
```

CreationModeleBayesien appelle la méthode de base pour créer les parties communes du modèle et implémente ensuite le code de modélisation spécifique à l'entraînement.

NombreDeTrajets est un type **Variable<int>** qui représente le nombre d'éléments dans le tableau de données d'entraînement. CreationModeleBayesien utilise New pour créer la variable NombreDeTrajets , puis utilise NombreDeTrajets pour initialiser indiceTrajet . NombreDeTrajets n'a pas de valeur à ce stade; il est spécifié ultérieurement en attribuant une valeur à la propriété ObservedValue .

Comme DureeCycliste2 a dix observations, il utilise un tableau de variables aléatoires pour représenter les valeurs observées au lieu d'objets Variable <double> séparés. Vous pourriez conditionner l'ensemble d'objets en tant que tableau .NET, mais le moteur d'inférence ne gérer pas ces tableaux efficacement. Une meilleure approche consiste à utiliser un objet VariableArray<T>, qui représente un tableau de variables aléatoires d'une manière qui peut être traitée efficacement par le moteur d'inférence.

Quelques détails explicatifs:

- VariableArray <T> est un type indexé, il peut donc être utilisé comme un tableau .NET standard. Cependant, vous devez utiliser un objet **Microsoft.ML.Probabilistic.Models.Range** en tant qu'index au lieu de l'entier habituel.
- L' objet **Range** est initialisé avec la longueur du tableau.
  Dans ce cas, la longueur du tableau est déterminée par NombreDeTrajets . La valeur NombreDeTrajets n'a pas été spécifié à ce stade, mais vous pouvez initialiser Range de cette manière à condition que vous spécifiez la valeur NombreDeTrajets avant d'interroger le moteur d'inférence.
- Pour créer un objet VariableArray <T> , appelez la méthode de création, statique **Variable.Array <T>** et transmettez-lui l' objet Range associé .
  Variable.Array <T> crée un objet VariableArray <T> , mais les variables ne sont pas définies statistiquement. Cette tâche doit être traitée séparément.

Remarque : Bien que NombreDeTrajets soit un objet Variable <int>, il représente en réalité une valeur définie: le nombre d'éléments du tableau. En faire un Variable<int> au lieu de int permet à DureeCycliste2 de gérer efficacement plusieurs sessions d'entraînement avec des données des tableaux de différentes longueurs. Si NombreDeTrajets était un int, DureeCycliste2 devrait créer un nouvel objet Range chaque fois que la longueur du tableau change, ce qui changerait le modèle et forcerait le compilateur de modèles à le recompiler. En utilisant un objet Variable<int>, le seul changement d'une session à l'autre est la propriété NombreDeTrajets ObservedValue, et la recompilation n'est pas nécessaire.

La dernière étape de la création du modèle d'entraînement consiste à définir statistiquement les éléments du tableau TempsDeTrajet. Parce que de trajet n'est pas un tableau standard, vous ne pouvez

pas utiliser *for* ou *foreach* pour parcourir le tableau. Au lieu de cela, Infer.NET fournit une méthode statique Variable.ForEach qui a le même objectif. ForEach prend un objet Range spécifié et itère sur la plage. L'instruction using (une norme C # courante) définit la portée du bloc ForEach et peut contenir un nombre arbitraire de déclarations.

b) Méthode CalculePosterieurs

DureeCycliste2 appelle *CalculePosterieurs* pour déduire les postérieurs de *DureeMoyenne* et *BruitTrafic* basés sur les prieurs spécifiés plus tôt dans *DefinirDistributions* et sur un tableau de données d'entraînement.

```
public DonneesCycliste CalculePosterieurs(double[] donneesObservees)
		{
			DonneesCycliste posterieurs;
			NombreDeTrajets.ObservedValue = donneesObservees.Length;
			TempsDeTrajet.ObservedValue = donneesObservees;
			posterieurs.DistribMoyenne =
MoteurInference.Infer<Gaussian>(Moyenne);
			posterieurs.DistribBruitTraffic =
MoteurInference.Infer<Gamma>(Bruit);

			return posterieurs;
		}
```

CalculePosterieurs assigne la longueur du tableau de données d'apprentissage à la propriété ObserveValue de la variable *NombreDeTrajets*. Il assigne ensuite les données d'entraînement à la propriété *ObservedValue* du tableau *TempsDeTrajet* et appelle le moteur d'inférence pour déduire les postérieurs. *CalculePosterieurs* retourne ensuite les postérieurs, conditionnés dans la structure *DonneesCycliste.*

3. Classe de prédiction

La classe *PredictionCycliste* implémente le modèle de prédiction. Elle hérite de CyclisteBase et a la structure de classe suivante:

```
public class PredictionCycliste : CyclisteBase
	{
		private Gaussian demainDistrib;
		public Variable<double> demainTemps;
		public override void CreationModeleBayesien()
		{
			…
		}
		public Gaussian EstimerTempsDemain()
		{
			…
		}
		public Bernoulli EstimerTempsDemainInferieurA(double duree)
		{
			…
		}
	}
```

Les champs représentent les éléments suivants :

• *dureeDemainDist* représente la distribution de la durée de trajet prévue.

• *demainTemps* est une variable aléatoire qui représente le temps de trajet prévu. Ce champ peut être privé aux fins de DureeCycliste2. Cependant, un champ public est utile pour construire des modèles qui impliquent plus d'un cycliste, discuté plus loin.

PredictionCycliste a trois méthodes :

• *CreationModeleBayesien* remplace la méthode de base et implémente la méthode spécifique à la partie prédiction du modèle.

• *EstimerTempsDemain* interroge le moteur d'inférence sur la distribution de durée prévue pour demain, à partir du modèle d'entraînement.

• EstimerTempsDemainInferieurA infère la probabilité que l'heure de demain soit moins longue que la valeur spécifiée.

PredictionCycliste traite les données du modèle de la même manière que EntrainementCycliste, donc elle ne remplace pas *DefinirDistributions*.

a) Méthode CreationModeleBayesien

DureeCycliste2 appelle PredictionCycliste.CreationModeleBayesien pour créer le modèle de prédiction.

```
public override void CreationModeleBayesien()
            {
                  base.CreationModeleBayesien();
                  demainTemps = Variable.GaussianFromMeanAndPrecision(
                   Moyenne, Bruit);
            }
```

CreationModeleBayesien appelle la méthode de base pour créer les parties communes du modèle. Elle implémente alors le code de modélisation spécifique à la prédiction, identique au code de DureeCycliste1.

b) Méthode EstimerTempsDemain

DureeCycliste2 appelle DefinirDistributions pour spécifier les prieurs du modèle de prédiction. Elle appelle alors *EstimerTempsDemain* pour obtenir la distribution prévue pour la durée du voyage de demain. Ces prieurs sont vraisemblablement les postérieurs qui ont été obtenus des sessions d'entraînement précédentes.

```
public Gaussian EstimerTempsDemain()
            {
                  demainDistrib = MoteurInference.Infer<Gaussian>(
                   demainTemps);
                  return demainDistrib;
            }
```

EstimerTempsDemain déduit le marginal sur la base des prieurs que DureeCycliste2 a spécifié quand il a appelé DefinirDistributions , puis renvoie la distribution à l'appelant.

Le calcul de l'inférence dans *EstimerTempsDemain* est plus efficace que le calcul correspondant dans DureeCycliste1, qui devait ré-entraîner le modèle à partir du début avant de calculer le temps prévu de

demain. DureeCycliste2 calcule *dureeDemainDist* en utilisant les postérieurs du modèle d'entraînement comme prieurs. Ces distributions représentent essentiellement les résultats de l'entraînement, il n'est donc pas nécessaire de répéter le calcul de l'entraînement.

c) Méthode EstimerTempsDemainInferieurA

EstimerTempsDemainInferieurA déduit la probabilité que le temps de trajet de demain soit inférieur à une durée spécifiée.

```
public Bernoulli EstimerTempsDemainInferieurA(double duree)
            {
                  return MoteurInference.Infer<Bernoulli>(demainTemps < duree);
            }
```

Ce code exécute la même tâche que le code correspondant dans DureeCycliste1.

EstimerTempsDemainInferieurA renvoie une distribution de Bernoulli, qui est caractérisée par un paramètre unique qui spécifie la probabilité que la variable soit vraie. Dans ce cas, vrai correspond à la probabilité que la durée du trajet de demain soit inférieure à la valeur spécifiée.

4. Utilisation du modèle

Cette section explique comment DureeCycliste2 utilise les classes décrites dans le

sections précédentes pour former le modèle et prédire le temps de déplacement de demain.

a) Entraînement

La première partie de DureeCycliste2 crée et entraîne le modèle, comme indiqué dans l'extrait suivant :

```
static void DureeCycliste2()
            {

                  // 1 - Entraînement

                  double[] donneesTrajets = new[] { 13, 17, 20, 25, 16, 11, 16, 14,
12.5 };
                  DonneesCycliste mesDistributions = new DonneesCycliste(
                    Gaussian.FromMeanAndPrecision(1, 0.01),
                   Gamma.FromShapeAndScale(2, 0.5));
                  EntrainementCycliste monEntrainement = new EntrainementCycliste();
                  monEntrainement.CreationModeleBayesien();
                  monEntrainement.DefinirDistributions(mesDistributions);
                  DonneesCycliste monPosterieur =
monEntrainement.CalculePosterieurs(donneesTrajets);
                  Console.WriteLine($"moyenne posterieure
{monPosterieur.DistribMoyenne.ToString()}");
                  Console.WriteLine($"bruit posterieur
{monPosterieur.DistribBruitTraffic.ToString()}");
…

            }
```

DureeCycliste2 définit les prieurs initiaux pour *DureeMoyenne* et *BruitTrafic* , en utilisant la structure *DonneesCycliste* .

DureeCycliste2 crée un nouvel objet *EntrainementCycliste* pour représenter le modèle d'entraînement. Il appelle *CreationModeleBayesien* pour créer le modèle d'entraînement et transmet les prieurs initiaux à *DefinirDistributions*. Enfin, DureeCycliste2 transmet les données d'apprentissage à *CalculePosterieurs*, qui renvoie les postérieurs.

Les résultats sont:

moyenne posterieure Gaussian(15,85, 1,762)

bruit posterieur Gamma(5,158, 0,01568)[mean=0,08087]

b) Prédiction

DureeCycliste2 utilise ensuite les postérieurs de la première session d'entraînement pour prédire les résultats de demain.

```
static void DureeCycliste2()
              {

                    // 1 - Entraînement

                    …

                    // 2 - Prédiction

                    PredictionCycliste maPrediction = new PredictionCycliste();
                    maPrediction.CreationModeleBayesien();
                    maPrediction.DefinirDistributions(monPosterieur);
                    Gaussian ditribDemain = maPrediction.EstimerTempsDemain();
                    Console.WriteLine($"Prédiction demain {ditribDemain.ToString()}");
                    Console.WriteLine($"Ecart type:
{Math.Sqrt(ditribDemain.GetVariance())}");
                    Console.WriteLine($"Probabilité durée < 13mn: " +

$"{maPrediction.EstimerTempsDemainInferieurA(13)}");

                    …

              }
```

La procédure est similaire à la phase d'entraînement. *DureeCycliste2* crée un nouveau *PredictionCycliste* pour représenter le modèle d'entraînement. Il appelle *CreationModeleBayesien* pour créer le modèle d'entraînement et passe les postérieurs de la phase d'entraînement à DefinirDistributions. Enfin, DureeCycliste2 appelle EstimerTempsDemain et EstimerTempsDemainInferieurA pour obtenir les résultats prévus, comme suit:

Compiling model...done.

Prédiction demain Gaussian(15,85, 17,1)

Ecart type: 4,13544715993872

Compiling model...done.

Probabilité durée < 13mn: Bernoulli(0,2456)

c) Apprentissage en ligne

DureeCycliste1 est limité à une seule phase d'entraînement et à un seul ensemble de prédictions. Les programmes probabilistes utilisent souvent un processus **appelé apprentissage en ligne**, qui permet à une application d'apprendre les paramètres du modèle rapidement en fonction de certaines données initiales, puis continuer à apprendre progressivement à mesure que plus de données entrent. Le processus de base est le suivant :

1. Commencez par les a priori initiaux généralement choisis pour avoir des distributions très larges.
2. Recueillez quelques données et calculez les postérieurs.
3. Mettez à jour les prieurs du modèle afin de refléter les nouvelles données en les remplaçant par les postérieurs de l'étape 2.
4. Utilisez le modèle mis à jour pour exécuter des prévisions.
5. Répétez les étapes 2 à 4 autant de fois que nécessaire, peut-être indéfiniment.

Au fur et à mesure que vous continuez le cycle de collecte de données et de mise à jour des statistiques a priori, les paramètres du modèle doivent approcher les valeurs réelles et la capacité prédictive du modèle devrait s'améliorer. De plus, si les conditions externes changent, l'entraînement en ligne permet au modèle de s'adapter à ces changements.

Il est important de noter que si vous êtes prêt à conserver les données, vous pouvez toujours réentraîner votre modèle à partir de zéro en utilisant les versions a priori et toutes les données. C'est la différence fondamentale entre l'apprentissage hors ligne et l'apprentissage en ligne.

(1) Mise à jour des postérieurs

Jusqu'à présent, DureeCycliste2 n'est qu'une version plus sophistiquée et efficace de DureeCycliste1, avec quelques données d'entraînement supplémentaires. Cependant, l'approche utilisée par DureeCycliste2 est plus flexible. Cette section montre comment utiliser les classes *EntrainementCycliste* et *PredictionCycliste* pour mettre en œuvre l'apprentissage en ligne.

DureeCycliste2 a déjà terminé une phase d'entraînement et utilisé les résultats pour la prédiction. En mettant en place une deuxième phase d'entraînement et de prédiction, vous pouvez : améliorer progressivement la précision des distributions de *DureeMoyenne* et *BruitTrafic*. L'approche de base peut être facilement étendue pour traiter autant de séances d'entraînement que de besoin.

L'exemple suivant montre comment DureeCycliste2 implémente la deuxième session d'entraînement :

```
static void DureeCycliste2()
            {

                    // 1 - Entraînement

                    …

                    // 2 - Prédiction

                    …

                    // 3 - Apprentissage en ligne

                    double[] semaineSuivante = new Double[] { 18, 25, 30, 14, 11 };
                    maPrediction.DefinirDistributions(monPosterieur);
                    DonneesCycliste posterieurSemaineSuivante =
monEntrainement.CalculePosterieurs(semaineSuivante);
                    Console.WriteLine($"moyenne posterieure semaine suivante
{posterieurSemaineSuivante.DistribMoyenne.ToString()}");
                    Console.WriteLine($"bruit posterieur semaine suivante
{posterieurSemaineSuivante.DistribBruitTraffic.ToString()}");

                    …

            }
```

Le deuxième ensemble de données d'entraînement est pour cinq jours. EntrainementCycliste peut gérer des ensembles de données de longueur arbitraire, de sorte qu'une session d'entraînement peut s'étendre sur n'importe quelle fenêtre temporelle, même quotidiennement.

Une instance de *EntrainementCycliste* existe déjà et le modèle a été créé au cours de la première session d'entraînement, les sessions d'entraînement suivantes ne nécessitent que deux lignes de code :

1. Appelez EntrainementCycliste.DefinirDistributions pour mettre à jour les prieurs du modèle d'entraînement avec les postérieurs de la session d'entraînement précédente.
2. Appelez EntrainementCycliste.CalculePosterieurs pour déduire de nouveaux postérieurs.

Les nouveaux postérieurs intègrent maintenant les prieurs initiaux et les données d'entraînement, ainsi que des informations du second ensemble de données d'entraînement. En pratique, l'influence des prieurs initiaux est maintenant négligeable, et les postérieurs reflètent les données d'entraînement.

(2) Nouvelles prédictions

```
        static void DureeCycliste2()
               {


…
                        // 3 - Apprentissage en ligne

…
                        // 4 - Nouvelle prédiction
                        maPrediction.DefinirDistributions(posterieurSemaineSuivante);
                        ditribDemain = maPrediction.EstimerTempsDemain();
                        Console.WriteLine($"Prédiction semaine prochaine
{ditribDemain.ToString()}");
                        Console.WriteLine($"Ecart type:
{Math.Sqrt(ditribDemain.GetVariance())}");
                        Console.WriteLine($"Probabilité durée < 13mn: " +

$"{maPrediction.EstimerTempsDemainInferieurA(13)}");

               }
```

DureeCycliste2 appelle *PredictionCycliste.DefinirDistributions* pour mettre à jour les prieurs de la prédiction du modèle avec les nouveaux postérieurs et appelle les deux méthodes d'inférence de *PredictionCycliste*, pour prédire le temps de trajet du lendemain.

Cette procédure démontre l'avantage d'utiliser des objets Variable <T> pour représenter les a priori. Lorsque vous mettez à jour les prieurs, *DefinirDistributions* attribue simplement les distributions spécifiées aux propriétés **ObservedValue** des variables *MoyenneAPriori* et *BruitAPriori*. Parce que seules les propriétés **ObservedValue** ont changé entre la première et la deuxième phase, les modèles d'entraînement et de prédiction sont inchangé et ne doivent pas être recompilés. Le moteur d'inférence peut simplement exécuter les modèles compilés de la première phase avec les nouvelles données d'entraînement et les précédents.

Les résultats pour la deuxième phase d'entraînement et de prédiction sont les suivants :

moyenne posterieure semaine suivante Gaussian(18,49, 9,785)

bruit posterieur semaine suivante Gamma(2,747, 0,01289)[mean=0,03541]

Prédiction semaine prochaine Gaussian(18,49, 54,19)

Ecart type: 7,3610514253304

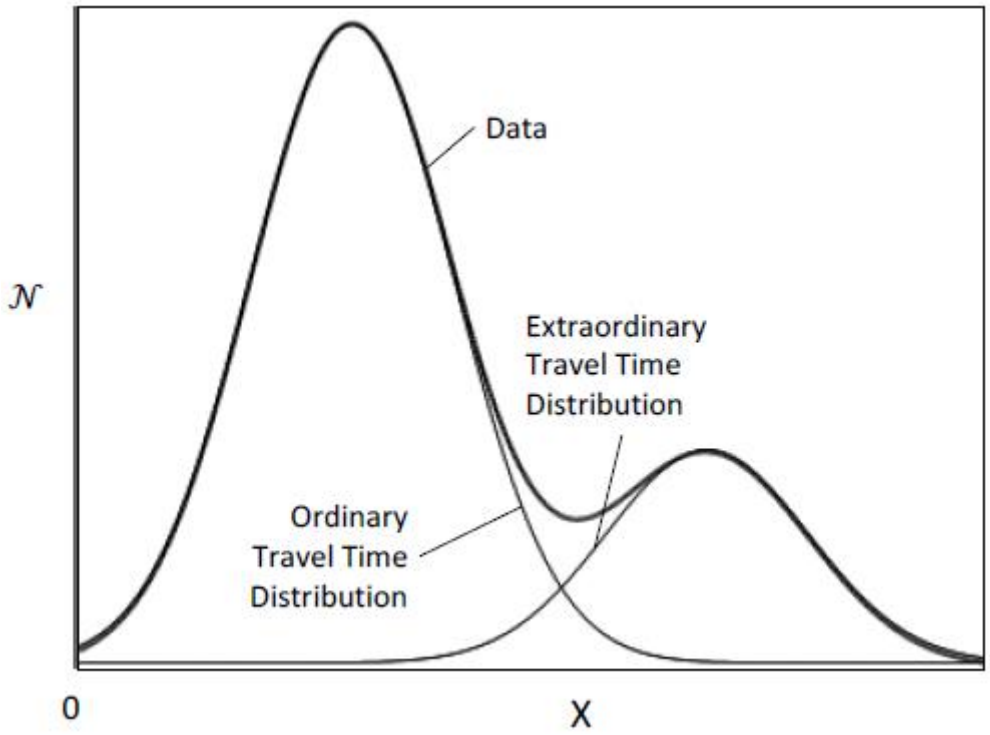
Compiling model...done.

Probabilité durée < 13mn: Bernoulli(0,2278)

D. Modèle mixte

Le scénario pour DureeCycliste1 et DureeCycliste2 supposait que la distribution des temps de trajet pourrait être représentés par une seule distribution gaussienne. Ceci est une hypothèse commune, mais les données ne sont pas nécessairement aussi coopératives.

Par exemple, tout cycliste sait que vous rencontrez parfois des circonstances extraordinaires, comme un pneu crevé ou une route fermée pour travaux, qui ajoutent un temps substantiel à votre durée de déplacement habituel. Une seule gaussienne pourrait être un modèle raisonnable pour les déplacements ordinaires, mais les données du scénario précédent pourraient sembler plus comme la ligne épaisse de la figure suivante :



L'effet des temps de trajet extraordinaires dans cet exemple est un ensemble de données avec deux pics — un correspondant aux trajets normaux et un correspondant aux trajets anormaux. Représenter cet ensemble de données par un seul gaussien, fusionne les effets des deux types de trajets dans un seul modèle relativement simple, qui pourrait ne pas être très précis. Une autre approche, souvent meilleure, consiste à représenter ces données sous forme de mélange de plusieurs Gaussiennes. Les **modèles mixtes** ont l'avantage supplémentaire d'être moins sensible aux données aberrantes qu'une seule distribution.

Pour spécifier un tel modèle, vous n'avez pas besoin de connaître la fréquence relative de chaque type de trajet, mais vous devez spécifier le nombre de Gaussiennes dans le mélange. Dans ce cas, deux Gaussiennes est le choix évident, mais le nombre optimal n'est pas toujours aussi apparent. Comme indiqué plus loin, l'inférence bayésienne permet de déterminer le nombre optimal de Gaussiennes dans le mélange en utilisant **les preuves du modèle**.

L'application DureeCycliste3 étend les modèles DureeCycliste2 pour gérer des événements extraordinaires, comme suit:

- La plupart des voyages sont ordinaires, avec un temps de trajet moyen d'environ 15 – 16 minutes.
- Cinq à dix pour cent sont extraordinaires et nécessitent dix à quinze autres minutes.

DureeCycliste3 gère ce scénario en représentant le système comme un mélange de deux Gaussiennes. Pour plus de commodité, la gaussienne qui correspond aux durées de trajet ordinaires est appelé la composante ordinaire et la gaussienne qui correspond aux durées de trajet extraordinaires est appelé la composante extraordinaire.

DureeCycliste3 est généralement similaire à DureeCycliste2, comme suit :

- Les modèles d'entraînement et de prédiction sont mis en œuvre par les classes *EntrainementCyclisteMixte* et *PredictionCyclisteMixte*.
- *EntrainementCyclisteMixte* et *PredictionCyclisteMixte* héritent d'une classe de base, *CyclisteBaseMixte*, qui implémente les parties communes des modèles.
- La structure de classe de base est identique aux classes correspondantes de DureeCycliste2. Les différences se limitent aux détails du code de modélisation.
- Une méthode statique DureeCycliste3 crée et entraîne le modèle d'entraînement, puis utilise les résultats et le modèle de prédiction pour prédire le temps de trajet de demain.

Par concision, DureeCycliste3 utilise une seule phase d'entraînement et de prédiction, mais peut facilement être étendu à plusieurs phases.

1. Classe de base

CyclisteBaseMixte possède les champs suivants, dont la plupart des variables aléatoires sont partagés par les classes d'entraînement et de prédiction :

```
public class CyclisteBaseMixte
	{

		public InferenceEngine MoteurInference;

		protected int NombreComposantes = 2;

		protected VariableArray<Gaussian> MoyennesAPriori;
		protected VariableArray<Gamma> BruitsAPriori;
		protected Variable<Dirichlet> MixeAPriori;

		protected VariableArray<double> Moyennes;
		protected VariableArray<double> Bruits;
		protected Variable<Vector> Mixe;


		public virtual void CreationModeleBayesien()
		{
			…
		}

		public virtual void DefinirDistributions(
			DonneesCyclisteMixte distribsApriori)
		{
			…
		}

	}
```

Les premières variables aléatoires ont la même fonction que les variables correspondantes dans DureeCycliste2. Cependant, un modèle de mélange nécessite une *dureeMoyenne* et *bruitTrafic* distincts pour chaque composant de mélange, il y en a donc deux.

Pour faciliter le calcul, les deux valeurs *dureeMoyenne* et *bruitTrafic* sont représentées sous forme de tableaux variables, de même que les a priori correspondants. *MixeAPriori* et *Mixe* sont liés au modèle de mélange et sont discutés plus tard.

a) Méthode CreationModeleBayesien

CyclisteBaseMixte.CreationModeleBayesien crée et définit les variables communes aux deux modèles. Il est similaire en principe à CyclisteBase.CreationModeleBayesien, mais les détails sont un un peu plus compliqués.

```
public virtual void CreationModeleBayesien()
            {
                  Range indiceComposants = new Range(NombreComposantes);
                  MoteurInference = new InferenceEngine(new
VariationalMessagePassing());

                  MoyennesAPriori = Variable.Array<Gaussian>(indiceComposants);
                  BruitsAPriori = Variable.Array<Gamma>(indiceComposants);
                  Moyennes = Variable.Array<double>(indiceComposants);
                  Bruits = Variable.Array<double>(indiceComposants);

                  using (Variable.ForEach(indiceComposants))
                  {
                        Moyennes[indiceComposants] =
Variable<double>.Random(MoyennesAPriori[indiceComposants]);
                        Bruits[indiceComposants] =
Variable<double>.Random(BruitsAPriori[indiceComposants]);
                  }

                  MixeAPriori = Variable.New<Dirichlet>();
                  Mixe = Variable<Vector>.Random(MixeAPriori);
                  Mixe.SetValueRange(indiceComposants);

            }
```

Le moteur d'inférence prend en charge plusieurs algorithmes d'inférence. L'algorithme par défaut, qui est utilisé par les exemples précédents, est la propagation d'espérance. CreationModeleBayesien spécifie plutôt l'algorithme de passage de message variationnel pour le moteur d'inférence.

Le passage de message variationnel est souvent un meilleur choix pour les modèles de mélange, et la propagation d'espérance pour DureeCycliste3 pose des problèmes numériques dans calcul d'inférence, donnant lieu à des messages incorrects.

L'objet Range, *IndiceComposant*, est initialisé avec le nombre de composants (2). Il définit la plage de plusieurs objets VariableArray <T> liés aux composants.

Pour plus de commodité, les variables aléatoires *DureeMoyenne* et *BruitTrafic* et leurs prieurs sont définis sous forme de tableaux à deux éléments. CyclisteBaseMixte définit ensuite *DureeMoyenne* et *BruitTrafic* en utilisant **Variable.ForEach** pour parcourir les deux composants de mélange et définir chaque élément de DureeMoyenne et BruitTrafic avec le prieur approprié.

Le dernier ensemble de variables aléatoires est nécessaire pour définir le mélange lui-même. Les composants du mélange n'ont généralement pas une influence égale sur la distribution globale, de sorte que la proportion relative de chaque composant dans le mélange est définie par une paire de coefficients de mélange, chacun spécifiant la proportion d'une des composantes.

CyclisteBaseMixte représente les coefficients de mélange par une variable aléatoire de type **Vector**, Mixe . Le type de vecteur - défini par Infer.NET dans l'espace de noms **Microsoft.ML.Probabilistic.Math** :

contient un ensemble de valeurs de type double, deux dans ce cas. Mixe est défini par une distribution a priori de type **Dirichlet** , *MixeAPriori*, qui est représentée par un objet Variable <T> afin qu'une valeur puisse lui être affectée ultérieurement.

En bref, une distribution de Dirichlet est une distribution sur un vecteur de probabilité, qui est un vecteur dont les éléments totalisent 1,0. Par exemple, un échantillon d'une telle distribution de dimension deux, tel que celui utilisé par DureeCycliste3, pourrait être (0,25, 0,75).

La dernière ligne appelle Variable.SetValueRange pour spécifier explicitement la plage de *Mixe*, car le compilateur de modèles Infer.NET ne peut pas toujours déduire la plage de valeurs qu'une variable aléatoire entière peut prendre.

b) Méthode DefinirDistributions

DureeCycliste3 appelle DefinirDistributions pour spécifier des a priori pour DureeMoyenne, BruitTrafic ,

et Mixe.

```
public virtual void DefinirDistributions(
                    DonneesCyclisteMixte distribsApriori)
             {
                    MoyennesAPriori.ObservedValue = distribsApriori.DistribMoyenne;
                    BruitsAPriori.ObservedValue = distribsApriori.DistribBruitTraffic;
                    MixeAPriori.ObservedValue = distribsApriori.DistribMixe;
             }
```

Pour plus de commodité, les données de modèle sont présentées sous forme de structure DonneesCyclisteMixed .

```
public struct DonneesCyclisteMixte
    {
        public Gaussian[] DistribMoyenne;
        public Gamma[] DistribBruitTraffic;
        public Dirichlet DistribMixe;

    }
```

2. Classe d'entraînement

La classe EntrainementCyclisteMixte hérite de CyclisteBaseMixte et implémente le modèle d'entraînement.

Le listing suivant montre la structure de la classe et ses champs :

```
public class EntrainementCyclisteMixte : CyclisteBaseMixte
	{

		protected Variable<int> NombreDeTrajets;

		protected VariableArray<double> TempsDeTrajet;
		protected VariableArray<int> ComposantesTrajets;


		public override void CreationModeleBayesien()
		{
			…
		}

		public DonneesCyclisteMixte CalculePosterieurs(
		   double[] donneesObservees)
		{
			…
		}
	}
```

Les champs *NombreDeTrajets* et *TempsDeTrajet* ont le même objectif que dans *EntrainementCycliste*. *ComposantesTrajets* est un nouveau tableau de variables aléatoires int avec la même plage de données que *TempsDeTrajet* . Il spécifie l'indice du composant de mélange qui a généré le temps de trajet correspondant.

Le processus génératif est le suivant :

1) Pour chaque composant de mélange, échantillonnez *DureeMoyenne* et *BruitTrafic* de leurs prieurs respectifs.
2) Echantillonnez les coefficients de mélange à partir du prieur de mélange.
3) Pour chaque trajet, procédez comme suit :
   a) Échantillon de l'index de composant pour déterminer quel composant de mélange a généré le temps de trajet.
   b) Échantillon de la valeur du temps de trajet d'une distribution gaussienne dont moyenne et précision sont les valeurs *DureeMoyenne* et *BruitTrafic* pour ce composant.

a) Méthode CreationModeleBayesien

EntrainementCyclisteMixte.CreationModeleBayesien implémente ce modèle, comme illustré ci-dessous.

```
public override void CreationModeleBayesien()
            {
                  base.CreationModeleBayesien();
                  NombreDeTrajets = Variable.New<int>();
                  Range indiceTrajet = new Range(NombreDeTrajets);
                  TempsDeTrajet = Variable.Array<double>(indiceTrajet);
                  ComposantesTrajets = Variable.Array<int>(indiceTrajet);

                  using (Variable.ForEach(indiceTrajet))
                  {
                        ComposantesTrajets[indiceTrajet] = Variable.Discrete(Mixe);
                        using (Variable.Switch(ComposantesTrajets[indiceTrajet]))
                        {
                              TempsDeTrajet[indiceTrajet].SetTo(
                               Variable.GaussianFromMeanAndPrecision(
                                Moyennes[ComposantesTrajets[indiceTrajet]],
                                Bruits[ComposantesTrajets[indiceTrajet]]));
                        }
                  }
            }
```

CreationModeleBayesien appelle la méthode de la classe de base pour créer et définir les variables communes. Il crée ensuite les variables de champs de la classe, et une variable **Range** initialisée par NombreDeTrajets. Le cœur du modèle d'entraînement est créé dans les blocs **using**.

Le bloc using externe utilise Variable.ForEach pour parcourir les éléments des tableaux *TempsDeTrajet* et *ComposantesTrajets*. Pour chaque itération, il définit l'élément *ComposantesTrajets* correspondant à l'aide d'une distribution **Discrete** dont les paramètres sont spécifiés par *Mixe*. Les coefficients de Mixe déterminent la probabilité que chacune des deux composantes du mélange produise la valeur donnée.

Le bloc **using** interne implémente les branches qui définissent le mélange. Avec des branches standard, telles que if-else, une branche s'exécute et les autres non. Avec la programmation probabiliste, au lieu de tout ou rien, chaque branche s'exécute avec une probabilité spécifiée, qui détermine la proportion de contribution de la branche au modèle.

La probabilité que chaque branche s'exécute est spécifiée par une variable aléatoire discrète appelée une condition. Si, par exemple, la condition a des probabilités égales pour toutes les valeurs possibles, chaque branche est activée avec une probabilité égale et contribue également au modèle.

DureeCycliste3, implémente des branches en utilisant la méthode **Variable.Switch**, qui représente plusieurs branches dans le code généré, une pour chaque valeur de condition possible. Chaque branche exécute le code dans le bloc **using**, comme suit :

- La probabilité de chacune des valeurs possibles de la condition détermine le pourcentage du temps d'exécution de la branche associée.
- Le code est essentiellement le même pour chaque branche, sauf que le code dans chaque branche utilise la valeur possible de la condition pour cette branche particulière.

Pour *EntrainementCyclisteMixte* :

- La condition de commutation est l'élément actuellement sélectionné dans *ComposantesTrajets* qui génère deux branches, une pour chaque composant du mélange.
- La probabilité d'exécution de chaque branche est déterminée par les coefficients du mélange.
- Le code de chaque branche définit l'élément *TempsDeTrajet* actuellement sélectionné par une distribution gaussienne caractérisée par *Moyennes* et *Bruits* pour le composant associé.

Remarque : *EntrainementCyclisteMixte* utilise **Variable.SetTo** pour définir *TempsDeTrajet* plutôt que l'opérateur '='. Ceci n'est pas strictement requis pour cet exemple, mais il est utile de prendre l'habitude

d'utiliser **SetTo** car il est requis dans une ou deux situations. Par exemple, SetTo est requis si vous fournissez la définition statistique d'une variable aléatoire scalaire dans les deux branches d'une construction If / IfNot. Parce qu'Infer.NET ne peut pas surcharger l'opérateur '=' de C#, la définition de la branche IfNot remplacera la définition dans la branche If.

b) Méthode CalculePosterieurs

CalculePosterieurs déduit les postérieurs sur la base des prieurs spécifiés précédemment dans DefinirDistributions et un tableau de données d'apprentissage.

```
public DonneesCyclisteMixte CalculePosterieurs(
                double[] donneesObservees)
            {
                  DonneesCyclisteMixte posterieurs;
                  NombreDeTrajets.ObservedValue = donneesObservees.Length;
                  TempsDeTrajet.ObservedValue = donneesObservees;
                  posterieurs.DistribMoyenne =
MoteurInference.Infer<Gaussian[]>(Moyennes);
                  posterieurs.DistribBruitTraffic =
MoteurInference.Infer<Gamma[]>(Bruits);
                  posterieurs.DistribMixe = MoteurInference.Infer<Dirichlet>(Mixe);

                  return posterieurs;
            }
```

DureeCycliste3 passe dans un tableau de double contenant les données d'apprentissage. *CalculePosterieurs* définit *NombreDeTrajets* avec la longueur du tableau et affecte les données d'apprentissage à *TempsDeTrajet*. Il en déduit ensuite les postérieurs pour *DureeMoyenne*, *BruitTrafic* et *Mixe*, et les renvoie à DureeCycliste3 en tant que structure *DonneesCyclisteMixed.*

3. Classe de prédiction

La classe *PredictionCyclisteMixte* implémente le modèle de prédiction. L'exemple suivant montre la structure de la classe. Il est presque identique à *PredictionCycliste*.

```
      public class PredictionCyclisteMixte : CyclisteBaseMixte
      {
            private Gaussian demainDistrib;
            public Variable<double> demainTemps;

            public override void CreationModeleBayesien()
            {
                  …
            }

            public Gaussian EstimerTempsDemain()
            {
                  …
            }

      }
```

PredictionCyclisteMixte hérite de *CyclisteBaseMixte* et ses champs et méthodes servent les mêmes objectifs que les champs et méthodes équivalents de *PredictionCycliste*. Les différences se trouvent dans les détails de la mise en œuvre.

a) Méthode CreationModeleBayesien

DureeCycliste3 appelle *PredictionCycliste.CreationModeleBayesien* pour créer le modèle de prédiction.

```
public override void CreationModeleBayesien()
{
	base.CreationModeleBayesien();
	Variable<int> indiceComposant = Variable.Discrete(Mixe);
	demainTemps = Variable.New<double>();
	using (Variable.Switch(indiceComposant))
	{

	demainTemps.SetTo(Variable.GaussianFromMeanAndPrecision(Moyennes[indiceComposant
], Bruits[indiceComposant]));
	}
}
```

Le modèle prédit un seul temps de trajet, il existe donc une variable d'index de composant unique, *indiceComposant*. Elle est définie par une distribution discrète à deux éléments qui est caractérisée par les coefficients de mélange, vraisemblablement le posterieur déterminé par la phase d'entraînement.

Le modèle utilise ensuite **Variable.Switch** pour définir *demainTemps* en utilisant un mélange des deux composants, avec les proportions définies par Mixe.

b) Méthode EstimerTempsDemain

DureeCycliste3 appelle la méthode *EstimerTempsDemain* pour obtenir la distribution prédite pour le temps de demain. La mise en œuvre est identique à *PredictionCycliste.EstimerTempsDemain*.

```
public Gaussian EstimerTempsDemain()
{
	demainDistrib = MoteurInference.Infer<Gaussian>(demainTemps);
	return demainDistrib;
}
```

4. Utilisation du modèle

DureeCycliste3 crée une instance de *EntrainementCyclisteMixte*, entraîne le modèle, puis utilise le modèle formé pour prédire le temps de déplacement de demain.

a) Entraînement

```
static void DureeCycliste3()
            {
                  // 1 - Entraînement

                  double[] donneesTrajets = new[] { 13, 17, 20, 25, 16, 11, 16, 25,
12.5, 30 };
                  DonneesCyclisteMixte mesDistribsAPriori;
                  mesDistribsAPriori.DistribMoyenne = new Gaussian[]
                  {
                        new Gaussian(15, 100), //Ordinaire
                        new Gaussian(30, 100) //Extraordinaire
                  };
                  mesDistribsAPriori.DistribBruitTraffic = new Gamma[]
                  {
                        new Gamma(2, 0.5), //O
                        new Gamma(2, 0.5)  //E
                  };
                  mesDistribsAPriori.DistribMixe = new Dirichlet(1, 1);

                  EntrainementCyclisteMixte monEntrainement = new
EntrainementCyclisteMixte();
                  monEntrainement.CreationModeleBayesien();
                  monEntrainement.DefinirDistributions(mesDistribsAPriori);

                  DonneesCyclisteMixte posterieur =
monEntrainement.CalculePosterieurs(donneesTrajets);

                  Console.WriteLine($"moyenne posterieure 1
{posterieur.DistribMoyenne[0].ToString()}");
                  Console.WriteLine($"bruit posterieur 1
{posterieur.DistribBruitTraffic[0].ToString()}");

                  Console.WriteLine($"moyenne posterieure 2
{posterieur.DistribMoyenne[1].ToString()}");
                  Console.WriteLine($"bruit posterieur 2
{posterieur.DistribBruitTraffic[1].ToString()}");
                  Console.WriteLine($"Coefficients du mélange:
{posterieur.DistribMixe}");

                  // 2 - Prédiction

                  …

            }
```

Le jeu de données d'apprentissage est similaire à celui utilisé par DureeCycliste2, à l'exception de durées supplémentaires qui représentent des événements extraordinaires.

DureeCycliste3 crée les prieurs suivants :

- Le composant de mélange ordinaire a les mêmes a priori que ceux utilisés par DureeCycliste2. Ils sont étiquetés O dans l'exemple.
- La composante de mélange extraordinaire fixe le temps de trajet moyen à 30, basé sur l'estimation que des événements extraordinaires ajoutent environ 15 minutes à un voyage. Les prieurs extraordinaires sont étiquetés E dans l'exemple. Les valeurs de précision et le prieur de BruitTrafic sont définis sur les mêmes valeurs que pour le composant ordinaire.
- Le prieur initial de *DistribMixe* est une distribution de Dirichlet (1, 1). Cette distribution correspond à une proportion égale des deux composants avec une relativement grande incertitude.

Important : Si vous ne pouvez pas estimer les prieurs initiaux des composants du mélange, vous pouvez généralement les définir sur les mêmes valeurs. Cependant, si vous utilisez des prieurs identiques pour tous composants du mélange, vous devez initialiser le modèle pour briser sa symétrie.

DureeCycliste3 crée une instance de *EntrainementCyclisteMixte*, appelle *CreationModeleBayesien* pour créer le modèle et appelle *DefinirDistributions* pour spécifier les prieurs. DureeCycliste3 appelle ensuite *CalculePosterieurs* pour déduire les postérieurs des moyennes, précisions et coefficients de mélange du modèle, et affiche les résultats suivants :

moyenne posterieure 1: Gaussian(15,07, 0,8674)

bruit posterieur 1: Gamma(5,498, 0,02972)[mean=0,1634]

moyenne posterieure 2: Gaussian(26,69, 1,146)

bruit posterieur 2: Gamma(3,502, 0,08199)[mean=0,2871]

Coefficients du mélange: Dirichlet(7,995 4,005)

La distribution 1 correspond aux temps de trajet ordinaires et la distribution 2 aux temps de trajet extraordinaires. Après avoir entraîné le modèle,

b) Prédiction

```
static void DureeCycliste3()
              {
                     // 1 - Entraînement

                     …

                     // 2 - Prédiction

                     PredictionCyclisteMixte maPrediction = new
PredictionCyclisteMixte();
                     maPrediction.CreationModeleBayesien();
                     maPrediction.DefinirDistributions(posterieur);
                     Gaussian ditribDemain = maPrediction.EstimerTempsDemain();
                     Console.WriteLine($"Prédiction demain {ditribDemain.ToString()}");
                     Console.WriteLine($"Ecart type:
{Math.Sqrt(ditribDemain.GetVariance())}");

              }
```

DureeCycliste3 crée une instance de *PredictionCyclisteMixte*, crée le modèle et définit les prieurs à partir des postérieurs du modèle qui vient d'être calculé.

Les résultats sont :

Prédiction demain Gaussian(18,48, 33,39)

Ecart type: 5,77865034061496

Le temps de trajet moyen prévu et l'écart type sont tous deux plus grands que pour DureeCycliste2, qui reflète l'inclusion d'événements extraordinaires dans le modèle.

E. Comparaison de Modèles

Nous disposons maintenant de deux modèles pour représenter le temps de trajet des cyclistes : l'un basé sur une seule gaussienne et l'autre sur un mélange de deux gaussiennes. Pourquoi ne pas envisager un modèle basé sur un mélange de trois gaussiennes, ou vingt ou trente gaussiennes ? Ou bien est-ce qu'une seule gaussienne suffit ? Comment choisir le meilleur modèle ?

En général, un modèle complexe avec plus de paramètres ajustables représente un ensemble de données particulier plus précisément qu'un modèle plus simple avec moins de paramètres. La question plus intéressante est de savoir si le modèle plus complexe offre une meilleure représentation qu'un modèle plus simple. Les modèles trop adaptés à un ensemble de données particulier ne seront pas utiles en pratique car ils ne s'ajusteront pas nécessairement bien aux nouvelles données.

Par exemple, lorsque vous ajustez un polynôme à un ensemble de points de données, vous pouvez toujours obtenir un ajustement exact en ajoutant suffisamment d'éléments au polynôme. Cependant, un modèle qui correspond exactement à chaque point de données fluctue généralement de manière erratique entre les points et ne correspondra donc pas bien aux nouvelles données. Ce phénomène est connu sous le nom de surapprentissage (overfitting).

La meilleure approche consiste à trouver un modèle qui ajuste raisonnablement bien les données sans être trop complexe. Comparer visuellement les modèles avec les données est subjectif et peu pratique pour des scénarios plus complexes. Ce qu'il vous faut est un critère objectif, comme le rasoir d'Occam, qui évalue quantitativement la qualité du modèle et détermine le compromis optimal entre précision et complexité. En inférence bayésienne, il existe un mécanisme robuste pour évaluer la qualité du modèle appelé preuve du modèle (evidence).

1. Calculer la preuve du modèle avec Infer.NET

Avec Infer.NET, la preuve est représentée par une variable aléatoire booléenne. La procédure de base est illustrée dans l'exemple suivant.

```
Variable<bool> Evidence = Variable.Bernoulli(0.5);
using (Variable.If(Evidence))
{
    // Implémenter le modèle d'entraînement à évaluer
}
// Observer les données d'entraînement
// Interroger le moteur d'inférence pour les postérieurs du modèle
// Interroger le moteur d'inférence pour la distribution de la preuve
```

Infer.NET définit des modèles à deux branches en utilisant `Variable.If` et `Variable.IfNot`, qui sont l'équivalent de `if-else` en C#. La condition est une variable aléatoire booléenne. La probabilité que la

condition soit vraie détermine la proportion de la branche `If` dans le mélange, et le reste du mélange est la branche `IfNot`.

Pour évaluer la preuve d'un modèle, utilisez `If/IfNot` comme illustré dans l'exemple, ce qui crée un mélange de deux modèles :

- Le modèle que vous souhaitez évaluer, représenté par la branche `If`.
- Un modèle "vide", représenté par la branche `IfNot` manquante.

La variable `Preuve` est la condition qui contrôle les proportions dans le mélange. Son a priori initial est généralement défini sur `Bernoulli(0.5)`. Pour déterminer la distribution réelle de la preuve, observez les données et calculez le postérieur de la variable `Preuve`.

2. Implémentation pour les cyclistes

La nouvelle application utilise la preuve pour évaluer si le modèle `EntrainementCycliste` ou `EntrainementCyclisteMixte` représente le mieux les données d'entraînement de `EntrainementCyclisteMixte`.

a) Classe CyclisteAvecPreuve

La classe `CyclisteAvecPreuve` évalue la preuve pour le modèle `EntrainementCycliste`.

```
public class CyclisteAvecPreuve : EntrainementCycliste
{
    protected Variable<bool> Preuve;

    public override void CreationModeleBayesien()
    {
        Preuve = Variable.Bernoulli(0.5);
        using (Variable.If(Preuve))
        {
            base.CreationModeleBayesien();
        }
    }

    public double CalculPreuve(double[] trainingData)
    {
        double logEvidence;
        DonneesCycliste posteriors = base.CalculePosterieurs(trainingData);
        logEvidence = MoteurInference.Infer<Bernoulli>(Preuve).LogOdds;
        return logEvidence;
    }
}
```

b) Classe CyclisteMixedAvecPreuve

La classe `CyclisteMixedAvecPreuve` évalue la preuve pour le modèle `EntrainementCyclisteMixte`.

```
public class CyclisteMixedAvecPreuve : EntrainementCyclisteMixte
{
    protected Variable<bool> Preuve;

    public override void CreationModeleBayesien()
    {
        Preuve = Variable.Bernoulli(0.5);
        using (Variable.If(Preuve))
        {
            base.CreationModeleBayesien();
        }
    }

    public double CalculPreuve(double[] trainingData)
    {
        double logEvidence;
        DonneesCyclisteMixte posteriors =
base.CalculePosterieurs(trainingData);
        logEvidence = MoteurInference.Infer<Bernoulli>(Preuve).LogOdds;
        return logEvidence;
    }
}
```

c) Calcul de la preuve

On calcule la preuve pour les deux modèles et on affiche les résultats.

```
double[] trainingData = new double[] { 13, 17, 16, 12, 13, 12, 14, 18, 16, 16,
27, 32 };

DonneesCycliste initPriors = new DonneesCycliste(
    Gaussian.FromMeanAndPrecision(15.0, 0.01),
    Gamma.FromShapeAndScale(2.0, 0.5));

CyclisteAvecPreuve cyclistWithEvidence = new CyclisteAvecPreuve();
cyclistWithEvidence.CreationModeleBayesien();
cyclistWithEvidence.DefinirDistributions(initPriors);
double logEvidence = cyclistWithEvidence.CalculPreuve(trainingData);
// Preuves pour le modèle CyclistMixedWithEvidence (CyclingTime3)
```

```
DonneesCyclisteMixte initPriorsMixed = new DonneesCyclisteMixte
{
    DistribMoyenne = new Gaussian[] { new Gaussian(15.0, 100), new
Gaussian(30.0, 100) },
    DistribBruitTraffic = new Gamma[] { new Gamma(2.0, 0.5), new Gamma(2.0,
0.5) },
    DistribMixe = new Dirichlet(1, 1)
};
CyclisteMixedAvecPreuve cyclistMixedWithEvidence = new
CyclisteMixedAvecPreuve();
cyclistMixedWithEvidence.CreationModeleBayesien();
cyclistMixedWithEvidence.DefinirDistributions(initPriorsMixed);
double logEvidenceMixed = cyclistMixedWithEvidence.CalculPreuve(trainingData);
// Affichage des résultats
Console.WriteLine($"Preuve logarithmique pour une gaussienne simple :
{logEvidence}");
Console.WriteLine($"Preuve logarithmique pour un mélange de deux gaussiennes :
{logEvidenceMixed}");
```

Compiling model...done.

Iterating:

.....|.....|.....|.....|.....| 50

Compiling model...done.

Iterating:

.....|.....|.....|.....|.....| 50

Preuve logarithmique pour une gaussienne simple : -45,79730358385471

Preuve logarithmique pour un mélange de deux gaussiennes : -40,98166057283106

Ces valeurs de preuve montrent que le modèle de mélange de deux gaussiennes est supérieur au modèle basé sur une seule gaussienne pour les données d'entraînement fournies.