Modèles de N-grammes

Adapté d'un sujet de TP d'agrégation

Consignes

La durée de l'épreuve est de 3h30.

Chaque partie contenant des questions de programmation est associée à un seul langage de programmation qu'il faudra respecter. Des rappels de programmation dans les deux langages sont donnés en annexe.

Il est attendu, au cours de l'épreuve, la rédaction d'un compte-rendu qui contiendra notamment :

- un résumé des questions n'apparaissant pas dans le sujet posées par l'examinateur ;
- pour les questions de programmation, le nom des fonctions répondant à la question (s'il n'est pas imposé par l'énoncé), ainsi qu'un résumé des tests effectués pour tester le code ;
- pour les questions n'attendant pas de code, un résumé de ce qui a été expliqué au jury (éventuellement via des schémas).

Pour simplifier l'étude, on supposera que tous les caractères considérés sont ceux de la table ASCII et sont représentés en C et OCaml par le type `char`, en ignorant tout problème relatif à l'encodage. L'annexe rappelle comment faire la conversion entre un caractère et le code entier ASCII associé, compris entre 0 et 127.

1 Contexte

On souhaite développer un système permettant de prédire la suite d'une séquence de caractères. Ce genre de système est utilisé pour compléter les messages mails ou SMS par exemple. L'utilisateur commence à taper son message (une séquence de caractères) et le système propose de le compléter.

L'approche considérée est basée sur les N-grammes par apprentissage sur un texte préexistant. Un N-gramme est une séquence de N caractères consécutifs apparaissant dans le texte. Cette approche est expliquée sur le texte suivant (un réel apprentissage nécessite un texte bien plus grand) :

`Bonjour,␣comment␣allez-vous␣?␣Ca␣va,␣ca␣va␣aller␣bien␣mieux.`

L'objectif de l'approche est :

1. de construire les N-grammes contenus dans le texte ;
2. d'identifier les caractères qui leur succèdent en calculant les probabilités associées.

La première étape nécessite de choisir la valeur de N, la taille des N-grammes. Dans l'exemple précédent, le texte contient, par ordre d'apparition :

- les 1-grammes : `"B"`, `"o"`, `"n"`, `"j"`, `"u"`, `"r"`, `","`, `'␣'`, `"c"`, ...
- les 2-grammes : `"Bo"`, `"on"`, `"nj"`, `"jo"`, `"ou"`, ...
- les 3-grammes : `"Bon"`, `"onj"`, `"njo"`, `"jou"`, ...
- etc.

La deuxième étape vise à identifier les caractères qui suivent chaque N-gramme, qu'on appellera les **successeurs**, et compter leurs occurrences pour connaître leur probabilité d'apparition. Par exemple :

- le 1-gramme `"a"` peut être suivi par les caractères `'l'`, `'␣'` et `','` avec respectivement 2, 3 et 1 occurrences, donc les probabilités $\frac{2}{6}$, $\frac{3}{6}$ et $\frac{1}{6}$;
- le 1-gramme `"u"` peut être suivi par les caractères `'r'` et `'s'` avec respectivement 1 et 1 occurrences, donc les probabilités $\frac{1}{2}$ et $\frac{1}{2}$;
- le 2-gramme `"le"` peut être suivi par les caractères `'z'` et `'r'` avec respectivement 1 et 1 occurrences, donc les probabilités $\frac{1}{2}$ et $\frac{1}{2}$.

En choisissant une taille de N-grammes, c'est-à-dire en fixant N, on dispose alors de probabilités conditionnelles qui permettent de prédire le caractère qui fait suite à une séquence de caractères donnée.

2 Modèle de 1-grammes

Cette partie doit être réalisée en C.

Dans une version simplifiée de modèle de 1-grammes, le caractère à prédire après un autre sera toujours le même, à savoir le plus fréquent parmi les successeurs. En cas d'égalité de nombres d'occurrences entre deux successeurs, on choisira arbitrairement le premier dans l'ordre alphabétique, à savoir celui qui a le plus petit code associé. Ainsi, le modèle n'a pas à garder en mémoire l'ensemble des successeurs pour chaque caractère, mais seulement celui qui sera prédit.

En reprenant les exemples précédents, le modèle prédira que :

- le 1-gramme `"a"` sera suivi par le caractère `'␣'` qui est le plus fréquent parmi les successeurs ;
- le 1-gramme `"u"` sera suivi par le caractère `'r'`, qui apparaît aussi fréquemment que `'s'` mais se trouve avant `'s'` dans l'ordre alphabétique.

On implémente un modèle de 1-grammes par un tableau d'entiers `M`, de taille 128, tel que si `c` est un caractère de code `k`, alors `M[k]` vaut :

- le code du caractère à prédire si `c` possède des successeurs ;
- le code 0 du caractère nul `'\0'` sinon.

Question 1 Écrire une fonction `int plus_frequent_successeur(char c, char* chaine)` qui prend en argument un caractère `c` et une chaîne de caractère et renvoie le code du caractère qui apparaît le plus fréquemment dans la chaîne comme successeur de `c`, ou le plus petit code en cas d'égalité. La fonction renverra le code du caractère nul si `c` n'apparaît pas dans la chaîne.

Question 2 En déduire une fonction `int* init_modele(char* chaine)` qui construit et renvoie un modèle de 1-gramme à partir d'un texte d'apprentissage donné en argument.

Question 3 Quelle est la complexité temporelle de la fonction précédente ? Peut-on faire mieux ? On ne demande pas d'implémenter une éventuelle solution mais de la décrire succinctement à l'examinateur.

Question 4 Écrire une fonction `int** matrice_confusion(int* M, char* test)` qui prend en argument un modèle de 1-gramme et une chaîne de caractères `test` et crée et renvoie une matrice de confusion de taille 128×128 comparant les prédictions données par le modèle et les successeurs de chaque caractère dans la chaîne de test.

Question 5 Évaluer le modèle de prédiction basé sur la phrase donnée en exemple, en calculant le pourcentage d'erreurs à partir de la matrice de confusion pour la phrase de test :

`Bonjour,␣ca␣va␣bien␣?␣Oui␣!␣Bien␣mieux,␣et␣vous,␣ca␣va␣?`

3 Modèle de N-grammes

Cette partie doit être réalisée en OCaml.

Par rapport au modèle précédent, on propose deux changements :

- la prédiction se fait non plus à partir du dernier 1-gramme lu, mais du dernier N-gramme, ou le dernier $(N-1)$-gramme si le N-gramme n'a pas de successeur, ou du dernier $(N-2)$-gramme si le $(N-1)$-gramme n'a pas de successeur, ... en allant si besoin jusqu'au 0-gramme (la chaîne vide), dont les successeurs sont l'ensemble des lettres du texte d'apprentissage ;
- la prédiction n'est plus déterministe, mais choisit aléatoirement un successeur pour un N-gramme en fonction de la probabilité associée.

Par exemple :

– le modèle prédira que le 1-gramme `"a"` sera suivi par le caractère `'l'` avec probabilité $\frac{2}{6}$, `'␣'` avec probabilité $\frac{3}{6}$ et `','` avec probabilité $\frac{1}{6}$;
– le 3-gramme `"ple"` n'a pas de successeur, donc le modèle utilisera le 2-gramme `"le"` pour faire la prédiction, qui sera le caractère `'r'` avec probabilité $\frac{1}{2}$ et `'z'` avec probabilité $\frac{1}{2}$;
– le 2-gramme `"ay"` n'a pas de successeur, et le 1-gramme `"y"` non plus, donc le modèle utilisera le 0-gramme pour faire la prédiction, qui sera le caractère `'B'` avec probabilité $\frac{1}{60}$, le caractère `'o'` avec probabilité $\frac{4}{60}$, le caractère `'n'` avec probabilité $\frac{3}{60}$, ...

Dans cette partie, un modèle de N-grammes contient, pour chaque k-gramme du texte d'apprentissage, $k \leqslant N$, l'ensemble de ses successeurs et du nombre d'occurrences correspondant, ou des probabilité associées.

Question 6 Décrire une structure de données pour encoder un modèle de N-grammes avec un N donné. Cette structure doit permettre de retrouver les modèles dont la taille est inférieure à N. Par exemple, un modèle de 3-grammes devra contenir les successeurs des 3-grammes, mais également des 2-grammes, des 1-grammes et du 0-gramme. On fera valider la structure par l'examinateur avant de créer un type correspondant en OCaml.

On donne ici quelques suggestions pour la question précédente :

– on peut représenter un ensemble de successeurs et de leurs occurrences par :
  * un tableau de taille 128, où l'élément d'indice k est le nombre d'occurrences de la lettre dont le code est k.
  * un dictionnaire dont les clés sont les caractères et les valeurs le nombre d'occurrences.

  On peut alors représenter un modèle de N-grammes comme un dictionnaire dont les clés sont les N-grammes et les valeurs les ensembles de successeurs et occurrences associés.
– On peut également représenter directement un modèle de N-grammes comme un arbre préfixe (ou *trie*), c'est-à-dire un arbre d'arité quelconque vérifiant :
  * la racine est étiquetée par le mot vide ;
  * les autres nœuds sont étiquetés par un couple (caractère, nombre d'occurrences) ;
  * à chaque nœud, on peut associer le mot constitué de la concaténation de tous les caractères lues dans le chemin de la racine à ce nœud ;
  * chaque nœud interne a pour enfants tous les successeurs du mot associé, avec le nombre d'occurrences correspondant.

Question 7 Écrire une fonction qui construit et renvoie un modèle de N-grammes à partir d'un texte d'apprentissage et d'un entier N donnés en argument.

Question 8 Écrire une fonction qui prend en argument un modèle et un N-gramme et renvoie un caractère prédit par le modèle selon le principe décrit précédemment.

Pour tester le modèle, on souhaite faire de la génération automatique de texte en *pseudo-français*, c'est-à-dire un texte dont les mots ressemblent à du français mais n'en sont pas forcément. L'idée est de partir d'une chaîne de caractère (vide ou non) et de prédire les caractères suivants, à chaque fois en considérant le dernier N-gramme, qui contient éventuellement les lettres déjà prédites. Voici par exemple un texte généré automatiquement à partir de la chaîne `"Bonjour"` en utilisant le modèle de la phrase donnée initialement en exemple, pour $N = 2$.

```
Bonjour, ca allez-vour, ca va, ca aller bien mien mieux.-vous ? Ca va, commen mieux.
va va aller bieux.va va, commen mient allez-vous ? Ca va va, ca va va va, ca va va va,
comment aller bieux.
```

Question 9 Écrire une fonction qui prend en argument un modèle de N-grammes, une chaîne de caractères et une taille cible et génère un texte aléatoire de taille cible à partir de la chaîne, en utilisant le modèle de prédiction, selon le principe décrit précédemment.

Question 10 Le fichier texte `cid.txt` contient un extrait du Cid (Corneille). Utiliser ce texte pour faire de la génération automatique, pour différentes valeurs de N.

4 Programmation concurrente

Cette partie doit être réalisée en OCaml.

La construction d'un modèle de N-grammes peut prendre du temps surtout si le texte d'entrée est très long. Pour optimiser la construction, on souhaite mettre en place une approche exploitant la concurrence, avec plusieurs fils d'exécution.

Question 11 Proposer à l'examinateur une approche pour répartir la construction du modèle sur plusieurs fils.

Question 12 Implémenter la solution précédente.

Annexe : rappels de programmation

Rappels en C

La compilation d'un fichier `source.c` peut se faire avec la commande `gcc source.c`. On peut rajouter des options avec la commande `gcc` *`options`* `source.c` où *`options`* doit être remplacé par une ou plusieurs options de compilation. On cite par exemple :

- choix du nom d'exécutable : `-o nom_exec`
- avertissements : `-Wall`, `-Wextra`
- alerte mémoire : `-fsanitize=address`

On peut lancer un fichier exécutable `nom_exec` par la commande `./nom_exec`.

On peut convertir un caractère `c` en son code ASCII associé par `(int) c`. On peut convertir un entier `k` entre 0 et 127 en le caractère associé par `(char) k`.

Rappels en OCaml

La compilation d'un fichier `source.ml` peut se faire avec la commande `ocamlc source.ml`. Comme en C, on peut rajouter des options de compilation avec `ocamlc` *`options`* `source.ml` où *`options`* doit être remplacé par une ou plusieurs options de compilation. On cite par exemple :

- choix du nom d'exécutable : `-o nom_exec`
- utilisation des fils d'exécution : `-I +threads unix.cma threads.cma` (en dernière option)

On peut tirer un entier pseudo-aléatoire entre 0 inclus et n exclu par la commande `Random.int n`. On peut écrire `Random.self_init ()` pour utiliser une graine aléatoire.

`Char.code : char -> int` et `Char.chr : int -> char` permettent de faire la conversion entre caractère et code associé. La fonction `String.sub : string -> int -> int` permet de renvoyer une sous-chaîne : `String.sub s deb taille` renvoie la sous-chaîne de `s` commençant à l'indice `deb` et de taille `taille`.

L'utilisation de dictionnaires peut se faire à l'aide du module `Hashtbl`. On dispose notamment des commandes :

- `Hashtbl.create : int -> ('a, 'b) Hashtbl.t` qui prend en argument un entier correspondant à une estimation du nombre de clés qui vont être associées dans la table (en pratique on peut simplement utiliser `1` car les tables sont redimensionnables dynamiquement) et crée une table vide ;
- `Hashtbl.add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` qui prend en argument une table, une clé et une valeur et rajoute une association ;
- `Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool` qui teste si une table contient une association pour une clé donnée ;

– `Hashtbl.find : ('a, 'b) Hashtbl.t -> 'a -> 'b` qui prend en argument une table et une clé et renvoie la valeur associée, ou lève l'exception `Not_found` si la clé n'a pas d'association ;
– `Hashtbl.iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit` qui prend en argument une fonction f et une table et applique la fonction f à chaque couple (clé, valeur) de la table.

L'utilisation des outils de synchronisation peut se faire à l'aide des commandes suivantes. Pour simplifier l'utilisation des sémaphores, on peut renommer le module avec :

```
module Sem = Semaphore.Counting
```

– `Thread.create : ('a -> 'b) -> 'a -> Thread.t` prend en argument une fonction `f` et un argument `x` et renvoie un fil d'exécution qui exécute `f x`. **Attention** : la fonction `f` ne doit avoir qu'un seul argument. Si on souhaite la faire travailler avec plusieurs arguments, on peut utiliser un tuple ;
– `Thread.join : Thread.t -> unit` prend en argument un fil d'exécution et attend qu'il ait terminé son exécution ;
– `Mutex.create : unit -> Mutex.t` renvoie un nouveau mutex ;
– `Mutex.lock : Mutex.t -> unit` verrouille un mutex ;
– `Mutex.unlock : Mutex.t -> unit` déverrouille un mutex ;
– `Sem.make : int -> Sem.t` prend en argument un entier k et renvoie un sémaphore avec un compteur initialisé à k ;
– `Sem.acquire : Sem.t -> unit` acquière un sémaphore (c'est-à-dire décrémente le compteur) ;
– `Sem.release : Sem.t -> unit` libère un sémaphore (c'est-à-dire incrémente le compteur).

Attention, il est nécessaire d'utiliser l'option de compilation prévue à cet effet. On peut également utiliser un interpréteur OCaml au lieu de compiler, auquel cas il sera nécessaire d'exécuter les commandes suivantes :

```
#directory "+threads";;
#load "unix.cma";;
#load "threads.cma";;
```