Word Générator C++

Clément Guillo, Dimitri Meunier

16 mai 2019

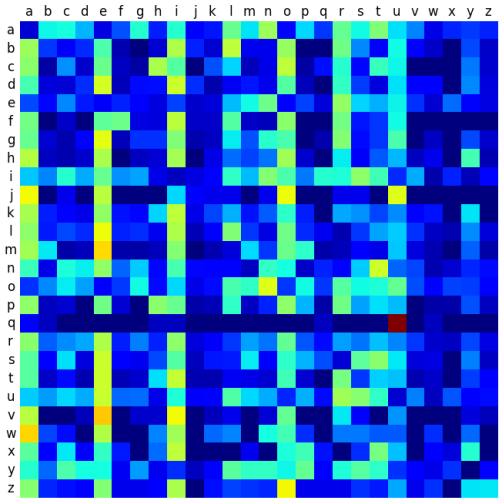
1 Introduction

Ce document, sert de descriptif au travail effectué dans le cadre du projet C++ générateur de mots. Dans un premier temps, on explicitera la démarche qui nous a mené au produit fini. Produit que nous détaillerons dans un second temps.

2 Fonctionnalités

L'idée derrière le générateur de mots est de calculer des probabilités de transition entre les lettres d'un alphbet (ou entplmpkmopkmjcvobpkjdfopbijre les mots d'un corpus), probabilités calculées empiriquement sur la base d'un ensemble de mots contenus dans un fichier. Ainsi, on aura par exemple si l'on travaille sur des mots français une probabilité d'arriver à la lettre s en partant de la lettre k très faible. On se sert alors de ces probabilités pour générer un mot. Ce principe est généralisable à des probabilités de transition entre les mots d'un texte ou bien à des probabilités de transition entre des groupements de lettres ou des groupements de mots de taille L.

FIGURE 1 – Probabilités de transition de l'alphabet français basées sur un dictionnaire de la langue française. Graphique généré à partir du Package *Python*, Matplotlib.



Ces probabilités de transition peuvent être modélisées comme celles d'une chaîne de Markov.
Soit E l'espace d'états de la chaîne de Markov que l'on note $(X_n)_{n\geq 0}$. Si on travaille sur du texte , $E = \{mot_1, \ldots, mot_n\}$ et si on travaille sur des mots $E = \{lettre_1, \ldots, lettre_n\}$
Le seul paramètre à fixer en amont est l'ordre de la chaîne de Markov, si on fixe l'ordre à L alors on fait l'hypothèse que

$$\mathbb{P}(X_n = x_n | X_{n-1} = x_{n-1}, \ldots, X_0 = x_0) = \mathbb{P}(X_n = x_n | X_{n-1} = x_{n-1}, \ldots, X_{n-L} = x_{n-L})$$

avec $(x_0, \ldots, x_n) \in E^n$

But du projet : apprendre les probabilités de transition de la chaîne de Markov à partir d'un corpus et s'en servir pour générer des mots et/ou du texte.

Ainsi, le rendu final est un générateur de mots ou de phrases. Dans les deux cas, le fichier en entrée est un fichier au format txt. Lors de l'écriture de la commande d'execution il faut spécifier si l'on veut que notre générateur fonctionne en mode "mot", auquel cas il génèrera des mots suite à l'apprentissage sur le fichier texte désigné, ou en mode "phrase" et se servira alors du fichier désigné pour générer des phrases. Il faut également spécifier lors du lancement le paramètre de mémoire L qui indique la taille des groupements de lettre ou de mots qui doit être considérée lors du calcul des probabilités de transition.

Un exemple de lancement sur console *./main ../data/EN.txt 3 0* Ici on charge le contenu du fichier "EN.txt" situé dans le dossier "data" avec un paramètre $L = 3$ et un fonctionnement en mode "mot".

3 Démarche : passage d'une vision matricielle à une vision graphe

L'utilisation d'une matrice de transition semble adaptée au problème de départ. En effet, prenons l'exemple de l'alphabet français : il suffit de générer une matrice carré M de taille 26 dont le terme m_{ij} représente le nombre de fois que la i-ème lettre de l'alphabet est succédée par la j-ème lettre de l'alphabet (comme sur la Figure 1). À partir de ces nombres d'occurences, en normalisant chaque ligne on peut calculer des probabilités de transition et ainsi générer des mots en tirant aléatoirement chaque lettre selon ces probabilités. Plusieurs problèmes se posent alors :

— Si l'on veut calculer des matrices de transition entre des groupements de lettres, il faut généraliser ces matrices de transition en cube ou hypercube de transition, ce que nous avons réalisé pour des groupements de lettre de taille 2 et 3 mais la généralisation à des tailles plus grandes rendait le modèle matriciel difficilement utilisable et il fallait réécrire le code pour chaque nouvelle dimension.

— Ces matrices peuvent être des objets très lourds si l'on parle de probabilité de transition entre les mots par exemple. Et,si on ne la crée pas de manière dynamique, beaucoup de cases peuvent être vides et occuper inutilement de l'espace. De plus la recherche de cases dans ces matrices peut prendre également beaucoup de temps.

Même si nous aurions pu éliminer une partie de ces problèmes en créant des classes de matrices (ou tenseurs) "sparse", ces constats nous ont conduit a changer de modèle de conception de notre générateur et a passer à un modèle de type graphe. Plutôt que de stocker les probabilités de transition de la chaîne de Markov sous forme matricielle il apparaissait plus efficace (en terme de mémoire) de stocker son graphe. Ceci étant fait, il n'y a plus qu'à l'initialiser et laisser notre chaîne naviguer dans le graphe pour générer aléatoirement mots ou phrases.

FIGURE 2 – Graphe d'une chaîne de Markov d'ordre 1 créé à partir du corpus banane-bois-loi

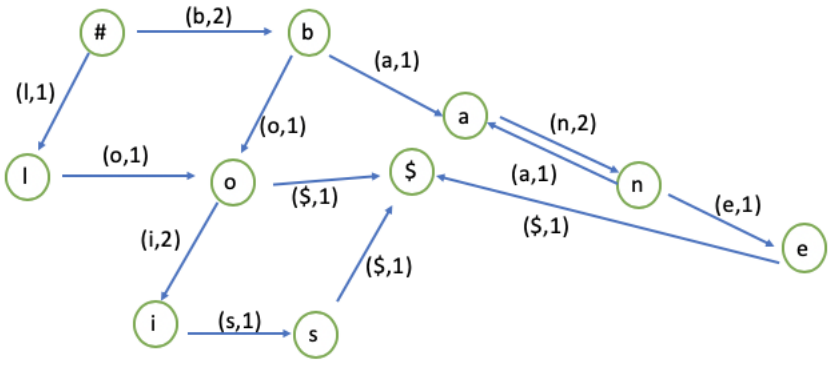
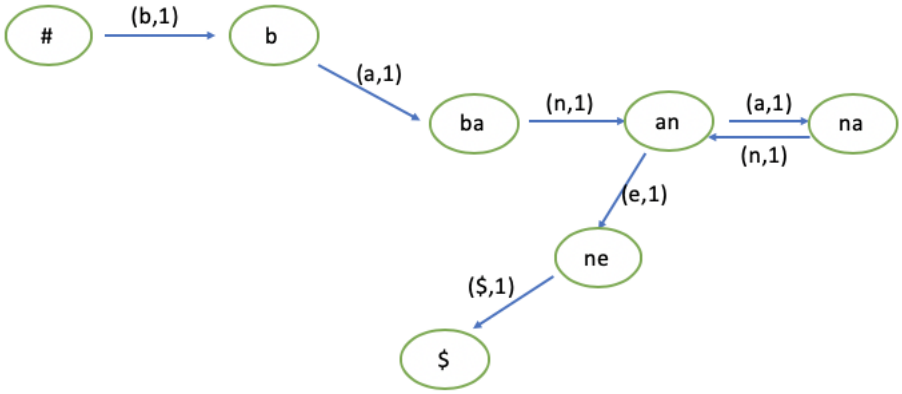


FIGURE 3 – Graphe d'une chaîne de Markov d'ordre 2 créé à partir du corpus uniquement composé du mot banane



Cette vision permet alors d'être plus dynamique et plus économe en espace mémoire lors du calcul des probabilités de transition puisque les états de notre chaîne sont créés seulement au moment où l'on observe leur existence. En outre il semble plus aisé avec cette méthode de garder en mémoire des groupements de lettres plus grands ou de remplacer nos lettres par des mots dans la mesure où il suffit seulement de changer l'ensemble d'appartenance des états de notre chaîne de Markov. Cette approche nous a donc permis d'écrire un unique code pour le traitement des mots et le traitement du texte et pour n'importe quel ordre de chaîne de Markov L. C'est donc cette dernière vision qui a été retenue pour la suite.

4 Structure du projet

Le code a été organisé selon la logique graphe présentée dans les paragraphes précédents. C'est pourquoi notre projet est articulé autour de 3 classes, les classes Vertex, Node et Automaton, que l'on va détailler ci-dessous.

Vertex : Un Vertex va représenter un des flux sortants d'un état de notre chaîne de Markov. Un vertex sera défini par le noeud vers lequel il pointe ainsi que par le nombre de fois qu'il pointe vers ce même noeud.

Attributs :

- *string nextNode :* Le nom du noeud pointé par le Vertex.
- *float weight :* Le nombre de fois que le noeud est pointé. Le type float utilisé permet de transformer ce nombre d'occurences en une probabilité une fois que le graphe de la chaîne de Markov est construit.

Méthodes

- *void increment() :* ajoute 1 à l'attribut weight.
- *void display() :* fonction d'affichage du Vertex.
- *float getweight() :* retourne le poids du Vertex.
- *string getNextNode :* retourne le noeud pointé par le Vertex.

Constructeur : Vertex(string nextNode, float weight) : prend en arguments, la chaîne de caractère nextNode et le float weight et se sert de ces arguments lors de l'instanciation d'un objet de la classe Vertex.

Node : Un node représente un des états (ou noeud) de notre chaîne de Markov. Noeud duquel partent un ensemble de Vertex. Ainsi un Node sera défini indirectement par l'ensemble des vertex qui en sortent.

Attributs :

- *map<string , Vertex> mapVertex :* Une map contenant l'ensemble des Vertex partant du Node. La clé de la map est la lettre avec laquelle on sort du noeud dans le cas d'un générateur de mots, ou le mot par lequel on sort dans le cas d'un générateur de texte. À titre d'exemple, si l'on travaille avec un générateur de mots et avec des noeuds de 3 lettres. Si on se trouve dans le Node nommé "abo" et qu'on sort de ce noeud par la lettre r, on va alors se retrouver sur le Node nommé "bor" et la map associée au Node "abo" contiendra r :("bor",1).

Méthodes

- *map<string , Vertex> getMap()* : Accesseur de la map de Vertex.
- *void add_map(string next , Vertex v) :* ajoute le Vertex v à l'attribut mapVertex.
- *void increment(string NextLetter)* : fait appel à la méthode increment() de la classe Vertex pour un des Vertex de l'attribut mapVertex. Cette méthode sera appelée lorsque l'on repasse d'un noeud déjà existant à un autre noeud déjà existant et permettra de compter ce passage supplémentaire dans le Vertex associé.
- *void display() :* affichage du Node, à savoir l'affichage de l'ensemble de ses Vertex.
- *string select_node_suivant() :* Cette méthode est appelée par la méthode generate_word() de la classe Automaton (voir ci-dessous). Elle permet en partant d'un Node et de l'ensemble des Vertex partant de ce dernier, de sélectionner aléatoirement le noeud suivant (ce qui revient à choisir un Vertex du Node) avec des probabilités proportionnelles au poids de chaque Vertex.

Constructeur : Node() : Dans le constructeur de node on initialise seulement une map de Vertex vide. Cette map sera alimentée par la suite lors de la création de notre Automate basé sur un ensemble de mots ou un corpus de texte. Ceci, de sorte à ce qu'elle contienne l'ensemble des façons possibles de quitter le noeud en cours.

Automaton : À partir d'un corpus de texte, une instance de la classe Automaton va stocker l'information sous forme de graphe. Les états peuvent être alors des groupements de L lettres concaténées dans le cas où l'on génère des mots ou des groupements de L mots séparés par un espace si l'on veut générer du texte. L représentant la mémoire de notre automate, i.e. le nombre de lettres ou de mots qui constituent un état. La définition d'un état dans cette section diffère légèrement de celle donnée dans la 2ème partie si on considère l'espace E définit précédemment et L l'ordre de la chaîne de Markov alors les états dont nous parlons ici seront toutes les parties de E à L éléments.

Attributs :

- *int memoryLength :* l'ordre de la chaîne de Markov (L)
- *int word :* Est égal à 1 si l'on veut que le graphe génère des mots, à 0 si l'on veut générer des phrases.
- *map<string,Node> mapNode :* une map d'éléments de la classe Node dont l'indexation se fait par une chaîne de caractères portant le nom du node. Cet attribut est en fait le graphe de Node que nous allons devoir alimenter par la suite à la lecture d'un corpus.

Méthodes :

- *void learnFromWord(string word) :* à partir d'un mot, cette méthode va venir alimenter le graphe en créant de nouveaux nodes si ceux ci n'ont pas été déjà créés, ou en créant de nouveaux vertex entre deux nodes pré-existants ou bien en incrémentant des vertex déjà existants (si on lit deux fois le même mot par exemple).
- *void add_map(string idNode, Node node) :* permet d'ajouter un Node dans l'attribut mapNode.
- *void display() :* fonction d'affichage d'une instance de la classe Automaton.
- *string get_init() :* permet d'initialiser le premier Node, pour le lancement de la méthode generate_word().
- *void generate_word() :* génère un mot à l'aide du graphe à espace d'états préalablement construit avec la méthode learnFromWord. Partant d'un noeud, des probabilités sont calculées pour chaque noeud suivant, on effectue donc un tirage du noeud suivant selon ces probabilités inégales. Cette fonction est récursive puisqu'on peut la rappeler sur le noeud ainsi sélectionné.

Constructeur : Automaton(string path, int memoryLength, int word) : le constructeur de la classe Automaton prend en entrée le chemin du fichier contenant une liste de mots ou un ensemble de phrases d'un corpus de texte, ainsi que la valeur de ses attributs memoryLength et word. À partir de ces arguments l'attribut mapNode est alimenté par le biais de la fonction learnFromWord.

5 Diagramme de Classe

Le diagramme de classe ci-dessous est assez simple mais permet d'apprècier rapidement la structure du projet.