Codage universel : la méthode arithmétique

1 La théorie du codage source de Shannon

1.1 Codes préfixes et lois de codage

Que ce soit pour les stocker ou pour les transmettre plus rapidement, il est souvent souhaitable de pouvoir compresser ses données, c'est-à-dire leur associer un fichier binaire de taille aussi petite que possible - c'est le service que proposent par exemple les programmes `gzip, compress, WinZip`. Nous nous intéressons ici au problème de la compression sans perte : nous souhaitons pouvoir retrouver exactement le message initial à la décompression ; c'est cela qui est souhaitable pour traiter les textes et les données numériques par exemple, mais pas pour les images ou le son pour lesquels l'utilisateur est prêt à subir une petite perte de qualité pour un gros gain de place. L'exemple utilisé ci-dessous pour les expérience est un texte contenant le début des fables de La Fontaine. Nous aurions également pu choisir une séquence ADN, une partition musicale, les relevés de température d'un poste météo, etc.

Pour tous ces exemples, le problème peut être modélisé de la façon suivante : il existe un *alphabet* $\mathcal{A}$, c'est-à-dire un ensemble fini, dont les éléments seront appelés *symboles*. On dispose d'un *message* x_1^n de taille $n \in \mathbb{N}$, c'est-à-dire d'un n-uplet de symboles $(x_1, \dots, x_n)$, auquel on souhaite associer un *mot de code* $\phi_n(x_1^n)$ qui soit une suite finie de bits, c'est-à-dire un élément de $\{0,1\}^* = \cup_{k\in\mathbb{N}}\{0,1\}^k$. La fonction $\phi_n : A^n \to \{0,1\}^*$ est appelée *code* et, d'après ce qui précède, on souhaite qu'elle soit injective. On souhaite en fait un peu plus : on veut que ϕ_n soit *uniquement décodable*, c'est-à-dire qu'il soit possible de retrouver les messages à partir de la concaténation de leurs mots de codes :

$$\phi_n(m_1).\phi_n(m_2)\dots\phi_n(m_k) = \phi_n(m'_1).\phi_n(m'_2)\dots\phi_n(m'_l) \implies k = l \text{ et } \forall j \in \{1,\dots,k\}, m_j = m'_j.$$

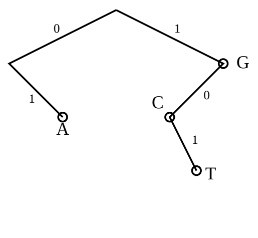
En fait, on peut imposer sans rien y perdre une condition plus forte : que le code ϕ_n soit *préfixe*, c'est-à-dire qu'aucun mot de code ne soit le début d'un autre. Il est clair qu'un code préfixe est uniquement décodable.

Si l'on considère l'arbre des mots de code de ϕ_n, cela revient à dire que tous les mots de codes y sont des feuilles (et aucun n'est un nœud interne).

Exemple 1 *Soit $\mathcal{A} = \{A, C, T, G\}$ et $n = 1$.*

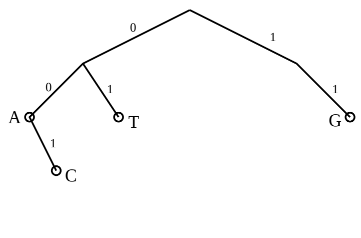
Le code $\phi^1 : \begin{cases} A & \mapsto & 01 \\ C & \mapsto & 10 \\ T & \mapsto & 101 \\ G & \mapsto & 1 \end{cases}$

n'est pas uniquement décodable.



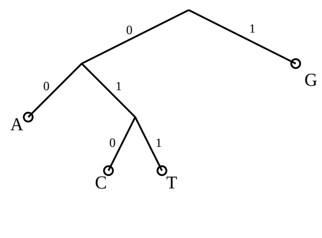
Le code $\phi^2 : \begin{cases} A & \mapsto & 00 \\ C & \mapsto & 001 \\ T & \mapsto & 01 \\ G & \mapsto & 11 \end{cases}$

est uniquement décodable, mais pas préfixe.



Le code $\phi^3 : \begin{cases} A & \mapsto & 00 \\ C & \mapsto & 010 \\ T & \mapsto & 011 \\ G & \mapsto & 1 \end{cases}$

est préfixe.



On vérifie facilement la propriété suivante :

Proposition 1 (Inégalité de Kraft) *Pour tout code préfixe ϕ_n, on a*

$$\sum_{x_1^n \in \mathcal{A}^n} 2^{-|\phi_n(x_1^n)|} \leqslant 1.$$

Cette propriété admet la réciproque suivante :

Proposition 2 *Soit $l : A^n \to \mathbb{N}$ telle que $\sum_{x_1^n \in \mathcal{A}^n} 2^{-l(x_1^n)} \leqslant 1$. Il existe un code préfixe ϕ_n tel que :*

$$\forall x_1^n \in \mathcal{A}^n, |\phi_n(x_1^n)| = l(x_1^n).$$

Notons que l'inégalité de Kraft peut également être démontrée pour tous les codes uniquement décodables. Cela explique pourquoi on ne perd rien en ne s'intéressant, parmi les codes uniquement décodables, qu'aux codes préfixes.

L'inégalité de Kraft montre qu'en posant

$$q_{\phi_n}(\cdot) = 2^{-|\phi_n(\cdot)|}$$

on définit une sous-probabilité q_{ϕ_n} que l'on appellera dans la suite *loi de codage* associée au code ϕ. Inversement, la réciproque permet de voir qu'à toute sous-probabilité q_n on peut associer un code ϕ_{q_n} vérifiant

$$|\phi_n(\cdot)| = \lceil -\log q_n(\cdot) \rceil\,,$$

où $\lceil a \rceil$ désigne le plus petit entier supérieur ou égal au réel a et log le logarithme à base 2. Ainsi, il existe une correspondance presque parfaite entre codes et probabilités de codage, et nous pourrons dans la suite l'exploiter en utilisant la théorie des probabilités pour étudier certains problèmes de codage.

1.2 La modélisation de Shannon

Bien entendu, il est impossible de compresser tous les messages. Il faut restreindre nos ambitions à essayer de construire des codes qui compressent beaucoup les messages fréquents, quitte à ce qu'ils ne soient éventuellement pas très efficaces pour les messages que l'on n'attend pas et que l'on voit peu. Dans son article fondateur de la théorie de l'information, Claude E. Shannon propose la modélisation suivante : le message x_1^n est issus d'une *source* P, c'est-à-dire est la réalisation d'une variable aléatoire X_1^n dont la loi P^n est la loi marginale du processus stationnaire ergodique de loi P. Le but est alors de minimiser la longueur de code *moyenne* sous cette loi :

$$\mathbb{E}_P[X_1^n] = \sum_{x_1^n \in \mathcal{A}^n} p(x_1^n)\, |\phi_n(x_1^n)|\,,$$

où l'on a noté $p(x_1^n) = P(X_1^n = x_1^n)$.

On est ainsi ramené à un petit problème d'optimisation : pour trouver le meilleur code préfixe ϕ_n pour la source P, on pose $l(\cdot) = |\phi_n(\cdot)|$. On cherche à minimiser $\sum_{x_1^n \in \mathcal{A}^n} p(x_1^n) l(x_1^n)$ sous la contrainte de Kraft $\sum_{x_1^n \in \mathcal{A}^n} 2^{-l(x)} \leqslant 1$. On obtient la définition et le théorème suivants :

Définition 1 *Soit Z une variable aléatoire à valeur dans l'ensemble fini $\mathcal{X}$. L'*entropie *de Z est définie par :*

$$H(Z) \triangleq \sum_{z \in \mathcal{X}} P(Z = z) \log \frac{1}{P(Z = z)}.$$

Théorème 1 (Shannon '48) *L'espérance sous P de la longueur de n'importe quel code sur les messages de taille n est supérieure à l'entropie de X_1^n :*

$$\mathbb{E}_P\left[|\phi_n(X_1^n)|\right] \geqslant H(X_1^n) = \sum_{x_1^n \in \mathcal{A}^n} p(x_1^n) \log \frac{1}{p(x_1^n)}.$$

De plus, il existe un code qui atteint cette borne à moins de 1 bit près.

L'entropie H est une mesure d'information. Le théorème précédent permet de l'interpréter comme le nombre minimal de bits nécessaires au codage d'une réalisation en moyenne.

Proposition 3 *Soit Z une variable aléatoire à valeur dans l'ensemble fini $\mathcal{X}$. L'entropie $H(Z)$ vérifie les propriétés suivantes :*

- $0 \leqslant H(Z) \leqslant \log|\mathcal{X}|$.
- *$H(Z)$ est nulle si et seulement si Z est déterministe, et prend sa valeur maximale $\log|\mathcal{X}|$ si et seulement si Z suit la loi uniforme sur $\mathcal{X}$.*

– *Soit Z' une variable aléatoire sur un ensemble fini $\mathcal{X}'$. L'entropie du couple (Z, Z') vérifie :*

$$H(Z, Z') \leqslant H(Z) + H(Z'),$$

avec égalité si et seulement si Z et Z' sont indépendantes.

Ces propriétés permettent de montrer que la suite des entropies normalisées converge quand la taille des messages augmente.

Proposition 4 *La suite de terme général $\frac{1}{n}H\left(X_1^n\right)$ admet une limite $H(X)$ appelée* entropie de la source *P. De plus, pour tout symbole $a \in \mathcal{A}$ la suite de variables aléatoires $\left(\mathbb{P}\left(X_0 = a | X_{-k}^{-1}\right)\right)_k$ converge presque-sûrement et dans L^1 : en notant $P\left(X_0 = a|X_{-\infty}^{-1}\right)$ sa limite on a*

$$H(X) = \mathbb{E}\left[-\log P\left(X_0 | X_{-\infty}^{-1}\right)\right].$$

La première partie de cette proposition résulte du fait que la fonction $n \mapsto H\left(X_1^n\right)$ est sous-additive. Pour démontrer la deuxième partie, on peut remarquer que la suite de terme général $\mathbb{P}\left(X_0 = a|X_{-k}^{-1}\right)$ est une martingale bornée relativement à la filtration $\left(\sigma\left(X_{-k}^{-1}\right)\right)_k$, souligner que la fonction $x \mapsto x \log \frac{1}{x}$ est bornée sur $[0, 1]$, et utiliser les moyennes de Césaro.

Cette proposition permet d'interpréter $H(X)$ comme le nombre minimal de bits par symbol nécessaire en moyenne pour coder les messages issus de P, ou de manière équivalente pour coder un symbole en connaissant tous les précédents. On vérifie facilement que si P est un processus sans mémoire (c'est-à-dire si les X_k sont indépendants identiquement distribués) alors $H(X) = H(X_1)$. De façon similaire, si X est une chaîne de Markov alors $H(X) = \mathbb{E}\left[-\log P\left(X_0|X_{-1}\right)\right]$.

Remarquons pour terminer que si l'on utilise une probabilité de codage q différente de P^n, les mots de codes obtenus sont en moyenne plus long. On appelle *redondance de la loi de codage q_n pour la source P* la différence entre l'espérance de la longueur d'un mot de code avec q_n et la n-entropie $H(X_1^n)$.

Définition 2 *Soit P et Q deux lois de probabilités sur le même ensemble $\mathcal{X}$, et Z une variable aléatoire de loi P. L'*information de Küllback-Leibler *entre la loi P et la loi Q est définie par :*

$$D(P, Q) = \begin{cases} \mathbb{E}\left[\log \frac{P(Z)}{Q(Z)}\right] & \text{si } P \ll Q, \\ \infty & \text{sinon.} \end{cases}$$

Il apparaît que la redondance de la loi de codage q_n pour la source P est égale à

$$D(P^n, q_n) = \mathbb{E}_P\left[\log \frac{P^n(X_1^n)}{q_n(X_1^n)}\right].$$

La proposition suivante permet d'interpréter l'information de Küllback-Leibler comme une “pseudo-distance” :

Proposition 5 *Soit P et Q deux lois de probabilités sur le même ensemble $\mathcal{X}$. On a*

$$D(P, Q) \geqslant 0$$

avec égalité si et seulement si $P = Q$.

Nous allons voir dans la section suivante comment construire en pratique des codes efficaces.

2 Le codage arithmétique

2.1 Principe

Le codage arithmétique a été mis au point au cours des années 1970 et 1980. Il ne s'agit pas à proprement parler d'un code, mais d'un algorithme permettant de construire un code effectif φ^q à partir de n'importe quelle loi de codage q_n qui lui est transmise comme paramètre. Il prend en argument un message x_1^n, et lui associe un mot de code $\varphi_n^q\left(x_1^n\right)$ de taille $\lceil q_n(x_1^n)\rceil + 1$. Son grand avantage est justement d'isoler l'aspect algorithmique du codage de la modélisation de la source : grâce au codage arithmétique une loi de codage donne directement un code implémentable.

Soit k un entier positif. Choisissons pour simplifier $\mathcal{A} = \{1, \dots, m\}$, et notons $\prec$ l'ordre lexicographique sur $\mathcal{A}^k$: pour $x_1^k, y_1^k \in \mathcal{A}^k$ on a $y_1^k \prec x_1^k$ si et seulement s'il existe $l \leqslant k$ tel que $y_1^{l-1} = x_1^{l-1}$ et $y_l < x_l$.

L'idée est la suivante : à chaque message x_1^n on associe le sous intervalle de $[0,1[$ noté $I(x_1^n) = [\lambda(x_1^n), \lambda(x_1^n) + q(x_1^n)[$ avec

$$\lambda(x_1^n) = \sum_{y_1^n \prec x_1^n} q_n(y_1^n).$$

Le mot de code $c = \varphi_n^q(x_1^n)$ est formé des $|c| = \left\lceil \log \frac{1}{q(x_1^n)} \right\rceil + 1$ premiers bits de l'écriture binaire du milieu $m(x_1^n)$ de l'intervalle $I(x_1^n)$. On a donc $m(x_1^n) \in \left[\overline{0.c}, \overline{0.c} + 2^{-|c|}\right[$, d'où l'on déduit facilement

$$\lambda(x_1^n) = m(x_1^n) - \frac{q(x_1^n)}{2} \leqslant m - 2^{-\left\lceil \log \frac{1}{q(x_1^n)} \right\rceil - 1} < \overline{0.c} < \overline{0.c} + 2^{-|c|}$$
$$\leqslant m(x_1^n) + 2^{-\left\lceil \log \frac{1}{q(x_1^n)} \right\rceil - 1} \leqslant m + \frac{q(x_1^n)}{2} = \lambda(x_1^n) + q(x_1^n)$$

Il ne reste qu'à remarquer que

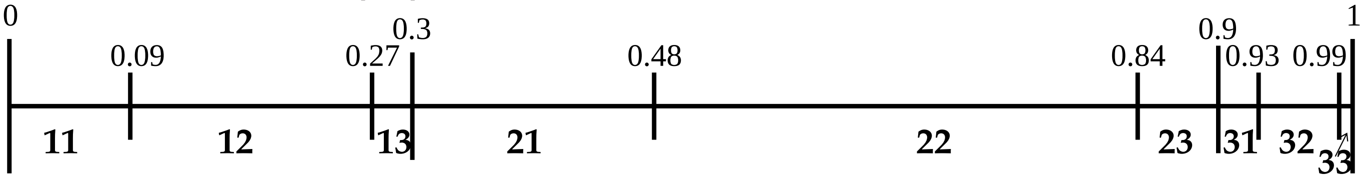
$$x_1^n \neq y_1^n \implies I(x_1^n) \cap I(y_1^n) = \emptyset$$

pour conclure :

Proposition 6 *Pour tout entier positif n et toute loi de codage q, le code φ_n^q est préfixe.*

Il est par conséquent uniquement décodable.

Exemple 2 *Pour $\mathcal{A} = \{1,2,3\}$, $n = 2$ et la loi de codage $q_2 = q_1 \otimes q_1$, où $q_1(1) = 0.3$, $q_1(2) = 0.6$ et $q_1(3) = 0.1$, la décomposition de l'intervalle $[0,1[$ est la suivante :*



Exemples :

- • $\mathbf{I(00) = [0, 0.09[}$:

$$m(00) = 0.045 = \overline{0.\mathbf{00001}01110\dots}$$

Comme $\left\lceil \log \frac{1}{0.09} \right\rceil + 1 = 5$, le mot de code $\varphi_2^q(00)$ est 00001.

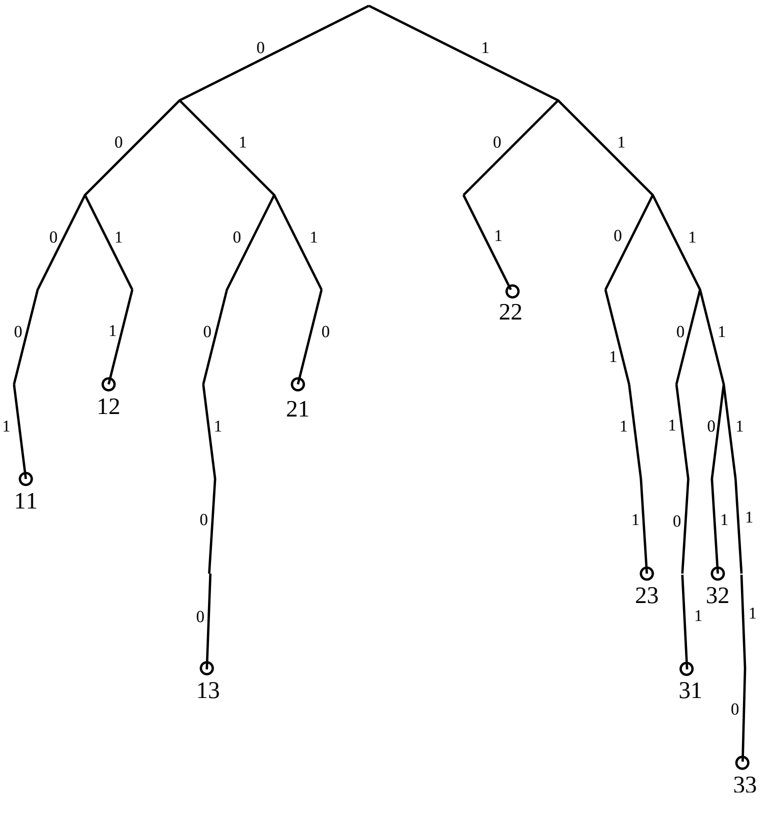
- • $\mathbf{I(22) = [0.48, 0.84[}$:

$$m(22) = 0.66 = \overline{0.\mathbf{101}0100011\dots}$$

Comme $\left\lceil \log \frac{1}{0.36} \right\rceil + 1 = 3$, le mot de code $\varphi_2^q(22)$ est 100.

On arrive ainsi au code φ_2^q dont l'arbre est représenté ci-contre.

Cet exemple montre bien que chaque mot de code peut être trop long d'un à deux bits. Si $q_2 = P^2$, alors $\mathbb{E}_P[\varphi_2^q(X_1^2)] = \sum_{x_1^2} q_2(x_1^2) \left|\varphi_2^q(x_1^2)\right| \approx 4.2$ alors que $H(X_1^2) = \sum_{x_1^2} q_2(x_1^2) \log \frac{1}{q_2(x_1^2)} \approx 2.6$.



2.2 Calcul itératif

Le grand avantage du code arithmétique est qu'il peut se faire par un calcul récursif. Pour $k \leqslant n$ et $x_1^k \in \mathcal{A}^k$, notons en effet $q_n(x_1^k) = \sum_{x_{k+1}^n \in \mathcal{A}^{n-k}} q_n\left(x_1^n\right)$, et convenons que pour le mot vide $\epsilon \in \mathcal{A}^0$ on note $q(\epsilon) = 1$. Pour $a \in \mathcal{A}$, notons également $q_n(a|x_1^k) = \frac{q_n(x_1^k a)}{q_n(x_1^k)}$. On remarque que

$$\lambda\left(x_1^k\right) = \lambda(x_1^{k-1}) + \left(\sum_{a<x_k} q_n(a|x_1^{k-1})\right) q_n\left(x_1^{k-1}\right).$$

On peut donc effectuer le codage de façon *itérative*, en traitant les symboles x_k au fur et à mesure. Pour le codage de x_k, il suffit de connaître la loi conditionnelle $q_n(\cdot|x_1^{k-1})$.

De même, il existe une façon très simple et itérative d'effectuer le décodage du mot de code c : soit τ le nombre réel de l'intervalle $[0,1[$ dont l'écriture en base 2 est $\overline{0.c} = \sum_{j=1}^{|c|} c_j 2^{-j}$. On commence par découper l'intervalle $[0,1[$ selon la loi $q(\cdot)$, et on regarde dans quel intervalle tombe τ : le symbole y_1 associé est le premier du message. On calcule alors $I(y_1) = [\lambda(y_1), \lambda(y_1)+q(y_1)[$, comme le codeur, et on découpe celui-ci selon la loi $q(\cdot|y_1)$. En regardant où tombe τ, on en déduit y_2, etc... En pseudo-code, codage et décodage arithmétiques sont donc réalisées par les fonctions suivantes :

function c=code(x)
λ <- 0, κ <- 1
for k <- 1 **to** n
  λ <- λ + $\sum_{a<x_k} q(a|x_1^{k-1}) * \kappa$
  κ <- κ * $q(x_k|x_1^{k-1})$
end
c = les $\lceil -\log \rceil + 1$ premiers bits de $l + h/2$.

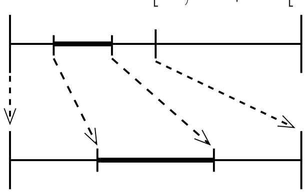
function y=decode(n, c)
λ <- 0, κ <- 1, τ <- $\overline{0.c}$
for k <- 1 **to** n
  y(k) <- le symbole associé à τ dans $[\lambda, \lambda+\kappa[$
  λ <- λ + $\sum_{a<x_k} q(a|x_1^{k-1}) * \kappa$
  κ <- κ * $q(x_k|x_1^{k-1})$
end

Malheureusement, ce pseudo-code ne prend pas en compte le fait que les calculs sont effectués par l'ordinateur avec une précision limitée. Dans le codage, au bout de quelques symboles, κ cesse d'être grand par rapport à la précision des calculs, et la procédure ne fonctionne plus. Lors du décodage, il est clair qu'on ne peut travailler numériquement avec $\tau = \overline{0.c}$ lorsque c comprend plusieurs milliers de symboles.

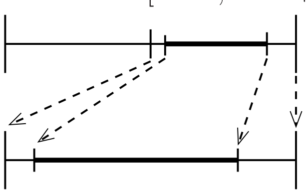
2.3 Implémentation pratique

Pour éviter cela, on va constuire le mot de code c au fur et à mesure de la lecture des symboles x_i, en travaillant avec l'intervalle $[l, l+h[$ qui est déduit de $I(x_1^n) = [\lambda, \lambda+\kappa[$ par l'application des règles de normalisation suivantes :

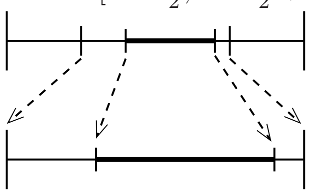
Règle 1 si l'intervalle $[l, l+h[$ est inclus dans $\left[0, \frac{1}{2}\right[$, alors l'écriture binaire de tous ses points débute par un 0 : on peut rajouter ce symbole au code, et travailler dans l'intervalle $[2l, 2l+2h[$.



Règle 2 si un intervalle $[l, l+h[$ est inclus dans $\left[\frac{1}{2}, 1\right[$, alors l'écriture binaire de tous ses points débute par un 1 : on peut rajouter ce symbole au code, et travailler dans l'intervalle $[2l-1, 2l-1+2h[$.



Règle 3 si un intervalle $[l, l+h[$ est inclus dans $\left[\frac{1}{4}, \frac{3}{4}\right[$, alors l'écriture binaire de tous ses points débute par deux symboles différents : on peut retenir cela, et travailler dans $[2l - \frac{1}{2}, 2l - \frac{1}{2} + 2h[$.



En appliquant ces règles autant de fois que possible, on se ramène toujours à un intervalle $[l, l+h[$ de largeur supérieure à $\frac{1}{4}$: on évite ainsi les problèmes dûs à la précision limitée des calculs par ordinateur.

On trouve dans le code **Matlab** 1 une sous-fonction `codearith`, qui implémente le codage arithmétique, précédée par quelques lignes qui montrent comment l'utiliser. Voici quelques remarques pour le comprendre :

- Pour des raisons pratiques, on travail avec des variables l et h entières comprises entre 0 et 2^S et non avec des variables flottantes. Sous **Matlab** , il faut impérativement que S soit au plus égal à 53. Si l'on ne veut pas entrer dans les détails, on peut se contenter de considérer l représente le nombre nombre réel $\frac{l}{2^S}$, et h le nombre réel $\frac{h}{2^S}$.
- La fonction `modele` prend en argument le dernier symbole codé, et renvoie la loi du prochain symbole conditionnellement aux précédents. Au début du codage, la fonction `code` lui transmet le "symbole" -1, qui ne fait pas partie de l'alphabet.
- Comme on utilise des entiers, il faut être certain que l'intervalle de travail $[l, l+h[$ contient au moins deux points distincts, donc que $h >= 2$. Une condition suffisante est :
$$\forall k \leqslant n, \forall x_1^k \in \mathcal{A}^k, q_n(x_k|x_1^{k-1}) \geqslant 2^{-S+3}.$$
Pour S=53, il suffit donc que toutes les probabilités conditionnelles utilisées soient supérieures à 10^{-15}, ce qui n'est pas très restrictif.
- Par ailleurs, l'utilisation d'arrondis aux entiers les plus proches entraîne une légère déformation de la loi de codage : au lieu de $q_n(x_k|x_1^{k-1})$ on utilise la probabilité $\tilde{q}(x_k|x_1^{k-1}) = \frac{E(hr(k+1))-E(hr(k))}{h}$. On voit que $\tilde{q}(x_k|x_1^{k-1}) - q_n(x_k|x_1^{k-1}) \leqslant 2^{-S+2}$. Même pour de petites valeurs de S, la dégradation dans la qualité de compression est imperceptible.

Pour prouver la correction de cet algorithme, on peut vérifier qu'aux passages des lignes 19 et 29 les relations suivantes sont toujours vérifiées :

$$\left\{ \begin{array}{rcl} \lambda(x_1^k) & = & \overline{0.c} + 2^{-|c|}\left(\frac{1}{2} - \left(\frac{1}{2} - l\right) 2^{-r}\right) \\ q\left(x_1^k\right) & = & 2^{-|c|-r}\, h \end{array} \right.$$

2.4 Expérience

Le fichier `debfables.txt` contient le début du livre 1 des fables de La Fontaine, et le vecteur `freq` du fichier `freqfcais.mat` qui contient une estimation des fréquences d'apparition des caractères ASCII en français.

Les lignes de code suivantes montrent comment utiliser ces données.

```
fid = fopen('debfables.txt') ;
[A, count] = fread(fid, inf, 'char') ;
x=A' ;
c=code(x, @iid_ascii) ;
disp(['Longueur du code : ', num2str(length(c)/8), ' octets']) ;

function p=iid_ascii(s)
persistent p0 ;
if s==-1, load 'freqfcais' ; p0=freq ; end
p=p0 ;
```

En faisant l'expérience, on trouve que cet algorithme compresse le fichier `debfables.txt`, qui fait initialement 21563 octets, en un mot de code de 12197 octets : on atteint donc un taux de compression de 57%. Pour obtenir un meilleur taux, on peut utiliser un modèle Markovien d'ordre 1, c'est-à-dire en modéliser le texte par une chaîne de Markov dont les probabilités $P(X_{k+1} = j|X_k = i)$ sont données par la matrice `trans` (également définie dans le fichier `statsfcais.mat`) à partir d'estimations. On obtient alors un mot de code de 9123 octets, soit un taux de compression de 42%.

Code Matlab 1 Codage arithmétique

```
function essai_codearith
x = [1 3 2 2 3 1 2 1 1 2 2]
c = codearith(x, @iid)

function qc = iid(s)
qc = [0.3 0.5 0.2] ;

function c = codearith(x, lcc)
      %% x : message à coder, lcc : loi de codage conditionnelle
S = 53 ;       %% valeur maximale : 53
l = 0 ; h = 2 ∧ S ; ret = 0 ; symb = -1 ; c = [] ;       %% initialisations
for k = 1 : length(x)
   qc = feval(lcc, symb) ;       %% récupère la loi conditionnelle pour le symbole suivant
   symb = x(k) ;       %% lit ce symbole
   r = [0, cumsum(qc)] ;       %% partitionne l'intervalle [0, 1[ selon la loi conditionnelle qc
   l = l + round(r(symb) * h) ;       %% met à jour la borne inférieure de l'intervalle
   h = round(h * r(symb + 1)) - round(h * r(symb)) ;       %% met à jour sa largeur
   if h <= 1, error(['S trop petit : h = ', num2str(h)]) ; end       %% [l, l+h−1[ est de largeur nulle !
         %% peut-on appliquer les règles de renormalisation ?
   while (l >= 2 ∧ (S - 1)) || (l + h <= 2 ∧ (S - 1))       %% le bit de gauche est acquis
      nb = floor(l / 2 ∧ (S - 1)) ;       %% identifie le bit à ajouter au code
      c = [c, nb] ; for j = 1 : ret, c = [c, 1 - nb] ; end ; ret = 0 ;       %% ajoute avec les complémentaires
      l = 2 * (l - nb * 2 ∧ (S - 1)) ; h = 2 * h - 1 ;       %% renormalise
   end
   while (l >= 2 ∧ (S - 2)) && (l + h <= 3 * 2 ∧ (S - 2))
      ret = ret+1 ;       %% I ⊂ [1/4, 3/4[ ⟹ retient qu'il y a un complémentaire de plus
      l = 2 * l - 2 ∧ (S - 1) ; h = 2 * h - 1 ;       %% renormalise
   end
end       %% reste à finir d'émettre le code
t = l + h / 2 ;       %% milieu de l'intervalle
nb = floor(t / 2 ∧ (S - 1)) ; c = [c, nb] ; for j = 1 : ret, c = [c, 1 - nb] ; end
t = 2 * ( t - nb * 2 ∧ (S - 1)) ; nb = floor( t / 2 ∧ (S - 1)) ; c = [c, nb] ;
if h < 2 ∧ (S - 1)
   t = 2 * (t - nb * 2 ∧ (S - 1)) ; c = [c, floor( t / 2 ∧ (S - 1))] ;       %% encore un à émettre
end
```

Pour comparaison : le programme `gzip` fabrique lui un mot de code de 9271 octets, alors que celui produit par `bzip2` fait seulement 8136 octets.

3 Codage universel par mélange

Le principal problème du code présenté ci-dessus, c'est qu'il nécessite la connaissance a priori des statistiques de la source - ou tout au moins d'une loi de codage q qui n'en diffère pas trop. Comment faire si on ne les connaît pas ? Comment faire si l'on veut construire un seul code pour toutes les sources à la fois ? Après tout, `gzip` doit fonctionner pour tous les types de fichiers, que leur contenu soit un texte en français, en anglais, ou des séquences ADN...

Nous restreignons ici au modèle $\mathcal{M}_0$ des sources sans mémoire sur l'alphabet $\mathcal{A} = \{1, \ldots, m\}$. Soit

$$\Theta = \left\{ \theta = (\theta_1, \ldots, \theta_m) \in [0,1]^m : \sum_{i=1}^{m} \theta_i = 1 \right\}$$

le simplexe sur $\mathbb{R}^m$. Pour tout $\theta \in \Theta$, notons p_θ la loi de probabilité sur $\mathcal{A}$ telle que pour tout symbole a de $\mathcal{A}$ on ait $p_\theta(a) = \theta_a$. L'application $\Theta \to \mathcal{M}_0$ qui à θ associe la loi P_θ d'un processus $(X_n)_n$ sans mémoire tel que $P_\theta(X_1 = \cdot) = p_\theta(\cdot)$ est une bijection : on dit que le modèle $\mathcal{M}_0$ est paramétré par Θ.

Nous allons voir qu'il est possible de constuire un code dit "universel" qui atteint un taux de compression par symbole égal à l'entropie de toute source $P \in \mathcal{M}_0$. L'idée que nous développons ici est qu'il suffit d'utiliser comme loi de codage q un *mélange* des différentes sources :

$$q^\mu(x_1^n) = \int_{\theta \in \Theta} P_\theta \left(X_1^n = x_1^n \right) d\mu \left(\theta \right).$$

Si μ est une loi de probabilité sur Θ, alors q^μ est une loi de probabilité sur A^n, dont la longueur de code associée pour le message x_1^n est $-\log q^\mu(x_1^n)$

On peut aisément calculer $q^\mu(x_1^n)$ si le mélange μ est de la forme :

$$\mu(\theta) = \prod_{i=1}^{m} \theta_i^{\alpha - 1},$$

où α est un nombre réel positif. Notons Γ la fonction Gamma d'Euler, définie pour tout $t > 0$ par

$$\Gamma(t) = \int_0^\infty x^{t-1} e^{-x} dx.$$

Pour tout i de l'alphabet $\mathcal{A}$, soit $n_i = n_i \left(x_1^n \right) = \sum_{k=1}^{n} \mathbb{1}_{x_k = i}$ le nombre d'occurences du symbole i dans x_1^n. On trouve :

$$q^\mu(x_1^n) = \frac{\prod_{i=1}^{n} \Gamma \left(n_i + \alpha \right)}{\Gamma \left(n + m\alpha \right)}.$$

On en déduit très facilement les probabilités conditionnelles

$$q(i|x_1^n) = \frac{n_i + \alpha}{n + m\alpha}.$$

Ces probabilités conditionnelles suffisent pour écrire des fonctions **Matlab** que l'on peut utiliser comme modèle dans le codage arithmétique (c'est-à-dire utiliser comme troisième argument dans l'appel à la fonction `code`). On peut remarquer, comme heuristique, qu'utiliser le mélange μ revient à faire comme si on avait vu, avant le message x_1^n, "α occurences" de chaque symbole $i \in \mathcal{A}$, et comme si on utilisait à chaque instant comme loi conditionnelle $q\left(\cdot | x_{-k}^{-1} \right)$ l'estimateur du maximum de vraisemblance sur l'ensemble de ces "observations".

Quand on expérimente cette méthode sur le fichier `debfables.txt`, on obtient un mot de code de 12197 octets, soit seulement 105 octets de plus qu'en utilisant la table des fréquences de la langue française !

3.1 Etude de la redondance

En fait, on peut montrer que la redondance des codages par mélange est petite : pour n'importe quelle source $P \in \mathcal{M}_0$, elle sa croissance n'est que logarithmique en fonction de n. Nous proposons dans la suite de montrer une version faible de ce résultat.

Théorème 2 *Pour un réel $\eta > 0$, soit $B_n^\eta = \{x_1^n \in \mathcal{A}^n : \forall i \in \mathcal{A}, n_i(x_1^n) > \eta n)$. Alors quel que soit η, il existe une une fonction $C = C(\theta, \eta)$ telle que on a*

$$\sup_{x_1^n \in B_n^\eta} \left\{ \log \frac{1}{q^\mu(x_1^n)} - \inf_{\theta \in \Theta} \log \frac{1}{\mathbb{P}_\theta(x_1^n)} \right\} \leqslant C(\eta) \frac{m-1}{2} \log n.$$

En fait, on peut prendre $C(\theta, \eta)$ indépendante de η, et $\eta = 0$ mais la preuve est plus délicate. Ici, il suffit d'écrire explicitement le maximum de vraisemblance, et d'utiliser l'équivalent de Stirling.

On peut montrer une égalité semblable dans l'autre sens, et également montrer qu'aucune toute loi de codage universelle q_n ne peut faire beaucoup mieux dans le pire des cas. Précisément :

$$\inf_{q_n} \sup_{\theta \in \Theta} D(P_\theta^n, q_n) = \frac{m-1}{2} \log \frac{n}{2\pi e} + \log \frac{\Gamma\left(\frac{1}{2}\right)^m}{\Gamma\left(\frac{m}{2}\right)} + o(1)$$

quand n tend vers $+\infty$.

4 Questions

Il n'est pas possible de traiter toutes les questions qui suivent. Le candidat choisira celles qu'il préfère, et pourra d'ailleurs choisir de développer d'autres points du texte qu'il jugera intéressants. Il veillera à la qualité de rédaction de ses réponses. Il lui est vivement conseillé d'illustrer ses réponses par des graphiques et des simulations.

Les fichiers `debfables.txt, freqfcais.mat, codearith.m` sont téléchargeables à l'adresse :

http ://www.dptmaths.ens-cachan.fr/ ∼ agarivier/agreg/codearith.html

1. Donner la preuve complète des affirmations, propositions et théorèmes présentés dans ce texte.
2. Si $\mathcal{A} = \{0, 1\}$ et si les X_i sont des variables indépendantes identiquement distribuées suivant une loi de Bernoulli de paramètre p, représenter graphiquement $H(X)$ en fonction de p. Commenter.
3. Si $\mathcal{A} = \{0, 1\}$ et si les X_i forment une chaîne de Markov de matrice de transition $\begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix}$, que vaut $H(X)$? Commenter.
4. Effectuer à la main le codage arithmétique du message $x_1^4 = 1322$ avec la loi de codage de l'exemple 2.
5. Avec l'aide du code **Matlab** 1, écrire un petit programme qui permet de vérifier tous les mots de code de l'exemple 2.
6. Prouver la correction de l'algorithme présenté dans le code **Matlab** 1.
7. Effectuer les expériences évoquées dans le texte, ainsi que toutes celles qui vous sembleront pertinentes.
8. Dans le mélange μ présenté ci-dessus, quelle valeur de α donne les meilleurs résultats ? Pourquoi ?
9. Peut-on imaginer des mélanges pour les modèles markoviens ?
10. Si l'on suppose que les fables de La Fontaine vraiment issus d'une source stationnaire ergodique, estimer l'entropie $H(X)$ de cette source.
11. Quelle est la complexité de l'algorithme de codage arithmétique ? Est-ce bien ce que l'on observe ? Pourquoi ?
12. Généraliser l'agorithme de codage arithmétique pour un alphabet des mots de code b-aire ($b \geqslant 2$). En quoi cela peut-il être utile en pratique ?
13. Ecrire une fonction **Matlab** `y=decodearith(n, c)` prenant en argument la taille n du message à décoder et le mot code arithmétique c, qui permet de reconstruire les messages codés par la fonction `code`.
14. Quelles sont les "raisons pratiques" qui font que l'on travaille avec des variables l et h entières plutôt que flottantes ? Adapter le programme pour qu'il travaille directement avec des variables flottantes, et comprendre ce qui ne fonctionne pas. Eventuellement, corriger ce problème.