Rapport Algorithme et programmation II

Charles Vin

Décembre 2020

Table des matières

1 Recette et Ingredients **3**
1.1 Démarche générale 3
1.2 Spécification et documentation des fonctions 3
1.2.1 Prétraitement 3
1.2.2 Recipe Chooser 3
1.3 Etude de la complexité 4

2 L'autocomplétions **4**
2.1 Démarche générale 4
2.2 Spécification et documentation des fonctions 5
2.3 Etude de la complexité 6

3 Interface **6**

Mon projet porte sur un mélange entre le sujet d'autocomplétions proposé et sur un algorithme de suggestion de recette à partir d'une liste d'ingrédients. Les ingrédients sont proposés et autocomplétés puis les recettes les plus pertinentes sont ouvertes dans le navigateur. La saisie se fait dans une interface tkinter.

La base de données contenant les recettes et les ingrédients vient du site Kaggle et se nomme *Recipe Ingredients and Reviews*. Elle est disponible ici.

1 Recette et Ingredients

1.1 Démarche générale

La première étape fut de nettoyer la base de données. Elle fournissait déjà un fichier nettoyé des recettes mais celle-ci compotaient malgré tout quelques ingrédients atypiques. On retrouvait beaucoup d'ingrédients avec une correspondance en pound, tablespoon, cups, etc. La liste des ingrédients avant nettoyage est disponible dans le fichier *ingredient_ list.csv* .

Ainsi, chaque mot indésirable de chaque ingrédients a été éliminé de la base de données. Puis dans un même temps, j'ai créé un petit Set de 4 recettes me permettant de tester l'algorithme du choix de recettes.

Cet algorithme prend en entrée une liste d'ingrédients et renvoie une liste de recettes les plus proches de ces ingrédients. Pour cela, j'ai associé chaque ingrédients à une liste de recettes dans lequel il est utilisé. Ainsi, pour obtenir un Set de recette contenant au moins un ingrédient choisis par l'utilisateur, il suffisait de fusionner la liste associé à chaque ingrédient.
Mais maintenant comment choisir la recette la plus proche de mes ingrédients ? Pour attribuer un score par recette, plusieurs méthodes s'offraient à moi et sont proposées à l'utilisateur :

- Méthode 1 : Privilégier les recettes où l'utilisateur possède le plus d'ingrédient **en prenant en compte** ceux qu'il n'a pas. C'est à dire faire un ratio entre le nombre d'ingrédients possédé et le nombre d'ingrédients nécessaire pour faire la recette
- Méthode 2 : Privilégier les recettes où l'utilisateur possède le plus d'ingrédient **sans prendre en compte** ceux qu'il n'a pas. C'est à dire simplement compter le nombre d'ingrédient possédé par recette

Une fois un score associé à chaque recette, il a donc suffi de renvoyer les dix avec le plus grand score.

1.2 Spécification et documentation des fonctions

1.2.1 Prétraitement

Dans cette section, nous traiterons du fichier *Prétraitement.py* et ses fonctions. Les docstings fournissent déjà une documentation détaillée, ici je vais décrire l'implication de chaque fonctions dans la démarche globale.

- kToInt : Cette fonction a permis de convertir la colonne *Review Count* de la base de donnée entièrement au format Int, convertissant les *1k* commentaire en *1000*.
- ingredientCleaner : Cette fonction permet de supprimer les mots indésirables des ingrédients.
- getCleanCsvIngredientList : Convertie la colonne *Ingredients* de la base de données en une liste de liste d'ingrédients tous nettoyés. Cette fonction utilise un dictionnaire pour mémoriser et associer un ingrédient nettoyé avec son ingrédient original. Evitant ainsi de passer chaque ingrédient de chaque recette dans toutes les conditions de la fonction *ingredientCleaner*.
- ingredientListOfList2set : Créé un Set de tous les ingrédients de toutes les recettes de la base de donnée. Elle intervient après *getCleanCsvIngredientList* car elle prend en paramètre la liste des ingrédients de chaque recette (liste de liste).
- getCleanIngredientList : Permet de nettoyer le Set d'ingrédient créé par *ingredientListOfList2set*.

L'obtention d'une liste d'ingrédients propre est importante pour l'autocomplétions.

1.2.2 Recipe Chooser

Dans cette section, nous traiterons du fichier *Recipe_ chooser.py* et ses fonctions. Les docstings fournissent déjà une documentation détaillée, ici je vais décrire l'implication de chaque fonctions dans la démarche globale.

— createRecipesPerIngredient : Une des fonctions les plus importantes, elle permet d'associer chaque ingrédient à toutes les recettes qui le contiennent. Exemple : {'flour', 'egg'} -> {'flour' : {1,2,3,4}, 'egg' :{2,5}}
— frqIgrd : A partir du dictionnaire créé par la fonction précédente, calcul le nombre d'occurrence de tous les ingrédients et le place dans un dictionnaire pour l'étape d'autocomplétion.
— getRecipe : Renvoie une pandas.Series correspondant à la recette demandé. L'ID de la recette correspond à sa ligne dans le csv.
— recipe_chooser : Cette fonction forme le cœur de l'algorithme. Elle met en lien toutes les fonctions du fichier pour renvoyer la liste des 10 recettes les plus proche en fonction des ingrédients disponibles. Cette liste est sous la forme d'une pandas DataFrame avec les liens photos sous la forme d'une image HTML.

Les fonctions *score* permettent d'attribuer un score à une recette en fonction des ingrédients disponible et des ingrédients de la recette. La méthode dépend de la fonctions :

— score : Fait un ratio nombre d'ingrédient disponible pour cette recette/nombre total d'ingrédient de la recette. Utilise *getRecipe* pour obtenir les ingrédient de la recette en question.
— score2 : Compte simplement les ingrédients disponibles parmis ceur nécessaire pour faire la recette. Utilise *getRecipe* pour obtenir les ingrédients de la recette en question.
— sortByScore : Trie les valeurs d'un dictionnaire dans l'ordre décroissant. Je l'utilise pour afficher les recettes les plus pertinentes mais également dans l'autocomplétions pour trier le dictionnaire des fréquence de mot.

Ces fonctions suivantes permettent, à partir du dictionnaire de *createRecipesPerIngredient* et de la liste des ingrédients disponibles, d'associer chaque recette à un score. Pour cela, elles font l'union des Set de recette de chaque ingrédient et le parcourt recette par recette. Elles utilisent différentes méthodes qui dépendent assez de l'intention de l'utilisateur.

— recipeIntersection : Renvoie les recettes qui contiennent exclusivement tous les ingrédients demandés.
— recipesScore : Utilise la fonction *score* pour calculer les scores
— recipesScore2 : Utilise la fonction *score2* pour calculer les scores

1.3 Etude de la complexité

Nous allons étudier la complexité de l'algorithme dans sa globalité en observant celle des principales fonctions et les éventuelles possibilités d'améliorations.

— createRecipesPerIngredient : En considérant les Sets fonctionnant comme des tables de Hachage où le calcul des clés est de temps constant, ajouter un élément serait d'une complexité $O(1)$. Puis en prenant m le nombre moyen d'ingrédient dans une recette, n le nombre de recette, la complexité de cette fonction est linéaire $O(nm)$.
— La complexité de *recipeScore* varie en fonction de la taille de la liste d'ingrédient disponible que nous appellerons n. L'union des Sets est d'une complexité égale à la somme de la taille des Sets à réunir. Elle est donc linéaire, disons $O(p)$.
Puis on calcule le score de chaque recette, les fonctions score ont une complexité linaire variant en fonction de n (la taille de la liste d'ingrédient disponible). On appel la fonction *score* dans le pire des cas p fois. Ou dans le cas moyen q fois avec q le nombre de recette moyen par ingrédient.
Ce qui nous donne pour les fonctions *recipeScore* une complexité $O(np)$ dans le pire des cas ou $O(nq)$ dans le cas moyen. On pourrait légèrement améliorer sa vitesse en faisant l'union et le calcul du score dans la même boucle mais la complexité resterait linéaire.

L'algorithme des recettes s'alignerait donc sur une complexité linéaire dépendant de beaucoup de variables. L'utilisation de Set au lieu de liste pour simplifier la lisibilité et la simplicité du code ne semble pas vraiment influer sur sa complexité, les Sets étant juste des tables de Hachage.

2 L'autocomplétions

2.1 Démarche générale

La représentation d'une Node sous forme d'une classe permet une visualisation de la trie et une lecture de code simplifiée. Chaque Node possède 4 propriétés :

1. Sa lettre
2. Un dictionnaire de ses nodes enfants

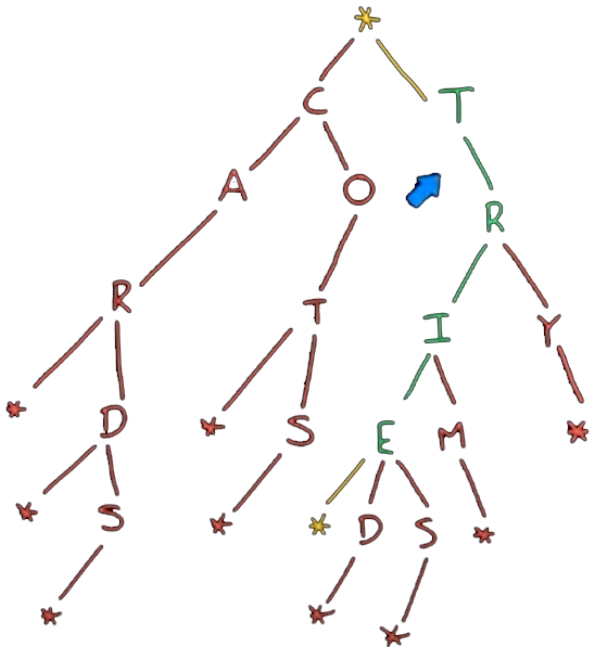

FIGURE 1 – Modèle d'une trie

3. Si elle représente un mot (ie. eles nodes étoiles dans la Figure 1), on stocke le mot en question
4. Si oui avec la fréquence de ce mot

Pour construire la trie, je parcours chaque lettre de chaque mot et regarde jusqu'où les nodes ont déjà été créés, si la lettre n'existe pas, je l'ajoute. Puis je recommence jusqu'à la fin du mot où j'ajoute une node étoile avec les caractéristiques associées au mot. Grâce à ces nodes étoiles, déduire une liste de mot possible sous une node est simple. Il suffit de chercher toutes les nodes étoiles en parcourant la trie sous la node puis de les trier en fonction de leur fréquence.

2.2 Spécification et documentation des fonctions

Dans cette section, nous traiterons du fichier *Autocomplete.py* et ses fonctions. Les docstings fournissent déjà une documentation détaillée, ici je vais décrire l'implication de chaque fonctions dans la démarche globale.

- Node.appendChild : Ajoute une node enfant (ie. une entrée dans le dictionnaire) reçu en paramètre à la node self.
- rechercheNode : Recherche la node correspondant ayant pour chemin le mot en paramètre. S'il la trouve, il la renvoie accompagnée d'un booléen True, sinon il renvoie la dernière node trouvée et un False. Le choix de renvoyer deux variables permet d'éviter de relancer une recherche sur le mot moins une lettre lors de la construction de la trie.
- buildTrie : cette fonction construit une trie à partir d'une liste de mot et d'un dictionnaire de fréquence. Pour cela elle parcourt chaque lettre de chaque mot et regarde si la node existe pour cette lettre. Si oui elle passe à la lettre suivante et descend dans la trie, sinon elle crée la node correspondant à la lettre manquante. Dans les deux cas on garde en mémoire la dernière node utilisée pour ajouter une node étoiles lorsque le mot a été créé
- printTrie : Cette fonction a surtout servi lors des tests. Elle permet de visualiser une trie sous une forme indentée avec des *print()*.
- getWord : est une fonction récursive qui explore toutes les sous nodes de celle fournis en paramètre et qui renvoie une liste des mots trouvés sous celle-ci. C'est elle qui va renvoyer la liste des suggestions en fonction du mot tapé.
- recherche : La fonction *recherche* utilise toutes les fonctions précédentes. Elle prend le mot tapé par l'utilisateur en entrée et renvoie la liste de suggestions triée en fonction de la fréquence des mots. Pour cela elle va d'abord chercher la node du mot en paramètre dans la trie(fonction *rechercheNode*). Si elle ne la trouve pas alors le mot n'existe pas et elle return False, sinon elle extrait tous les sous-mot de cette node (fonction *getWord*) puis renvoie cette liste triée en fonction de la fréquence d'occurrence des mots (fonction *sortByScore*).

2.3 Etude de la complexité

L'utilisation de liste de mot bien plus grande pour la création de la trie peut rapidement engendrer des temps de création de celle-ci bien plus long. Cette structure de donnée étant de base optimisée pour une recherche rapide, il me semble plus intéressant d'étudier la complexité au moment du build plutôt que lors de la recherche de mot. Nous allons donc étudier la fonction *buildTrie* et *rechercheNode*. Le nombre de tours de boucle de ces deux fonctions dépend du nombre de lettres du mot.
Pour un mot de taille m et dans le pire des cas, c'est à dire si le mot existe dans la trie. La fonction *rechercheNode* effectue alors m fois une condition (try/except) et une assignation. Ce qui donne en moyenne moins de $2m$ opérations élémentaires.
Dans la boucle des lettres, la fonction *buildTrie* appelle la fonction *rechercheNode* avec une lettre de plus à chaque fois. C'est à dire pour un mot de m lettres : $2+4+...+2m = 2(1+2+...+m) = 2*\frac{m(m+1)}{2} = m^2+m$ opérations élémentaires pas tour de boucle mot. Rajoutons à cela les $5m$ assignations, on obtient pour un mot de m caractères $m^2 + 6m$ dans le pire des cas (complexité maximum, majoration). Prenons une moyenne de 9 caractères par mots (source). Et donc avec une liste de n mot on obtient $T(n) = n*(m^2+6m) = nm^2+mn = 18n+54n = O(n*m^2)$, c'est à dire une complexité quadratique (polynôme de degré 2) $T(n) = m^2 + 6m = O(n^2)$.
On pourrait améliorer cette complexité en évitant de chercher les nodes lettre par lettre mais en les ajoutant directement de manière récursive ! C'est ce que j'ai fait dans la fonction *buildTrie2*. Pour un tour de boucle mot elle fait $T(n) = n(2m+2) = 2mn+2n = O(mn)$ opérations élémentaires, c'est à dire une complexité linéaire.

D'autres pistes d'améliorations sont possibles. Nous avons étudié la compléxité temporelle lors de la création de la trie. Mais en commençant par ne pas stocker le mot entier dans une node spécialisée (étoile), on pourrait améliorer la mémoire utilisée par la trie. On pourrait également compresser la trie en fusionnant chaque noeud n'ayant qu'un seul enfant et ainsi optimiser la place en mémoire et peut être également améliorer la vitesse d'execution de la recherche en dépit de celle de construction.

3 Interface

L'interface n'étant pas le but de l'exercice, je ne vais pas m'attarder dessus. L'interface vas venir lier l'autocomplétions au *recipe_ chooser*. La Class AutoComplete fut inspirée de celle de cet utilisateur de stackexchange et adaptée au projet.

Par un appuis de la touche *Entrée* ou sur le bouton correspondant, l'utilisateur peut ajouter un ingrédient à la liste. Si celui si n'est pas dans les suggestions ou est déjà dans la liste il ne peut pas être ajouté. C'est la fonction *addIgrd* qui gère cette action.
Pour valider sa liste, l'utilisateur appuis sur le bouton *Find Recipe !* qui appelle la fonction *printRecipe* qui vas convertir la DataFrame de recette reçu au format html pour ensuite l'ouvrir dans le navigateur de l'utilisateur.