Projet C++

Sujet choisi : Générateur de textes/mots. À l'aide d'un corpus de texte ou de mots on souhaite obtenir des textes/mots générés automatiquement à l'aide d'un modèle de chaîne de Markov.

Table des matières

Description du programme

L'objectif de ce programme est de créer des tweets qui aurait pu être écrit par Donald Trump. Pour ce faire on est parti d'un corpus de texte comportant de nombreux tweets réellement écrits par Donald Trump, à partir de cela en utilisant des chaines de Markov, on obtient alors des tweets fictifs mais syntaxiquement correct et proche du style de Trump.

Nous avons utilisé ce corpus de texte afin de rendre la simulation plus ludique. De plus la base étant assez fournis, cela nous permettait d'avoir un échantillon d'apprentissage intéressant pour la génération de phrases mais également sur le plan plus technique de travailler sur un gros volume de données.

Chaine de Markov

« En mathématiques, une chaine de Markov est un processus de Markov à espace d'états discret. Un processus de Markov est un processus stochastique possédant la propriété de Markov : l'information utile pour la prédiction du futur est entièrement contenue dans l'état du processus et n'est pas dépendante des états antérieurs (le système n'a pas de « mémoire »). » Wikipédia

Généralement la chaine de Markov est accompagnée d'une matrice de probabilité de transition, dans notre cas nous n'avons pas directement créer de matrice. En créant un dictionnaire, composé d'une clef faites d'un doublon de mot, et d'un vecteur qui contient l'ensemble des mots qui suivent ce doublon, d'après le corpus de texte initiale. Dans ce vecteur, il peut y avoir plusieurs fois le même mot, car chaque fois que l'on rencontre le doublon dans le corpus de texte, on ajoute le mot qui suit dans le vecteur. C'est en conservant les répétitions que l'on compense l'absence de matrice.

Lors de la génération de phrase on vient piocher dans le dictionnaire le mot suivant, on rentre une clef, et on prend aléatoirement le mot suivant dans le vecteur associé à la clef.

De plus, une chaine de Markov s'accompagne en général de probabilités initiales qui permettent de déterminer dans quel état se trouve la chaine à l'instant 0. Nous avons déterminé une liste des débuts de tweets de D.Trump avec aussi des répétitions qui permet de ne pas calculer de probabilités initiales.

Critique des problèmes rencontrés et des solutions adoptées

Plusieurs problèmes sont apparus lors de la création du programme.

- **Retraitement d'un fichier texte**

Nous travaillons avec un fichier non structuré contenant des milliers de caractères, afin de travailler convenablement il nous a fallu trouver un moyen de réorganiser les données de façon plus structurée. Après beaucoup de recherches nous avons trouvé une solution, par décomposition caractères par caractères.

- **Génération d'un chaine de Markov**

Ensuite ce fut la mise en place de la méthode de Markov et la matrice de probabilités de transition, qui fut compliqué à mettre en place. Pour ce faire, après plusieurs essaies infructueux via des vecteurs, nous avons décidés d'utiliser un dictionnaire, dit « map », permettant une gestion plus simple des outputs et des inputs, mais nous a aussi permis de ne pas créer de matrice de transition, comme expliqué précédemment. La map elle-même ne fut pas simple à générer puisque nous avions alors des objets assez complexes tels que des map<string,<vector<string>> mais le temps que nous avons passé à élaborer ces objets a faciliter tous le développement du programme ensuite.

- **Une programmation objet claire et concise**

Nous avons ensuite souhaité créer des classes indépendantes, claires et précises, cela ne nous a pas posé de problèmes particuliers, mais à demander du temps et des recherches. Nous avons aussi cherché à rendre cohérent le programme en appliquant des principes de la programmation objet tels que l'héritage. Notre objectif était donc de ne pas s'encombrer avec d'innombrables objets mais que chaque objet fasse un travail simple pour que le code soit lisible mais également modifiable de façon indépendante. Nous avions commencé par écrire de nombreuse ligne de codes dans le main puis voyant que la debbugage n'était pas simple nous avons opté pour l'élaboration de classes ce qui nous a également aidé dans les nombreuses heures de débogage que nous avons fais puisque nous pouvions détecter de quelle classe provenait les erreurs.

- **Temps de calcul**

Enfin le dernier problème fut celui du temps de processus. Nous avons en effet construit principalement le programme sans faire appel aux pointeurs. La génération du dictionnaire et ensuite son utilisation étaient très longs, car le dictionnaire fait plusieurs milliers de lignes. En introduisant des pointeurs aux endroits pertinent, le temps de process s'est nettement diminué.

En effet, ce projet nous a bien permis de comprendre l'utilité des pointeurs et de la gestion de la mémoire. En effet, notre programme contient des objets très volumineux et à chaque appel dans une méthode, le programme faisait une copie de ces objets. En comprenant ça, nous avons introduit des pointeurs pour faire référence à ces objets et nous avons pu optimiser le code qui prenait près de 10 minutes au départ pour arriver à 2 min avec l'utilisation des pointeurs.

Description de l'architecture générale du programme

Le programme est décomposé principalement de deux classes:

-Decomposition

- Generator

Ces fichiers sont ensuite utilisés dans le fichier nommé « ProjetMain » afin de générer les tweets.

Les fichiers sources ont été pensé comme des fichiers pouvant être utilisés dans d'autres programmes et donc indépendants de notre programme.

Ces classes vont nous permettre par la suite de créer des phrases en créant des chaines de Markov. La classe décomposition nous permet d'obtenir toute la matière première en retravaillant le fichier source (tokenization, création des dictionnaires, etc…). Le second fichier texte peut être vu comme la boîte à outils (départ de la chaine, transition, etc…).

Decomposition

La classe décomposition prend en entrée n'importe quel fichier texte afin d'en extraire deux dictionnaires et une liste. Ces éléments sont utiles pour la création d'une chaine de Markov, cependant cette méthode de décomposition peut être utilisée au sein de d'autres méthodes.

Le premier dictionnaire a pour clef une suite de deux mots, et pour lien un vecteur comportant l'ensemble des mots qui les suivent dans le corpus de texte de base.

Le second ressence l'ensemble des fins de phrases, avec pour clef les deux avant derniers mots, et pour lien l'élément final.

Le vecteur lui est composé de paires formant un début de phrase et dont on est sûr qu'une suite est possible.

Ces dictionnaires et vecteurs vont nous assurer la création d'une phrase cohérente.

- ***Ensemble des attributs*** :
  - `std::string m_chemin`
    Un des seuls attributs de la classe est le chemin qui permet au programme de travailler sur le fichier texte. Déclaré en protected pour respecter un principe de la programmation objet, l'utilisateur ne doit pas pouvoir le modifier.

- ***Ensemble des méthodes*** :
  - `Decomposition(string chemin)`
    Déclatation du chemin

  - `ouvertureFichier(ifstream &inFile)`
    Ouverture du fichier grâce au chemin indiqué

  - `mrMrs(std::string word)`
    Cas particulier lors de la décomposition du corpus de texte, afin de ne pas prendre en compte les ponctuations de ces cas comme fin de phrases.

- `f_listeDebut(std::vector<std::vector<std::string>> &listeDemarrage, std::string mot1, std::string mot2 )`
  Liste comprenant l'ensemble des débuts de phrases présent dans le corpus de texte initial.

- `f_dicoFin(map<string, string> &dicoFin, std::string mot1, std::string mot2)`
  Liste comprenant l'ensemble des fins de phrases présent dans le corpus de texte initial.

- `dictionnaire(map<string, vector<string>> & dico, map<string, string> &dicoFin, vector<vector<string>> &listeDemarrage)`
  Création du dictionnaire. Clef : deux mots consécutifs dans le corpus, lien : vecteur des mots associés à la clef trouvés dans le corpus de texte initial.

Generator

La classe source generator, est la boîte à outil de ce programme, et prend en entrée la décomposition du fichier.

Cette classe est la classe fille de la classe Decomposition. L'héritage nous permet en effet de déclarer des objets Decomposition dans la classe Generator et d'hériter des méthodes de la classe Decomposition. Cela nous est bien utile puisque la classe Generator a besoin des dictionnaires crées dans la classe Decomposition pour créer des chaines markoviennes et générer des phrases.

Cette boîte à outil peut être utilisé dans d'autres programme, tant que les entrées sont correctes. Cependant, un lien est fait entre les deux fichiers, afin de pouvoir générer une phrase en limitant les étapes dans le Main.

Ce fichier sert à créer une phrase, dans un premier temps en choisissant un élément du vecteur comportant les éléments initiant les phrases du corpus de texte. Nous cherchons ici à démarrer la chaine de façon cohérente, la chaine comportant des répétitions nous n'avons pas besoin d'établir une distribution de probabilité.

La phrase étant commencé, on la continue, en cherchant une suite cohérente via le dictionnaire, qui contient pour chaque doublon une liste de mots suivants possibles. On passe de mot en mot en utilisant le principe de key/item des map, chaque mot précédant devenant une clé. Si jamais rien n'est trouvé, la phrase est terminée, et une nouvelle phrase est commencée.

Nous cherchons ici à générer un tweet, ce qui nécessite de limiter la taille de la phrase. Une méthode (finPhrase) a donc été créée pour contraindre la phrase, les derniers doublons de mots vont être analysé pour savoir s'ils peuvent mener à une fin de phrase, grâce au dictionnaire comprenant l'ensemble de ces doublons et de la suite associé. Si le doublon regardé ne propose pas de fin, il est alors modifié, dans le respect de la syntaxe et l'intelligibilité de la phrase.

L'ensemble des mots générés, sont dans un premier temps stocké dans un vecteur, qui se transformera en phrase à la fin.

- ***Ensemble des méthodes*** :
  `initialisationDonnee(map<string, vector<string>> &dico, map<string, string> &dicoFin, vector<vector<string>> &listeDemarrage)`

Initialisation des données, via l'appel à la classe déclaration

- `initialisationPhrase(vector<vector<string>> &listeDemarrage, string &clef, int &taille, vector<string> &phraseFinale, int & alea)`
  Fonction permettant de créer un début de phrase syntaxiquement cohérent. Liste de démarrage viable via la chaîne de Markov.

- `tirageDico(string clef, map<string, vector<string>> &dico, string &sortie)`
  Toujours sous principe de la chaîne de Markov, le dictionnaire a pour clef une suite de deux mots, et pour lien un vecteur comportant l'ensemble des mots qui les suivent dans le corpus de texte de base. En fonction de la clef, on continue la phrase en allant chercher le mot/ ponctuation suivant dans le dictionnaire.

- `finPhrase(vector<string> &phraseFinale, string clef, map<string, vector<string>> &dico, map<string, string> &dicoFin)`
  Permet de finir une phrase sous contrainte

- `generePhrase(vector<string> &phraseFinale, map<string, vector<string>> &dico, map<string, string> &dicoFin, vector<vector<string>> &listeDemarrage)`
  Génération de l'ensemble de la phrase syntaxiquement correcte.

- `phraseFinale(std::vector<std::string> &phraseFinale, std::string &leTweet)`
  Mise en forme sous forme de string pour afficher la phrase créée.

Main

Le main déclare l'ensemble des données, et se base sur les objets Décomposition et Generator. On déclare des objets Decomposition et Generator et on peut ainsi utiliser toutes les méthodes de ces classes.

Le projet est interactif, et demande à l'utilisateur le fichier source voulu, on peut donc choisir de créer des phrases qui ne sont pas des tweets de Trump, si l'on possède un corpus de texte.

Une fois le corpus de texte initial choisit, la génération du dictionnaire à lieu, cette partie est la plus chronophage, cependant grâce aux pointeurs cette partie a été optimisée, et ne dure à peine 2 minutes contre près de 10 minutes quand le programme n'était pas optimisé.

Une fois le dictionnaire créé, autant de phrases que voulues peuvent être formées. On donne la possibilité à l'utilisateur de créer d'autres phrases en entrant des commandes sur la console ou de s'arrêter s'il n'en a plus voulu. En tous les cas, les dictionnaires étant déjà générés lors du premier run, la génération d'autres phrases ne pose pas de problème en termes de calcul.

Durant l'ensemble du programme, plusieurs vérifications sont faites afin d'éviter les erreurs et l'arrêt du programme.