Annexe D

Le bit comme mesure de l'information

D.1 Les expériences de Shannon

Dans son article fondateur de la théorie de l'information, Shannon illustre la façon dont un texte contient de l'information en mesurant combien les suites de lettres sont prévisibles et caractéristiques d'un langage. Il utilise la notion de *n-grammes*, suites de n lettres consécutives dans un texte. Il montre que tous les n-grammes n'ont pas la même fréquence [1], et comment connaissant leurs fréquences il est possible d'engendrer des textes aléatoires qui ressemblent à des textes naturels.

Nous proposons de réaliser un programme (c-à-d. un ensemble de fonctions) qui

- mesure des fréquences de lettres ou de n-grammes,
- en déduit la quantité d'information portée par les lettres et l'entropie de la source utilisée,
- et de réaliser le programme inverse qui produit de façon aléatoire des textes qui vérifient une distribution donnée a priori.

Cette programmation met en jeu des structures de contrôle qui seront utilisées ailleurs, comme les listes d'association, et des tours de main, comme la façon d'engendrer des valeurs selon une distribution pré-établie.

On pourra faire ensuite des expériences avec les langues connues par les auteurs des programmes.

D.1.1 Analyse

Avant de commencer, il convient de préciser la notion de n-gramme. On abordera la question en se demandant quels sont les 5-grammes de « interdit d'interdire ». Le tableau D.1 les liste tous en indiquant leur nombre d'occurrences.

On doit faire attention aux points suivants :

- On décompte tous les groupes de n lettres consécutives, et pas les groupes consécutifs de n lettres.
- On considère les lettres en un sens très général, en fait tout ce qui concourt à écrire : lettres, ponctuations et espaces entre lettres (ici notés par ␣). On distinguera entre lettres minuscules et lettres majuscules et entre les différentes accentuations possibles.

Du nombre d'occurrence on peut passer à la probabilité d'occurrence en divisant chaque nombre d'occurrence par le nombre d'occurrences de n-grammes. La phrase « interdit d'interdire » contient donc 15 occurrences de 5-grammes, dont seulement 12 sont différents. La probabilité d'occurrence du 5-gramme « inter » est de $2/15$, tandis que celle de « erdir » est de $1/15$.

5-grammes	nombres d'occurrences
inter	2
nterd	2
terdi	2
erdit	1
rdit␣	1
dit␣d	1
it␣d'	1
t␣d'i	1
␣d'in	1
d'int	1
'inte	1
erdir	1

TABLE D.1 – Les 5-grammes de « interdit d'interdire »

4-grammes	5-èmes lettres	nombres d'occurrences
inte	r	2
nter	d	2
terd	i	2
erdi	t	1
	r	1
rdit	␣	1
dit␣	d	1
it␣d	'	1
t␣d'	i	1
␣d'i	n	1
d'in	t	1
'int	e	1

TABLE D.2 – Les 5-grammes de « interdit d'interdire »

En imaginant que la lecture se fasse lettre par lettre de gauche à droite, on voit qu'à partir de la n-ième lettre, chaque lettre lue est la fin du n-gramme qui commence par les $n-1$ lettres qui précèdent. De façon duale, la production d'un texte aléatoire suivant une distribution de n-grammes données pourra commencer par tirer un premier n-gramme, puis poursuivra en tirant une lettre en fonction de la probabilité qu'elle a de suivre les $n-1$ dernières lettres lues.

Le principe de base de ce programme sera donc de ne pas représenter directement des n-grammes, mais de représenter à la place des $n-1$-grammes accompagnés de la liste des

1. Plus précisément, il rappelle ce fait, bien connu des imprimeurs, et déjà utilisé en 1838 par Samuel Morse pour concevoir son fameux code.

lettres qui peuvent les prolonger, en accompagnant chaque lettre d'un nombre d'occurrence ou d'une probabilité relativement au $n-1$-gramme qu'elle suit. Cela correspond au tableau D.2. De cette façon, on pourra synthétiser des lettres en cherchant dans la table la ligne qui correspond aux $n-1$ dernières lettres synthétisées, et en sélectionnant la lettre suivante parmi les lettres qui peuvent suivre ce $n-1$-gramme.

D.1.2 Spécification

La fonction `string->distribution` prend deux paramètres :

1. un entier $n \geq 1$ qui décrit le grain de la mesure des fréquences : 1 pour les lettres isolées, 2 pour les digrammes, 3 pour les trigrammes, etc.
2. une chaîne de caractères s qui représente un texte. Ce texte est supposé de longueur supérieure à n.

Elle retourne la distribution d des n-grammes dans la chaîne de caractères s. Une distribution d sera codée en Scheme par une *liste d'associations* (*association list*, ou même *alist*, en anglais ; voir le rappel page 86) qui fait correspondre à chaque $(n-1)$-gramme rencontré une autre liste d'association qui fait correspondre à chaque lettre rencontrée après le $(n-1)$-gramme le nombre de fois où elle a été rencontrée. Le texte représenté par la chaîne de caractères s sera appelé le *texte de référence* de d.

La fonction `string->information` prend un paramètre : une chaîne de caractères s qui représente un texte de référence. Elle retourne la quantité d'information portée par chaque caractère du jeu de caractère employé dans le texte de référence. Le résultat sera représenté par une liste d'association.

La fonction `string->entropy` prend un paramètre : une chaîne de caractères s qui représente un texte de référence. Elle retourne l'entropie du jeu de caractère employé. Le résultat sera représenté par un nombre.

La fonction `distribution->string` prend un paramètre : une distribution d de n-grammes. Elle retourne une chaîne de caractères s dont la distribution des n-grammes suit la distribution d.

Représentation des distributions de n-grammes

Une distribution de lettres sera représentée par une liste d'associations $L \mapsto F$ où L est l'ensemble des lettres du texte de référence de la distribution et F est l'intervalle $[0\ T]$ où T est le nombre total de lettre. La fréquence f associée à une lettre l sera le nombre d'occurrences de l. Par exemple, `(list (cons #\a 25) (cons #\b 75))` représente une distribution où les a ont 25 occurrences et les b 75. C'est une façon de dire que les a sont trois fois moins fréquents que les b. On observe que la somme des fréquences vaut T.

Une distribution de n-grammes sera représentée par une liste d'associations $L^{n-1} \mapsto (L \mapsto F)$ où L est l'ensemble des lettres du texte de référence et F est l'intervalle $[0\ T]$ où T est le nombre d'occurrences du $n-1$-gramme correspondant.

Par exemple,

```
(list (cons (list #\a #\b)
            (list (cons #\a 1) (cons #\b 2)))
      (cons (list #\b #\a)
            (list (cons #\a 3) (cons #\b 4))))
```

représente une distribution des trigrammes `aba`, `abb`, `baa` et `bab`. La fréquence f associée à une lettre l dans la liste associée $n-1$-gramme p est le nombre d'occurrences de l qui suivent immédiatement une occurrence de p. La somme de toutes les fréquences de lettres qui suivent p est égale au nombre d'occurrences de p.

Dans les spécifications des fonctions à programmer, le type des distributions sera donc `(alist (list letter) (alist letter frequency))`. C'est-à-dire une aliste dont les clés sont des listes de lettres, et dont les valeurs sont des alistes dont les clés sont des lettres et les valeurs des fréquences.

Génération de lettres aléatoires

On trouve facilement dans les bibliothèques des langages de programmation un générateur de nombres aléatoires pour une distribution uniforme dans un intervalle donné. Par exemple, la bibliothèque de Scheme offre la fonction `random` qui engendre des nombres uniformément répartis entre 0 et 1. Dans tous les cas, il ne s'agit que de nombres pseudo-aléatoires. En effet, ils sont produits par un processus strictement déterministe, mais dont la progression est suffisamment chaotique pour que l'illusion d'un comportement aléatoire soit convaincante. On se contentera de cela. Par contre, on ne trouve généralement pas de générateur de nombre aléatoire selon une distribution donnée a priori. Il faudra donc réaliser cette fonction nous-même.

Comment engendrer des nombres aléatoires selon une distribution a priori quand on sait engendrer des nombres aléatoires selon une distribution uniforme ? On a vu plus haut (voir « Représentation des distributions de n-grammes ») que la distribution donnée a priori associe à chaque lettre l_i une fréquence f_i pour un préfixe de longueur $n-1$ donné. L'idée est d'allouer dans l'intervalle [0 1] des plages de tailles proportionnelles aux fréquences f_i. Puisque la distribution fournie par le système est uniforme, les nombres aléatoires qu'il produit tomberont dans ces plages selon une fréquence qui sera proportionnelle à la taille des plages. Il reste à trouver un moyen d'associer chaque lettre l_i à une plage $[l_i\ h_i]$. Pour commencer, les tailles des différentes plages sont calculées en normalisant les fréquences de façon à ce qu'elles soient toutes dans l'intervalle [0 1] ; il suffit de diviser les fréquences par T. Ainsi la taille de la plage associée à la lettre l_i sera f_i/T. On remarque que la somme des fréquences normalisées fait toujours 1 ; $\Sigma(f_i/T) = (\Sigma f_i)/T = T/T = 1$.

On associera donc la plage $[0\ f_1/T]$ à la lettre l_1, la plage $]f_1/T\ \ (f_1+f_2)/T]$ à la lettre l_2, la plage $](f_1+f_2)/T\ (f_1+f_2+f_3)/T]$ à la lettre l_3, etc. Plus généralement, on associe la plage $](\Sigma_{i=1}^{k-1} f_i)/T\ \ (\Sigma_{i=1}^{k} f_i)/T]$ à la lettre l_k. Il faut mémoriser ces limites de plages sous la forme d'une *distribution cumulative normalisée* qui associe à chaque lettre l_i la valeur $(\Sigma_{i=1}^{k} f_i)/T$ plutôt que sa fréquence f_i. Étant donné un nombre pseudo-aléatoire n on cherche la plus petite fréquence cumulée qui le dépasse ; elle indique la lettre qui est choisie.

Traitement des cas limites

Un autre aspect de ce projet est de programmer ces calculs d'une façon qui ne dépende pas de la taille des n-grammes. On souhaite que les mêmes fonctions traitent le cas des lettres isolées, des digrammes, des trigrammes et des n-grammes en général. C'est une caractéristique du produit final qui ne préjuge pas des étapes intermédiaires du développement. En particulier, nous recommandons de commencer par programmer le cas particulier qui ne traite que les lettres isolées, et seulement quand toute la chaîne de traitement sera bien comprise pour ce cas simplifié, il conviendra de généraliser à tous les n-grammes.

Le traitement des n-grammes introduit des cas limites qui paraissent particuliers mais qu'il convient de traiter de la façon la plus générique possible. Un de ces cas limites est celui des $n-1$-èmes premières lettres du texte de référence qui ne peuvent pas former de n-gramme puisqu'elles ne sont pas assez nombreuses. Nous proposons de traiter ce cas limite en concaténant devant le texte de référence suffisamment de caractères de bourrage pour que toutes les lettres du texte de référence puissent marquer la fin d'un n-gramme. Un caractère de bourrage est un caractère qui ne fait pas partie de l'alphabet utilisé dans le texte de référence ; on supposera que le caractère '^' convient (noté `#\^` en Scheme). Par exemple, si on analyse le texte de référence en termes de trigrammes, ses deux premières lettres forment un cas limite, mais si on fait précéder le texte de suffisamment d'occurrences du caractère '^' ce problème est résolu comme dans « ^^Mignone allons voir si la rose… » où « ^^M » et « ^Mi » forment des trigrammes tout à fait légitimes. De façon symétrique, on peut vouloir analyser ce qui se passe à la fin du texte de référence pour connaître quelles sont les lettres qui peuvent terminer un texte. Pour traiter cet autre cas limite on utilisera le caractère de bourrage $ (noté `#\$` en Scheme).

D.1.3 Plan de développement

On propose le plan de développement suivant. On programmera dans un premier temps les fonctions `string->distribution` et `distribution->string` pour le cas particulier où n vaut 1. Dans un second temps, on généralisera ces deux fonctions au cas des n-grammes quelconques. On ne programmera les fonctions `string->information` et `string->entropy` que pour le cas où n vaut 1.

1. Programmer une fonction `IncFreq` qui prend en paramètre une distribution de lettre et une lettre, et retourne une nouvelle distribution où la fréquence associée à la lettre est augmentée de 1.

   ```
   ;; increments the frequency of a given
   ;; letter in a distribution.
   ;; IncFreq:
   ;;   (alist letter freq) * letter
   ;;   -> (alist letter freq)
   ```

   Bien penser aux deux cas possibles, selon que la lettre est déjà présente dans la distribution ou ne l'est pas.

2. Programmer une fonction `AnalFreq` qui prend en paramètre une distribution de lettre et un texte représenté par une liste de lettres, et retourne une nouvelle distribution où la fréquence associée à chaque lettre est augmentée du nombre de ses occurrences dans le texte.

   ```
   ;; increments the frequency of all letters
   ;; of a text in a distribution.
   ;; AnalFreq:
   ;;   (alist letter freq) * (list letter)
   ;;   -> (alist letter freq)
   ```

3. Programmer la fonction `string->distribution` qui ne traite que les distributions de lettres isolées.

   ```
   ;; computes a distribution of letters from a
   ;; character string.
   ;; string->distribution:
   ;;   string -> (alist letter freq)
   ```

   On pourra utiliser la fonction `string->list` de Scheme pour calculer la liste des caractères qui forment une chaîne.

4. Programmer la fonction `TotalFreq` qui calcule la somme des fréquences d'une distribution.

   ```
   ;; computes the sum of all frequencies in a
   ;; distribution.
   ;; TotalFreq:
   ;;   (alist letter freq) -> totalfreq
   ```

5. Programmer la fonction `string->information` qui calcule la quantité d'information portée par chaque caractère du texte de référence (voir la définition 8.1).

   ```
   ;; computes information conveyed by every
   ;; letter from a character string.
   ;; string->information:
   ;;   string -> (alist letter info)
   ```

   On pourra utiliser la fonction `log` de Scheme.

6. Programmer la fonction `string->entropy` qui calcule l'entropie du jeu de caractère du texte de référence (voir la définition 8.2).

   ```
   ;; computes the entropy of the character set
   ;; of a character string.
   ;; string->entropy: string -> entropy
   ```

7. Programmer la fonction `CumulDistrib` qui calcule la distribution normalisée et cumulée d'une distribution. Cette fonction prend en paramètre une distribution, sa fréquence totale, et un cumul initial. Typiquement, cette fonction est appelée par `(CumulDistrib` d `(TotalFreq` d`) 0)`.

   ```
   ;; computes the cumulative distribution of a
   ;; distribution.
   ;; CumulDistrib:
   ;;   (alist letter freq)
   ;;   * totalfreq * cumul
   ;;   -> (letter -> cumulfreq)
   ```

8. Programmer la fonction `FindFirstGreaterInCD` qui recherche dans une distribution cumulée normalisée la première lettre dont la distribution cumulée est supérieure à une valeur donnée.

```
;; searches the first letter whose cumulated
;; distribution is bigger than a given value.
;; FindFirstGreaterInCD:
;;   (alist letter cumulfreq) * freq
;;   -> letter
```

9. Programmer la fonction `RandLetter` qui engendre une lettre de façon aléatoire mais selon une distribution cumulée normalisée donnée en paramètre.

   ```
   ;; Generates a random letter following a
   ;; given cumulative distribution.
   ;; RandLetter:
   ;;    (alist letter cumulfreq) -> letter
   ```

   On pourra utiliser la fonction `random` de Scheme.

10. Programmer la fonction `RandText` qui engendre un texte aléatoire dont les lettres suivent une distribution cumulée normalisée donnéee. Une longueur maximum est passée en second paramètre.

    ```
    ;; Generates a random text following a
    ;; given cumulative distribution.
    ;; RandText:
    ;;   (alist letter cumulfreq) * integer
    ;;   -> (list letter)
    ```

11. Programmer une première version de la fonction `distribution->string` qui ne traite que les distributions de lettres isolées. On considère ici une distribution telle que rendue par la fonction `string->distribution`. Un second paramètre indique la longueur de la chaîne à construire.

    ```
    ;; computes a string following a given
    ;; distribution.
    ;; distribution->string:
    ;;   (letter -> freq) * integer -> string
    ```

    On pourra utiliser la fonction `list->string` de Scheme pour obtenir la chaîne dont on connaît la liste des caractères.

Cette fonction clôt la première partie de ce projet qui ne traite que des distributions de lettres isolées, par opposition aux distributions de n-grammes. Traiter les distributions de n-grammes demande de ne plus considérer les lettres isolément mais dans le contexte d'un préfixe formé des $n-1$ lettres qui précèdent. De cette façon, le préfixe permet de chercher la bonne distribution avant de retrouver le cas précédent. Bien comprendre qu'une analyse se fait avec des n-grammes qui sont tous de la même taille, et donc des préfixes qui sont tous de taille $n-1$.

1. Programmer la fonction `IncFreq/pfx` qui prend en paramètre une distribution de lettre, un préfixe représenté par une liste de lettres, et une lettre, et retourne une nouvelle distribution où la fréquence associée à la lettre précédée du préfixe est augmentée de 1.

   ```
   ;; increments the frequency of a given
   ;; letter in a distribution.
   ;; IncFreq/pfx:
   ;;   (alist prefix (alist letter freq))
   ;;   * prefix * letter
   ;;   -> (alist prefix (alist letter freq))
   ```

2. Programmer la fonction `AnalFreq/pfx` qui prend en paramètre une distribution de lettre, un préfixe représenté par une liste de lettres, et un texte représenté par une liste de lettres, et retourne une nouvelle distribution où la fréquence associée à chaque lettre est augmentée du nombre de ses occurrences dans le texte en tenant compte du préfixe de chaque occurrence.

   ```
   ;; increments the frequency of all
   ;; letters of a text in a distribution.
   ;; AnalFreq/pfx:
   ;;    (alist prefix (alist letter freq))
   ;;    * prefix * text
   ;;    -> (alist prefix (alist letter freq))
   ```

   Noter que dans un texte $l_1 l_2 \ldots l_T$, les $n-1$ lettres qui précèdent une lettre l_i constituent le préfixe de l_i. Cela veut dire que dans les appels récursifs successifs de `AnalFreq/pfx` les lettres passent du paramètre texte au paramètre préfixe, puis sont oubliées :

   (a) `(AnalFreq/pfx` d_1 "abc" "$defghijk$"`)`
   (b) `(AnalFreq/pfx` d_2 "bcd" "$efghijk$"`)`
   (c) `(AnalFreq/pfx` d_3 "cde" "$fghijk$"`)`
   (d) `(AnalFreq/pfx` d_4 "def" "$ghijk$"`)`
   (e) `(AnalFreq/pfx` d_5 "efg" "$hijk$"`)`
   (f) `(AnalFreq/pfx` d_6 "fgh" "ijk"`)`
   (g) `(AnalFreq/pfx` d_7 "ghi" "jk"`)`
   (h) `(AnalFreq/pfx` d_8 "hij" "k"`)`
   (i) `(AnalFreq/pfx` d_9 "ijk" ""`)`

3. Programmer la fonction `string->distribution/pfx` qui traite les distributions de n-grammes quelconques. La taille n des n-grammes est passée en paramètre. On pourra utiliser la fonction `make-string` de Scheme pour engendrer les caractères de bourrage.

   ```
   ;; computes a distribution of n-grams from a
   ;; character string.
   ;; string->distribution/pfx:
   ;;   integer * string
   ;;   -> (alist prefix (alist letter freq))
   ```

   Ne pas oublier les caractères de bourrage.

4. Programmer la fonction `CumulDistrib/pfx` qui calcule la distribution cumulée normalisée d'une distribution avec préfixe. Cette fonction prend en paramètre une distribution de n-grammes.

   ```
   ;; computes the cumulative distribution of a
   ;; distribution.
   ;; CumulDistrib/pfx:
   ;;   (alist prefix (alist letter freq))
   ;;   -> (alist prefix (alist letter freq))
   ```

   Noter qu'une distribution de n-grammes comporte autant de distribution de lettres isolées que de préfixes différents. Ce sont ces distributions de lettres isolées qui font l'objet d'un cumul, et il y a autant de cumuls différents que de distributions de lettres isolées.

5. Programmer la fonction `RandLetter/pfx` qui engendre une lettre de façon aléatoire mais selon une distribution de n-grammes et un préfixe donnée en paramètre.

```
;; Generates a random letter following a
;; given distribution.
;; RandLetter/pfx:
;;   (alist prefix (alist letter cumulfreq)
;;   * prefix
;;   -> letter
```

6. Programmer la fonction `RandText/pfx` qui engendre un texte aléatoire dont les lettres suivent une distribution cumulée normalisée donnéee. Un préfixe initial est aussi passé en paramètre.

```
;; Generates a random text following a given
;; distribution.
;; RandText/pfx:
;;   (alist prefix (alist letter cumulfreq))
;;   * prefix * integer
::   -> (list letter)
```

7. Programmer la fonction `distribution->string/pfx` qui traite les distributions de n-grammes quelconques.

```
;; computes a string following a given
;; distribution.
;; distribution->string/pfx:
;;   (alist prefix (alist letter freq))
;;   -> string
```

Faire des expériences avec des langues et des tailles de n-grammes variées. Admirer la poésie involontaire de certains des textes produits.