Contents

Code source disponible ici (sous licence MIT) : ForkEat

Organisation dans l'équipe

Rôle fonctionnel de l'application

Nous avons créé une application de gestion de stock de nourritures, afin de limiter le gaspillage et de faciliter la planification des repas. Dans l'état actuel, notre application permet d'entrer les stocks de produits en base manuellement, de créer des recettes et de les chercher par noms ou ingrédients. De plus, nous avons commencé le développement d'un système de recommandation de recette, mais qui n'est pas encore intégré à l'application.

Analyse du domaine

Outils et langages

Application Mobile

Nous avons opté pour le framework Flutter qui permet de développer des applications mobiles cross-plateforme avec le langage Dart.

Serveur d'application

Le serveur d'application est écrit avec le langage C# sur le runtime cross-plateforme et open-source .NET 6 (nous avons commencé sur la version 5 puis continué sur la version 6 sortie en novembre). Nous avons choisi C# et .NET, car ce sont des outils modernes qui permettent de développer du code fiable, et ce avec un bon confort de développement.

Pour le protocole de communication avec le client, nous avons sélectionné REST, le standard en la matière de service Web. Notre application backend est donc une API Web REST développée le framework web fournis avec le runtime .NET, ASP .NET Core.

La sécurité de l'API et gérée par une authentification utilisant JSON Web Tokens. Cette option est la plus compatible avec ReST, respectant son caractère sans état.

Pour la persistance des données dans la base de donnée relationnelle, nous avons utilisé le standard de facto en la matière d'ORM pour .NET, Entity Framework Core.

Prototype UI

Figma

Serveur de base de donnée

Pour le serveur de base de donnée, nous avons sélectionné PostgreSQL, un serveur de base de donnée relationnel moderne et open-source très performant et qui présente des fonctionnalités spécifiques intéressantes, comme un super pour la full-text search.

Machine learning

Comme la plupart des implémentations de machine learning, nous avons choisi python comme langage de programmation. Pour faciliter le développement et la séparation claire des étapes, nous avons utilisé jupyter notebook comme outil de développement. De plus, pour gérer plus facilement les dépendances aux diverses bibliothèques utilisées, nous nous sommes aidé de miniconda pour créer un environnement indépendant des machines.

Architecture technique et méthodologies de développement

Front-end

Back-end

Mobile

Technologies

Pour la partie mobile nous avons décidé d'utiliser flutter comme outil de développement, car il contient beaucoup d'avantage. Dans un premier temps, grâce au langage Dart, il est possible de rafraîchir rapidement l'affichage de l'écran en prenant en compte les modifications avec le "hot reload", ce qui évite d'avoir à fermer l'application et de la relancer. De plus, il possède toutes les fonctionnalités des plates-formes natives comme le défilement, la navigation, etc. Mais également des widgets et designs qui lui sont propres ce qui le rend indépendant de la plateforme et lui permet de couvrir à la fois IOS et Android. L'écosystème est open source et tous les packages sont sous licence BSD ou MIT ce qui les rend totalement libre d'utilisation.

Dart est un langage de programmation optimisé pour plusieurs plateformes. C'est un langage orienté objet asynchrone qui supporte le fonctionnement de code en parallèle et permet de fournir des réponses en temps réel.

Architecture

Nous avons choisi de faire une architecture clean patern dans le but de faire des tests unitaires que nous n'avons malheureusement pas eu le temps de faire.

// Détail de l'archi des fichiers

Infra (thomas)

La partie infrastructure est principalement celle qui détermine la "Quality of Service". C'est pour cela qu'il est important de bien dimensionner l'infrastructure. Ainsi afin de garantir un service optimal et fiable nous avons identifié deux types d'infrastructure. La première étant la CI soit la "Continous Integration" qui nous permet à chaque itération de valider les ajouts avec une architecture la plus proche de la production. La seconde étant la CD soit la "Continious Delivery" qui nous permet après chaque validation d'itération de pousser en production les nouvelles versions.

Pour effectuer cette mission, nous avons choisi Drone.io qui est une solution Open-Source et selfhosted qui nous permet de tester les différents services (tests unitaires, test d'intégration, construction de l'image). La grande force de cet outil est qu'il est nativement compatible avec Docker.

Continous Integration

La CI, Continuous Integration ou intégration continue en français est une étape permettant de tester continuellement et de manière automatique un projet tout au long de son cycle de vie.

Ainsi, il est possible grâce à l'intégration continue d'automatiser différentes tâches comme :
- Lancer la suite de tests unitaires afin de vérifier chaque composant d'une application individuellement. - Lancer la suite de tests d'intégration afin de vérifier si l'interfaçage avec les différents composants de l'application ne présente aucune régression. - Lancer la suite de tests fonctionnels afin de vérifier si fonctionnellement parlant, l'application ne présente aucune anomalie ou de régression. Lorsque des tests d'intégration sont disponibles, cette étape est généralement exécutée après, afin d'utiliser l'infrastructure logicielle fraîchement déployée. - Analyser et remonter les possibles problèmes de qualité de code. - Remonter les possibles failles de sécurité embarquée par des modules tiers téléchargés par un gestionnaire de dépendances comme npm, gem, pip/Poetry, cargo, nuget, maven, etc ...

L'ensemble de ces tâches constitue un pipeline. Généralement, il est exécuté à la suite d'évènements sur la forge comme la création d'une demande de fusion (Merge/Pull request selon la forge), lors d'un commit, ou encore lors de la création d'une étiquette (tag).

Si nous prenons l'exemple du backend, nous avons deux pipelines lors d'un commit sur la branch master:

- L'une qui va lancer une base de donnée en docker en parallèle, puis qui va lancer les tests unitaires et d'intégration. Puis nous lançons un test de code coverage afin d'éviter d'avoir une réduction de celui en fonction des itérations. Puis nous déployons l'image de l'intégration sur le Docker Hub grâce au Dockerfile.
- La seconde pipeline a un aspect de documentation, car elle permet de synchroniser les readme entre Github et le DockerHub.

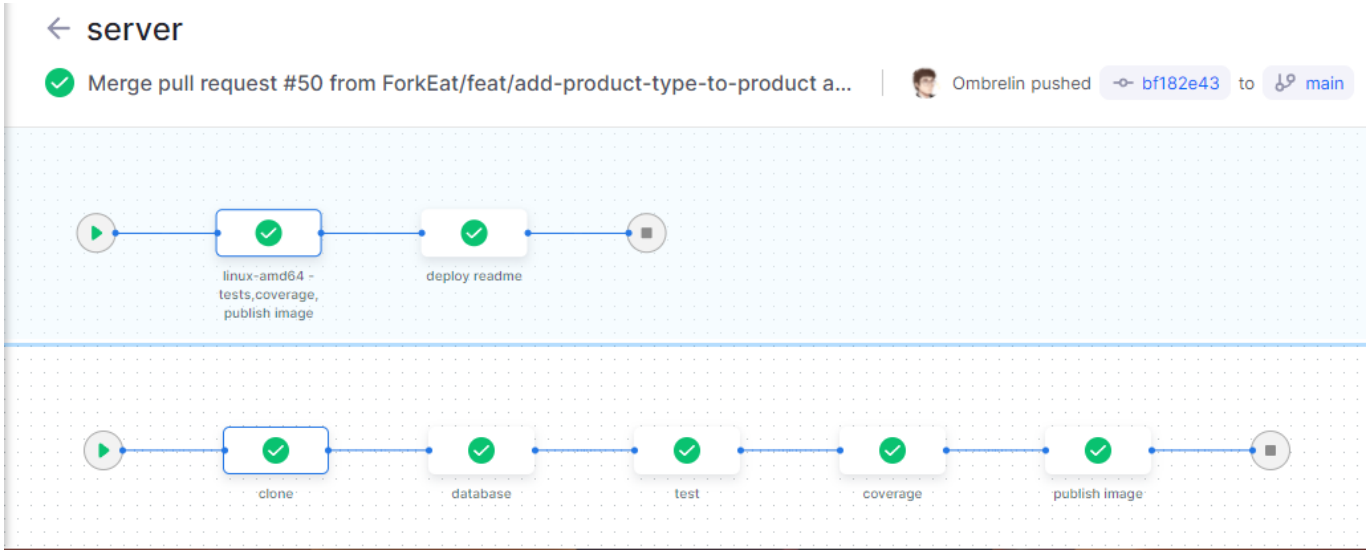

Figure 1: Pipeline

Cependant, nous nous commitons notre code sur une autre branch alors nous aurons uniquement le lancement des tests unitaires et d'intégrations avec le code coverage. Nous n'avons pas besoin de lancer le déploiement d'une image Docker. A contrario, si nous créons un tag sur le dépot Github, la CI va lancer le déploiement de l'image en utilisant les tags comme version :

Voici à quoi ressemble le pipeline chargé de lancer nos tests sur la partie Backend (Les parties mobile et embarquée disposent, elles aussi de pipeline chargé de valider les itérations) :

```yaml
services:
- name: database
  image: postgres:alpine3.15
  environment:
    POSTGRES_USER: user
    POSTGRES_PASSWORD: password
    POSTGRES_DB: test

steps:
- name: test
  image: mcr.microsoft.com/dotnet/sdk:6.0
  pull: always
  environment:
    JWT_SECRET: rdtfyguhijgfrdtyg45678
    TEST_DATABASE_URL: postgres://user:password@database:5432/test
    DATABASE_URL: postgres://user:password@database:5432/test
  commands:
    - sleep 15
    - cd ForkEat/
    - dotnet test --logger "console;verbosity=detailed" /p:CollectCoverage=true /p:CoverletOutputFormat=op

- name: coverage
  image: plugins/codecov
  settings:
    token:
      from_secret: codecov-token
    files:
      - ForkEat/ForkEat.Core.Tests/coverage.opencover.xml
    required: true

- name: publish image
  image: plugins/docker
  settings:
    auto_tag: true
    auto_tag_suffix: linux-amd64
```

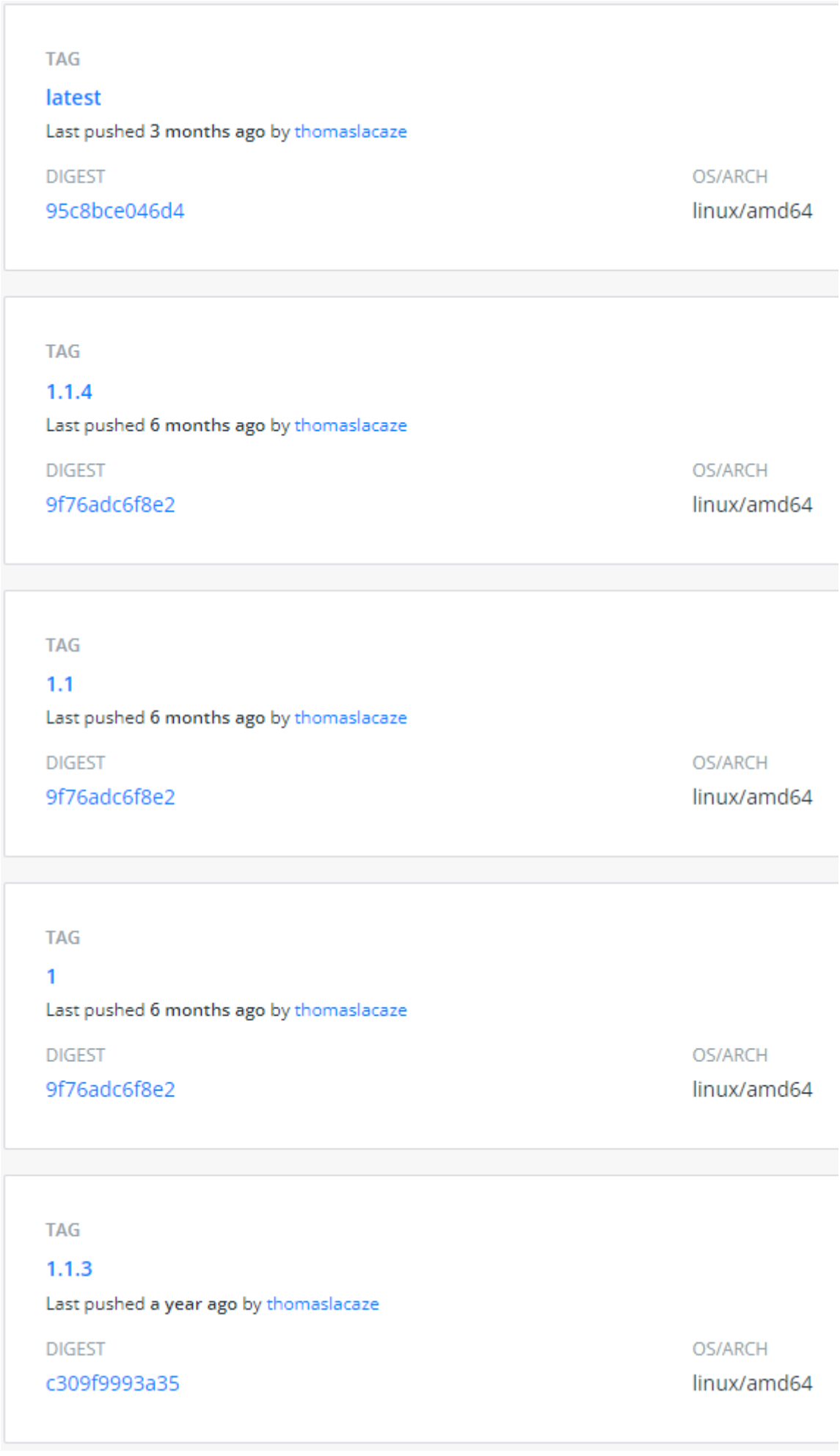

Figure 2: DockerHub

```
    repo: thomaslacaze/forkeat-server
    username:
      from_secret: docker_username
    password:
      from_secret: docker_password
    dockerfile: ForkEat/ForkEat.Web/Dockerfile
  when:
    event:
    - push
    - tag
    branch:
    - master
    - main
```

Continous Delivery

La CD désigne à la fois la livraison en continue (Continuous Delivery) ainsi que le déploiement continue (Continuous Deployment). La livraison continue permet de livrer une nouvelle version d'un logiciel en générant des paquets et binaires prêts à l'emploi pour les différents environnements dont l'application peut être exécutée. Le déploiement continue permet d'automatiser le processus de mise en production de la nouvelle version d'une application sur un ou plusieurs environnements comme un environnement de stage, pré-production ou de production

Embarqué (thomas)

Image

Ce qui est fait, pourquoi

Limites rencontrées

Machine learning

Pour développer un système de recommandation, il fallait déjà que nous comprenions plus en détail ce que c'était et quel type de système choisir. Il y a donc d'abord eu une importante phase de recherche afin de savoir ce que nous devions faire. Nous allons résumer ici tout ce que nous avons découvert au cours de cette phase.

Recherches

Les systèmes de recommandation sont une sous-catégorie importante des algorithmes d'apprentissage automatique, et plus précisément des systèmes de filtrage de l'information. Un système de filtrage de l'information est un système qui gère un flux important d'informations, et qui est utilisé pour supprimer les informations redondantes ou indésirables, afin d'exposer les utilisateurs aux informations pertinentes. Un système de recommandation est créé pour prédire la préférence d'un utilisateur pour des articles divers. Il existe différents types d'approches de filtrage : le filtrage collaboratif et le filtrage basé sur le contenu.

Filtrage collaboratif Le filtrage collaboratif utilise les similitudes entre différents utilisateurs, l'idée derrière est la "sagesse de la foule". Pour prédire ce qu'une personne va aimer, nous allons étudier ce que d'autres utilisateurs similaires ont aimé. Si l'utilisateur A et l'utilisateur B ont aimé les mêmes articles dans le passé, il est fort probable qu'ils aient des goûts similaires et que nous puissions recommander à l'utilisateur A certains articles que l'utilisateur B a achetés, ou vice versa. Pour que cela fonctionne, nous avons besoin du profil de l'utilisateur, c'est-à-dire de l'historique des achats et des évaluations des articles, le modèle n'a pas besoin de comprendre quels sont les articles, ce qui est un avantage dans certains cas. Il existe deux approches pour le filtrage collaboratif :

- Explicite : l'utilisateur doit explicitement donner une opinion sur l'article, généralement avec une note, pour indiquer au modèle s'il a aimé ou non l'article. Cette méthode est utilisée par exemple par Netflix, elle permet au système de reconstruire facilement l'historique d'un utilisateur et d'éviter d'agréger des informations qui ne sont pas liées à un utilisateur unique (plusieurs utilisateurs utilisant le même ordinateur).
- Implicite : ce système doit faire une analyse plus poussée pour comprendre le comportement du client et ses goûts, puisqu'il n'y a pas d'information explicite sur ce qu'il aime ou n'aime pas. Cela peut se faire en analysant le nombre de fois qu'un utilisateur a vu un article et en conservant l'enregistrement des achats par exemple (un article que vous avez vu plusieurs fois est susceptible de vous intéresser). Cette méthode est utilisée par Amazon (généralement achetée ensemble) et Facebook. Le principal avantage est qu'il n'est pas demandé à l'utilisateur d'évaluer ses articles et qu'aucune des données récoltées ne contient de biais de déclaration (si de nombreux utilisateurs ont aimé un article, l'évaluation de l'utilisateur peut être influencée).

Avantages et inconvénients Le plus grand avantage du filtrage collaboratif est qu'aucune connaissance du domaine n'est nécessaire, puisque la nature du produit n'a pas d'importance pour le système. De plus, ce système peut aider l'utilisateur à découvrir de nouveaux intérêts. Mais il a ses inconvénients comme le démarrage à froid (pour les nouveaux utilisateurs/articles), si un article n'est pas vu pendant la partie entrainement, le système ne peut pas créer de relation et ne sera pas en mesure d'interroger ce modèle (cependant certaines techniques comme la projection dans WALS peuvent aider à résoudre ce problème particulier). Il y a aussi le problème de la sparsité, sur internet, il y a tellement d'articles que même le plus populaire n'aura que quelques commentaires, ce qui rend le calcul plus difficile.

Filtrage basé sur le contenu Le filtrage basé sur le contenu exploite les similitudes entre les caractéristiques des produits et les préférences de l'utilisateur pour formuler des recommandations. Les préférences de l'utilisateur sont créées sur la base de ses actions précédentes, et les similarités entre les objets sont basées sur les mots-clés et les attributs calculés par le modèle. Par exemple, si un utilisateur achète un téléphone, le système de recommandation proposera probablement des étuis de téléphone, des films de protection d'écran ou d'autres accessoires de smartphone pour le bon modèle de smartphone, puisqu'ils auront probablement des mots clés communs comme smartphone, le nom du fabricant, le modèle, etc. Cette méthode est généralement adaptée aux applications qui contiennent des informations sur le produit (plus il y en a, mieux c'est), mais peu ou pas d'informations sur le profil de l'utilisateur.

Avantages et inconvénients Le principal avantage de cette méthode est que nous n'avons pas besoin des informations d'autres utilisateurs, contrairement au filtrage collaboratif, et que la recommandation est très pertinente pour l'utilisateur puisqu'elle est adaptée uniquement à ce que les utilisateurs ont acheté, la recommandation d'articles de niche est plus susceptible de se produire avec ce type de filtrage. Ce faisant, les recommandations sont plus compréhensibles par l'utilisateur. Avec le filtrage collaboratif, si la recommandation peut effectivement faire découvrir à l'utilisateur un nouvel intérêt, il est également possible que l'utilisateur ne soit pas du tout intéressé par ces nouveaux articles et ne comprenne pas pourquoi ils sont recommandés. Cependant, le système de recommandation sur mesure peut présenter un manque de diversité, ne recommandant que des éléments que l'utilisateur sait déjà qu'il va aimer. Le problème du démarrage à froid existe aussi d'une manière différente, un utilisateur doit avoir fait une action avant que le modèle puisse recommander avec pertinence, donc un nouvel utilisateur n'aura pas immédiatement une bonne recommandation. De plus, à chaque fois qu'un produit est ajouté, nous devons définir son mot-clé pour trouver des similarités entre les produits. Enfin, contrairement au filtrage collaboratif, nous avons besoin d'une connaissance du domaine pour définir les mots clés et les autres relations entre les articles et ce, pour chaque article, et ces liens entre les articles sont ce qui rend le système de recommandation bon ou mauvais, ils sont donc très importants.

Il existe d'autres techniques de filtrage (comme la recommandation par session ou la recommandation mobile), mais nous ne les verrons pas ici puisque les trois développées précédemment sont les principales.

Nous avions prévu d'avoir une version de l'application où de nombreux utilisateurs pourraient partager des recettes, créant ainsi un réseau entre les utilisateurs. Cependant, cela a été considéré comme trop compliqué puisque les algorithmes basés sur le contenu sont plus faciles à mettre en œuvre et qu'il s'agissait de notre premier système de recommandation (de plus, nous n'avions pas prévu d'avoir suffisamment d'utilisateurs pour créer un véritable réseau collaboratif entre eux, nous avons donc décidé de nous en tenir au filtrage basé sur le contenu). De plus, comme cette partie de l'application n'allait être mise en œuvre que tardivement dans le développement, nous ne pouvions pas attendre cette mise en œuvre pour travailler sur la partie Machine learning du projet. Nous avons donc opté pour un filtrage des données basé sur le contenu.

Pour ce projet, nous avons voulu mettre en œuvre un système de machine learning afin de recommander des recettes en fonction des informations de l'utilisateur. Nous avions deux idées de cas d'utilisation qui pouvaient être intéressantes pour notre application. Le premier était de recommander des recettes en fonction de ce que l'utilisateur avait à sa disposition dans sa cuisine, le second était de recommander des recettes en fonction des préférences et des goûts de l'utilisateur.

Nous avons pensé que le système de recommandation le plus simple serait celui basé sur les ingrédients restants. Sur la page d'accueil de l'utilisateur, nous enverrions au système de recommandation les ingrédients disponibles en ce moment dans la cuisine, et il vous proposerait une recette utilisant la plupart de ces ingrédients (avec la nécessité d'en ajouter certains bien sûr). Cependant, ce système ne semblait pas aussi intéressant que le second, nous avons donc décidé de passer au suivant.

Difficultés prédites Nous avons décidé de nous tourner vers l'autre système de recommandation : baser les recommandations sur les goûts de l'utilisateur. La mise en œuvre de cette méthode posait cependant deux gros problèmes.

- Si nous voulions faire une recommandation sur ce que l'utilisateur aimait, nous avions besoin d'un indicateur pour savoir si un utilisateur aimait une recette ou non. Nous avions déjà mis en place un bouton "J'aime", mais il s'agissait davantage d'un usage favori que d'un indicateur permettant de savoir si l'on veut toujours quelque chose de similaire. Par exemple, je veux accéder rapidement à ma recette de chocolat préférée, mais je ne veux pas que mon application ne me propose que de la tarte au chocolat ou autre. Nous ne voulions pas mettre en place un système d'évaluation, car il n'était pas présent dans la première version de l'application et ce type de

fonctionnalité est surtout utilisé pour les applications en ligne (applications avec de nombreux utilisateurs pour évaluer les recettes des autres utilisateurs). Nous avons donc pensé à un moyen de calculer combien de fois un utilisateur a vu la recette dans les dernières semaines/mois et dans l'ensemble (si un utilisateur fait une recette beaucoup de fois, il doit l'aimer) mais c'était en fait trop compliqué à mettre en œuvre dans le temps disponible, pour une première version au moins. Nous avons donc décidé d'envoyer au modèle une recette choisie pour sélectionner 10 recettes similaires.

- Il y avait maintenant le problème des données elles-mêmes. Comme tout algorithme d'apprentissage automatique, nous avons besoin d'un volume suffisant de données pour obtenir des résultats corrects. Et dans l'application elle-même, les recettes étant, dans les deux premières versions, ajoutées par l'utilisateur, nous n'avions pas assez de données pour entraîner notre modèle à trouver lui-même des similarités invisibles. Nous avons donc décidé d'importer un jeu de données de recettes préexistant au premier lancement de l'application (l'application a déjà pré-rempli des recettes) pour avoir un jeu de données plus conséquent sur lequel travailler.
- Un système de recommandation n'est pas un modèle d'apprentissage automatique comme les autres que nous avons vus auparavant, le jeu de données n'est pas séparé en deux parties : formation et test. La recommandation va trouver des similitudes entre différents éléments et baser sa recommandation sur ces similitudes. Ainsi, il n'est pas possible de séparer les données de l'utilisateur et les données importées, c'est pourquoi nous avons dû importer les données dans l'application, nous ne pouvions pas utiliser les nouvelles données uniquement pour la formation. Ainsi, chaque fois que l'utilisateur ajoute une nouvelle recette, nous devons régénérer un fichier et ré-entraîner le moteur pour le mettre à jour avec les nouvelles données. Ce n'est pas très optimisé et évolutif, mais c'est le plus gros problème des systèmes basés sur le contenu. Vous pouvez en fait l'ajouter en fonction de l'algorithme que vous utilisez en python, mais nous n'avons pas eu le temps de l'implémenter dans la version actuelle.

NLP (Natural language processing) Après toutes ces recherches, nous avons réalisé qu'avant d'implémenter un modèle de recommandation, il allait falloir utiliser les algorithmes NLP pour pré-processer les données, car les informations pertinentes d'une recette ne sont pas des simples chiffres ou mots (nous n'allons pas nous baser sur le temps de cuisson ou le nombre d'ingrédients), mais des textes relativement complexes.

Implémentation

Problèmes rencontrés

Front-end

Back-end

Mobile

Nous n'avons pas eu de difficultés particulières sur le développement.

Embarqué (thomas)

Limite de la caméra du pi pour lire des codes barre

Infra (thomas)

Limite d'avoir une base de données persistante, on a dû mettre une db dans la pipeline qui se crée automatiquement un db temporaire

Machine learning

Pour la partie recommandation, la plus grosse difficulté à été de trouver des informations pour créer un modèle adapter à la recommandation de recettes. En effet, comme nous n'avons jamais eu de cours sur les systèmes de recommandation ainsi que la gestion d'information textuelle, il a fallu fait un gros exercice de recherche pour comprendre le fonctionnement de NLP ainsi que des systèmes de recommandation (assez différent de système de classification ou de régression classique que nous avions déjà vu). Une fois un tutoriel assez clair trouvé, il a fallu corriger les erreurs contenues ce qui a été assez difficile.

Amélioration possible

Comme nous avons pu le voir entre l'application actuelle et celle que nous avions imaginée, nous n'avons pas eu le temps de développer toutes les fonctionnalités que nous avions souhaitées. Nous aurions particulièrement aimé réaliser la feature permettant de planifier les repas et de les présenter sous forme de calendrier. Au niveau du machine learning, nous aurions aussi aimé avoir plus de temps pour mieux gérer les hyper-paramètres du modèle obtenu, afin d'améliorer les recommandations. En effet, même si elles restent pertinentes, certaines recommandations peuvent être assez difficiles à comprendre. De plus, il serait particulièrement intéressant de réussir à intégrer ce système à notre application, et de trouver un moyen d'ajouter des recettes au modèle sans avoir à le réentrainer entièrement.