Curry et Bayes sont sur un bateau

Samy Zarour

A propos de moi

Samy Zarour

- Data-scientiste junior chez ebiznext
- Au quotidien: data engineering Spark/Scala
- Apprenti polyglotte : Scala / R / Python

FABERNOVEL TECHNOLOGIES

tinyclues LUNATECH Teads.tv ebiznext criteo labs

vente-privee GROUP Valraiso lizeo group mfg labs

Remerciements

Ebiznext

Equipe SU Data

Quelques mots sur le titre

Haskell Curry (1900-1982): logicien, mathématicien. Précurseur de la programmation fonctionnelle.

Thomas Bayes (1702-1761): mathématicien, inventeur du théorème de Bayes en probabilités.

Au programme

- Comprendre les probabilités via la programmation fonctionnelle
  - Implémenter la théorie des probabilités
  - Générer des données
  - Introduction au probabilistic programming
- Permettre aux développeurs et data scientist de communiquer
  - Comprendre les notions statistiques et probabilistes avec du code

Cas d'utilisation

- Génération de données
  - Pour tester des applications sans data

- Aide à la décision/prédiction (degré de confiance)
  - Construire des modèles probabilistes
  - Répondre à une question grâce aux probabilités

Génération de données

On souhaite générer des utilisateurs qui viennent sur notre site à une date donnée.

```scala
  val idProb: Prob[String] = Prob.fromGet(() => UUID.randomUUID().toString)

  val ipProb: Prob[String] = {
   val ipPartProb: Prob[Int] = Prob.range(100, 255)
   for {
     i1 <- ipPartProb
     i2 <- ipPartProb
     i3 <- ipPartProb
     i4 <- ipPartProb
   } yield s"$i1.$i2.$i3.$i4"
  }

  val firstTsProb: Prob[Timestamp] = {
   val i = Instant.now()
   Prob.range(0, 24 * 3600).map(s => new Timestamp((i.getEpochSecond + s) *
1000))
  }
```

Génération de visites

```scala
case class User(id: String, ip: String, firstInteraction: Timestamp)

val userProp: Prob[User] = for {
 userId <- idProb
 ip <- ipProb
 ts <- firstTsProb

} yield {
 User(userId, ip, ts)
}
 userProp.samples(10).foreach(println)
```

```
User(93c8e050-6824-4443-af7e-5014f3fab89f,164.165.191.218,2017-10-26
17:25:38.0)
User(644fe06d-5e19-4ee9-8976-016e7cbc98d7,174.171.238.145,2017-10-26
21:36:38.0)
User(d2c3f419-4bc3-4b2f-81ea-f43108c50baf,156.185.133.106,2017-10-26
03:41:05.0)
```

Génération de visites

Les concepts

- Espaces mesurables
- Probabilités
- Variables aléatoires
- Probabilistic programming

Espace mesurable

- Etant donné une collection d'éléments, on cherche à construire un espace sur lequel on définit une mesure.
- Une mesure quantifie la “proportion" d'un élément de l'espace.

```scala
//Définition d'un espace mesurable et d'une mesure
trait Mesurable[T] {
 def mes(a: T): Double
}

object Mesurable {
 //helper
 def fromF[T](f: T => Double): Mesurable[T] = new Mesurable[T] {
   override def mes(a: T): Double = f(a)
 }
}
```

Définition de l'espace mesurable

Espace mesurable

Un espace $(X, \mathcal{X})$ est dit mesurable lorsque X est un ensemble et $\mathcal{X}$ une **tribu** sur X:

- $\mathcal{X}$ est non vide
- $\mathcal{X}$ est stable par complémentaire `def complementaire(a:Set[T]):Set[T]`
- $\mathcal{X}$ est stable par union dénombrable `def union(a:Set[T], b:Set[T]): Set[T]`

Exemple: lancé de pièce

```
sealed trait Piece
case object Pile extends Piece
case object Face extends Piece

def union(a:Set[Piece], b:Set[Piece])= a ++ b

def complementaire(a:Set[Piece]) = Set(Pile,Face).remove(a)
```

Mesure

Soit $(X, \mathcal{X})$ un espace mesurable.

Une mesure est un application $\mu : \mathcal{X} \to \mathbb{R}^+$ vérifiant:

- $\mu(\varnothing) = 0$
- Pour toute famille dénombrable disjointe deux à deux $(E_i)_{i \in I \subseteq A}$,

$$\mu(\cup E_i) = \sum \mu(E_i)$$

```
def mesUnion(s:T*) = s.distinct.map(mes).sum
```

```scala
//Piece = {Pile, Face}
sealed trait Piece
case object Pile extends Piece
case object Face extends Piece

val mes1: Mesurable[Piece] = Mesurable.fromF({
 case Pile => 50
 case Face => 20
})

val mes2: Mesurable[Piece] = Mesurable.fromF({
 case Pile => 0.5
 case Face => 0
})

s"mes1(Face) = ${mes1.mes(Face)}, mes1(Pile) = ${mes1.mes(Pile)}"
//=> mes1(Face) = 20.0, mes1(Pile) = 50.0
s"mes2(Face) = ${mes2.mes(Face)}, mes2(Pile) = ${mes2.mes(Pile)}"
//=> mes2(Face) = 0.0, mes2(Pile) = 0.5
```

Implémentation d'une mesure

Mesure de probabilité

On souhaite avoir une **mesure** qui quantifie la **certitude** d'un événement.

On définit donc une **probabilité**: c'est une mesure à valeur dans **[0,1]**

Vocabulaire:

probabilité ⇔ distribution ⇔ loi ⇔ mesure de probabilité

```scala
trait Prob[T] extends Mesurable[T] {

 self =>

 def get: T

 def prob(pred: T => Boolean): Double

 require(
  prob((a:T) => true) + prob((a:T) => false) > 0.99)

 override def mes(a: T): Double = prob(_ == a)
}
```

Espace probabilisable

Mesure de probabilité

Soit (Ω, A) un espace mesurable.

Une probabilité est une mesure $\mathbb{p} : A \rightarrow [0,1]$ telle que :

- $\mathbb{p}(\Omega) = 1$

Exemple: lancé de dé

$\Omega = \{ 1, 2, 3, 4, 5, 6 \}$ et $A = \mathcal{P}(\Omega)$

$\forall\ i \in \{1,...,6\}, \mathbb{p}(i) = \frac{1}{6}$

$\mathbb{p}(\{1, 2\}) = \mathbb{p}(1) + \mathbb{p}(2) = \frac{1}{6} + \frac{1}{6} = \frac{1}{3}$

$\mathbb{p}(\Omega) = \mathbb{p}(\{1,2,3,4,5,6\}) = 1$

c'est à dire qu'il est certain que le résultat soit dans {1,...,6}

Variable aléatoire

Ce n'est pas le résultat de l'évènement qui nous intéresse, mais plutôt une **fonction de l'évènement**.

Exemple: lancé de dé

- Quelle est la probabilité que le résultat soit pair?
- Quelle est la probabilité d'avoir >3 ?

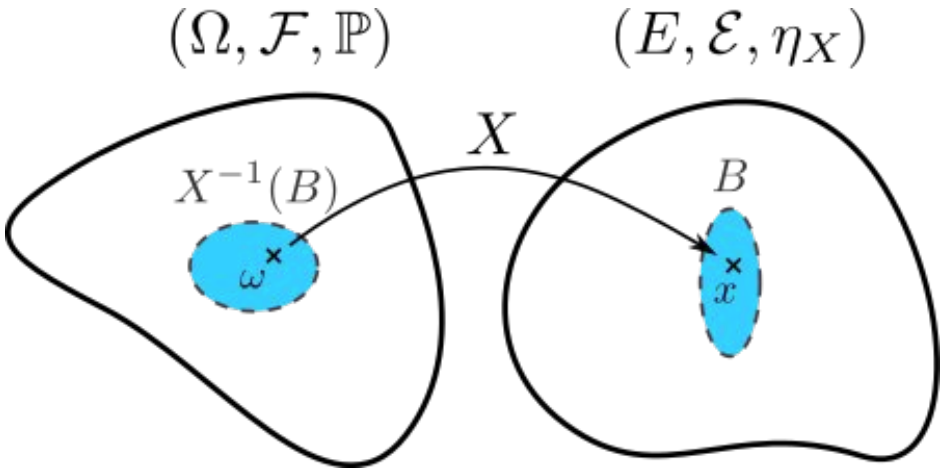
Variable aléatoire

Soit $(\Omega, \mathcal{F})$ et $(E, \boldsymbol{\varepsilon})$ mesurables. Soit $\mathbb{p}$ une probabilité sur $(\Omega, \mathcal{F})$.

Une **variable aléatoire** X est une application $X : \Omega \to E$

On appelle **loi de probabilité de X** la fonction $\mathbb{p}_X : \mathcal{F} \to [0,1]$ définie par:

$\mathbb{p}_X : \omega \to \mathbb{p}(\{\omega, X^{-1}(\omega) \in \mathcal{F}\})$




```
def randomVariable[Omega, E](prob:Prob[Omega],  X: Omega => E): Prob[E] = prob.map(X)
```

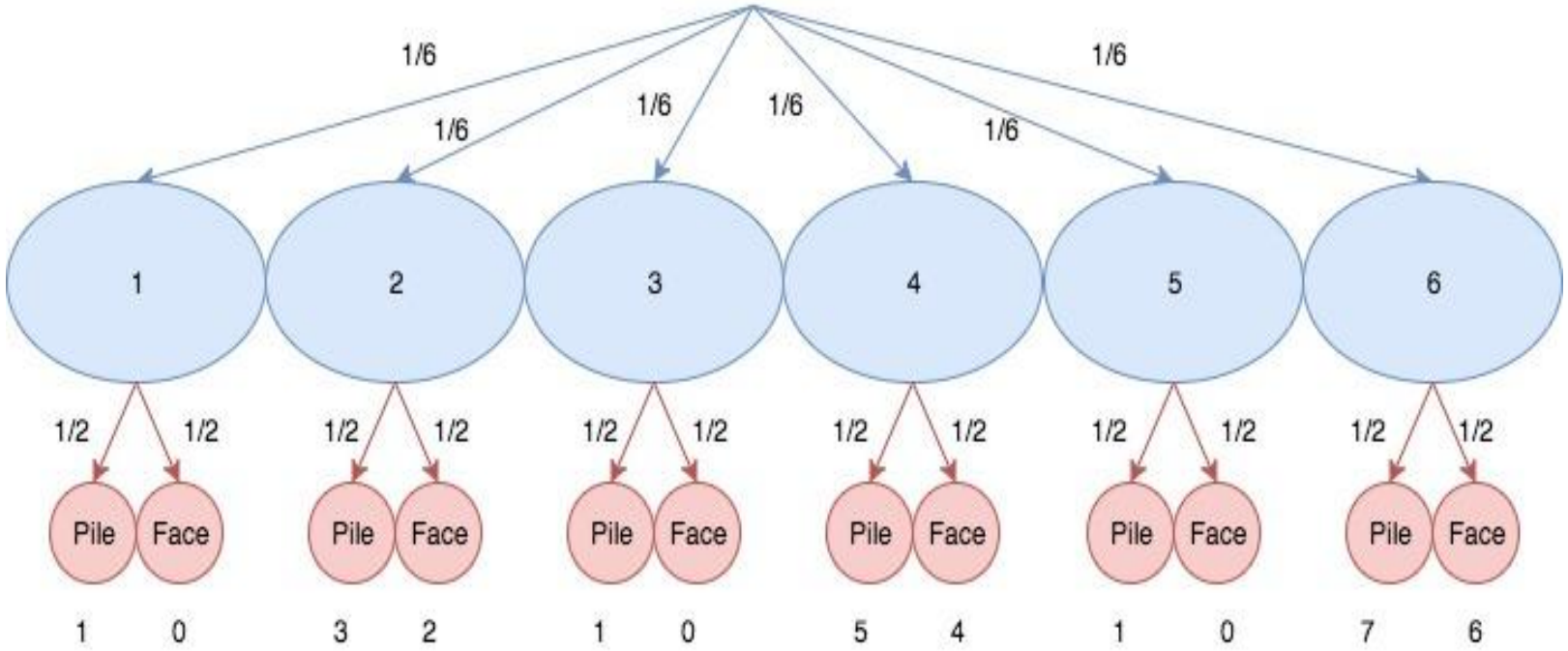
Exemple

Lancé d'un dé et d'une pièce de 1€.

- Si le dé est pair on gagne le gain du dé + celui de la pièce. Sinon on ne gagne que les gains de la pièce
- On s'intéresse aux gains finaux.

On définit une variable aléatoire X qui correspond aux gains.

Probabilités conditionnelles



$$\mathbb{P}(A) = \sum_{i \in I} \mathbb{P}(A|B_i)\mathbb{P}(B_i).$$

P(piece=pile) = P(piece=pile|de=1)P(de=1)+...+P(piece=pile|de=6)P(de=6)

```
def probTotal[Piece, De](probB:Prob[De], probA_kB: De => Prob[Piece]): Prob[Piece]
                = probB.flatMap(probA_kB)
```

```scala
object Prob {

 def choose[T](xs: Seq[T]): Prob[T] = Prob.fromGet(() => xs(Random.nextInt(xs.size)))

 def range(start: Int, end: Int): Prob[Int] = choose(start to end)

 def pure[T](e: T): Prob[T] = fromGet(() => e)

 def fromGet[T](_get: () => T): Prob[T] = {
   new Prob[T] {
     override def get: T = _get()
   }
 }

def generateNormal(mean: Double, std: Double): Prob[Double] = Prob.fromGet(() =>
util.Random.nextGaussian() * std + mean)


 def generateUniform(min: Double, max: Double): Prob[Double] = Prob.fromGet(() =>
util.Random.nextDouble() * (max - min) + min)

}
```

Boîte à outils

```scala
//De = {DeValue(1), ..., DeValue(6)}
sealed trait De
case class DeValue(i:Int) extends De {
 require(i>0 && i<7)
}

//Piece = {Pile, Face}
sealed trait Piece
case object Pile extends Piece
case object Face extends Piece

val probDe: Prob[De] = choose((1 to 6).map(DeValue.apply))

val probPiece: Prob[Piece] = choose(Seq(Pile,Face))
```

Implémentation dé + pièce

```scala
def gainPiece(p:Piece): Int = p match {
 case Pile => 1
 case Face => 0
}

def f(de:De): Prob[Int] = Prob.fromGet(() => de match {
 case DeValue(imp) if imp%2==1 => probPiece.map(gainPiece)
 case DeValue(pair) => probPiece.map(p => gainPiece(p) + pair)
}).flatMap(identity)

val X: Prob[Int] = probDe.flatMap(f)
```

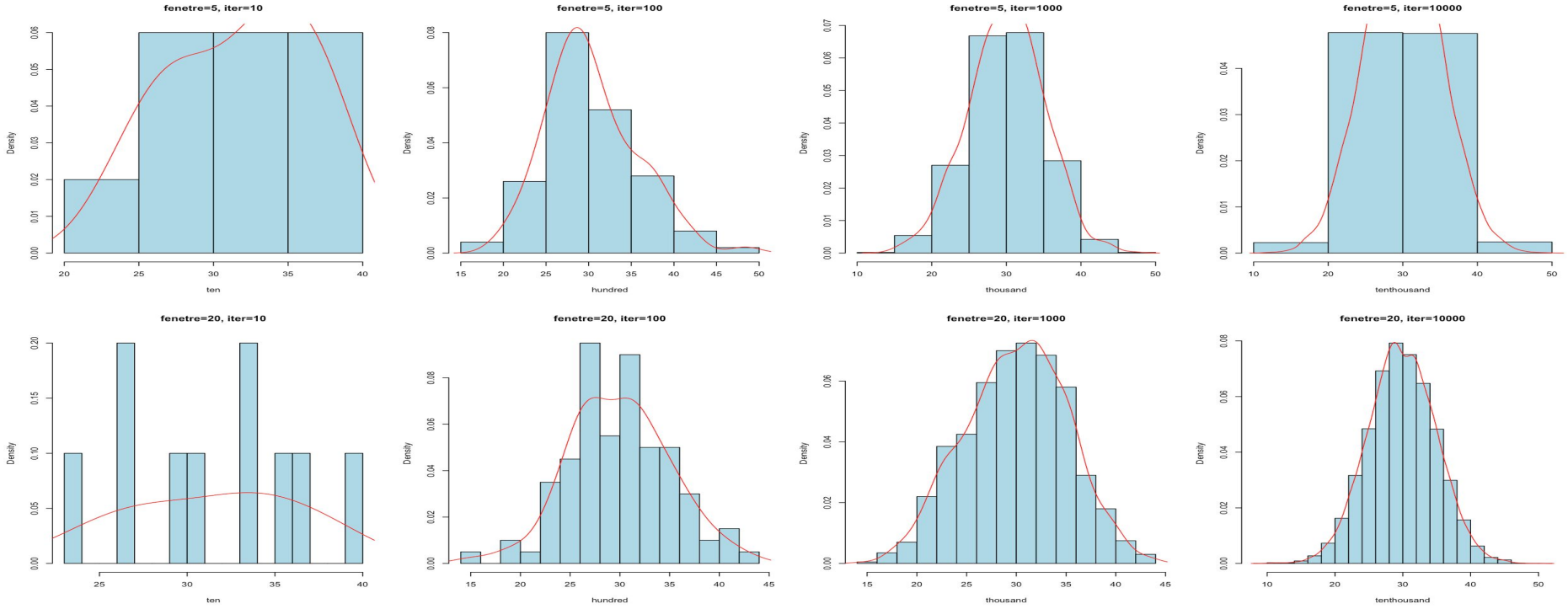
Implémentation dé + pièce

Densité

Représentation de la **répartition** des valeurs possibles d'une loi.

Pour une variable continue, on doit **discrétiser**
(c'est à dire grouper dans des intervalles)

Exemple: Âge des personnes dans la salle

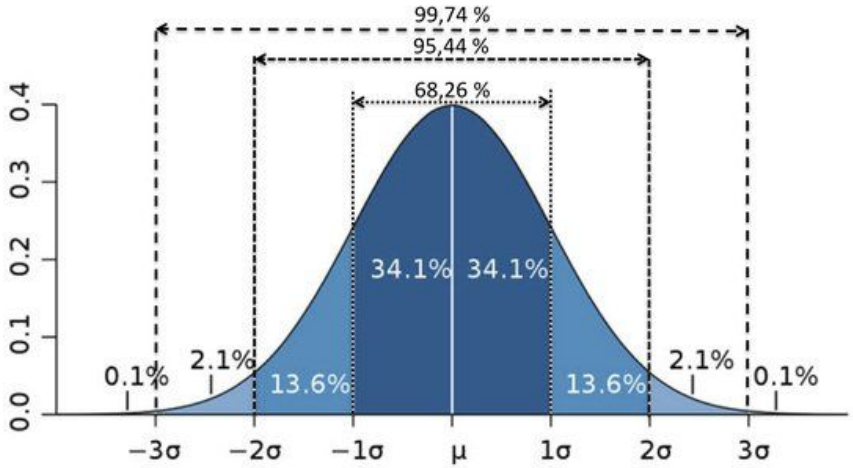


Densité

C'est une fonction `f` **continue**, **positive** dont *l'intégrale* vaut 1.

Exemple: âge

$\sigma = 5, \mu = 30$




```
def samples(n: Int): Seq[T] = Stream.fill(n)(get)

def density(factor: T => T = identity,
            nb: Int = 1000000): Map[T, Double] = {
 samples(nb).groupBy(factor).mapValues(_.size.toDouble / nb)
}
```

Comment définir une loi de probabilité en programmation fonctionnelle ?

On doit pouvoir:

- Générer des valeurs: get + sample
- Définir un **foncteur**: map
- Définir une **monade**: flatMap + pure

```scala
object Prob {
 def fromGet[T](_get: () => T): Prob[T] = {
   new Prob[T] {
     override def get: T = _get()
   }
 }

def pure[T](e: T): Prob[T] = fromGet(() => e)
}
```

Boîte à outils

```scala
trait Prob[T] extends Mesurable[T] {

 self =>
 def get: T
 def samples(n: Int): Seq[T] = Stream.fill(n)(get)
 def flatMap[U](f: T => Prob[U]): Prob[U] = Prob.fromGet(() => f(self.get).get)
 def map[U](f: T => U): Prob[U] = flatMap(f.andThen(Prob.pure)) // pure o f
// ou encore map[U](f:T=>U): Prob[U] = Prob.fromGet(() => f(self.get))
 def density(factor: T => T = identity, nb: Int = 1000000): Map[T, Double] = {
   samples(nb).groupBy(factor).mapValues(_.size.toDouble / nb)
 }
 override def mes(a: T): Double = prob(_ == a)
 def prob(pred: T => Boolean): Double = {
   val d: Map[Boolean, Double] = map(pred).density()
   d(true)
 }
}
```

Implémentation d'une loi de probabilité

```scala
//Piece = {Pile, Face}
sealed trait Piece
case object Pile extends Piece
case object Face extends Piece

val probPiece: Prob[Piece] = choose(Seq(Pile,Face))

val densityPiece: Map[Piece, Double] = probPiece.density()
//=> Map(Face -> 0.498954, Pile -> 0.501046)

def gainPiece(p:Piece): Int = p match {
 case Pile => 1
 case Face => 0
}
val probGain: Prob[Int] = probPiece.map(gainPiece)
val prob1euro: Double = probGain.prob(_==1)
//=> 0.499237

val distribNormal: Map[Double, Double] = Prob.generateNormal(30,
5).density((d:Double) => math.round(d))
distribNormal.filter(x => x._1>=25 && x._1<=35).values.toSeq.sum
//=> 0.7282919999999999
```

Définir des statistiques descriptives

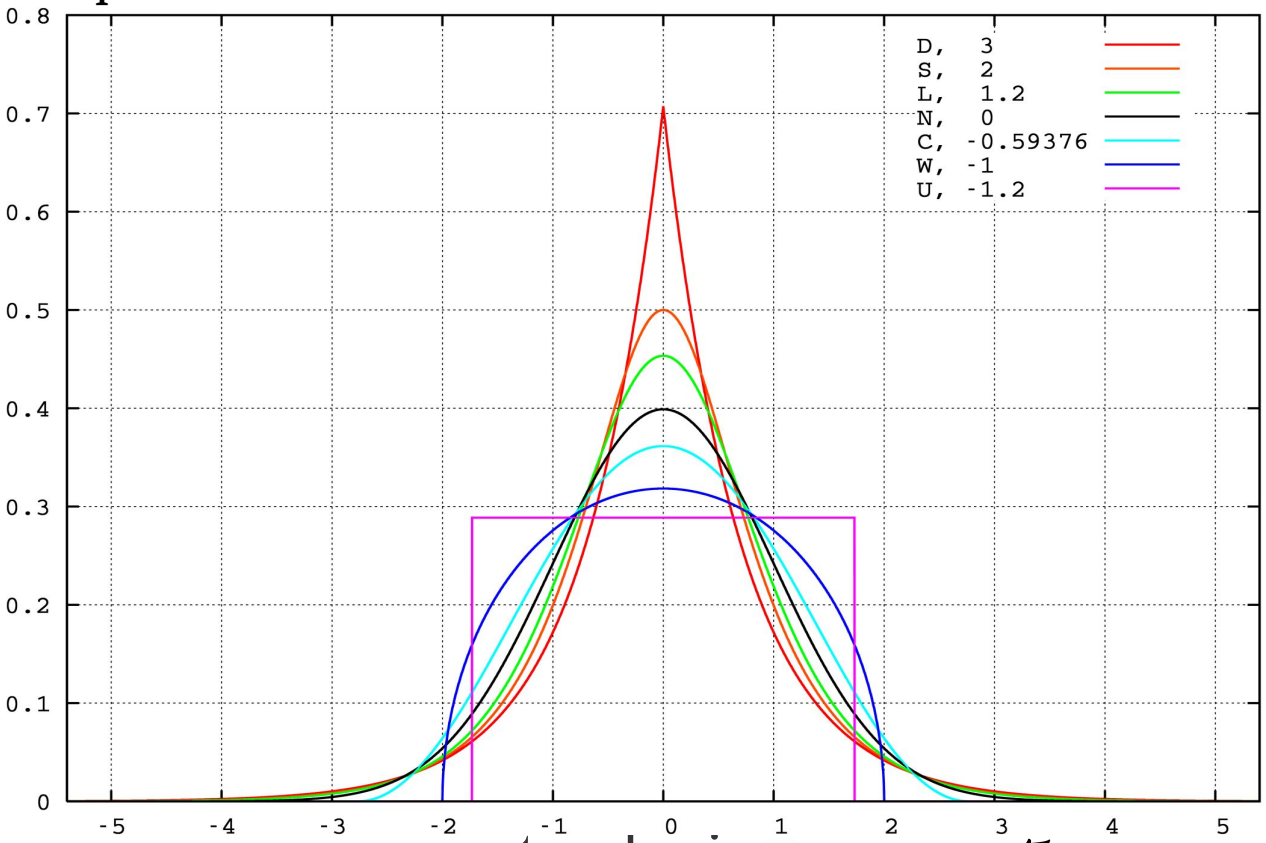
Soit $(x_1, ..., x_n)$ un échantillon issu de X.

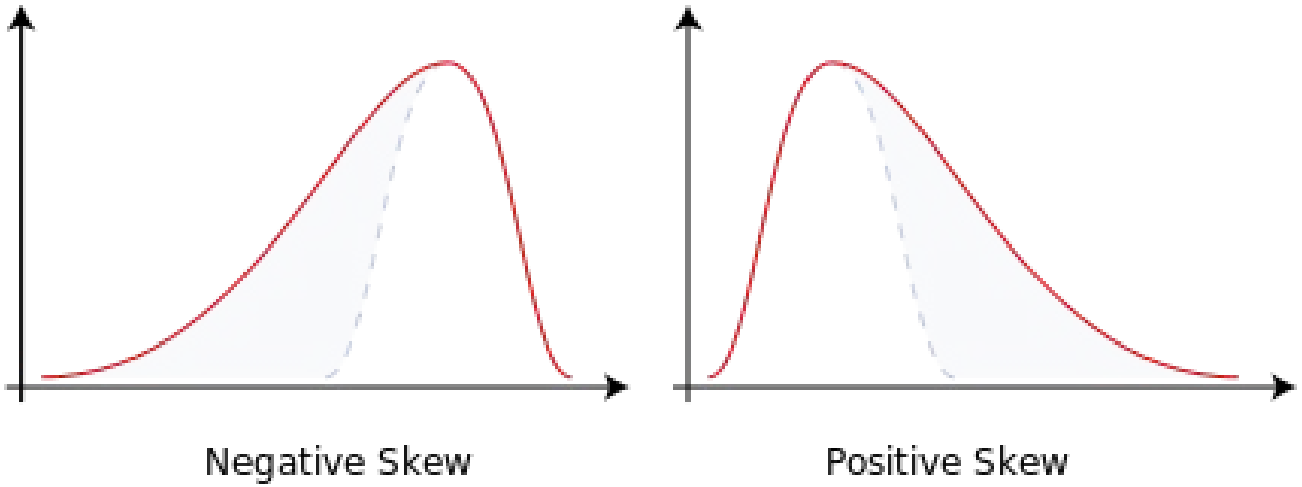
Pour comprendre les données, on utilise des statistiques diverses

- Moyenne
  - $e = (x_1 + ... + x_n) / n$
- Médiane
  - Valeur qui sépare en deux parties égales la série ordonnée
- Variance / écart-type
  - $v = ((x_1 - e)^2 + ... + (x_n - e)^2) / n$
  - $\sigma = sqrt(v)$
- Moment centré réduit d'ordre k:
  - $E[[(X-e)/\sigma]^k]$
  - Kurtosis: ordre
  - Asymétrie: ordre 3

Kurtosis<0 : aplatissement

Kurtosis>0: pic





asymétrie<0: queue à gauche

asymétrie>0: queue à droite

```scala
implicit class DoubleProb(prob: Prob[Double]) {
 val size = 1000001

 def esp: Double = prob.samples(size).sum / size

 def variance: Double = {
   val e = esp
   prob.samples(size).map(_ - e).map(x => x * x).sum / size
 }

 def std: Double = math.sqrt(variance)

 def median: Double = {
   val sortedProb: Seq[Double] = prob.samples(size).sorted
   val medSize: Int = size / 2
   if (size % 2 == 1) sortedProb(medSize + 1) else (sortedProb(medSize + 1) +
sortedProb(medSize)) / 2
}

}
```

Implémentation des statistiques de base

```
def moment(ordre: Int) = {
   val (e,s) = (esp, std)
   prob.samples(size).map{x =>
      math.pow((x - e) / s, ordre)
}.sum / size
 }
def asym: Double = moment(3)

def kurtosis: Double = moment(4)
```

Implémentation des statistiques de base

```
val normalAge = Prob.generateNormal(30,5)

s"mediane = ${normalAge.median}"
//=>mediane = 30.002552912237398
s"esp = ${normalAge.esp}"
//=>esp = 29.99329357110289
s"kurtosis = ${normalAge.kurtosis}"
//=>kurtosis = 3.016174522762754
s"asym = ${normalAge.asym}"
//=>asym = 0.006522548889749082
s"variance = ${normalAge.variance}"
//=>variance = 25.012177563579556
s"std = ${normalAge.std}"
//=>std = 5.000275776151668
```

Statistiques sur une loi normale

Programmation probabiliste

Construire un modèle et répondre à une question

On peut traiter les problèmes dans l'autre sens

- Inversement:
  - A partir des data, répondre à une question
    - Par l'échantillonnage
    - Par la mise à jour des probabilités conditionnelles

- **Probabilistic programming**: Paradigme de programmation dans lequel on manipule des variables aléatoires

Figaro

- Langage de programmation probabiliste
- Ecrit en Scala
- Construction de modèles via une API riche
- Algorithmes d'inférence

```
val piece: Element[Piece] = Flip(0.5).map(b => if (b) Pile else Face)

val de: Element[Int] = Select(1.0 -> 1, 1.0 -> 2, 1.0 -> 3, 1.0 -> 4, 1.0 -> 5, 1.0 -> 6)

val gain = for {
 p <- piece
 d <- de
} yield {
 (p, d % 2 == 0) match {
   case (Pile, true) => d +1
   case (Face, true) => d
   case (Pile, _) => 1
   case (Face, _) => 0
 }
}

val alg = BeliefPropagation(10, gain)
alg.start()
val density: Stream[(Double, Int)] = alg.computeDistribution(gain)
val prob: Double = alg.probability(gain, (g:Int) => g > 5)
alg.kill()
println(density.toList)
//List((0.08333333333333333,3), (0.25000000000000006,1), (0.08333333333333333,4),
(0.08333333333333333,7),(0.08333333333333333,5), (0.08333333333333333,2), (0.25000000000000006,0),
(0.08333333333333333,6))
println(s"P(X>5) = $prob")
//P(X>5) = 0.16666666666666666
```

Exemple avec Figaro

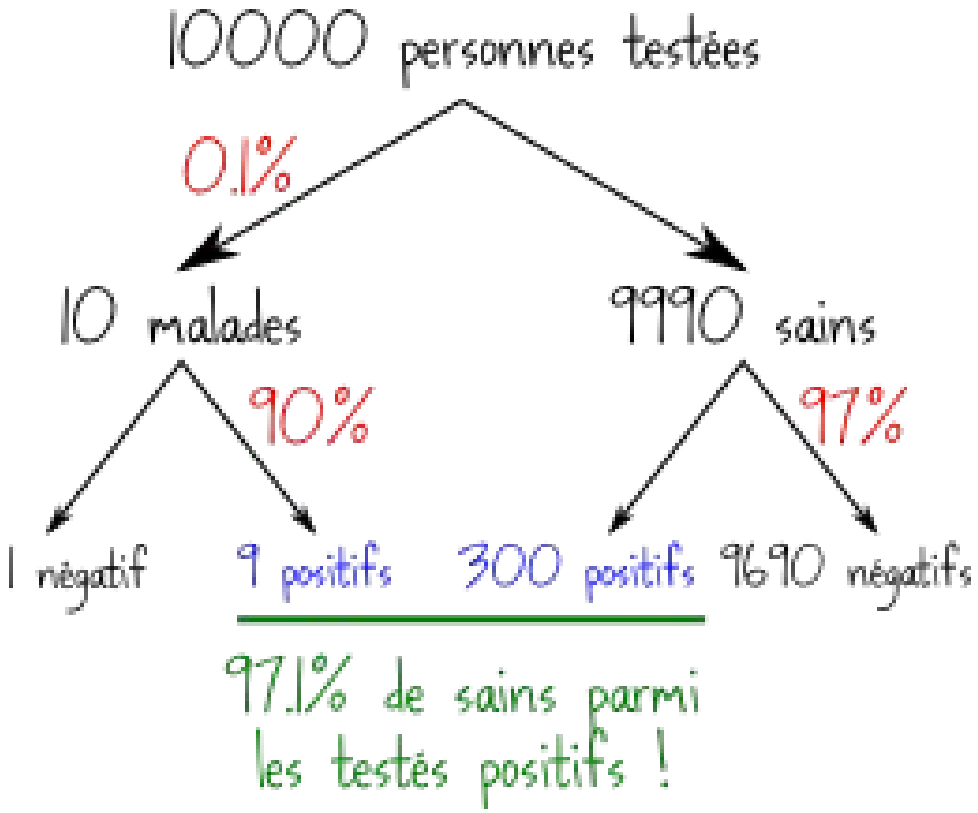
FABERNOVEL TECHNOLOGIES

tinyclues LUNATECH Teads.tv ebiznext criteo labs

vente-privee GROUP Valraiso lizeo group mfg labs

Retour sur les probabilités conditionnelles

Quelle est la probabilité d'être malade sachant qu'on est positif ?



Théorème de Bayes

Soit A et B deux événements. On note A^C le complémentaire de l'événement A.

$$\mathbb{p}(A \mid B) = \mathbb{p}(B \mid A)\mathbb{p}(A)/\mathbb{p}(B)$$

Probabilités totales:

$$\mathbb{p}(B) = \mathbb{p}(B \mid A)\mathbb{p}(A) + \mathbb{p}(B \mid A^C)\mathbb{p}(A^C)$$

$$\mathbb{p}(A \mid B) = \mathbb{p}(B \mid A)\mathbb{p}(A)/[\mathbb{p}(B \mid A)\mathbb{p}(A)+\mathbb{p}(B \mid A^C)\mathbb{p}(A^C)]$$

Théorème de Bayes: application

M = "le patient est malade"

P = "le patient est positif"

$\mathbb{p}(M \mid P) = \mathbb{p}(P \mid M)\mathbb{p}(M)/\mathbb{p}(P)$

$\mathbb{p}(P \mid M) = 0.9$

$\mathbb{p}(M) = 0.001$

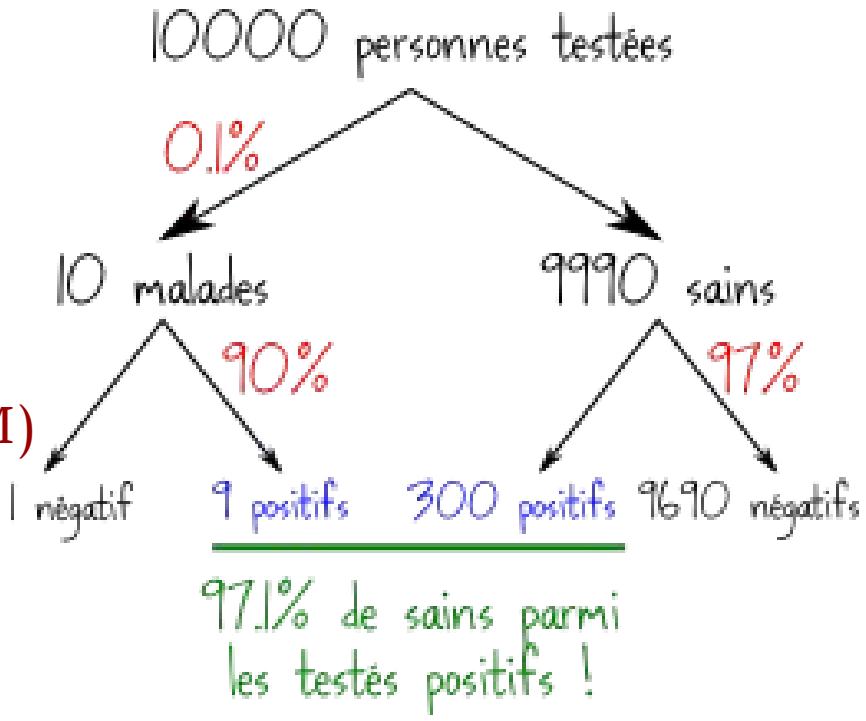
$\mathbb{p}(P) = \mathbb{p}(P \mid M^C)\mathbb{p}(M^C) + \mathbb{p}(P \mid M)\mathbb{p}(M)$

$= 0.03*0.999 + 0.9*0.001$

$= 0.03087$

D'où: $\mathbb{p}(M \mid P) = 0.9*0.001 / 0.03087$

$= 0.02915$



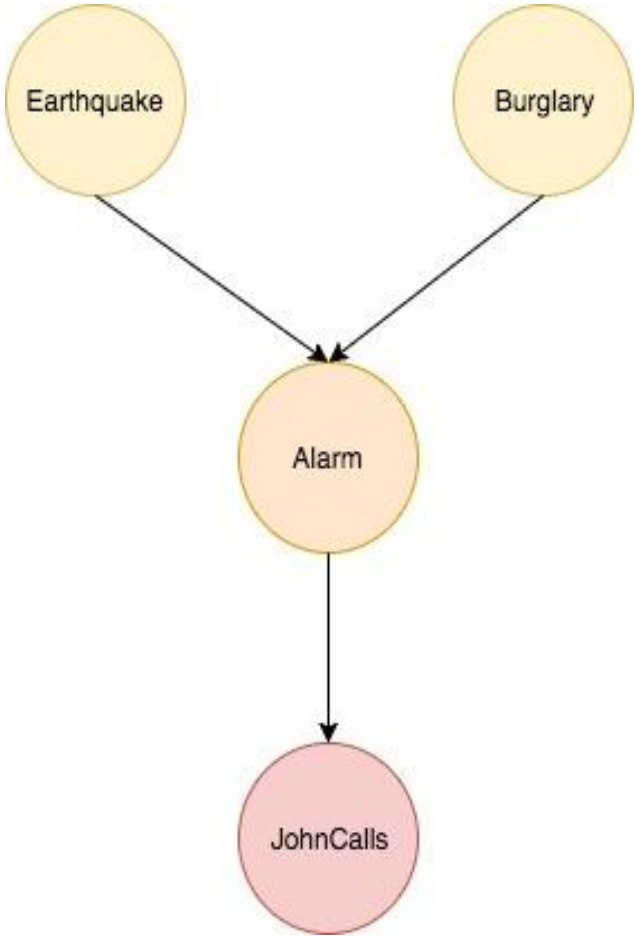
2.9% de chance d'être malade sachant qu'on est positif au test !

```
  private val burglary = Flip(0.01)

  private val earthquake = Flip(0.0001)

  private val alarm = CPD(burglary, earthquake,
    (false, false) -> Flip(0.001),
    (false, true) -> Flip(0.1),
    (true, false) -> Flip(0.9),
    (true, true) -> Flip(0.99))

  private val johnCalls = CPD(alarm,
    false -> Flip(0.01),
    true -> Flip(0.7))
}
```



Exemple Figaro

```scala
def main(args: Array[String]) {
 val alg1 = VariableElimination(burglary, earthquake)
 alg1.start()
 println("Probability of burglary: " + alg1.probability(burglary, true))
 //=> Probability of burglary: 0.01

 alg1.kill

 johnCalls.observe(true)
 val alg = VariableElimination(burglary, earthquake)
 alg.start()
 println("Probability of burglary | john calls: " + alg.probability(burglary, true))
 //=> Probability of burglary | john calls: 0.3733781172643905
 alg.kill

 println(MetropolisHastings.probability(burglary, true))
 //=>0.009514
 johnCalls.observe(true)
 println(MetropolisHastings.probability(burglary, true))
 //=>0.3644864673490166
}
```

Exemple Figaro

FABERNOVEL TECHNOLOGIES

tinyclues LUNATECH Teads.tv ebiznext criteo labs

vente-privee GROUP Valraiso lizeo group mfg labs

Conclusion

- Avantages
  - L'algorithme de prédiction est explicable au métier
  - Règles métiers
    - Connaissance a priori des paramètres
    - Mettre des contraintes sur les variables aléatoires
  - Complément au machine learning
  - Prédiction avec certitude
  - Pas besoin de beaucoup de données pour prédire

- Inconvénients
  - Vite complexe
    - Beaucoup de variables
    - (Etude approfondie des données)
  - Beaucoup de feature engineering

Merci de votre attention

Slides + code:
github.com/zaroursamy/ScalaIO2017

Questions/échange:
samy.zarour8430@gmail.com

Lecture:
Practical Probabilistic Programming,
Avi Pfeffer