IA naïve de génération de textes

Introduction

Le but de ce projet est d'implémenter une IA simple pour générer des textes en utilisant les informations extraites d'un ensemble de textes.

Pour cela, étant donné un ensemble de textes en entrée, nous allons d'abord calculer la fréquence de toutes les séquences de deux mots dans les textes d'origine. On utilisera ces informations pour produire aléatoirement de nouvelles phrases respectant les fréquences de ces séquences.

Cette méthode est connue sous le terme de chaîne de Markov. À partir du texte d'entrée, nous calculons une table de distribution qui associe à chaque mot la liste des mots qui peuvent apparaître après lui, avec leurs fréquences relatives.

Par exemple, si nous examinons le texte :

`I am a man and my dog is a good dog and a good dog makes a good man`

et en le délimitant avec `"START"` et `"STOP"` pour identifier le début et la fin de la phrase, nous obtenons la table de distribution suivante :

Mot	mots suivants	fréquence
`"START"`	`"I"`	100%
`"I"`	`"am"`	100%
`"am"`	`"a"`	100%
`"a"`	`"man"`	25%
	`"good"`	75%
`"man"`	`"and"`	50%
	`"STOP"`	50%
`"and"`	`"my"`	50%
	`"a"`	50%
`"my"`	`"dog"`	100%
`"dog"`	`"is"`	33%
	`"and"`	33%
	`"makes"`	34%
`"good"`	`"dog"`	66%
	`"man"`	34%
`"is"`	`"a"`	100%
`"makes"`	`"a"`	100%

Cette table peut ensuite être utilisée pour générer un nouveau texte qui ressemble à l'entrée de la manière suivante :

- en partant du mot `"START"`, on choisit aléatoirement un des mots qui peut apparaître après, avec la probabilité associée dans la table

- on l'ajoute à la liste des mots de sortie,
- on répète le processus jusqu'à ce que le mot `"STOP"` soit trouvé.

Ci-dessous, quelques phrases exemples produites en utilisant cette table.

```
START I am a good  man STOP

START I am a good dog is a good dog and my dog and my dog is a man
and my  dog and a man STOP

START I am a good dog is a man and my dog makes a good man STOP

START I am a good dog makes a good dog is a good dog  and a good
dog makes a good dog is a man STOP
```

Plan

Ce projet se décompose en 3 parties :

A. D'abord, on va construire un prototype qui prend une phrase en entrée et génère aléatoirement des phrases avec la table de distribution déterminée grâce à l'entrée.

B. Ensuite, on va utiliser une structure de donnée plus efficace pour améliorer les performances afin de pouvoir utiliser en entrée des textes plus longs comme de petits livres.

C. Finalement, on améliorera la qualité de nos entrées et de nos sorties en analysant un peu mieux l'ensemble de texte en entrée et en considérant des séquences de plus de deux mots.

Partie A : prototype

Notre premier objectif est de construire une table de distribution et de générer des phrases à partir de celle-ci, en utilisant des listes et leurs opérateurs prédéfinis. Pensez à utiliser autant que possible le module `List` (`List.assoc`, `List.length`, `List.nth`, `List.fold_left`, *etc.*) et ne pensez pas à l'efficacité.

Dans cet exercice, nous utiliserons des listes associatives comme structure de données pour lier chaque mot à ses suivants possibles. Les listes associatives sont souvent utilisées dans des prototypes ou des programmes non critiques car elles sont très faciles à utiliser et à déboguer. Leur principal inconvénient est la complexité de la recherche d'un élément.

Le type d'une liste associative qui associe des clés de type `string` à des valeurs de type `'a` est:

```
(string * 'a) list
```

Chaque paire représente une association d'un mot à une valeur de type `'a`. Ainsi, associer `3` à `"x"`, par exemple, consiste à ajouter `("x",3)` en tête de la liste associative. La fonction qui recherche la valeur associée à une clé est déjà définie dans la bibliothèque standard avec `List.assoc` ou `List.assoc_opt`. Notons que ces deux fonctions retournent la première association, pas les suivantes.

Par exemple, l'application:

```
List.assoc_opt "aa" [("bb", 2); ("aa", 1); ("cc", 3); ("aa", 4)]
```

retourne la valeur

```
Some 1
```

Pour cette partie, le type des tables est :

```
type followers_assoc = (string * string list) list
```

Question 1

Nous travaillerons sur des phrases en anglais. On considère qu'un mot est une séquence de lettres et de chiffres et que tout le reste sont des séparateurs. Écrivez la fonction `is_alphanum : char -> bool`.

`is_alphanum c` vaut `true` si `c` est un caractère alpha-numérique et `false` sinon.

Question 2

Pour commencer, nous avons besoin d'une fonction `words : string -> string list` que découpe une phrase en une liste de mots. On considéra tous les caractères non-alpha-numériques comme des séparateurs. Pour écrire la fonction `words`, suivez l'algorithme suivant :

- remplacer tous les caractères non-alphanumériques par le caractère `' '` (espace)
- utiliser `String.split_on_char` pour découper la phrase en une liste de mots
- retirer les mots vides

Par exemple, l'application:

```
words "Ocaml is a functional programming language."
```

retourne la valeur

```
["Ocaml"; "is"; "a"; "functional"; "programming"; "language"]
```

L'ordre est important.

Question 3

Écrivez une fonction `add_follower_assoc : string -> string -> followers_assoc -> followers_assoc` qui prend en entrée un mot, son suivant et une table de distribution et retourne cette même table dans laquelle a été ajouté le nouveau suivant.

Par exemple l'application :

```
add_follower_assoc "good" "dog" [("a", ["good"; "good"]); ("good", ["man"])]
```

retourne la valeur :

```
 [("a", ["good"; "good"]); ("good", ["dog"; "man"])]
```

Notez que pour la liste associative elle-même autant que pour la composante droite des paires, l'ordre n'a pas d'importance (c'est d'ailleurs un indice que les listes ne sont pas la structure la plus adaptée).

Autrement dit

```
 [("good", ["man"; "dog"]); ("a", ["good"; "good"]) ]
```

est aussi un résultat valide.

Question 4

En utilisant `add_assoc`, écrivez la fonction `build_assoc : string list -> followers_assoc` qui construit une liste associative liant chaque mot présent dans le texte d'entrée à tous ses suivants possibles (y compris les doublons). La table doit également contenir "START" qui pointe vers le premier mot et "STOP" qui est pointé par le dernier mot.

Par exemple, une table correcte (et minimale) pour `"xx yy zz yy xx yy"` ressemble à :

```
    [ ("zz", [ "yy" ]);
      ("xx", [ "yy" ; "yy" ]);
      ("START", [ "xx" ]);
      ("yy", [ "xx" ; "zz" ; "STOP" ]) ]
```

Question 5

Écrivez la fonction `get_follower_assoc : followers_assoc -> string -> string` qui prend une table, un mot donné et retourne un suivant aléatoire parmi ceux valides pour ce mot. Votre fonction devrait respecter la distribution de probabilité (ce qui devrait être assuré trivialement par la présence des doublons dans les listes de suivant).

Par exemple, l'application :

```
get_follower_assoc (build_table_assoc (words `"x y z y x y")) y
```

peut retourner avec la même probabilité `"x"`, `"z"`, ou `"STOP"`.

Pour obtenir un nombre aléatoire, utilisez le module `Random` de la librairie standard. `Random.int` retourne un entier aléatoire compris entre `0` et `n-1`

Question 6

Écrivez la fonction de génération aléatoire `generate_text_assoc : followers_assoc -> string list` qui prend une table et retourne une séquence de mots formant une phrase aléatoire valide (sans les `"START"` et `"STOP"`).

Générez quelques exemples en utilisant les textes prédéfinies dans `predefined.ml` et mettez les en commentaires de votre fonction.

Vous pouvez utiliser la fonction fournie `print_quote: string list -> unit` pour afficher le texte généré.

Par exemple, l'application :

```
generate_text_assoc
   (build_table_assoc
      (words "I am a man and my dog is a good dog"))
```

peut retourner (c'est aléatoire !)

```
    ["I"; "am"; "a"; "man"; "and"; "my"; "dog"]
```

Partie B : Amélioration des performances

Pour la suite, nous allons utiliser une structure de données plus efficace afin de pouvoir traiter des entrées plus importantes et construire de plus grandes tables de distribution.

Comme ce cours porte sur la programmation fonctionnelle, nous utiliserons le module `Map` qui prédéfinit des dictionnaires en OCaml, mais une table de hachage serait encore plus pertinente. Pour les dictionnaires, les opérations d'insertion et de recherche sont logarithmiques et donc bien plus rapides que la liste associative pour laquelle ces opérations sont de complexité linéaire.

Dans la partie précédente, nous avons stocké pour chaque mot la liste complète des mots suivants possibles, y compris les doublons. C'est une structure de données valide à utiliser lors de la construction de la table puisque ajouter un nouveau suivant en tête de liste est rapide. Mais lors de la génération, cela signifie calculer la longueur de cette liste à chaque fois, et accéder à son élément par indice, ce qui est lent si la liste est longue.

Pour cet exercice, on souhaite donc améliorer les performances. On associe chaque mot à une distribution, plutôt qu'à une liste de mot.

```
type distribution =
     { occurrences : int ;
       followers : (string * int) list }
```

Exemple :

La distribution associée au mot `"x"` dans le texte `"x y z y x y"` est :

```
{ occurrences = 2 (* le nombre d'occurrences de "x" *)
  followers = [("y", 2)]}
```

alors que celle de `"y"` est :

```
{ occurrences = 3
  followers = [("z", 1); ("STOP", 1); ("x", 1)]} (* l'ordre n'importe pas *)
```

Question 7

Avant toute chose, il faut construire le dictionnaire avec `Map.Make`.

Construire le module `StrMap` qui associe à un mot à un type `'a` (qui sera `distribution` dans notre cas).

Le module dictionnaire `Map` dispose de nombreuses fonctions utiles. Les fonctions suivantes en particulier peuvent être utiles pour le projet :

- `StrMap.map : ('a -> 'b) -> 'a StrMap.t -> 'b StrMap.t`
- `StrMap.bindings : 'a StrMap.t -> (string, 'b) list` : permet inspecter le contenu de votre map en la transformant en une liste associative
- `StrMap.update : string -> ('a option -> 'b option) -> 'a StrMap.t -> 'b StrMap.t` (aller voir la doc !

Le type de la table de distribution pour cette partie est :

```
type followers_map = distribution StrMap.t
```

Question 8

Écrivez `compute_distribution : string list -> distribution` qui prend une liste de mots et retourne une paire contenant sa longueur et une association entre chaque mot présent dans la liste originale et son nombre d'occurrences.

Par exemple, l'application :

```
compute_distribution ["a"; "b"; "c"; "c"; "c"; "c"; "c";  "b"]
```

retourne

```
{ occurrences = 10 ;
  followers = [("c", 5); ("b", 2); ("a", 1)] }
```

Conseil : Une première étape qui simplifie le problème est de trier la liste (utiliser `List.sort`).

Question 9

Définisez `build_table_map : string list -> followers_map` qui crée un dictionnaire. Cette fonction fait la même chose que `build_table_assoc` mais retourne un `followers_map` à la place d'une `followers_assoc`.

Comme pour la liste associative, le dictionnaire est indexé par les mots, chaque mot étant associé à ses suivants. Mais au lieu de stocker la liste des suivants, on utilise ici une `distribution`.

Conseil : Avant de calculer la distribution, une première étape possible est de réunir tous les suivants d'un mot dans une liste avec des doublons comme dans la partie précédente, et les stocker dans un dictionnaire intermédiaire de type `string list StrMap.t` (plutôt que dans une liste associative comme en Partie A).

Commencez par définir `add_follower_map : string -> string -> string list StrMap.t -> string list StrMap.t`, puis écrivez la fonction qui construit le dictionnaire intermédiaire (de type `string list StrMap.t`) en utilisant `add_follower_map`. Finalement, écrivez la fonction `build_table_map` à partir de cela.

Question 10

Définissez `get_follower_map : followers_map -> string -> string` qui fait la même chose que `get_follower_assoc` pour le nouveau format de table. Attention, retourner un successeur aléatoire est un peu plus compliqué !

Question 11

Enfin, définissez `generate_text_map : followers_map -> string list`.

Générez quelques exemples en utilisant les textes prédéfinies dans `predefined.ml` et mettez les en commentaires de votre fonction.

Partie C : amélioration de la qualité du traitement

Pour générer des phrases à partir d'ensemble plus larges, il faut prendre en compte la ponctuation. En particulier, la structure en phrases du texte : on veut pouvoir générer du texte en utilisant non seulement le début du texte original, mais le début de n'importe quelle phrase qu'il contient.

Question 12

On souhaite définir une fonction qui divise une chaîne de caractère en une liste de phrases (donc une liste de listes de mots `string list list`) telles que :

- les séquences ininterrompues de lettres romaines, de chiffres et de caractères non ASCII (dans la plage de `'\128'` à `'\255'`) sont des mots ;

- les caractères de ponctuation uniques `';'`, `','`, `':'`, `'-'`, `'"'`, `'''`, `'?'`, `'!'` et `'.'` sont considérés comme des mots ;
- les caractères de ponctuation `'?'`, `'!'` et `'.'` terminent les phrases ;
- tout le reste est un séparateur ;

Pour faire cela, en utilisant le type énuméré ci-dessous :

```
type kind = Valid | Terminate | Single | Separator
```

définissez la fonction `kind_of_char : char -> kind` qui associe à chaque caractère sa catégorie `kind`.

Question 13

Chaque catégorie de caractère correspond à un comportement différent à appliquer quand un tel caractère est rencontré dans le texte d'entrée. Par exemple :

- `Valid` : un caractère de cette catégorie doit être concaténé au mot en cours de construction ;
- `Separator` : un caractère de cette catégorie marque la fin du mot en cours, qui doit être ajouté à la phrase en cours s'il n'est pas vide.

Déterminez les actions associées aux deux dernières catégories puis utiliser `String.fold_right` pour construire la fonction `sentence : string -> string list list` qui découpe une entrée en liste de phrases suivant les règles énoncées au dessus.

Conseil La fonction que `String.fold_right` nécessite 3 accumulateurs : le mot en cours, la phrase en cours et la liste de phrases terminées.

Question 14

Défnissez le module `SLMap.t` en utilisant `Map.Make` tel que les clés soient des `string list`.

Pour la suite, on va considérablement améliorer les résultats obtenus en faisant correspondre des séquences de plus de deux mots. Nous mettrons donc à jour le format de nos tables :

```
type followers_multi = { prefix_length : int;
                         table : distribution SLMap.t }
```

Question 15

Il nous faut donc identifier des séquences de N mots dans le texte. Dans ce cas, le champ `prefix_length` contiendrait la valeur N - *1*. Le champ `table` associe chaque liste de N - *1* mots contenu dans le texte à la distribution de ses successeurs possibles.

Le tableau ci-dessous donne la table de distribution pour l'exemple donné au début du projet :

```
I am a man and my dog is a good dog and a good dog makes a good man
```

et une taille de 2. Vous pouvez voir que les points de branchement sont moins nombreux et ont un peu plus de sens.

Séquence de mots	Mots suivants	Fréquence
`["START"; "START"]`	`"I"`	100%
`["START"; "I"]`	`"am"`	100%
`["I"; "am"]`	`"a"`	100%
`["am"; "a"]`	`"man"`	100%
`["man"; "and"]`	`"my"`	100%
`["is"; "a"]`	`"good"`	100%
`["and"; "my"]`	`"dog"`	100%
`["my"; "dog"]`	`"is"`	100%
`["makes"; "a"]`	`"good"`	100%
`["a"; "good"]`	`"man"`	33%
	`"dog"`	66%
`["dog"; "is"]`	`"a"`	100%
`["and"; "a"]`	`"good"`	100%
`["good"; "dog"]`	`"makes"`	50%
	`"and"`	50%
`["dog"; "and"]`	`"a"`	100%
`["a"; "man"]`	`"and"`	100%
`["good"; "man"]`	`"STOP"`	100%
`["dog"; "makes"]`	`"a"`	100%

Comme vous pouvez le voir, nous utiliserons `"STOP"` comme marqueur de fin comme auparavant. Mais au lieu d'un seul `"START"`, nous utiliserons comme marqueur de départ un préfixe de la même taille que les autres, rempli de `"START"`.

Question 15

Tout d'abord, définissez `start : int -> string list` qui crée le préfixe de départ pour une taille donnée.

`start 0`

retourne

`[]`

`start 2`

retourne

```
[ "START" ; "START" ]
```

Question 16

Définissez `shift : string list -> string -> string list`. Elle supprime l'élément de tête de la liste et place le nouvel élément à la fin.

Par exemple

```
shift [ "A" ; "B" ; "C" ] "D"
```

retourne

```
[ "B" ; "C" ; "D" ]
```

et

```
shift [ "B" ; "C" ; "D" ] "E"
```

retourne

```
[ "C" ; "D" ; "E" ]
```

Question 17

Définissez `build_table_multi : string list list -> int -> followers_multi` qui construit le dictionnaire pour une longueur de préfixe donnée, en utilisant les deux fonctions précédentes. Notez que la fonction prend en entrée des listes de phrases et non de mots.

Conseil : Écrivez d'abord la fonction

```
build_table_multi_words :
     string list -> followers_multi -> int -> followers_multi
```

qui prend une liste de mots en entrée et ajoute les successeurs à la table en entrée.

Question 18

Définissez `generate_text_multi : ptable -> string list` qui génère une phrase à partir d'une `ptable` donnée. Inspirez vous de `generate_table_map`.

Puisque nous avons maintenant un découpage correct des phrases, vous pouvez générer des textes de plusieurs phrases, en choisissant aléatoirement de continuer à partir du début après avoir rencontré un `"STOP"`.

Testez ! Vous pouvez utiliser le fichier `data.ml` qui contient des textes pris sur le site gutenberg.org