Café sans-fil : Système de recommandation

Rapport de projet dans le cadre du cours :
IFT 3150 - Projet d'informatique

Auteur : Bio Samir Gbian (20250793)

Superviseur : Louis-Edouard LAFONTANT

Department d'Informatique et de Recherche Opérationnelle (DIRO)
Université de montréal
9 août 2024

Table des matières

Table des figures

Listings

1 Introduction

1.1 Contexte

Afin d'améliorer le service des cafés étudiants de l'Université de Montréal (UdeM), le projet Café sans-fil a été initié au trimestre d'automne 2023 durant lequel furent construits l'infrastructure backend et un premier prototype de l'application. Le projet a été poursuivi au trimestre d'hiver durant lequel l'application fut évaluée et enrichi par l'ajout d'éléments à caractères sociaux tel que les évènements, reflétant mieux la nature sociale des cafés étudiants. Le lancement de la plateforme est prévue pour la fin de cet été, intégrant le travail réalisé à l'hiver et des améliorations mineures additionnelles.

1.2 Problématique

Cependant, faute d'exploitation des données fournis entre autre par les utilisateurs, l'expérience utilisateur actuelle manque grandement de personnalisation, impactant la satisfaction client. En effet, si un étudiant possède certaines restrictions ou préférences alimentaires ou des allergies, la plateforme ne l'aide nullement à satisfaire ses exigences (critères), résultant sur un plus lourd travail de recherche, pouvant décourager certains. Cette lacune a aussi des conséquences sur les cafés, impactant les ventes et la fidélisation des clients ainsi que la prise de décision en vue d'une quelconque amélioration de leurs services et du menu. En effet, sans analyse de données, il est difficile pour les gérants d'optimiser le menu ou les services pour éviter des pertes et mieux répondre aux clients du café.

1.3 Proposition

En réponse aux problèmes énoncés, ce projet vise à mettre en place un système de recommandation utilisant les données fournies par les utilisateurs. Pour créer un moyen efficace de collecte et d'analyse de données nécessaire au système de recommandation, nous envisageons enrichir la plateforme avec de nouvelles fonctionnalités permettant aux utilisateurs de communiquer leurs préférences et réagir avec plus de choix aux propositions des cafés.

Ce projet est motivé par l'ajout des fonctionnalités suivantes :

- **Amélioration de l'expérience décisionnel du gérant** : En effet, plus il y a de données disponibles, plus nous pouvons informer le gérant des possibles améliorations à faire au niveau du menu. Une profile permettant à l'utilisateur de spécifier ses préférences permettront au gérant de savoir quel item devrait être rajoutés ou retirés du menu à long terme.

- **Amélioration de la recherche d'items** : Pour un utilisateur, il peut être parfois pénible de choisir un item respectant ses préférences sans connaître tous le menu. Ainsi, l'ajout d'une fonctionnalité de recommendation facilliterait cette étape de recherche.

Cafe sans fil est un projet open source. Le code peut être retrouvé sur *github*.

2 Système de recommendation

2.1 Introduction et définition

À l'ère numérique actuelle, la vaste quantité d'informations et de choix disponibles peut être écrasante pour les utilisateurs, qu'ils naviguent en ligne pour acheter des produits, choisir un film à regarder ou sélectionner un restaurant pour dîner. Les systèmes de recommandation sont devenus des outils essentiels pour aider les utilisateurs à naviguer dans cette abondance d'options en leur suggérant des éléments qui correspondent à leurs préférences et à leurs besoins. Ces systèmes sont largement utilisés dans divers secteurs, tels que le commerce en ligne, le divertissement et les réseaux sociaux, où ils jouent un rôle crucial dans l'amélioration de l'expérience utilisateur et le renforcement de l'engagement.

Ainsi, un système de recommandation est une application logicielle qui prédit et suggère des éléments aux utilisateurs en fonction de leurs préférences, comportements et interactions. Ces éléments peuvent varier, allant des produits, services et contenus aux connexions sociales, voire aux expériences. L'objectif principal d'un système de recommandation est de filtrer de grandes quantités de données et de fournir des recommandations personnalisées qui sont les plus pertinentes pour l'utilisateur individuel. Les systèmes de recommandation fonctionnent grâce à différentes techniques, telles que le filtrage collaboratif, le filtrage basé sur le contenu et les approches hybrides qui combinent plusieurs méthodes qui sont présentées dans la sous section suivante.

2.2 Algorithmes

Dans cette section, je présente les principaux algorithmes de recommendation utilisés dans le cadre du projet. Pour chacun de ces algorithmes nous verront comment il fonctionne ainsi que ses points fort et ses points faibles.

2.2.1 Filtrage collaboratif

Le filtrage collaboratif [**?**] est une technique de recommandation qui repose sur l'idée que les utilisateurs ayant montré des préférences similaires dans le passé auront des préférences similaires à l'avenir. L'algorithme peut être divisé en deux sous-catégories principales : le filtrage collaboratif basé sur les utilisateurs (user-based) et le filtrage collaboratif basé sur les éléments (item-based).

Filtrage collaboratif basé sur les utilisateurs
L'idée derrière le filtrage collaboratif basé sur les utilisateurs est de recommander à un utilisateur des éléments qui ont été appréciés par des utilisateurs similaires. La similarité entre les utilisateurs peut être mesurée par des techniques telles que la corrélation de Pearson ou le cosinus de l'angle entre les vecteurs d'attributs des utilisateurs ou encore la similarité jaccard :

$$Jaccard(x, y) = \frac{\sum_{i=1}^{n} X[i] \wedge Y[i]}{\sum_{i=1}^{n} X[i] \vee Y[i]} \tag{1}$$

où Jaccard(x, y) est la similarité entre l'utilisateur x et l'utilisateur y. X correspond à une liste d'attribut propre à l'utilisateur x et Y correspond à une liste d'attribut propre à l'utilisateur y. Ainsi, la similarité est calculé en prenant le quotient entre le nombre d'attributs que les deux utilisateurs ont en commun et le nombre total d'attributs.

$$\rho(u,v) = \frac{\sum_{i \in I_{uv}} (r_{u,i} - \overline{r}_u)(r_{v,i} - \overline{r}_v)}{\sqrt{\sum_{i \in I_{uv}} (r_{u,i} - \overline{r}_u)^2} \sqrt{\sum_{i \in I_{uv}} (r_{v,i} - \overline{r}_v)^2}} \tag{2}$$

où $\rho(u,v)$ est la similarité entre l'utilisateur u et l'utilisateur v, $r_{u,i}$ est la note de l'utilisateur u pour l'élément i, et I_{uv} est l'ensemble des éléments évalués par les deux utilisateurs u et v.

$$cos(X,Y) = \frac{\langle X, Y \rangle}{||X||||Y||} \tag{3}$$

où $cos(X,Y)$ est la similarité entre le vecteur d'attributs X de l'utilisateur x et le vecteur d'attributs Y de l'utilisateur y.

Filtrage collaboratif basé sur les éléments
Le filtrage collaboratif basé sur les éléments recommande des éléments similaires à ceux qu'un utilisateur a déjà appréciés. La similarité entre les éléments peut être calculée de manière analogue à la similarité entre les utilisateurs, en utilisant des mesures comme la corrélation ou la distance cosinus ou la similarité jaccard.

Cet algorithme fait partie des algorithmes les plus utilisés pour effectuer des recommendations. Il est souvent utilisé pour faire des recommendation de films (sur Netflix par exemple) à des utilisateurs basé sur le profile des personnes amis et/ou similaire à l'utilisateur à qui les recommendations sont faites. Il est également présent dans les recommendation faite au niveau des réseaux sociaux comme instagram, X(Twitter), TikTok. [**?**]

2.2.2 Filtrage basé sur le contenu

Le [**?**] filtrage basé sur le contenu repose sur les caractéristiques des éléments eux-mêmes pour effectuer des recommandations. Donc, il n'a besoin que des informations liées à l'utilisateur (historique de visionnage ou d'achat, items aimés...). L'idée est de recommander à l'utilisateur des éléments similaires à ceux qu'il a déjà appréciés en se basant sur les attributs des éléments. Par exemple, dans un système de recommandation de films, on pourrait représenter chaque film par un vecteur de caractéristiques ou d'attribut (comme le genre, les acteurs ou le réalisateur). Le système recommande alors des films similaires à ceux que l'utilisateur a déjà notés positivement, en utilisant des mesures de similarité.

2.2.3 Recommendation basé sur les connaissances

Les [**?**] systèmes de recommandation basés sur les connaissances utilisent des informations explicites sur les préférences et les besoins des utilisateurs pour faire des recommandations. Contrairement au filtrage collaboratif ou au filtrage basé sur le contenu, qui dépendent des données historiques d'utilisation, les systèmes basés sur les connaissances utilisent des règles ou des contraintes spécifiques pour effectuer des recommandations.

Par exemple, un système de recommandation de voitures pourrait utiliser des informations sur les besoins spécifiques d'un utilisateur (comme la taille de la voiture, le budget, et les préférences en matière de carburant) pour recommander des voitures adaptées.

L'une des approches les plus courantes est la correspondance entre les caractéristiques de l'utilisateur et celles des éléments, en utilisant des règles définies par les experts du domaine. Par exemple, si un utilisateur indique qu'il a un budget limité et préfère une voiture économique, le système recommandera des voitures correspondant à ces critères.

2.2.4 Avantages et Inconvénients

Chaque méthode de recommandation a ses propres avantages et inconvénients. Le filtrage collaboratif est puissant mais souffre du problème de démarrage à froid (cold start), où il est difficile de faire des recommandations pour de nouveaux utilisateurs ou de nouveaux éléments. Le filtrage basé sur le contenu peut contourner ce problème, mais il peut parfois être trop restrictif, ne recommandant que des éléments très similaires à ceux déjà appréciés par l'utilisateur. Les systèmes basés sur les connaissances, bien qu'ils puissent fournir des recommandations très pertinentes, nécessitent une connaissance experte et peuvent être difficiles à maintenir à grande échelle.

2.2.5 Systèmes hybrides

Un [**?**] système de recommandation hybride combine plusieurs approches de recommandation pour améliorer la qualité et la précision des recommandations. Contrairement aux systèmes de recommandation purement collaboratifs ou basés sur le contenu, les systèmes hybrides utilisent des techniques variées pour exploiter les forces et compenser les faiblesses de chaque méthode individuelle. Un exemple de système hybrique serait un système qui intégrerait les algorithmes de filtrage collaboratif, filtrage basé sur le contenu et filtrage basé sur les connaissances.

Cela permettrait de réduire le problème de démarage à froid tout en faisant des recommendations qui ne sont pas trop restrictifs. Un tel système aura aussi la possibilité de faire des recommendation basé sur les préférences de l'utilisateur.

En combinant plusieurs sources d'informations, les systèmes hybrides peuvent fournir une couverture plus large des éléments recommandés, augmentant ainsi la diversité des recommandations proposées aux utilisateurs.

Cependant, la mise en œuvre d'un système hybride peut être plus complexe en raison de la nécessité de gérer et d'intégrer différentes techniques de recommandation. Ce type de système peut-être plus difficile à optimiser en raison des différentes combinaisons possible d'algorithmes. Elle peuvent être également couteuse en terme de calcul et de stockage.

2.2.6 Autre algorithmes

Recommendation basé sur des models
Un [**?**] système de recommandation basé sur les modèles est un type de système de recommendation qui fournit des recommandations personnalisées aux utilisateurs en utilisant des modèles d'apprentissage automatique (réseaux de neurones, classification Bayésienne, machines à vecteurs de support) pour prédire une note qu'un utilisateur attribuerait à un article ou des items qu'un utilisateur aimerait par exemple. Après que le modèle a été entraîné, l'article avec le plus grand score est recommandé à l'utilisateur.

Une sous-catégorie des systèmes de recommandation basés sur les modèles est [**?**] le filtrage collaboratif basé sur les modèles. Il utilise des modèles pour prédire l'évaluation par les utilisateurs des éléments non notés. Il existe deux grands types de filtrage collaboratif par article : les modèles binaires qui se basent uniquement sur le fait qu'un utilisateur ait acheté/sélectionné/coché ou non un bien donné, et les modèles avec évaluations où les utilisateurs sont invités à noter les différents produits.

3 Conception

Dans cette section, je présenterai comment chacun des algorithmes de recommendation est intégré à l'application web *Café sans-fil*. La figure ci-dessous présente une vue haut niveau des différents algorithmes de recommendations. Chaque diagramme (représentant un algorithme) sera présenté dans les sections suivantes.

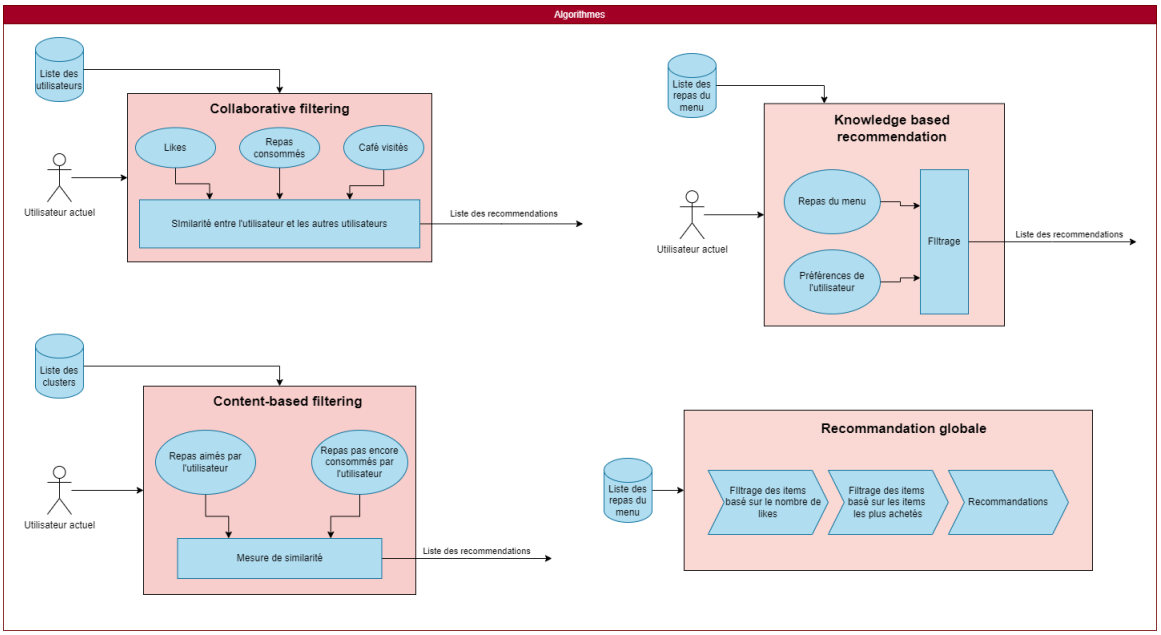

FIGURE 1 – Architecture des algorithmes

3.1 Recommendation public (globales) d'items

Les recommendations publiques s'addressent à toute la communauté étudiante ainsi que les professeurs. Elles se basent sur les items les plus aimés (likés) et les items les plus achetés par l'ensemble des utilisateurs. Ces recommendations sont affichées à titre indicatif pour l'utilisateur dans le sens où il pourra savoir, de façon global, qu'est-ce que les autres utilisateurs apprécient.

Cet algorithme fonctionne principalement par filtrage. En effet, pour un menu donné (un ensemble d'items), un premier filtrage est éffectué en conservant uniquement les items possédent le plus grand nombre de likes. Ensuite, parmi les items les plus aimés, un second filtrage est effectué pour conserver les items les plus achetés. Le nombre d'items conservé est limité à 10 pour ne pas surchargé l'interface au niveau du front-end. Voici un pseudo code présentant le fonctionnement de l'algorithme :

Algorithm 1 Recommandation globale

Description : Recommandations à tout les utilisateurs.
Entrée : Liste des repas du menu (M)
Sortie : Liste des recommandations

Algorithme(M[1...n]) :
    $k \leftarrow$ nombre de repas à recommander
    $I \leftarrow$ items les plus achetés
    retourner $items\_plus\_aimes(I, k)$

---

items_plus_aimes(I[1...n],k)) :
    $likes \leftarrow [vide]$
    $res \leftarrow [vide]$
    Pour chaque $item \in I$:
        likes.insérer(item.likes.length)
    Pour $i \leftarrow 0$ à k
        $max\_like \leftarrow max(likes)$
        $index \leftarrow likes.index(max\_likes)$
        $res.inserer(I[index])$
        $likes[index] \leftarrow -1$
    retourner res

3.2 Recommandations Personnalisées

Contrairement aux recommandations publiques, les recommandations personnalisées sont spécifiquement adaptées à chaque utilisateur. Elles s'appuient sur les trois algorithmes présentés précédemment dans la *section 2.2*. Les recommandations personnalisées se déclinent en deux catégories : les recommandations d'items et les recommandations de cafés. Le système de recommandations d'items suggère des articles aux utilisateurs en utilisant les trois algorithmes mentionnés. Pour les recommandations de cafés, le filtrage collaboratif est utilisé de manière similaire, à la différence que les *outputs* des recommandations sont des cafés. Cependant, le filtrage basé sur le contenu est partiellement appliqué aux café alors que les recommandations basées sur les connaissances ne sont pas appliqués aux cafés.

3.2.1 Filtrage Collaboratif

Dans le cadre de *Café sans-fil*, l'algorithme de filtrage collaboratif repose sur la similarité entre un utilisateur et les autres. La similarité entre deux utilisateurs est calculée en utilisant la *similarité Jaccard*. Les caractéristiques utilisées pour évaluer cette similarité incluent : les items aimés par l'utilisateur, les items achetés, et les cafés visités. Ces caractéristiques ont été sélectionnées car elles reflètent fidèlement les habitudes de consommation propres à chaque utilisateur. Une fois la similarité calculée sur la base de ce vecteur d'attributs, seuls les utilisateurs présentant une similarité supérieure à un certain seuil sont retenus. Ce seuil est fixé à 60% pour les données de test, mais il sera ajusté lors du déploiement sur des données réelles.
Si un item acheté par un utilisateur sélectionné n'a pas encore été acheté par l'utilisateur cible, un score (correspondant à la similarité entre les deux utilisateurs) est attribué à cet item. Pour les recommandations de cafés, ce score est attribué au café. Voici un pseudo code présentant le fonctionnement de l'algorithme :

Algorithm 2 Collaborative filtering

Description : Recommandations basé sur les similarités entre utilisateurs.
Entrée : Liste des utilisateurs de l'application (U), utilisateur (u)
Sortie : Liste L des recommandations

Algorithme(U[1...n], u) :
    $L \leftarrow [vide]$
    $\tau \leftarrow$ seuil de similarité
    $S \leftarrow U.retirer(u)$
    $L_u \leftarrow$ [[u.likes], [u.repas_consommés], [u.cafés_visités]]
    Pour chaque utilisateur $x \in S_{n\times 1}$
        $L_x \leftarrow$ [[x.likes], [x.repas_consommés], [x.cafés_visités]]
        $J \leftarrow [vide]$
        Pour $i \leftarrow 0$ à $L_x.length$
            $j \leftarrow Jaccard(L_u[i], L_x[i])$
            $J.ajouter(j)$
        $score \leftarrow sum(J)$
        Si $score \geq \tau$:
            $L \leftarrow (L_x[0] \cup L_x[1]) \backslash (L_u[0] \cup L_u[1])$
        $S.retirer(x)$
    retourner L

3.2.2 Filtrage Basé sur le Contenu

Dans le contexte de *Café sans-fil*, les recommandations basées sur le contenu, en ce qui concerne les items, se fondent sur les groupes d'items les plus appréciés ainsi que sur la similarité entre ces items. Lors du prétraitement des données, les items sont regroupés en fonction de leur similarité à l'aide de l'algorithme de clustering *KMeans*. Chaque cluster reçoit un score représentant le nombre d'items aimés par l'utilisateur dans ce cluster. Les clusters ayant le plus grand nombre de likes correspondent donc aux préférences de l'utilisateur. Ensuite, pour chaque cluster préféré, un score est attribué aux items de ce cluster que l'utilisateur n'a pas encore achetés, correspondant au score du cluster.
En ce qui concerne les cafés, aucun clustering n'est effectué. Seuls les cafés offrant le plus d'items appréciés par l'utilisateur sont proposés. Dans ce cas, le score représente le nombre d'items aimés par l'utilisateur dans un café. Le pseudo code de cet algorithme peut être retrouvé plus bas.

3.2.3 Recommandations Basées sur les Connaissances

Tout comme les recommandations globales, les recommandations basées sur les connaissances fonctionnent par filtrage. Une interface utilisateur (profil nutritionnel) a été ajoutée pour collecter les préférences nutritionnelles des utilisateurs. Cette interface est présentée plus en détail dans la *section 4*. À travers cette interface, il est possible de récupérer diverses préférences utilisateur telles que les régimes alimentaires suivis, les préférences nutritionnelles, et les allergies. Ensuite, les items du menu sont filtrés pour ne conserver que ceux compatibles avec le régime alimentaire choisi par l'utilisateur. Un second filtrage est appliqué pour respecter les autres préférences de l'utilisateur. Enfin, si certains items contiennent des allergènes de niveau 2 ou 3, ils sont retirés de la liste. Le pseudo code de cet algorithme peut être retrouvé plus bas.

Algorithm 3 Content based filtering

Description : Recommandations basé sur les habitudes de consommation de l'utilisateur.
Entrée : Menu du cafe (M), utilisateur (u)
Sortie : Liste L des recommandations

Algorithme(M[1...n], u) :
    $L \leftarrow [vide]$
    $P \leftarrow$ Items pas encore achetés (***Algorithme 1***)
    $clusters \leftarrow regrouper\_par\_cluster(M)$
    $cf \leftarrow [vide]$
    Tant que cf.length < clusters.length :
        $c \leftarrow cluster\_favoris(C, u)$
        $cf.inserer(c)$
        $C \leftarrow C.retirer(c)$
    Pour chaque cluster $c \in cf$:
        $L.ajouter(P \cap c)$
    retourner L

---

cluster_favoris(C[[1...n],[1...m],...k], u) :
    $L \leftarrow [empty]$
    Pour chaque cluster $c[1...n] \in C$:
        $tmp \leftarrow [empty]$
        Pour chaque repas $r \in c[1...n]$:
            SI $u \in r.likes$:
                $tmp.ajouter(r)$
        $L.ajouter(tmp.length)$
    $i \leftarrow L.index\_du\_max$
    retourner $C[i]$

---

regrouper_par_cluster(M) :
    $groupes \leftarrow \{vide\}$
    Pour chaque item $\in M$:
        Si $item.cluster \notin groupes$:
            $groupes[item.cluster] = [vide]$
        $groupes[item.cluster].inserer(item)$
    retourner groupes

Algorithm 4 Knowledge based filtering

Description : Recommandations basé sur les spécifications de l'utilisateur.
Entrée : Utilisateur actuel (u), Liste des repas du menu (M)
Sortie : Liste des recommandations.

Algorithme(M[1...n], u) :
    $A \leftarrow u.liste\_allergens$
    $allergenes \leftarrow repas\_allergens(A)$
    $P \leftarrow u.preferences$
    $R \leftarrow clusters(M)$ // ***Algorithme 3***
    $E[0...n] \leftarrow$ récupérer les bons repas en fonction du régime (P[0]) et des
    catégories (P[1]) spécifiées par l'utilisateur
    retourner E

---

repas_allergenes(A[1...n], M[1...m]) :
    $L \leftarrow [vide]$
    Pour chaque repas $r \in M$:
        Si $r.allergens \cap A \neq \emptyset$:
            L.inserer(r)
    retourner L

3.2.4 Robot de Recommandation Santé

Le robot de recommandation santé est conçu pour intégrer des items sains dans les recommandations personnalisées et publiques. Les items sont triés par ordre croissant de leur score santé (plus le score est bas, plus l'item est sain) et les items les plus sains sont ensuite intégrés aux recommandations. Cela vise à encourager la communauté étudiante à consommer davantage de produits sains.

3.3 Score santé

Les recommendations ne sont pas uniquement faites aux utilisateurs en se basant sur leur préférences et habitudes de consommation. En effet, chaque item possède un score santé représentant à quel point cet item est bon pour la santé des utilisateurs. Ensuite, en prenant la moyenne arithmétique du score des items d'un café, on attribut un score café aux cafés. Le score utilisé pour les items est le *Nutri-score* [**?**]. Le pseudo code peut être retrouvé plus bas.

Nutri-score

Le Nutri-Score est un système d'étiquetage nutritionnel visant à fournir une évaluation globale de la qualité nutritionnelle des aliments et boissons.

Il est basé sur un algorithme qui calcule un score nutritionnel global pour chaque produit. Ce score est déterminé en prenant en compte plusieurs critères nutritionnels, tels que :

- Éléments favorables : Les nutriments et composés bénéfiques pour la santé, tels que les fibres, les protéines, et les fruits et légumes. Plus la présence de ces éléments est élevée, plus le score est favorable.

- Éléments défavorables : Les nutriments et composés potentiellement nocifs, comme les acides gras saturés, les sucres ajoutés et le sel. Plus la présence de ces éléments est

élevée, plus le score est défavorable.

Le score final est calculé en soustrayant le total des points obtenus pour les éléments défavorables du total des points obtenus pour les éléments favorables. Lorsque le score est affiché aux consommateurs, ce dernier est ensuite converti en une lettre allant de **A** (meilleure qualité nutritionnelle) à **E** (moins bonne qualité nutritionnelle). Cependant, dans le projet, puisque ce score n'est pas afficher au grand publique, il est concervé en valeur décimal dans le but de faciliter les calculs effectuer plus tard.

Pourquoi le Nutri-score ?

Plusieurs raisons justifient le choix de ce score :

- Standardisation : Le Nutri-Score offre un standard uniforme pour l'évaluation nutritionnelle, ce qui permet une comparaison cohérente entre différents produits, indépendamment des variations dans les étiquetages ou les allégations marketing.

- Support pour les politiques de santé publique : En fournissant une information nutritionnelle transparente et accessible, le Nutri-Score soutient les initiatives de santé publique visant à réduire les maladies liées à l'alimentation et à promouvoir une meilleure santé globale.

Algorithm 5 Health score

Description : Calcul du score santé
Entrée : item
Sortie : Score

```
1:  Health_score(item) :
2:      negative_points_max ← 10
3:      positive_points_max ← 5
4:      energy_points ← min(max(item.calories / 335), 0), negative_points_max)
5:      sugar_points ← min(max(item.sugar / 4.5), 0), negative_points_max)
6:      saturated_fat_points ← min(max(item.saturated_fat / 1), 0), negative_points_max)
7:      sodium_points ← min(max(item.sodium / 90), 0), negative_points_max)
8:      negative_points ← energy_points + sugar_points + saturated_fat_points + sodium_points
9:      fiber_points ← min(max(item.fiber / 0.9), 0), positive_points_max)
10:     protein_points ← min(max(item.proteins / 1.6), 0), positive_points_max)
11:     positive_points ← fiber_points + protein_points
12:     score ← negative_points - positive_points
13:     return score
```

4 Implémentation

4.1 Spécification

Cette section présente principalement les détails techniques de l'implémentation (principalement backend) propre au projet.

4.1.1 Technologies

Les technologies utilisés sont les même que celles qui était déjà présente dans l'application web à savoir :

- **Python** pour sa popularité pour le fait qu'il possèdes plusieurs librairies facilitant le dévellopement.
- **MongoDB** pour la gestion de la base de données en rainson de sa flexibilitée et sa capacité à gérer la diversité.
- **FastAPI** pour le dévellopement de l'interface de programmation principalement en raison de ses performances élevées et sa facilité d'utilisation.
- **ReactJS** pour le dévellopement front-end (*profil nutritionnel*) em raison de sa popularité, la capacité de dévelloper des applications SPA(Single Page Application) ainsi que sa modularité.
- **TailwindCSS** pour le design de l'interface en raison de sa légereté.

Deux nouvelles librairies ont été rajouté dans le projet à savoir *numpy* et *sklearn*. C'est librairies sont utilisé pour effectuer le clustering des items. Nous avons choisie choisi la librairie *sklearn* car elle est l'une des plus populaire pour effectuer du clustering. De plus, elle possède précisément les méthodes dont nous avons besoin. La pluspart des méthodes de *sklearn* qui sont utilisées retournent des listes de type *np.array* (numpy array) donc, l'utilisation de la librairie *numpy* est nécessaire pour pouvoir utiliser *sklearn*.

4.1.2 Exécution

Dans un premier temps, il faudrait tout d'abord exécuter les routines pour regrouper les items (clustering des items), assigner un score santé à ces items et également assigner un score santé aux cafés. Cela est nécessaire car les algorithmes ont besoins du cluster et des score santé pour fonctionner. Ensuite, il n'y a pas d'ordre spécifique pour exécuter les algorithmes.

En raison du temps d'exécution assez long (à cause du manque d'optimisation présenté dans la *section 5.2*), nous avons pensé temporairement retirer un algorithme (filtrage collaboratif) car, ce dernier prend le plus temps pour s'exécuter. En effet, les algorithmes de filtrage collaboratifs, filtrage basé sur le contenu et filtrage basé sur les connaissances, exécutés tous ensemble, prennent approximativement 2h30 pour produire des recommendations. L'algorithme de filtrage collaboratif contribue en grande à ce temps d'exécution. Ainsi, l'algorithme sera rajouté de nouveau après qu'il ait subit quelques modifications qui sont présentés dans la *section 5.2*. En appliquant cette modification, les algorithmes sont planifié être exécutés toutes les 2h environ.

4.1.3 Base de donnée

Une nouvelle base de données pour les recommendation a été crée. Elle contient essentiellement de nouveaux attributs propre aux recommendations :

- **User recommendation** : Cette collection contient les recommendations personnalisés pour chaque utilisateur (recommendations d'items et de cafés) ainsi que les informations uniques à chaque utilisateur (nom d'utilisateur et identifiant).

- **Cafe for recommendation** : Cette collection contient la liste des recommendations publiques, le slug et le score santé de chaque café.

- **Items** : Cette collection contient l'identifiant, le slug, le score santé et le cluster de chaque item.

4.1.4 API

Au niveau de l'API, 13 endpoints ont été rajoutés pour gérer les recommendations.

- **Cafe et Items** : Quatre endpoints ont été rajoutés pour pouvoir récupérer et modifier les informations (score santé, recommendations publiques) des cafés et des items.

- **Users** : Cinq endpoints ont été rajoutés pour récupérer un utilisateur, ses recommendations (recommendation de cafés et d'items) et modifier les recommendations.

- **Public** : Deux endpoints pour récupérer et modifier les recommendations publiques.

- **Bot** : Un endpoint a été rajouté pour récupérer les recommendations du robot de recommendation. Les recommendations sont calculés lors de l'appel de l'API donc, ces dernières ne sont pas stockés dans la base de données, ce qui jsutifie l'absence d'endpoints pour modifier les recommendations.

- **Gérant** : Un endpoint a été rajouté pour récupérer les items désirés par les utilisateurs mais absent des cafés.

4.2 Illustration

Les modifications apportés dans le front-end se présentes à trois niveaux : l'ajout du profile nutritionnel, l'ajout de sections pour les recommendations publiques et personnalisés et l'ajout d'une section pour les recommendations de cafés personnalisés. Toutes les interfaces présenté plus bas peuvent être vu en *annexe*.

A. Profil nutritionnel
Le profile nutritionnel a été rajouté dans le but de pouvoir récupérer toutes les informations nécessaire pour pouvoir faire les recommendations basés sur les connaissances. Le profile est séparé en trois sections mais les préférence nutritionnelles et les allergies sont regroupées ensemble car elles sont similaires.

1. **Régimes alimentaires** : Cette section permet à l'utilisateur de sélectionner un ou plusieurs régimes alimentaires qu'il souhaite. Initalement, trois régimes sont présents : Cétogène, Méditéranéen et Végétarien. Chaque régime possède un nom, une description (facultative), une liste d'ingrédients/aliments qui ne doivent pas être consommés dans le régime et une liste des items que l'utilisateur voudrait avoir dans le régime. Une fonctionnalité qui n'a pas pu être rajouté à temps est l'affichage du nombre de cafés dans lesquels il existe des items respectant les spécification du régime alimentaire. L'utilisateur a l'option de rajouter un nouveau régime alimentaire s'il le souhaite. Cependant, seule les régimes que ce dernier a rajouté sont éditable et supprimable. Les trois régimes affichés initalements ne sont ni éditable, ni supprimable par l'utilisateur.

2. **Préférences nutritionnelles et allergies** : La section pour les préférences nutritionnelles permet à l'utilisateur de sélectionner les nutriments qu'il souhaite retrouver dans les aliments qui lui sont recommendés et à quelle proportion ils devraient se retrouver dans l'item (faible, moyen ou élevé). La section des allergies permet à l'utilisateur de sélectionner les aliments auquels il est allergique (lait/lactose, arachide/noix) et à quel degré ce dernier est allergique à ces aliments (faible, moyen, élevé).

B. Recommendation publiques et personnalisés

Pour l'affichage des recommendations publiques et personnalisés d'items, deux novelles boites ont été rajoutée dans le menu des cafés. Une boite "*Pour moi*" contenant les items recommendés à l'utilisateur et une autre boite "*Produits recommendés*" contenant les items recommendés à toute la communauté étudiante. La boite de recommendation personnalisée n'est visible que pour les utilisateurs qui sont connecté à l'application web.

C. Recommendation de cafés

L'affichage des recommendations de cafés se fait sur la page d'acceuil. En effet, une nouvelle section nommée "*Recommendation*" a été rajoutée dans sur la page d'acceuil. Cette section contient les cafés qui ont été recommendés. Elle section n'est visible que pour les utilisateurs qui sont connecté à l'application web. En dessous de cette section se trouve la section "*Tous les cafés*" qui contient tous les cafés de l'udem. Les cafés de cette section sont triés en fonction de leurs score santé de sorte à ce que les cafés vendant les items les plus sains soient affichés en premiers.

5 Évaluation

Pour s'assurer du bon fonctionnement des algorithmes intégrés à l'application web, des tests ont été éffectués tout au long du dévellopement. Les tests encourus sont divisés en 2 catégories :

- **Tests fonctionnels (à la boite noire)** : Ils vérifient la conformité du code aux exigences en évaluant si les sorties correspondent aux attentes pour des entrées données, sans se soucier des détails internes du programme.

- **Tests d'utilisabilité** : Ils permettent de déterminer dans quelle mesure un produit ou un service (généralement une interface utilisateur ou un site web) est facile à utiliser pour les utilisateurs réels. Ces tests sont conçus pour identifier les problèmes d'utilisabilité, recueillir des données qualitatives et quantitatives, et améliorer l'expérience utilisateur.

Pour vérifier et valider que l'ajout de nouveaux endpoints ne provoque pas de changements indésirables dans l'application, une série de tests unitaires et de tests d'intégration a été réalisée.

5.1 Tests

La librairie *unittest* de Python a été choisie pour effectuer les tests unitaires en raison de sa simplicité d'utilisation et de mon expérience antérieure avec cet outil.

5.1.1 Tests unitaires

Un total de 74 tests unitaires ont été réalisés. Environ 63 tests ont été créés pour garantir le bon fonctionnement des algorithmes, et 10 autres pour vérifier le bon fonctionnement des nouveaux endpoints de l'API. Les tests API sont structurés de manière cohérente avec ceux déjà en place.

Les tests des algorithmes sont structurés comme suit :

1. Chaque script contenant les tests d'un algorithme est subdivisé en *class*. Chaque *class* contient les différents tests correspondant à une méthode intermédiare utilisé dans l'algorithme. Ainsi, chaque méthode dans la *class* s'assure de tester d'une ou de plus manière la fonction à tester. L'algorithme possède aussi une *class* contenant ses tests. Si des méthodes intermédiaires ne nécéssitent qu'un seule cas de test, ces dernières sont inclusent dans la même *class* de test que l'algorithme principale.

2. Dans le but de controller le mieux possible l'environement de test, les mocks sont utilisés pour éviter de biaiser le fonctionnement des algorithmes. Ainsi, ces derniers sont utilisés uniquement sur les méthodes faisant appel à l'API. Cela permet de s'assurer que si un test échoue, l'erreur est uniquement lié à l'algorithme lui même ou à son implémentation et non à un problème au niveau de l'API.

3. Chaque *class* de test contient, si nécessaire, une méthode *setUp* qui permet d'initialiser les objets utilisés par plusieurs tests. Si un objet spécifique est requis pour faire un test et qu'il n'est requis que pour ce test, il est rajouté uniquement dans la méthode test.

4. Les tests unitaires suivent le style **AAA** : **Arrange** (section du test initialisant les objets utilisé par la méthode à tester), **Act** (section où l'on invoque la méthode testée avec les paramètres créés précédemment) et **Assert** (section vérifiant que la méthode produit le comportement désiré).

La figure ci-dessous est un test unitaire vérifiant que l'algorithme de filtrage basé sur le contenu retourne bien les items qu'il devrait retourner dans le cas où un utilisateur a déjà acheté tous les items d'un café.

```python
class TestContentBasedFiltering(unittest.TestCase):
    def setUp(self):
        self.user1 = {'user_id': 'user1', 'username': 'username1'}
        self.cafes = [
            # Cafe1 doesn't contain user2
            {
                'slug': 'cafe1',
                'menu_items': [
                    {'slug': 'item1', 'likes': []},
                    {'slug': 'item2', 'likes': ['user1']},
                    {'slug': 'item3', 'likes': ['user5', 'user3']},
                    {'slug': 'item4', 'likes': ['user1', 'user3']},
                    {'slug': 'item5', 'likes': ['user6', 'user3', 'user4']}
                ]
            },
      ...
    # autres methodes de la classe
    @patch('recommender_systems.utils.api_calls.CafeApi.get_all_items')
    @patch('recommender_systems.utils.db_utils.get_user_likes_in_cafe')
    @patch('recommender_systems.utils.utilitaries.items_not_bought_in_cafe')
    def test_main_4(self, mock_items_not_bought_in_cafe, mock_get_user_likes,
    mock_api_get_all_items):
        # ARRANGE: Recuperation des objets de la methode setUP
        user = self.user1
        cafe = self.cafes[0]

        # Mock d'un appel de l'API
        mock_api_get_all_items.return_value = ([
            {'slug': 'item1', 'likes': []},
            {'slug': 'item2', 'likes': ['user1']},
            {'slug': 'item3', 'likes': ['user5', 'user3']},
            {'slug': 'item4', 'likes': ['user1', 'user3']},
            {'slug': 'item5', 'likes': ['user6', 'user3', 'user4']}
        ], 200)

        # Mocking
        mock_items_not_bought_in_cafe.return_value = set()
        mock_get_user_likes.return_value = ['item2', 'item4']

        # ACT et ASSERT
        self.assertCountEqual(main(user, cafe), ['item2', 'item4'])

        # Mocking
        mock_get_user_likes.return_value = []
        mock_items_not_bought_in_cafe.return_value = set()

        # ACT et ASSERT
        self.assertCountEqual(main(user, cafe), ['item5', 'item4', 'item3', '
    item2', 'item1'])
```

Listing 1 – Test unitaire test_main_4 content based filtering

5.1.2 Tests d'utilisatbilité

Une fois une première version de l'interface pour le profil nutritionnel complétée, des tests d'utilisabilité ont été réalisés. Lors d'une foire aux projets qui s'est tenue le jeudi 25 juillet, il a été possible de recueillir des retours de la part des autres étudiants, ce qui a permis d'améliorer la présentation de l'interface utilisateur ainsi que l'expérience utilisateur du profil.
De plus, j'ai présenté l'application à certains membres de ma famille peu familiers avec les outils informatiques, dans le but d'obtenir des retours plus diversifiés. Ces commentaires ont été très pertinents pour améliorer l'expérience utilisateur de l'interface initiale.
Un commentaire fréquemment revenu concernait le manque d'animations dans le profil, qui sera une piste d'amélioration pour l'avenir.

5.2 Discussion

Cette section est dédiée à effectuer des remarques et critiques sur les algorithmes. Les critiques seront effectués par rapport à l'optimalité (complexité et Scalabilité) et la robustesse de l'algorithme. Puisque les algorithmes de recommendation des cafés sont très similaires aux algorithmes de recommendations personnelles, les critiques de ces dernières sont aussi valables pour les cafés.

5.2.1 Filtrage collaboratif

- **Complexité temporelle :** Cet algorithme calcule la similarité entre l'utilisateur cible et tous les autres utilisateurs, ce qui peut entraîner une complexité de l'ordre de $O(n \times m)$, où n est le nombre d'utilisateurs et m est le nombre d'attributs comparés. Bien que cette approche soit fonctionnelle pour des jeux de données de taille modérée, elle devient peu optimale lorsque le nombre d'utilisateurs augmente significativement. Pour des systèmes à grande échelle comme l'*UdeM* (l'université contient environ 67000 étudiants !), cela pourrait ralentir les performances, nécessitant potentiellement des techniques de réduction de la dimensionnalité comme les méthodes *PCA* et *MDS* pour réduire les nombre d'éléments à comparer lors du calcul de la similarité tout en conservant le plus d'information dans les données. En effet, l'utilisation de la similarité jaccard donne des résultats précis mais cet algorithme devient moins efficace pour de large données. L'utilisation des méthodes de clustering comme *KMeans* pour regrouper les utilisateurs permettrait aussi d'améliorer sinificativement l'efficacité.

- **Scalabilité :** Comme mentionné précédement, cet algorithme peut rencontrer des difficultés à s'adapter à un grand nombre d'utilisateurs ou de données en raison de la nécessité de calculer des similarités entre chaque paire d'utilisateurs. Dans les cas où le nombre d'utilisateurs est très élevé, des approches alternatives comme le filtrage collaboratif basé sur des modèles (qui se trouve dans la famille des recommendations basé sur les models) seraient plus efficaces.

- **Robustesse :** Les différents scénarios possibles sont pris en compte par l'algorithme mais une meilleure gestion des erreurs et des exceptions, notamment en ajoutant des logs plus détaillés, améliorerait sa robustesse et faciliterait le débugage.

- **Réutilisabilité :** L'algorithme est étroitement couplé avec des appels à des utilitaires spécifiques, ce qui pourrait limiter sa réutilisabilité dans d'autres contextes ou projets. Une approche consistant à abstraire ces dépendances rendrait l'algorithme plus modulaire et adaptable à différents environnements ou bases de données.

- **Extensibilité :** L'algorithme pourrait être plus flexible en permettant l'utilisation de différentes métriques de similarité ou en adaptant dynamiquement le seuil de similarité en fonction des besoins spécifiques de l'utilisateur ou du contexte.

En somme, pour une portentille amélioration de l'algorithme, il faudrait donc diminuer la dépendance de l'algorithme à d'autres modules de l'application, utiliser des méthodes de clustering ou de réduction de dimentionnalité pour augmenter son éfficacité et afficher des logs plus détaillés pour faciliter la maintenance des algorithmes.

5.2.2 Filtrage basé sur le contenu

- **Complexité temporelle :** L'algorithme regroupe les items en clusters et itère sur ces derniers pour trouver ceux que l'utilisateur a le plus appréciés. La complexité de cette opération dépend du nombre de clusters et d'items par cluster. Similairement à l'algorithme de filtrage collaboratif, cette méthode pourrait rencontrer des problèmes de performance lorsque le nombre de clusters ou d'items est très élevé. L'utilisation d'une approche de pré-traitement, comme l'indexation inversée, pourrait potentiellement réduire les coûts de recherche.

- **Scalabilité :** Le processus de sélection des clusters les plus appréciés pourrait être optimisé en pré-calculant et en stockant les résultats des préférences de l'utilisateur, plutôt que de les recalculer à chaque exécution de l'algorithme. Donc, il serait plus efficade de créer une routine qui calculerait les clsuters favoris de l'utilisateur et qui les mettrait à jour dans la base de données.

- **Robustesse :** Comme pour le filtrage collaboratif, l'ajout de logs plus détaillés permettront faciliteront le débugage et augmentront la robustesse de l'algorithme.

- **Réutilisabilité :** L'algorithme est assez spécifique dans l'ensemble même si certaines composantes (script pour trouver les clusters favoris par exemple) peuvent être réutilisés dans d'autres contextes.

En somme, l'utilisation d'indexage inversé, la réduction de la dépendance de l'algorithem avec d'autres modules de l'application ainsi l'ajout de meilleurs logs pour le débugage seraient des modifications qui améliorerait l'algorithme dans son ensemble.

5.2.3 Filtrage basé sur les connaissances

- **Complexité temporelle et scalabilité :** La complexité temporelle est de l'ordre de $O(n \times m \times k)$ dans le pire cas (l'utilisateur spécifie des préférences nutrives et des régimes alimentaires) où n est le nombre d'items, m, le nombre de régimes alimentaire et k, le nombre de filtrages appliqués. Cet algorithme est acceptable avec une quantité faible ou moyenne de données mais, lorsque le nombre de données est élevé, la perfommance de l'algorithme pourrait significativement diminuer en raison des multiples filtrages qui sont appliqués. L'algorithme pourrait être amélioré en utilisant d'avantages des structures de données appropriées (les ensembles et dictionnaires), en évitant les calculs répétitifs (comme les comparaisons redondantes) et en réduisant le nombre d'opérations coûteuses.

- **Robustesse et réutilisatbilité :** Les commentaires faites aux algorithmes précédents s'appliquent aussi à cet algorithme.

- **Extensibilité :** Le point fort de cet algorithme serait son extensibilité en ce sens qu'il est très facile de rajouter de nouveaux filtres. Cependant, il serait quand même plus inérressant pour facilité l'ajout de nouveaux filtres, de regrouper les différents filtres dans un script différent de celui de l'algorithme.

En résumé, cet algorithme souffre des même problème de robustesse et de réutilisabilité que les autres algorithme en ce sens qu'il dépend beaucoup trop des modules spécifiques à l'application web *Café sans-fil*. L'algorithmes serait donc plus optimal dans l'ensemble en réduisant la dépendance avec d'autres modules de l'application et en ajoutant de meilleurs logs pour le débugage.

6 Conclusion

Ce rapport a présenté une approche complète pour le développement d'un système de recommandation de repas personnalisé, en tenant compte des préférences nutritionnelles, des restrictions alimentaires, et des préférences de l'utilisateur. Les différentes sections ont mis en lumière l'importance de l'intégration de multiples algorithmes, chacun étant conçu pour optimiser un aspect spécifique du processus de recommandation. L'utilisation d'algorithmes de filtrage collaboratif, de filtrage basé sur le contenu, et de recommandations basées sur la connaissance a permis de créer un système robuste capable de s'adapter aux besoins variés des utilisateurs.

Les tests d'utilisabilité effectués ont permis d'affiner l'interface utilisateur, rendant le système plus intuitif et accessible. Les retours des utilisateurs ont été essentiels pour identifier des pistes d'amélioration, notamment l'intégration d'animations et d'améliorations visuelles pour enrichir l'expérience utilisateur.

Enfin, les critiques apportées aux algorithmes, en termes de complexité, scalabilité, et robustesse, ont permis de souligner les points forts et les défis à relever dans l'avenir. L'optimalité des algorithmes, bien que satisfaisante dans le cadre actuel, pourrait bénéficier de futures optimisations pour traiter des jeux de données plus vastes et diversifiés.

En conclusion, ce travail jette les bases d'une application de recommandation alimentaire personnalisée, avec un potentiel d'évolution et d'adaptation pour répondre aux nouvelles exigences des utilisateurs et du marché. Les prochaines étapes incluront l'amélioration continue des algorithmes et l'intégration de nouvelles fonctionnalités pour mieux servir les utilisateurs et enrichir leur expérience.

Remerciement

Il est important de mentionner que ce projet n'aurait pas vu le jour sans le support de **l'Université de Montréal** et plus précisément, **Louis-Édouard LAFONTANT**, qui a supervisé et fourni une précieuse aide durant le développement et **Dr. Stefan Monnier**, PhD qui chapeaute ce cours et a accepté de me laisser poursuivre ce projet pendant la session.

Annexe

Images de l'implémentation du front

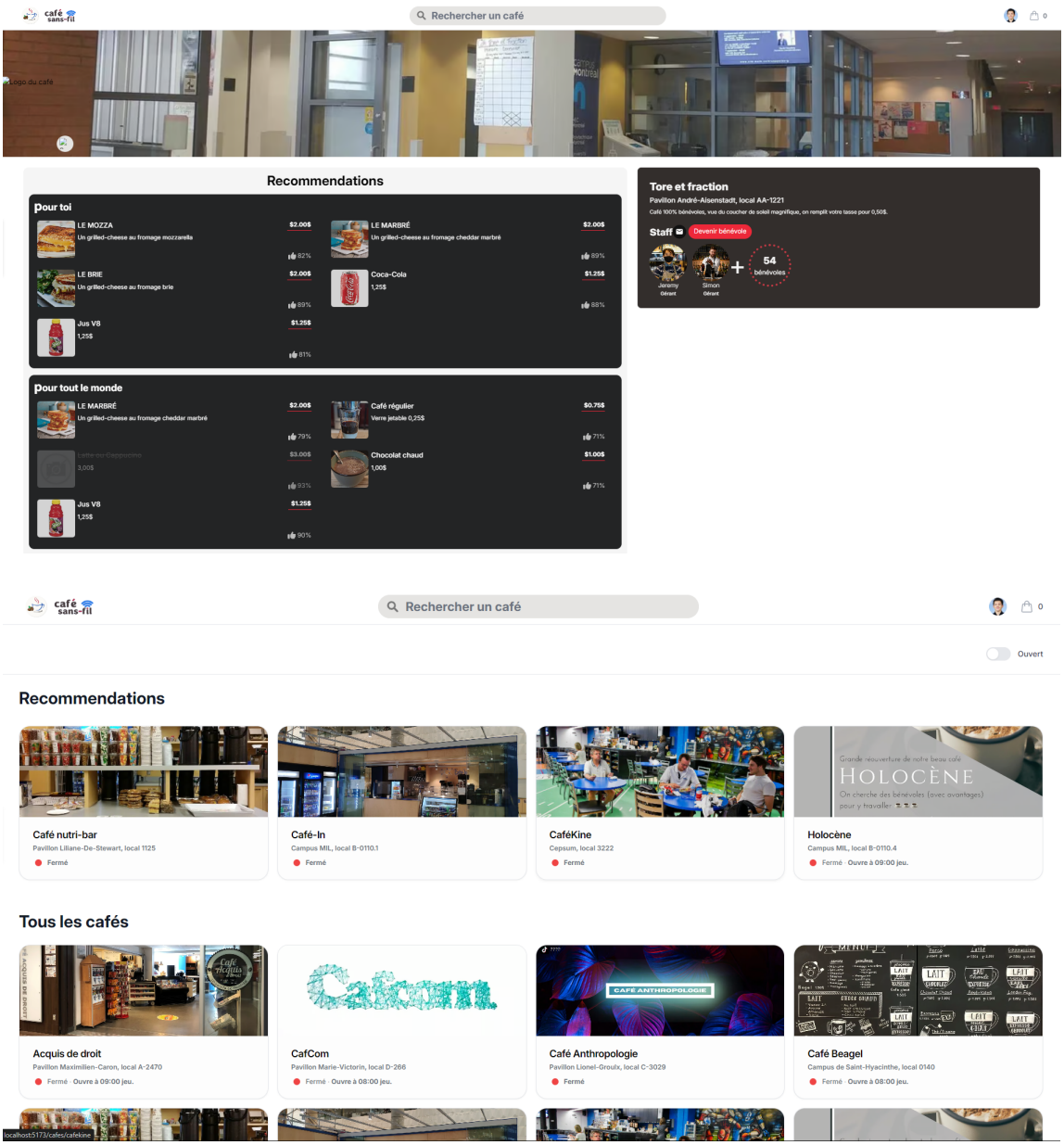

FIGURE 2 – Images des recommendations

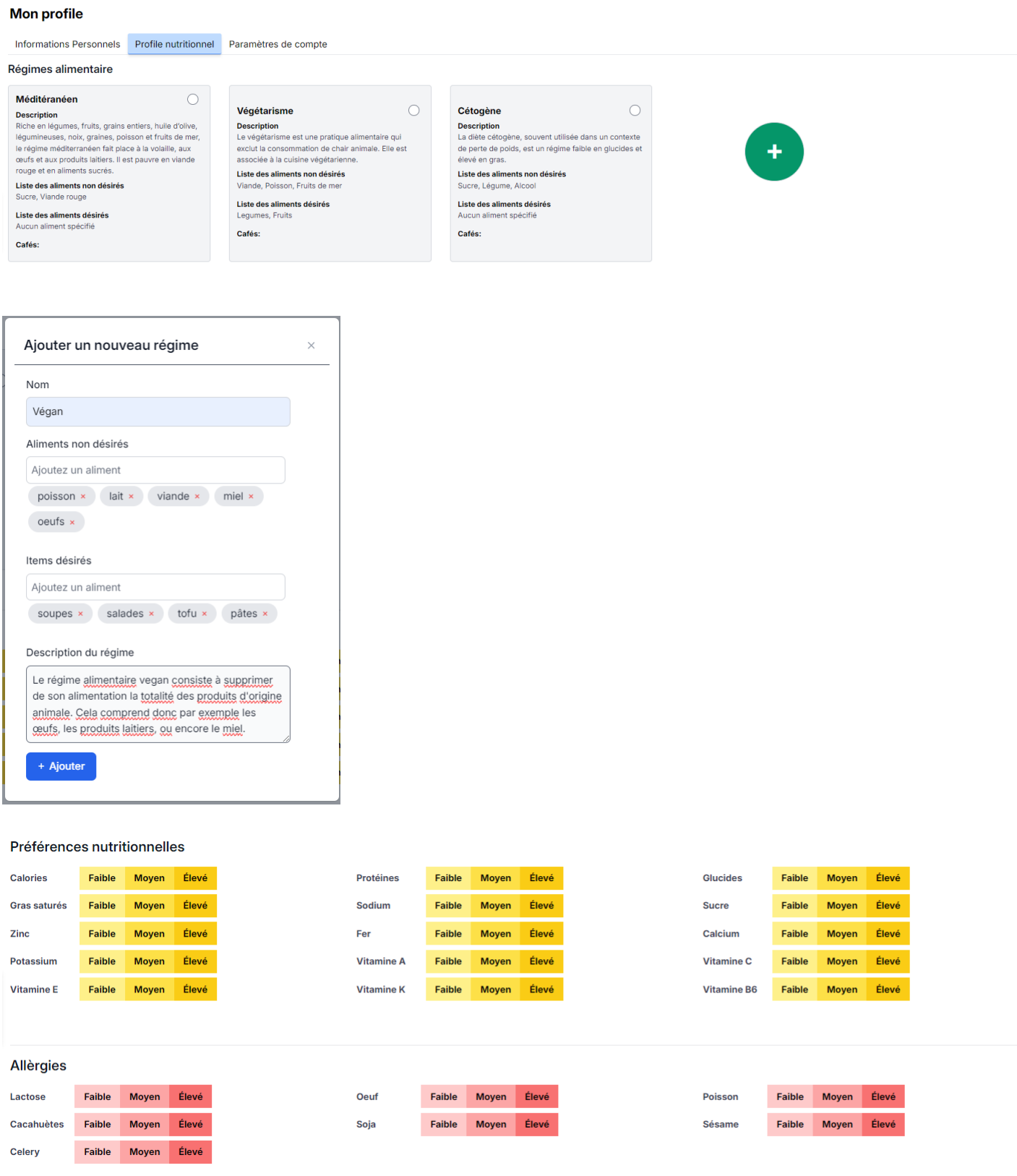

FIGURE 3 – Image du profile nutritionnel

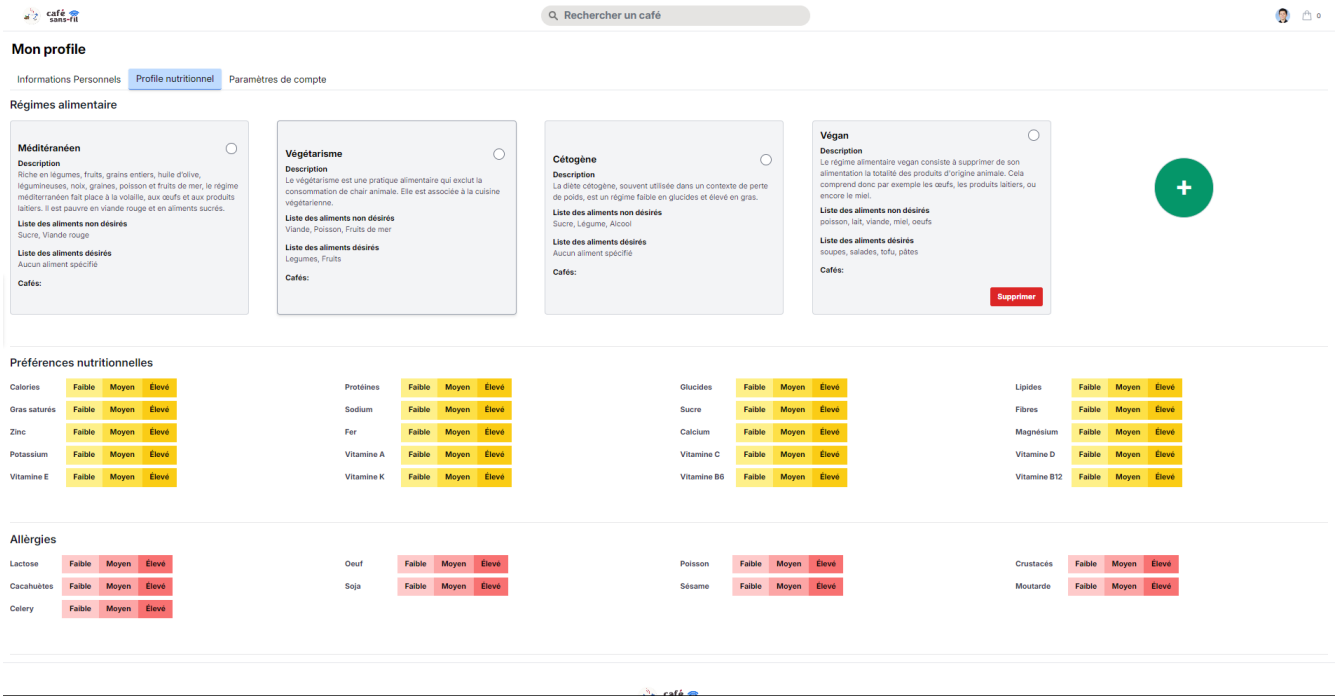

FIGURE 4 – Image complète du profil

Méthodes utilitaires

Algorithm 6 Repas pas encore consommés

Entrée : Menu du restaurant (M), utilisateur actuel (u)
Sortie : Liste de repas

Algorithme(M[1...n], u)
    $L \leftarrow u.historique\_achat$
    Pour chaque repas $r \in L$
        Si $r \in M$:
            $M.retirer(r)$
    retourner M

Algorithm 7 Mesure similarité : Jaccard

Entrée : Deux listes ou ensembles (L_1, L_2)
Sortie : Nombre décimal représentant la similarité

Jaccard$(L_1[1...n], L_2[1...m])$ ***:***
    $union \leftarrow L_1 \cup L_2$
    $inter \leftarrow L_1 \cap L_2$
    retourner $inter.length/union.length$

Algorithm 8 Clusters en fontion des préférences de l'utilisateur

Entrée : Menu du restaurant (M)
Sortie : Liste des régimes contenant les catégories de repas

clusters(M[1...n])
    // Nous avons k régimes différents et p catégories de repas
    $R \leftarrow [[0...n]...k]$
    Pour chaque repas $r \in M$:
        mettre le repas dans un ensemble en fonction de son régime
        rajouter les différents régimes de repas à R
    Pour chaque régime $T[1...\ell] \in R$
        mettre chaque repas du même régime dans une catégorie
    retourner $R[[[0...n]...p]...k]$