Mémoire d'alternance au sein de Pymma Software

Réalisé par
Fofana Cheick Tidiane

Travail présenté à
M. Nicolas Héron
M. Ioan-Marius Bilasco

Encadré par
M. Nicolas Héron
M. Ioan-Marius Bilasco

Du département d'Informatique
Faculté des Sciences et Technologie
Université De Lille
Cristal

20 Juin 2022

Table des matières

Introduction

Dans le cadre de ma formation en master « Machine Learning » à l'Université de Lille, j'ai effectué mon alternance au sein de la société Pymma Software. Durant cette alternance, j'ai été amené à travailler sur plusieurs sujets abordant les thématiques des systèmes de recommandations et du machine learning. Les sujets sur lesquels j'ai travaillé sont les suivants :

- Mise en place d'un système de recommandation utilisant le machine learning
- Mise en place d'un système de suggestion de termes associés
- Développement de traitements pour exploiter les logs d'activités des drivers et du solveur de contraintes Optaplanner
- Mise en place d'un système de prédiction de délais de prise de livraison

Ce mémoire a pour objet de présenter les travaux effectués durant ces deux années. Pour ce faire, nous commencerons dans un premier temps par présenter les travaux effectués avec les systèmes de recommandation. Ensuite, nous poursuivrons avec la procédure de mise en place d'un système de suggestion, avant de finaliser avec des travaux réalisés sur la mise en place de logs et leur exploitation pour la prédiction de retards de prise de colis dans le projet *Almady*.

Pour chacune de ces parties, nous présenterons si nécessaire le projet et le contexte dans lequel s'inscrit la mission réalisée. La structure de chaque partie reste globalement la même, à savoir, une mise en contexte, l'identification du/des problème(s) et la mise en place de solution(s) adaptée(s). Nous discuterons également des résultats obtenus pour chacune des solutions proposées.

1 Les Systèmes de Recommandations avec des données explicites

1.1 Contexte de Départ : recommandation de formations et métiers

L'Onisep (Office National d'Information sur les Enseignements et les Professions) est un opérateur de l'État qui élabore et diffuse des informations sur les formations et métiers. Pymma software a mis en place une application destinée aux élèves qui propose la découverte de formations et métiers. Dans cette application, les élèves reçoivent des recommandations en fonction de leur navigation. Le système de recommandation actuel repose sur le moteur de règles Drools. Ce moteur de règles incrémente/décrémente des scores liés à chaque formation ou métier en fonction de la navigation des élèves. Par exemple, un utilisateur qui "Like" le métier d'administrateur système verra le score des formations/métiers liés à l'informatique augmenter. Ainsi, le moteur de règles recommandera plus souvent des formations/métiers liés à l'informatique. Inversement, l'utilisateur a aussi la possibilité de "Dislike" une formation/métier, ce qui entraînera la diminution des scores associés.

1.2 Problème(s) identifié(s) : pas de recommandations basées sur les autres utilisateurs

Le système de recommandation utilisé par Onisep, reposant sur le moteur de règles Drools, propose des recommandations liées à la navigation des utilisateurs, cependant ce système souffre de l'effet "bulle de filtres". Un utilisateur ayant tendance à aimer les métiers liés à la finance, ne se verra le plus souvent proposés que des formations/métiers liés à la finance, l'empêchant ainsi parfois de découvrir d'autres domaines potentiellement intéressants. Une autre limite de cette technologie est que l'utilisateur ne profite pas de la navigation des autres utilisateurs avec des profils similaires ce qui pourrait en partie contrecarrer l'effet "bulle de filtres".

Une solution à la problématique soulevée plus haut est l'utilisation d'approche de "Machine Learning" permettant de mettre en place un système de recommandation parallèle prenant en compte les similarités entre les utilisateurs et de capturer les centres d'intérêts de ces utilisateurs.

1.3 Solution(s) proposée(s) ML

Dans cette section, nous présenterons les solutions proposées afin d'adresser les problèmes évoqués plus haut. Ces solutions devraient pouvoir tirer parti de l'aspect collectif de la navigation, et atténuer l'effet "bulle de filtres". Ci -dessous nous présenterons une solution adaptée appelé : filtrage collaboratif.

Le filtrage collaboratif [1](ou Collaborative Filtering) analyse les relations entre les utilisateurs et les interdépendances entre les items de recommandations pour identifier de nouvelles associations utilisateur-item. L'un des principaux attraits du filtrage collaboratif est qu'il n'est pas lié à un domaine particulier, et qu'il peut traiter des aspects des données qui sont souvent insaisissables et difficiles à profiler à l'aide d'autres méthodes de recommandation.

Bien qu'il soit adapté à notre problème, le filtrage collaboratif souffre de ce que l'on appelle le problème du démarrage à froid, en raison de son incapacité à traiter les nouveaux items et utilisateurs du système. Nous proposerons plus bas une solution simple à ce problème dans notre cas bien précis.

Les deux principaux domaines du filtrage collaboratif sont les méthodes de voisinage et les modèles de facteurs latents. Avant de vous présenter ces méthodes, nous allons expliquer comment les données sont modélisées et extraites.

1.3.1 Modélisation, Collecte & Extractions des données

L'objectif ici est de modéliser la navigation des utilisateurs. La navigation de l'utilisateur se définit comme les actions suivantes : aimer un item, ne pas aimer un item et marquer un item comme favori. Marquer un item en favori signifie qu'il s'agit de l'item préféré de l'utilisateur. Dans notre contexte, les items sont représentés par des formations et des métiers. La navigation (avis) des utilisateurs est sauvegardée dans une base de données MongoDB. La modélisation et la collecte de données a été réalisée en utilisant *Spring Data MongoDB*.

L'extraction des données consiste à extraire le contenu de la table dans laquelle sont stockés les avis des utilisateurs sous la forme d'un objet python (*Dataframe*) afin qu'ils soient accessibles pour l'étape d'apprentissage. Cette extraction est faite en python à travers la librairie *MongoClient*. Après extraction, nous obtenons le modèle de données suivant :


```
_id: "60c35e2d94e0c7743e37dfde1216305"
_class: "fr.onisep.moa.persist.ezPublish.model.MLRecommandationData"
class_identifier: "metier"
userId: "60c35e2d94e0c7743e37dfde"
class_identifier_object_id: 1216305
rating: 1
```


FIGURE 1 – Instance du modèle de données

Les attributs importants sont :

- **class_identifier** : représente le type d'item, il s'agit d'une formation ou d'un métier
- **userId** : représente l'identifiant de l'utilisateur
- **class_identifier_object_id** : représente l'identifiant de l'item avec lequel l'utilisateur interagit
- **rating** : représente l'avis donné par l'utilisateur à l'item donné

L'attribut **rating** prend les valeurs suivantes en fonction de l'avis de l'utilisateur :

A) **1** lorsque l'utilisateur u a aimé l'item i

B) **-1** lorsque l'utilisateur u n'a pas aimé l'item i

C) **2** lorsque l'utilisateur u a marqué comme favori l'item i

Il est important de noter que les choix de ces valeurs sont importants pour le processus d'apprentissage car, dans les étapes suivantes, nous serons amenés à faire des calculs de similarités entre les différents utilisateurs. Nous allons maintenant passer proprement dit au filtrage collaboratif.

1.3.2 Méthode de voisinage

Le principe de la méthode de voisinage consiste à trouver les utilisateurs qui partagent les mêmes centres d'intérêts. On les qualifiera ainsi d'utilisateurs voisins. Avec cette méthode, un utilisateur est représenté par un vecteur, qui se caractérise par l'ensemble de ces avis pour chaque item dans un ordre bien précis (Voir Figure 2). Les items sans avis sont remplacés par 0. Notons qu'avec cette représentation, on se retrouve avec des matrices très éparses car, en pratique, les utilisateurs ne donnent leurs avis que sur un ensemble très réduit de formations/métiers.

```
>>> user1 = [2,0,0,1,1,-1,1,2,0,0]
>>> user2 = [-1,2,0,1,-1,-1,0,1,0]
```

FIGURE 2 – Représentation d'un utilisateur

Afin de déterminer les utilisateurs "voisins" pour un utilisateur en particulier, il faut définir une mesure de similarité. Nous pouvons citer la similarité cosinus (1) ou encore mieux la similarité de Pearson (2) :

$$\text{cosine\_similarity(A,B)} = \cos(\theta) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{1}$$

$$\text{pearson\_similarity(A,B)} = \frac{(\mathbf{A}-\overline{\mathbf{A}})\cdot(\mathbf{B}-\overline{\mathbf{B}}))}{\|\mathbf{A}-\overline{\mathbf{A}}\|\|\mathbf{B}-\overline{\mathbf{B}}\|} \tag{2}$$

L'avantage de la similarité de Pearson par rapport à la similarité cosinus est qu'elle supprime le biais utilisateur. Ce biais utilisateur se caractérise par des utilisateurs qui attribuent très souvent des notes hautes/basses en moyenne. Avec cette mesure, la similarité entre tous les couples d'utilisateurs est calculée, puis les k utilisateurs les plus proches d'un utilisateur donné sont utilisés afin d'inférer l'avis de cet utilisateur.

Afin de calculer la complexité de cette méthode, nous définirons n comme le nombre d'utilisateurs, m le nombre de formations/métiers noté(e)s. Ainsi, cette méthode a une complexité de $\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \approx n^2$ pour le calcul de similarité entre les différents couples, de n pour déterminer la sélection des voisins, et de m pour le calcul de la note. La complexité globale est ainsi de $n^2 + n + m$, et peut être améliorée en $O(n+m)$ au détriment de l'espace mémoire pour stocker la matrice de similarité.

Cette méthode a pour avantage d'être simple à implémenter. Nous pouvons aussi travailler sur une base de données constamment mise à jour. Cependant, le désavantage principal est qu'il faut parcourir la table à chaque recommandation pour identifier les voisins. Cela peut rendre le système de recommandation beaucoup plus lent lorsque qu'il est sollicité très souvent même s'il existe des structures qui permettent de rendre plus efficaces les calculs arithmétiques sur ce type de matrice (*Compressed Sparse Matrix* de *scipy*).

1.3.3 Factorisation matricielle

L'une des approches par modèle est la factorisation matricielle. Pour faire simple, l'objectif est de définir tous les utilisateurs et les items à travers des critères appelés facteurs latents. Nous assumons leur existence, sans pour autant pouvoir y associer une sémantique. À travers les données, on pourra définir chaque utilisateur et chaque item par ces facteurs latents. Ces facteurs latents sont représentés par des vecteurs de tailles fixes que nous appellerons des *embeddings*. On devra apprendre un vecteur p_u pour chaque utilisateur et un vecteur q_i pour chaque item. Le vecteur p_u représente l'affinité de l'utilisateur u pour chaque facteur latent. Similairement, le vecteur q_i représente à quel point l'item i est caractérisé par chaque facteur latent. La note attribuée à un item par un utilisateur est calculée de la manière suivante :

$$\theta_{ui} = p_u \cdot q_i = \sum_{f \in facteurs\ latents} affinite\ de\ u\ pour\ f\ \times\ affinite\ de\ i\ pour\ f \tag{3}$$

En d'autres termes, si l'utilisateur u aime bien un facteur latent f et que l'item i est fortement caractérisé par ce facteur latent, alors, on voudrait que la note inférée θ_{ui} soit élevée. Et dans le cas contraire, elle devrait être basse si cet utilisateur n'aime pas ce facteur latent f. Ainsi, on doit trouver l'affinité de chaque utilisateur/item pour les différents facteurs latents. Cela peut être vu comme un problème d'optimisation avec comme objectif la minimisation de l'erreur empirique entre la note r_{ui} observée dans l'ensemble d'apprentissage et la note calculée θ_{ui} par notre modèle.

En considérant que le modèle ne dispose que d'un nombre restreint d'avis pour l'apprentissage, il peut facilement surapprendre (en apprenant à prédire spécifiquement les notes pour les couples utilisateur-item de l'ensemble d'entraînement) ou être influencé par des valeurs extrêmes. Cela justifie l'ajout d'un terme de régularisation dans l'équation. L'objectif de la régularisation est de réduire les risques d'obtenir des valeurs élevées pour chacun des vecteurs p_u et q_i. Considérons la matrice U comme l'ensemble des vecteurs p_u , V l'ensemble des vecteurs q_i et R l'ensemble des notes attribuées par les utilisateurs aux différents items (formations/métiers). Le problème d'optimisation se formalise de la manière suivante :

$$(\mathbf{U}, \mathbf{V}) = \text{argmin}_{\mathbf{U},\mathbf{V}} \sum_{\forall (u,i) \in S} (r_{ui} - \mathbf{U}_u \mathbf{V}_i^T)^2 + \lambda(||\mathbf{U}||_F^2 + ||\mathbf{V}||_F^2) \tag{4}$$

avec $||.||_F^2$ représente la norme Frobenius au carré, et λ le paramètre de régularisation.

Afin de résoudre le problème d'optimisation ci-dessus (Voir équation 4) plusieurs méthodes d'optimisations existent telles que la décomposition en valeurs singulière (SVD), la descente de gradient (SGD) ou encore *Alternating Least Squares* (ALS).

Ce problème ci-dessus peut être vu comme un problème de «*Low-rank approximation*». «*Low-rank approximation*» est un problème de minimisation dans lequel on essaie de réduire (selon une fonction de coût) la différence entre une matrice donnée et une approximation de cette matrice. D'après (**Eckart and Young, 1936**), les propriétés de l'opération de décomposition spectrale (SVD) sur la matrice donnée assurent l'obtention de la solution optimale en norme de Frobenius du problème de «*Low-rank approximation*». L'utilisation de SVD requiert une matrice R remplie, cependant la matrice R que nous avons est très éparse, ceci conduit à remplir la matrice R (par exemple avec des zéros ou la moyenne des avis de l'utilisateur ou de l'item). Cela implique des opérations coûteuses en calcul et en espace. Cette raison nous pousse plus à opter pour l'utilisation d'autres méthodes comme SGD.

Stochastic Gradient Descent (**SGD**) est une méthode d'optimisation itérative. Elle initialise d'abord les vecteurs p_u et q_i de manière aléatoire. Puis, l'étape suivante consiste à calculer l'erreur empirique entre la note observée r_{ui} et la note θ_{ui} calculée par notre modèle. Cette erreur sera ensuite utilisée pour mettre à jour la valeur de ces vecteurs. Cette étape est réitérée jusqu'à atteinte du critère d'arrêt (convergence ou nombre d'itération fixé). SGD à une complexité linéaire en la taille de la donnée ce qui est plutôt positif. Cependant elle est sensible à l'initialisation des vecteurs (p_u et q_i) et aussi aux hyperparamètres (learning rate α et le paramètre de régularisation λ).

ALS Alternating Least Squares (**ALS**) vise également à résoudre l'équation 4. Comme U et V sont tous deux inconnus, la fonction objective à minimiser dans l'équation (4) est non convexe. L'idée derrière ALS consiste à résoudre le problème de manière itérative, en fixant U (respectivement V) et en résolvant l'équation pour V(respectivement U). Cette méthode est plus stable que SGD et converge beaucoup plus rapidement, mais a une complexité beaucoup plus élevée car elle implique des inversions de matrices.

Dans l'implémentation mis en place, la méthode d'optimisation utilisée est Adam, une version améliorée de SGD avec un learning rate adaptatif (afin d'éviter les minimums locaux). Ce choix a été fait par rapport à la simplicité d'implémentation et d'évaluation de cette méthode avec des outils comme *Pytorch*.

Après apprentissage, cette méthode de factorisation matricielle projette les utilisateurs et les items dans un espace de k dimensions (k étant la taille des embeddings choisis) en s'assurant que les utilisateurs et les items proches se retrouvent proches dans cet espace de projection. Nous pouvons observer sur l'image 3 une représentation visuelle en 3D de ces vecteurs. Cette représentation a été obtenue en appliquant une méthode de réduction de dimensions sur tous ces vecteurs appelée **PCA**. La représentation est plutôt fiable car nous conservons 80% de la variance sur ces 3 dimensions. Les points bleus représentent les utilisateurs et les points rouges représentent les items. Étant donné que les notes entre utilisateurs et items sont calculées à travers le produit scalaire, nous déduisons que plus deux points sont proches et plus ils sont semblables.

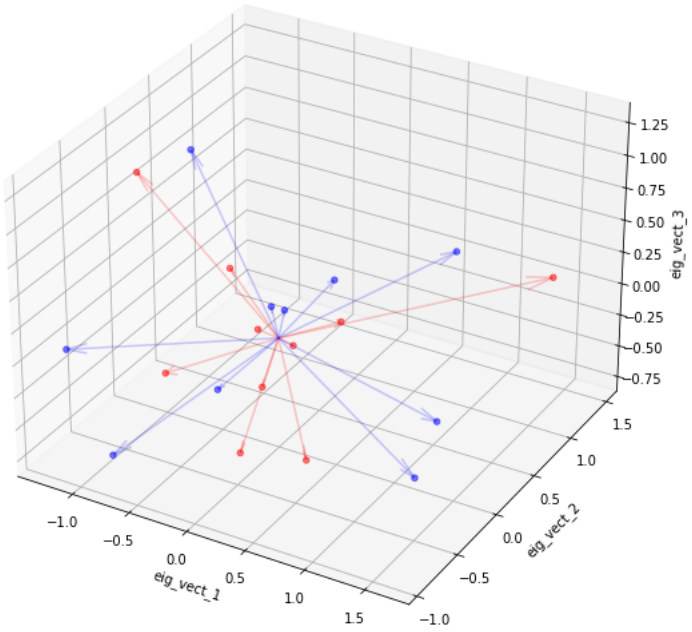


FIGURE 3 – Représentation visuelle des utilisateurs et des items

Globalement, cette méthode de factorisation matricielle comparée à la méthode de voisinage est plus rapide au niveau des prédictions, car nous ne recalculons pas toute la matrice de similarité. Elle utilise aussi beaucoup moins d'espace puisque nous avons juste besoin de matrices U et V qui sont beaucoup plus petites que la représentation des utilisateurs avec la méthode de voisinage.

1.4 Résultats : Comparaison des méthodes selon différentes métriques

Après l'implémentation des méthodes présentées ci-dessus, nous allons étudier les hyperparamètres de ces méthodes puis nous interpréterons les résultats obtenus. Afin d'évaluer les performances de nos algorithmes, l'introduction de métriques est nécessaire. Les approches par filtrage collaboratif ont historiquement visé à prédire les notes qu'attribueront les utilisateurs aux items, d'où l'utilisation des métriques suivantes basés sur l'accuracy :

— **Mean Absolute Error (MAE)** mesure l'écart absolu moyen entre un classement réel et un classement prédit. Il est facile à calculer, mais ne pénalise pas les erreurs importantes.

$$\text{MAE} = \frac{1}{|R|} \sum_{(u,i)\in R} |r_{u,i} - \widehat{r}_{u,i}| \tag{5}$$

— **Mean Squared Error (MSE)**, comparé à MAE, MSE pénalise de manière plus importante les gros écarts entre les prédictions et les vraies valeurs.

$$\text{MSE} = \frac{1}{|R|} \sum_{(u,i)\in R} (r_{u,i} - \widehat{r}_{u,i})^2 \tag{6}$$

— **Root Mean Squared Error (RMSE)** est la racine carré de la valeur de MSE, et est la métrique la plus utilisée. Il s'agit de la métrique standard utilisée dans les papiers de recherches.

$$\text{RMSE} = \sqrt{MSE} \tag{7}$$

avec R l'ensemble d'apprentissage, $r_{u,i}$ la vraie note et $\widehat{r}_{u,i}$ la note calculée par le modèle.

Pour ces différentes métriques, plus la valeur obtenue est basse, et plus le système est performant. Cependant, si les mesures de l'accuracy évaluent la capacité du système à estimer les notes, elles ne se concentrent pas sur la satisfaction de l'utilisateur. Il existe des mesures permettant d'évaluer plus précisément la capacité du système de recommandations à faire des recommandations pertinentes, en comparant la liste des éléments recommandés avec les vraies préférences des utilisateurs. Nous ne les citerons pas ici.

1.4.1 Étude des hyperparamètres

Les hyperparamètres sont des paramètres réglables qui vous permettent de contrôler le processus d'entraînement d'un modèle. Dans la méthode de voisinage, il s'agit du nombre de voisins, et pour la méthode de factorisation matricielle, nous avons le nombre d'itérations de l'apprentissage, le paramètre de régularisation λ, le taux d'apprentissage α et le nombre de facteurs latents.

Ci-dessous, nous pouvons observer la racine de l'erreur quadratique moyenne (Voir Equation7) obtenue sur l'ensemble d'apprentissage et l'ensemble de test (simulation des avis d'utilisateurs issu de la plateforme d'Onisep) en faisant varier ces hyperparamètres. Afin d'obtenir des résultats plus fiables, les expériences ont été réalisées plusieurs fois puis moyennées.

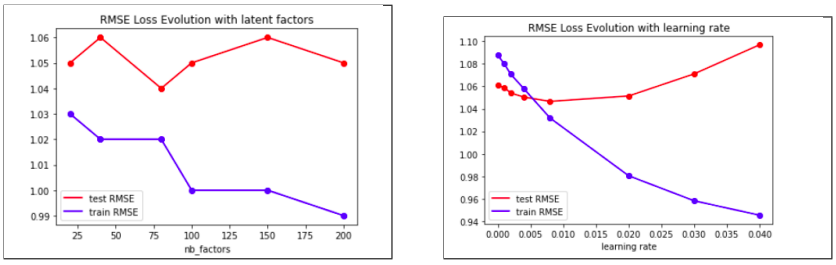


FIGURE 4 – Variation des hyperparamètres α et la taille des embeddings

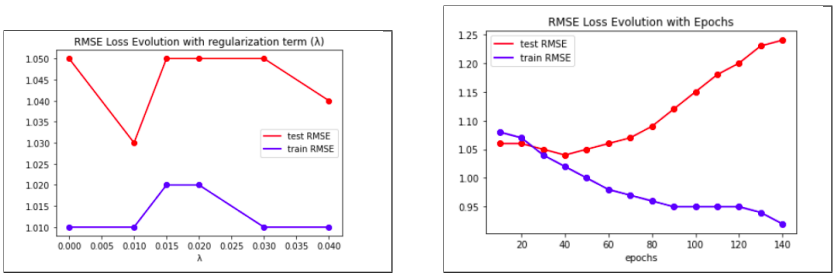
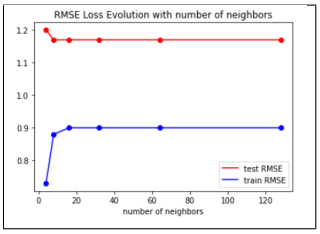


FIGURE 5 – Variation des hyperparamètres λ et du nombre d'itération

Nous observons que la perte est minimale sur l'ensemble de test lorsque le nombre de facteurs latents est de 80 (Figure 4). Il est quand même important de noter que les résultats ne sont pas lissés, ce qui peut cacher un manque de stabilité dans l'apprentissage. Concernant le taux d'apprentissage ou learning rate α, nous observons que la valeur optimale se situe entre 0.008 et 0.015 (Figure 4). Quant au taux de régularisation λ (variation entre 0 et 0.8), nous observons que la perte sur l'ensemble de test ne varie pas drastiquement (Figure 5). Enfin, après avoir fixé tous les hyperparamètres évoqués plus haut, nous observons que le nombre d'itérations nécessaire à la convergence tourne autour de 40 ((Figure 5)). Il est important d'évoquer que la valeur de ces hyperparamètres peut être amenée à évoluer, car la taille de l'ensemble d'apprentissage évolue avec le temps.




```
KNN : Training Done
RMSE[train]: 0.89
RMSE[test ]: 1.15
RMSE[valid]: 1.17
```

```
MF : Training Done
RMSE[train]: 1.02
RMSE[test ]: 1.04
RMSE[valid]: 1.01
```

FIGURE 6 – Variation du nombre de voisins, résultat ensemble de validation des méthodes

En ce qui concerne le nombre de voisins optimal k, nous observons sur la figure 6 qu'à partir de 20 voisins, la perte ne varie plus, nous pouvons en conclure que $k = 21$ est suffisant afin de déterminer de manière consistante l'avis de l'utilisateur.

Après adaptation des hyperparamètres sur les deux méthodes, nous avons une perte (RMSE) sur un ensemble de validation de 1.15 pour la méthode de voisinage et de 1.04 pour la méthode de factorisation matricielle (Voir Figure 6).

1.5 Intégration de la solution

Après avoir présenté les différentes solutions, nous expliquerons ici comment cette solution est intégrée dans l'application actuelle. Plusieurs scripts et fichiers de configurations ont été mis en place :

- **data.py** : permettant d'extraire les données depuis la base de donnée MongoDB.
- **train.py** : permettant d'entraîner un modèle sur les données extraites par le script *data.py* . L'exécution de ce fichier produit deux fichiers, un premier fichier *knn.pickle* qui représente le modèle utilisant la méthode de voisinage et un second fichier *matrix_factorization.pt* qui représente le modèle appris avec la méthode de factorisation matricielle. L'apprentissage est journalisé et ainsi pour chaque apprentissage, nous conservons une trace de la perte sur l'ensemble d'entraînement/test, les hyperparamètres et le temps d'entraînement (*learning.log*). Nous conservons aussi les 3 derniers modèles appris. Cela pourra être utile lorsqu'il faudra revenir sur une ancienne version d'un modèle.
- **requirement.txt** : contient la liste des librairies pythons nécessaire au bon fonctionnement des scripts pythons.
- **Dockerfile** : spécifiant les actions afin de créer une image docker.
- **recommandationMLService.py** : permet de créer un serveur (à travers la librairie *Flask* qui intègre les deux modèles appris précédemment. Ce serveur est accessible via une requête HTTP contenant l'UUID de l'utilisateur.

FIGURE 7 – Réponse API de recommandations

Une requête au serveur Flask renvoie un résultat similaire à celui sur l'image 7. Les attributs sont les suivants :

- *name* : représente l'identifiant de l'utilisateur.
- *recommandations* : liste d'identifiant d'item à recommander à l'utilisateur.
- *type_reco* : liste des types d'item associés à chaque recommandation (métier ou formation).
- *taille* : nombre de recommandations faites.

Les recommandations sont obtenues en utilisant les deux modèles (paramétrable afin de sélectionner qu'un seul des modèles), seuls les items recommandés par les deux modèles seront recommandés à l'utilisateur. En ce qui concerne l'une des limites du filtrage collaboratif qui est l'effet «*cold start*» (démarrage à froid), nous obligeons l'API à retourner des recommandations seulement si l'utilisateur a au moins 20 avis, ce qui permet d'avoir un minimum d'informations sur ses préférences. La réponse de la requête est ensuite utilisée afin de créer un objet Java «recommandation» qui sera ajouté aux recommandations de cet utilisateur.

L'ensemble des APIs indispensables au fonctionnement de l'application sont générées et instanciées avec Docker. Docker est une plateforme permettant de lancer certaines applications dans des conteneurs logiciels. Les fichiers présentés plus haut permettent de créer une image Docker. À travers cette image Docker, nous instancions un conteneur Docker avec l'application de recommandations accessible sur le port 5000.

Dans ce conteneur Docker, nous utilisons l'outil «*Cron*» qui permet d'exécuter des scripts périodiquement afin d'extraire les données et procéder à la mise à jour des modèles quotidiennement.

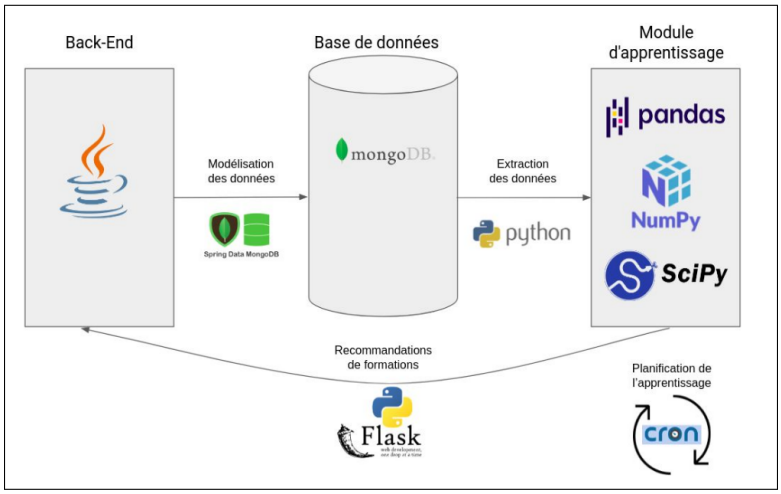


FIGURE 8 – Fonctionnement globale du système de recommandations

Ainsi, pour reprendre globalement le fonctionnement de ce nouveau système de recommandations, nous pouvons nous baser sur l'image 8. Les données sont extraites et utilisées pour l'entraînement quotidiennement. Les modèles générés sont ensuite utilisés par l'API de recommandations afin de fournir des recommandations aux utilisateurs. Tout ceci repose dans un conteneur logiciel qu'on peut générer en même temps que toutes les autres APIs à travers le fichier «*docker-compose.yaml*».

1.6 Bilan

Pour conclure cette première partie, nous allons récapituler les travaux qui ont été menés. Pour une remise en contexte, nous proposons un système de recommandation de formations/métiers à des apprenants en fonction de leur navigation. Ce système de recommandation devra tirer parti des utilisateurs avec des profils similaires, et aussi pouvoir contrecarrer l'effet bulle de filtre évoqué plus haut. La solution proposée dans cette section est la mise en place d'un système de recommandation utilisant le filtrage collaboratif. Dans un premier temps, un état de l'art a été réalisé sur le filtrage collaboratif menant à l'implémentation d'algorithmes basé sur les méthodes de voisinage et de facteurs latents. Ensuite, l'extraction des données a été réalisée en utilisant Spring Data. Après extraction, les algorithmes implémentés ont été évalués sur ces données, puis les hyperparamètres optimaux ont été sélectionnés. Un serveur a été développé pour de rendre notre système de recommandation accessible. Ce serveur lui-même étant encapsulé avec Docker afin de faciliter son déploiement. Un point d'amélioration aurait été intéressant d'implémenter des tests automatiques pour s'assurer du bon fonctionnement du serveur.

2 Les systèmes de recommandations avec des données implicites

2.1 Contexte de Départ : recommandation de formations et métiers

Dans cette section, nous continuons sur le projet Onisep et nous traiterons toujours la problématique de recommandations de formations/métiers aux collégiens/lycéens. Ici, nous nous intéresserons au potentiel intérêt d'un utilisateur pour un item. Nous n'essayons plus de prédire la note qu'attribuerait un utilisateur à un item, mais plutôt l'intérêt potentiel qu'un utilisateur aurait pour cet item.

2.2 Problème(s) identifié(s) : inférer intérêt des utilisateurs

Nous allons brièvement formaliser le problème ci-dessus puis nous discuterons des solutions existantes pour le filtrage collaboratif avec des données implicites (*implicit feedback*). Nous récapitulerons ensuite brièvement sur le modèle de factorisation matricielle précédemment utilisé, en mettant en évidence ses limites en raison de l'utilisation de l'opération de multiplication interne.

Nous définissons l'interaction d'un utilisateur u et d'un item i par $y_{ui} \in \{0,1\}$. $y_{ui} = 1$ lorsqu'il y a eu une interaction entre l'utilisateur u et l'item i, autrement $y_{ui} = 0$. Le problème de recommandation avec des données implicites peut être formulé comme un problème d'estimation de la probabilité qu'un utilisateur u soit intéressé par un item i avec lequel il n'a jamais interagi. Plus formellement, cela revient à apprendre $\hat{y}_{ui} = f(u, i|\theta)$, avec f représentant une fonction d'estimation (paramétré par θ) de probabilité d'interaction entre utilisateurs et items. Nous avons vu dans la section 1.3.3 qu'un estimateur adapté pour ce type de problème peut être un modèle basé sur la factorisation matricielle.

De manière informelle, nous rappelons que cette méthode vise à représenter les utilisateurs et les items dans un même espace à n-dimensions en s'assurant que le produit entre utilisateurs et items soit élevé lorsque l'utilisateur à des centres d'intérêts fortement caractérisés par cet item. Cependant, dans l'exemple qui suit, nous allons montrer que l'opération permettant de calculer y_{ui} qui combine simplement la multiplication (opération linéaire) des éléments des vecteurs latents de manière linéaire n'est pas forcément assez expressive pour capturer la structure complexe des interactions utilisateurs, surtout dans le contexte de données implicites.

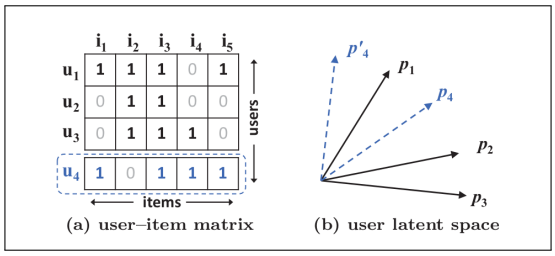


FIGURE 9 – Limite de la factorisation matricielle classique

Sur l'image ci-dessus, nous observons une matrice composée de 4 utilisateurs et 5 items. Les entrées à 0 signifient que l'utilisateur u n'a pas interagi avec l'item i, et 1 signifie qu'il a réagi avec l'item i. D'après la matrice (a), les avis de l'utilisateur u_4 se rapprochent beaucoup plus de u_1, suivi de u_3, et enfin de u_2. Cependant, en représentant les facteurs latents (extraits d'un modèle de factorisation matricielle entraîné) de ces utilisateurs (b), on s'aperçoit qu'en plaçant p_4 à côté de p_1, cela rendrait p_4 plus proche de p_2 que de p_3 ce qui engendrerait une grande erreur de classement.

L'exemple ci-dessus montre la limitation possible de la MF (factorisation matricielle) causée par l'opé-

ration de multiplication pour estimer les interactions complexes entre utilisateur et item. Une solution serait d'utiliser un grand nombre de facteurs latents K. Cependant, cela peut nuire à la généralisation de la méthode.

2.3 Solution(s) proposée(s) ML

Dans cette section, nous présenterons une nouvelle méthode nommée neuMF pour *Neural Matrix Factorization* proposé par « Xiangnan He et Al.» en 2017 [2] qui permettrait de mieux modéliser les interactions utilisateur-item grâce à des réseaux de neurones. Nous présenterons ici cette nouvelle méthode, ainsi que tout ce que ça implique d'implémenter et d'entraîner un tel modèle.

2.3.1 Modélisation, Collecte & Extractions des données

Ici, nous nous concentrons sur les commentaires implicites, qui reflète indirectement la préférence des utilisateurs à travers de comportements tels que regarder des vidéos, acheter des produits et cliquer sur des articles. Dans notre cas bien précis, il s'agit de cliquer sur les fiches de descriptions de formations/métiers. Par rapport au feedback explicite (c'est-à-dire les évaluations et les critiques), le feedback implicite peut être suivi automatiquement et est donc beaucoup plus facile à collecter. Cependant, il est plus difficile de l'utiliser, car la satisfaction des utilisateurs n'est pas directement observable.

Soient M et N le nombre d'utilisateurs et items, on définit la matrice d'interactions entre utilisateur-item $\mathbf{Y} \in \mathbb{R}^{M \times N}$ comme :

$$y_{ui} = \begin{cases} 1 & \text{si l'utilisateur } u \text{ à interagit avec l'item } i \\ 0 & \text{sinon.} \end{cases}$$

Nous obtenons ainsi une table donc la taille est au plus $M \times N$ contenant uniquement la valeur 1, contenant uniquement les utilisateurs et les items ayant interagi ensemble. La valeur 1 indique une interaction entre l'utilisateur u et l'item i ; cependant, cela ne signifie pas que u aime réellement i. De même, une valeur de 0 ne signifie pas nécessairement que u n'aime pas i, il se peut juste que l'utilisateur n'ait pas connaissance de cet item. Cela pose des problèmes pour l'apprentissage à partir de données implicites, car elles ne fournissent que des signaux bruyants sur la préférence des utilisateurs.

Dans notre cas, nous considérons qu'il y a une interaction entre un utilisateur et un item (formation/métier) lorsque cet utilisateur clique sur la fiche de description de cet item. La modélisation a été réalisée avec *Spring Data JPA* comme précédemment et les données sont stockées dans une base de données *MongoDB*, d'où l'utilisation de la librairie python *MongoDBClient* pour l'extraction des données.

2.3.2 Apprentissage supervisé : Neural Matrix Factorization

Comme présenté plus haut, nous abordons la recommandation d'item avec des feedbacks implicites comme un problème de classification binaire. Nous avons également vu que la factorisation matricielle n'était pas la plus adaptée et nous allons présenter ci-dessous une nouvelle méthode plus adaptée à cette tâche de classification. Nous décrirons cette méthode nommée **Neural Matrix Factorization** à travers son architecture (voir Image 10).

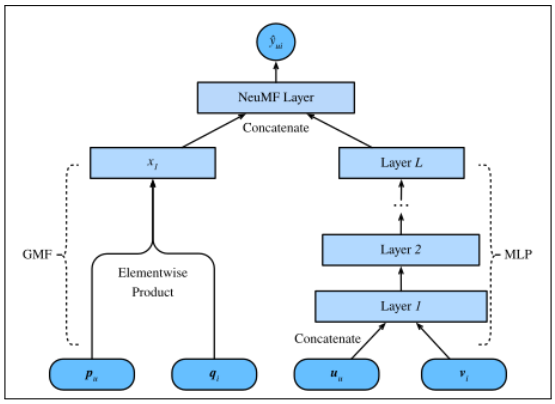


FIGURE 10 – Architecture de Neural Matrix Factorization

Sur l'image ci-dessus, nous observons l'architecture de **NeuMF** décomposable en 3 parties. Un premier sous réseau nommé **GMF** pour **Generalized Matrix Factorization**, et un second réseau nommé **MLP** pour **Multi-Layer Perceptron**. Les sorties de ces deux réseaux concaténés correspondent ensuite à l'entrée du dernier réseau de neurones, qui en sortie fournit la probabilité qu'un utilisateur u soit intéressé par un item i.

Generalized Matrix Factorization est un réseau de neurones avec une seule couche cachée qui prend en entrée les embeddings de u, i et nous fournit en sortie une représentation latente de l'interaction entre u et i. Il s'agit d'une généralisation de la méthode de factorisation matricielle en remplaçant l'opération interne de multiplication (linéaire) par une transformation non linéaire induite par la couche de neurones. Dans la configuration actuelle de **NeuMF**, nous utiliserons une spécialisation de **GMF** qui correspond à l'utilisation de l'opération de multiplication interne à la place de la couche cachée (Voir Image 10).

Multi-Layer Perceptron correspond au second réseau de neurones qui comme le précédent réseau prend en entrée les embeddings de u et de i et nous fournit en sortie une représentation latente de l'interaction entre u et i. Il s'agit de plusieurs couches denses de neurones (Voir Image 10). Nous utiliserons ReLU (pour Rectified Linear Unit qui permet tout simplement de remplacer les résultats négatifs par zéro) comme fonction d'activation de ces couches cachées.

Ce modèle réunit les forces de la linéarité de la **MF** (qui est une spécialisation de **GMF**) et de la non-linéarité de **MLP** pour modéliser de manière plus complexe les interactions entre utilisateurs et items.

Pour apporter à **NeuMF** une explication probabiliste, nous devons contraindre la sortie de ce modèle $\hat{y}_{ui}$ dans l'intervalle $[0, 1]$, ce qui peut être obtenu en utilisant une fonction probabiliste comme la fonction Logistique $\rho(x) = 1/(1 + e^x)$ pour la dernière couche de sortie. Nous utiliserons ReLU comme fonction d'activations des couches cachées pour le modèle **MLP**.

En ce qui concerne la fonction objectif, la plus adaptée pour ce problème de classification binaire est l'entropie croisée binaire :

$$L = -\sum_{(u,i)\in Y} log\ \hat{y}_{ui} - \sum_{(u,i)\in Y^-} log(1-\hat{y}_{ui}) = -\sum_{(u,i)\in Y\cup Y^-} y_{ui}\ log\ \hat{y}_{ui} + (1-y_{ui})\ log(1-\hat{y}_{ui}) \qquad (8)$$

avec Y représentant l'ensemble des interactions utilisateur-item et Y^- représente ce qu'on appellera des *negative samples*. Ces échantillons sont générés en sélectionnant aléatoirement des utilisateurs et des items auquels on attribuera la classe 0. Le taux de *negative samples* généré est paramétrable et est donc un hyperparamètre.

Cette méthode a été implémentée et dans la section 2.4, vous trouverez les résultats de la comparaison de cette méthode à d'autres méthodes puis de l'étude de ses hyperparamètres.

2.3.3 Apprentissage par renforcement : Adaptation du bandit manchot

Dans cette section, nous allons voir comment on peut utiliser une adaptation du problème du **Bandit Machot** ou **multi-armed bandit problem** afin de proposer une solution au problème d'inférence d'intérêts pour certains items. Nous évoquerons aussi la principale difficulté de cette méthode qui est l'évaluation [3].

Le problème Bandit Manchot modélise des situations où un agent, plongé à chaque instant t dans un certain contexte, doit choisir séquentiellement une suite d'actions a qui lui assurent un certain gain aléatoire s, et qui influent sur ses observations futures. Il s'agit de concevoir et d'analyser des règles de décision dynamiques, appelées *politiques*, utilisant les observations passées pour optimiser les choix futurs. Une bonne politique doit trouver un équilibre entre l'exploitation (dans notre contexte, utiliser nos connaissances sur l'utilisateur) des actions qui se sont révélées payantes par le passé et l'exploration de nouvelles possibilités qui pourraient s'avérer encore meilleures. L'objectif initial est de maximiser les gains futurs attendus. Par soucis de simplicité, nous ne formaliserons pas cette approche.

Nous allons nous restreindre à une version binaire du problème du bandit manchot appelée **Bernoulli Multi-armed Bandit** où le gain est binaire. Les actions possibles par l'agent à chaque instant t sont la suggestion d'un item appartenant à une catégorie particulière d'item. Par exemple :

- A1=suggérer une formation liée à l'informatique
- A2=suggérer un métier liée à la santé
- A3=suggérer un métier lié au sport

Les retours (gains) possible après chaque action correspondent à 1 lorsque l'utilisateur u aime l'item i suggéré et 0 lorsque u ignore l'item i. Ainsi, en récoltant les avis des utilisateurs, l'agent devrait apprendre les centres d'intérêts des utilisateurs. Il devra ensuite proposer des items aux utilisateurs en fonction de leurs centres d'intérêts (exploitation de sa connaissance des utilisateurs) ou prendre un risque d'explorer en proposant des items complètement aléatoire (exploration). Il s'agit du dilemme «*Exploitation-Exploration*»

La première approche qui vient à l'esprit est de calculer pour chaque action l'estimation de son espérance, puis de choisir l'action avec l'espérance la plus élevée. Une telle politique peut être trompée par des observations un peu malchanceuses lors des premiers essais de la meilleure action, et ne réalisera jamais son erreur si cette action n'est plus jamais sélectionnée. Il existe plusieurs algorithmes de bandits qui sont séparables en trois groupes. Nous les développerons très brièvement.

Upper-Confidence Bound se fie pour chaque action non à un estimateur de son espérance, mais plutôt à une borne supérieure de confiance (UCB). L'algorithme consiste à chaque instant t à calculer la borne supérieure suivante donnée par le théorème de *Hoeffding* :

$$\mathrm{UCB}(a) = \frac{S_a(t-1)}{N_a(t-1)} + \sqrt{\frac{f(t)}{2N_a(t-1)}} \tag{9}$$

avec N_a(t–1) le nombre de sélections de l'action a à l'instant t-1 et S_a(t–1) la récompense cumulée associée à cette action à l'instant t–1. Nous remarquons que le rapport de S_a(t–1) sur N_a(t–1) correspond à une estimation de l'espérance de l'action a, le second terme «uncertainty term» agit comme un régulateur de la valeur de UCB en fonction de la confiance de l'espérance obtenue, il dépend du nombre de fois que cette action a été tirée puis d'un paramètre $f(t)$ qui permet de régler le niveau de confiance.

Ainsi, plutôt que d'effectuer une exploration en sélectionnant simplement une action arbitraire choisie avec une probabilité qui reste constante, l'algorithme UCB modifie son équilibre exploration-exploitation à mesure qu'il acquiert davantage de connaissances sur l'environnement. Il passe de l'exploration, où les actions qui ont été le moins essayées sont préférées, à l'exploitation, en sélectionnant l'action dont la récompense estimée est la plus élevée.

Les méthodes **Softmax**, consistent à sélectionner une action selon une loi de probabilité sur l'ensemble des actions. Ces méthodes doivent maintenir à jour une loi de probabilité sur l'ensemble des actions et doivent veiller à ne pas donner toutes les chances à l'action qui semble la meilleure à l'instant t. La politique la plus représentative de cette famille est appelée «**Exponential Weights for Exploration and Exploitation**». Plusieurs variantes de cet algorithme existent, le plus simple se décrit de la manière suivante :

- donner un poids initial $w_a^t = 1$ à chaque action a.
- tirer à l'instant t, l'action a_t avec une probabilité $p_t(a)$ proportionnelle à son poids w_a^t.
- une fois la récompense S_t observée, mettre à jour le poids de l'action a_t en le multipliant par $exp(_t X_t / p_t(a))$, où $_t$ est un paramètre de l'algorithme. Ce facteur de multiplication est variable en fonction du problème et de l'intervalle des gains.

Thompson Sampling ou encore **Bayesian Bandits Algorithm** associe initialement une loi de probabilité *à priori* ou *prior* (généralement une loi uniforme) à chaque action. Pour chaque action a sélectionnée, cette loi est mise à jour en une loi à posteriori ou *posterior*. Pour déterminer l'action choisie à l'instant t, un tirage aléatoire (indépendant de tout le reste) est effectué sous la loi a posteriori associé à chaque action. Pour faire simple, cette méthode vise à construire itérativement une loi de probabilité pour chaque action en mettant à jour les paramètres de la loi *prior* choisie à l'initialisation. Pour notre problème, nous pouvons utiliser le « **Bernoulli Thompson Sampling** » qui utilise la *distribution Beta* comme loi de probabilité *prior* en initialisant ses paramètres α et β à 1, ces paramètres seront ensuite mis à jour après chaque interaction avec l'environnement. Ce dernier présente de grandes qualités pratiques et des garanties théoriques d'optimalité ont récemment été montrées.

La principale difficulté de ce type d'approche pour les systèmes de recommandation est l'étape d'évaluation de ces algorithmes. Il est le plus souvent impossible de mettre en place de vraies procédures de test où des recommandations sont envoyées à des utilisateurs dont on peut enregistrer les réactions. Pour résoudre cette difficulté, une solution classique consiste à utiliser un jeu statique de préférences exprimées pour mimer synthétiquement des interactions.

2.3.4 Apprentissage semisupervisé

Jusqu'à présent, toutes les solutions proposées n'utilisent aucune information autre que les avis des utilisateurs. Pourtant, nous disposons pour chaque item (formation ou métier) d'un ensemble d'informations utilisables. Chaque item est stocké dans une base de données **mongoDB** de manière structurée dans des collections. Nous avons ainsi la collection *EzpubFormation* qui contient l'ensemble des formations avec plus de 5000 documents et la collection *EzpubMetier* qui contient l'ensemble des métiers avec plus de 8000 documents. Chaque document contient pour chaque formation (respectivement métier) des informations relatives au métier (respectivement formation) lié à cet item et aux domaines (Sport, Santé, Informatique, Mathématique) qui caractérise cet item. L'idée est d'utiliser les informations que nous avons sur l'ensemble des items afin d'en faire un graphe de connaissance («*Knowledge Graph*») (Voir Image 11). Ce graphe liera les formations et métiers entre eux en fonction de ce qui les caractérise et ainsi faire de l'apprentissage dessus. Chaque item sera représenté par un nœud et une arête entre deux nœuds signifiera que ces deux items sont liés. La sélection du critère de liaisons entre deux nœuds n'est pas négligeable, car il ne faudrait ni un graphe avec trop d'arêtes et ni un graphe avec pas assez d'arêtes. La solution la plus simple et naturelle sera de mettre une arête entre deux nœuds si un l'un des items référence le second item.

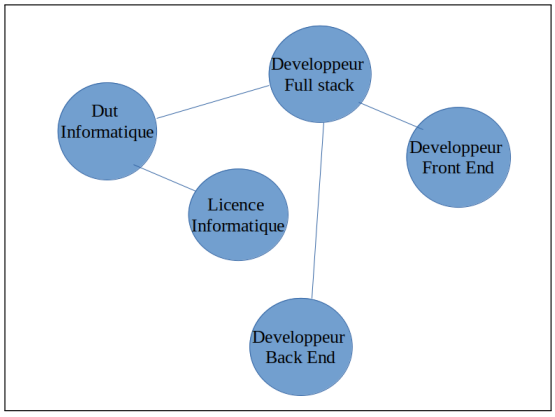


FIGURE 11 – Exemple de graphe de connaissance

Pour la construction de graphe, il faudra parcourir entièrement les collections *EzpubFormation* de taille N_f et *EzpubMetier* de taille N_m. Nous aurons ainsi $N = N_f + N_m$ nœuds et au plus $M = k \times N$ arêtes avec k représentant le nombre de références moyen des autres items ($k \ll N$). Après la construction du graphe, il faut maintenant associer un label à chaque nœud. Nous attribuerons la valeur 1 aux nœuds aimés ou simplement vus (sans avis négatif) par les utilisateurs, la valeur -1 pour les nœuds avec un avis explicitement négatif (Dislike). À la suite de la labellisation des nœuds, nous avons un graphe connexe avec des labels pour chaque nœud visité. Il est important de noter que le nombre de nœuds labellisés est très petits par rapport au nombre de nœuds total.

Il faut aussi décider si on conserve un graphe indépendant pour chaque utilisateur, ou si un graphe commun pour tous les utilisateurs est plus judicieux. La seule solution pour savoir lequel de ces choix est meilleur est la comparaison des résultats obtenus avec un grand jeu de données pour lequel nous avons beaucoup d'avis pour chaque utilisateur.

Pour parler du problème d'apprentissage proprement dit, nous avons un graphe avec un ensemble de nœuds labellisés (par 1 ou -1), et nous voudrions inférer en utilisant la structure du graphe (voisinage des nœuds) les labels des autres nœuds. Ce type de problème peut être traité en utilisant l'apprentissage semi-supervisé. L'apprentissage semi-supervisé est une approche de l'apprentissage automatique qui combine une petite quantité de données labellisées avec une grande quantité de données non labellisées durant l'apprentissage. Nous citerons ici des modèles d'apprentissage profond comme les Graph Convolutional Network (GCN)[4] ou encore plus récemment Self-supervised Graph Learning (SGL)[5]. Par souci de concision, nous nous limiterons à l'existence de cette approche et de ces modèles.

2.4 Résultats : Comparaison des résultats de NeuMF

NeuMF a été implémenté en utilisant *Pytorch*. La fonction de perte utilisée est l'entropie croisée binaire (Voir equation 8). Cette fonction sera optimisée en utilisant Adam. Les paramètres du modèle (embeddings, poids du réseau de neurones) sont initialisés aléatoirement selon une distribution gaussienne comme suggéré dans le papier de recherche [2]. En ce qui concerne les hyperparamètres, nous citerons la taille des batchs utilisés pour l'apprentissage, nous avons testé les tailles suivantes : [128, 256, 512, 1024]. Nous avons aussi fait varier le learning rate α (entre 0.0001 et 0.005). Le taux de *negative samples* ρ varie entre 1 et 5, 1 signifiant qu'il y a autant de sample négatifs d'interaction utilisateur-item et 5 signifiant qu'il y en a 5 fois plus. Nous ferons aussi varier le nombre de neurones dans les couches cachées de **MLP** et également la taille de la dernière couche cachée de **NeuMF** (voir Image 10) représentant la puissance d'expressivité de notre modèle. Sans mention spéciale, nous emploierons 3 couches cachées pour le **MLP** ; par exemple, si la sortie de **MLP** est de 8, l'architecture des couches neuronales sera $32 \rightarrow 16 \rightarrow 8$.

Dans le papier de recherche, il évoque l'importance de l'initialisation des paramètres dans la convergence du modèle, ainsi il suggère le *pre-training* qui correspond à un entraînement préalable des modèles **GMF** et **MLP** puis à l'utilisation de leurs paramètre comme paramètre de départ pour **NeuMF**. Nous avons fait une comparaison entre un modèle avec une initialisation Gaussienne et un second avec le **pre-training** et les résultats étaient identiques à un modèle entraîné sans pré-training avec juste plus d'itérations, ce qui semble logique.

Les modèles seront évalués sur les données extraites. Les données sont triées par *timestamp*. Les données sont composées de 60% des interactions de chaque utilisateur, les 40% restant correspondent aux dernières interactions des utilisateurs, 20% seront utilisés pour les tests et les 20% restant serviront d'ensemble de validation.

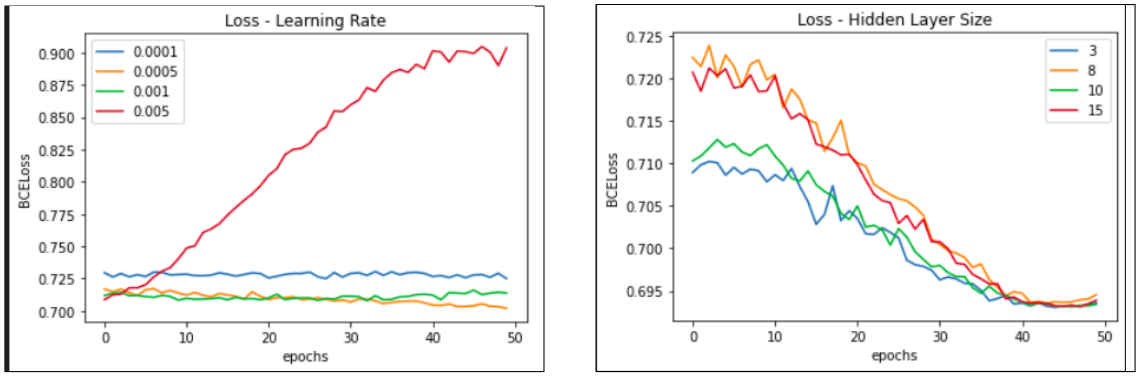


FIGURE 12 – Variation des hyperparamètres α et le nombre de couches cachées

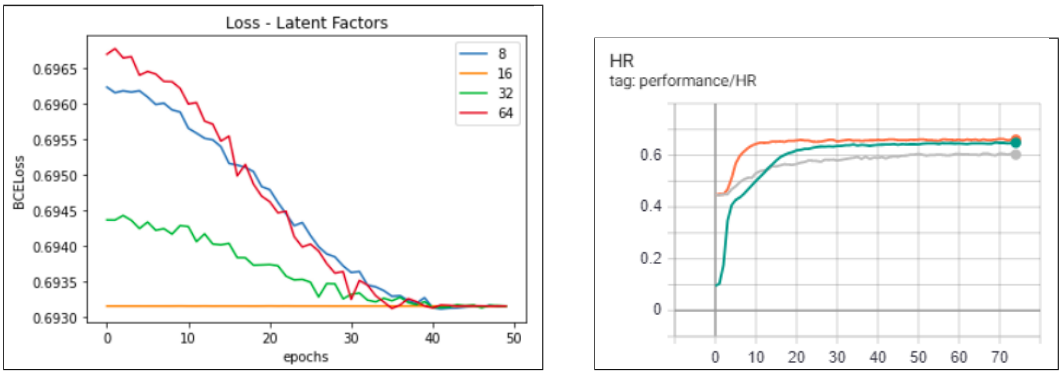


FIGURE 13 – Variation du nombre des facteurs latents et mesure du hit ratio entre les modèles NeuMF, GMF et MLP

Sur la figure 12, nous observons deux courbes d'apprentissage. Ces courbes d'apprentissages montrent l'évolution de l'entropie croisée binaire (**BCELoss**) du modèle NeuMF en faisant varier le learning rate et le nombre de couches cachées. Nous observons que $\alpha = 0.0005$ correspond au learning rate le plus adapté pour notre modèle et notre jeu de donnée, nous pouvons remarquer que la valeur 0.005 est trop élevée, raison pour laquelle le modèle ne converge pas. Ensuite, il est important de noter qu'après un certain nombre d'itérations, 3 couches cachées sont suffisantes afin de faire converger le modèle. Cela permet d'avoir un modèle plus léger. La figure 13 quant à elle montre comment le nombre de facteurs latents agit sur l'apprentissage et qu'à la suite d'un certain nombre d'itérations, des embeddings de taille 8 sont suffisants pour l'apprentissage. Un nombre d'embedding relativement bas permet une meilleure généralisation et permettrait d'éviter l'overfitting. Le dernier graphique montre la comparaison entre NeuMF, GMF et MLP. MLP est initialisé avec les paramètres de GMF. La métrique utilisée ici est le *Hit Ratio* (**HR**), il s'agit simplement de la fraction d'item bien recommandé sur l'ensemble des proposés, dans notre cas, il s'agit de l'ensemble des items recommandé sur tout l'ensemble de test. Sachant que nos modèles fournissent en sortie une probabilité, nous avons choisi un seuil à 0.8 pour considérer qu'un item intéresse un utilisateur. Ce seuil peut être considéré comme une mesure de confiance de la recommandation. Nous observons que le **HR** pour les modèles après 70 itérations est dans l'intervalle [0.6, 0.65], la courbe verte représente **MLP**, le modèle gris représente **GMF** et enfin la courbe orange représente **NeuMF**. Nous remarquons que le modèle **NeuMF** est légèrement mieux que **MLP** qui est aussi à son tour légèrement supérieur **GMF**. Nous ne pouvons pas juger la supériorité de ce modèle en utilisant juste cette métrique. Il serait judicieux d'introduire d'autres métriques comme le *Normalized Discounted Cumulative Gain* (**NDCG**).

2.4.1 Bilan

Dans cette seconde partie, nous restons toujours sur un problème de recommandation, mais là en utilisant les avis implicites des utilisateurs. Nous remarquons que cette partie est beaucoup plus axées sur des concepts non implémentés. Comme dans la section précédente, un état de l'art a aussi été réalisé et différentes approches ont été adoptées. Nous avons réimplémenté l'extraction des données et transformé les données de la section précédente afin les adapter à notre problème. Ces données ont ensuite été utilisées pour évaluer l'algorithme NeuMF selon différentes métriques. Cependant, cette nouvelle méthode évaluée sur notre jeu de données ne donne pas de résultats largement meilleurs que d'autres algorithmes plus simples. Les approches par apprentissage par renforcement et semi-supervisé n'ont malheureusement pas été implémentées et évaluées car le projet a dû prendre fin.

3 Un système de suggestion de termes de recherches associées

3.1 Contexte de Départ : barre de recherche de formation/métier

Dans cette section, nous continuons également sur le projet Onisep. Nous avons jusque-là traité des problématiques liées aux systèmes de recommandation utilisant les avis des utilisateurs afin de proposer des formations/métiers plus adaptés. Nous allons maintenant travailler sur un système de suggestion de termes de recherche.

3.2 Problème(s) identifié(s) : résultat non optimaux

L'application mise en place par Pymma Software permet de découvrir des formations/métiers et permet aussi de recommander des formations/métiers en fonction de la navigation des utilisateurs. L'un des désavantages de ce type d'application est qu'elle atténue fortement les interactions élève-enseignant. Dans notre cas, on peut citer la recherche de formations/métier par des collégiens. La présence de l'enseignant est nécessaire afin d'assister ces collégiens dans leurs recherches. Par exemple, un collégien recherchant le métier *"d'apiculteur"*, saisira dans la barre de recherche *"éleveur d'abeilles"*, avant d'être repris par son enseignant. Cependant, ce type d'interaction est impossible pour ces mêmes collégiens utilisant l'application depuis leur domicile. L'idéal pour ces collégiens serait d'avoir accès à ces suggestions à tout moment dès qu'ils saisissent dans la barre de recherche.

Ainsi, l'objectif est la mise un système de suggestion de termes aux collégiens mimant les interactions enseignant-élève. Nous voulons aussi pouvoir faire profiter tout utilisateur d'une correction apportée par un enseignant à un autre utilisateur.

3.3 Solution(s) proposée(s) ML

Ici, nous présenterons une solution afin de permettre de partager les corrections apportées par l'enseignant à tout utilisateur. Cette solution consiste à mettre en place un système de suggestion de termes associés (apportés potentiellement par l'enseignement) au terme saisi par le collégien. Pour ce faire, il faudra avant tout collecter l'ensemble des termes saisis par tous les utilisateurs, supprimer les termes non cohérents, puis inférer ce qui potentiellement est apporté par l'enseignant, avant de suggérer ce(s) termes(s) aux autres utilisateurs lorsqu'une recherche similaire (pas forcément les mêmes termes recherchés) est faite. Nous expliquerons plus en détails chacune de ces étapes plus bas.

3.3.1 Modélisation, Collecte & Extractions des données

L'objectif ici est de modéliser les recherches faites par les utilisateurs. Nous considérons qu'une correction apportée par un enseignant à un élève est également une recherche de terme. Chaque terme recherché par un utilisateur est sauvegardé dans une table. Le modèle de données pour une recherche est le suivant :


```
_id: ObjectId("60dc4625d8db5b6ea3e044eb")
_class: "fr.onisep.moa.persist.context.model.ContexteRecherche"
timer: 2021-06-30T10:23:33.178+00:00
UserId: "60c35e2d94e0c7743e37dfde"
time: 1625048613178
SearchExpression: "docteur"
```


FIGURE 14 – Instance du modèle de données d'une recherche

Les attributs importants sont :

— *userId* : représente l'identifiant de l'utilisateur

— *time* : nombre de millisecondes depuis le 1 janvier 1970 (utile pour l'étape de traitement plus bas)

— *SearchExpression* : correspond au terme recherché par l'utilisateur

La modélisation et la collecte de données a été réalisée en utilisant *Spring Data*, et les données sont stockées dans une BDD *MongoDB*. L'extraction des données consiste à extraire le contenu de la table dans laquelle est stockés les termes recherchés des utilisateurs sous la forme d'objet python (Dataframe) afin qu'ils soient accessibles pour les étapes suivantes. Cette extraction est faite en python à travers la librairie *MongoClient*.

3.4 Traitement des Données & Word2Vec

Les données extraites précédemment sont ensuite utilisées pour le traitement. Le traitement consiste à :

— Former des sous groupes

— Vérifier l'intégrité de chaque sous groupes

— Conserver que les sous groupes cohérents

La première étape consiste à former des sous groupes. Ces sous groupes sont formés selon les attributs *time* et *userId* évoqués précédemment, ainsi nous considérons qu'un ensemble de mots saisis par un même utilisateur dans un laps de temps bien défini correspond à un même sous-groupe (une même recherche). Nous considérons aussi que le dernier terme recherché par un utilisateur correspond à la correction suggérée par l'enseignant.

Ensuite, nous vérifions l'intégrité de chaque sous-groupe. Cela permet de supprimer les recherches hasardeuses et sans relation directe. La vérification de l'intégrité revient à vérifier que les mots d'un sous groupe sont des synonymes. Pour faire cela, nous utilisons *Gensim*, une librairie qui implémente des méthodes de Natural Language processing (**NLP**). Nous utiliserons l'algorithme **Word2vec**, un modèle de réseau de neurones qui apprend les associations de mots à partir d'un large corpus de textes. Une fois entraîné, ce modèle peut détecter des mots synonymes ou suggérer des mots supplémentaires pour une phrase partielle. Comme son nom l'indique, **Word2vec** représente chaque mot distinct par un vecteur. Les vecteurs sont choisis tels que la similarité cosinus (Voir Equation 1) entre ces vecteurs indique le niveau de similarité sémantique entre les mots représentés. Ces vecteurs sont également connus sous le nom de *Word Embedding*. Le modèle **Word2vec** utilisé ici est déjà entraîné sur un corpus de textes en Français.

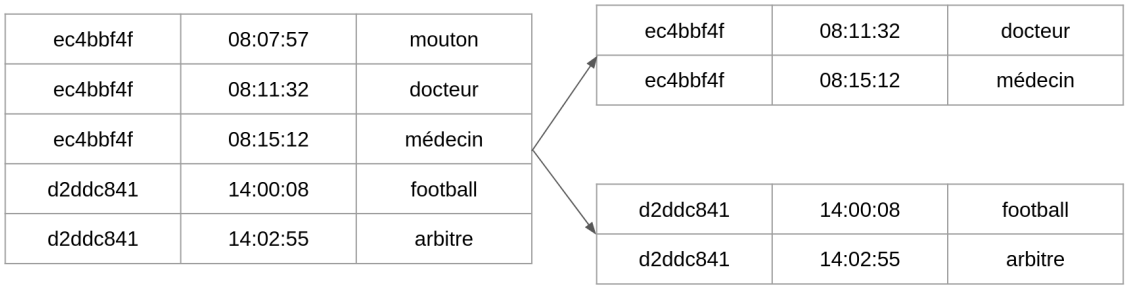


FIGURE 15 – Exemple de formation des sous-groupes

L'image 15 ci-dessous illustre avec un exemple formation des sous groupes basés sur le *userId* et le *timestamp*. Nous utilisons ici un format horaire plus compréhensible juste pour l'exemple. Nous observons la formation de deux groupes et la suppression de la recherche hasardeuse du terme « mouton » durant le filtrage des sous-groupes. Ce filtrage est obtenu en utilisant le modèle **Word2vec** afin de mesurer la similarité cosinus entre chaque couple de mots dans chaque sous groupe. Cette mesure est ensuite utilisée pour supprimer les mots incohérents ou les groupes incohérents. Il est important de préciser qu'un groupe est composé d'au moins de deux mots.

À partir de ces sous-groupes, nous formons un dictionnaire. La clé sera le mot cherché initialement par un utilisateur, puis la valeur sera le terme apporté par l'enseignant. Ainsi, pour chaque recherche d'un utilisateur, si le terme recherché est un synonyme d'une des clés du dictionnaire, alors la valeur est renvoyée. Cette valeur sera l'un des termes à suggérer.

3.5 Résultat : Démonstration d'exécution

Ici, nous montrons brièvement un exemple d'exécution de la solution proposée plus haut.

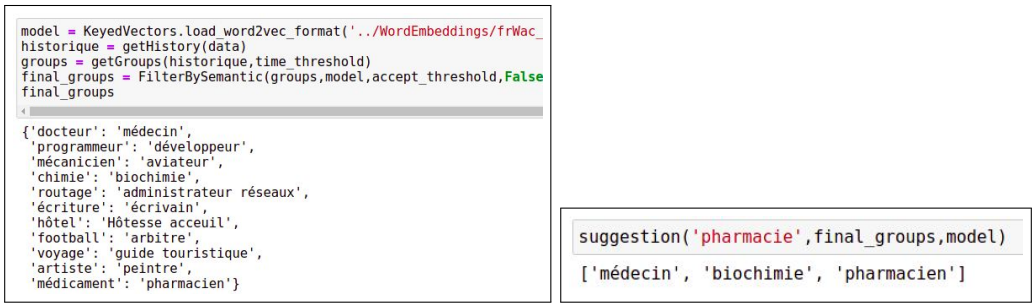


FIGURE 16 – Dictionnaire de suggestion et exemple de suggestion

Nous observons sur l'image 16, un exemple d'exécution, nous voyons le chargement du modèle, l'extraction des données avec la méthode *getHistory*, la formation des groupes puis le filtrage sémantique des groupes qui renvoie un dictionnaire. Nous voyons que les couples clé-valeur sont très cohérents. On peut affiner cette cohérence en variant le seuil (d'acceptabilité de la mesure cosinus) lors de la formation des groupes. Sur l'image 16, nous observons aussi le résultat de notre solution après avoir reçu en entrée un terme saisi par un utilisateur. La solution exploite le dictionnaire créé ou mis à jour lors de l'entraînement en utilisant le modèle **Word2Vec**. Les trois mots suggérés sont très cohérents.

3.6 Intégration de la solution

Nous allons maintenant expliquer comment cette solution a été intégrée dans l'application actuelle. Cette section ressemble fortement à la section 1.5. Nous avons plusieurs scripts :

- **data.py** : permettant d'extraire les données depuis la BDD MongoDB.
- **fit.py** : permettant de faire tout le traitement depuis l'importation des données fourni par **data.py** jusqu'à la génération du dictionnaire. L'exécution de ce fichier produit « *dictionnary.pickle* » qui représente le dictionnaire appris. Lorsque ce fichier *dictionnary.pickle* existe déjà, il est chargé puis les nouveaux couples clé-valeur sont ajoutés à celui-ci. Comme précédemment, l'apprentissage est journalisé.
- **suggestionMLService.py** : permet de créer un serveur (à travers la librairie « Flask ») qui intègre la fonction de suggestion précédente utilisant le dictionnaire appris préalablement. Ce serveur est accessible via une requête HTTP contenant le terme recherché de l'utilisateur.
- **requirement.txt** : contient la liste des librairies Python nécessaires au bon fonctionnement des scripts Python.
- **Dockerfile** : spécifiant les actions afin de créer une image docker.
- **frWac_non_lem_no_postag_no_phrase_200_cbow_cut100.bin** : fichier binaire permettant l'instanciation du modèle **Word2vec**.

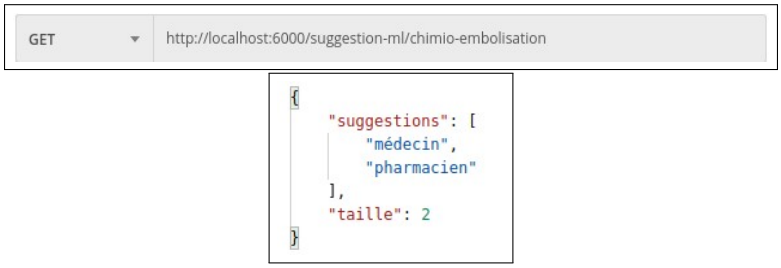


FIGURE 17 – Requête HTTP sur port 6000 et réponse de l'api de suggestion

Une requête au serveur Flask renvoie un résultat similaire à celui sur ci-dessus (Voir Image 17). Les attributs *suggestion* qui représentent les termes suggérés à l'utilisateur et *taille* qui correspond nombre de termes suggérés. Ce résultat est ensuite utilisé pour obtenir toutes les fiches liées aux termes suggérés. Ces fiches seront ensuite ajoutées aux suggestions affichées pour cet utilisateur.

4 Un système de prédiction de retards dans la chaîne logistique

4.1 Contexte de Départ : Livraison de colis

Almady est une entreprise de logistique saoudienne basée à Riyad. Cette entreprise assure la livraison à domicile des colis des clients. Elle emploie des coursiers afin d'assurer des livraisons. Ces coursiers ont des disponibilités particulières, une position géographique et des véhicules (avec différentes tailles et autres caractéristiques). Cela représente un problème classique de la logistique, mais particulièrement complexe. Pymma Software a mis en place un système de planification permettant d'attribuer à chaque coursier une ensemble de livraisons possibles en fonction de ses disponibilités, sa position géographique et son type de véhicule.

4.2 Problème(s) identifié(s) : Retard potentiel de prise de colis

Le problème d'attribution de colis aux coursiers peut être modélisé comme un problème de planification de tâches sous certaines contraintes. Il s'agit d'un problème d'optimisation multiobjectifs. L'objectif est de minimiser le coût de livraison et le temps de livraison en prenant en compte les contraintes liées aux coursiers (disponibilité, véhicule, position) et aux colis (position, taille, fragilité, conservation). Pymma software a utilisé un outil nommé **OptaPlanner**. **OptaPlanner** est un solveur de contraintes permettant de résoudre des problèmes d'optimisation. Il est basé sur l'utilisation des métaheuristiques (*Tabu Search*, *Simulated Annealing*, *Late Acceptance*, etc...) et fournit des résultats très efficaces en trouvant un compromis entre toutes ces contraintes. Certains colis peuvent transiter entre plusieurs coursiers avant d'atteindre leur destination finale. Ces colis sont déposés dans des points relais avant d'être récupérés par le coursier suivant. Cependant, **Optaplanner** ne peut pas prévoir dans ses planifications le retard d'un coursier (embouteillages, pannes), ce qui peut être gênant, car cela entraîne récursivement des retards de prise/livraison de colis par les coursiers suivants.

L'objectif ici, sera de pouvoir prédire en fonction des données que nous avons sur les déplacements des coursiers, un retard potentiel de la prise/livraison de colis par un coursier. Cette information pourra être utilisée dans un premier temps pour prévenir le client d'un délai sur sa livraison. Un usage ultérieur de cette information serait son ajout à **Optaplanner** afin de la prendre en compte dans le problème d'optimisation. À cette étape de mon alternance, seule la collecte des données a été réalisée. Nous parlerons néanmoins des étapes que nous réaliserons par la suite.

4.3 Solution(s) proposée(s)

Des travaux non négligeables ont été réalisés afin de *logger* entièrement l'application. Ces travaux ont été réalisés de paire avec un autre alternant nommé Noé Andrianarifidy. Davantage de logs ont été ajoutés notamment autour de l'outil d'Optaplanner, depuis la réception des commandes jusqu'à leur envoi à Optaplanner. Une interface graphique a également été mise en place afin d'accroître la tracabilité des trajets en cours d'optimisation. Nous apportons plus de détails ci-dessous.

Après la réception d'une commande, des objets Optaplanner sont générés à travers les informations de la commande. Ces objets sont ensuite envoyés à Optaplanner à travers JMS. JMS (**Java Message Service**) est une API qui permet de créer, d'envoyer et de lire des messages. Elle permet une communication asynchrone et fiable entre différents services. JMS permet d'envoyer directement des objets Java ce qui est très pratique puisque l'API et optaplanner fonctionnent en java. Après réception de ces objets, **Optaplanner** génère des trajets qui sont ensuite enregistrés. La figure 18 ci-dessous nous montre un schéma du fonctionnement global.

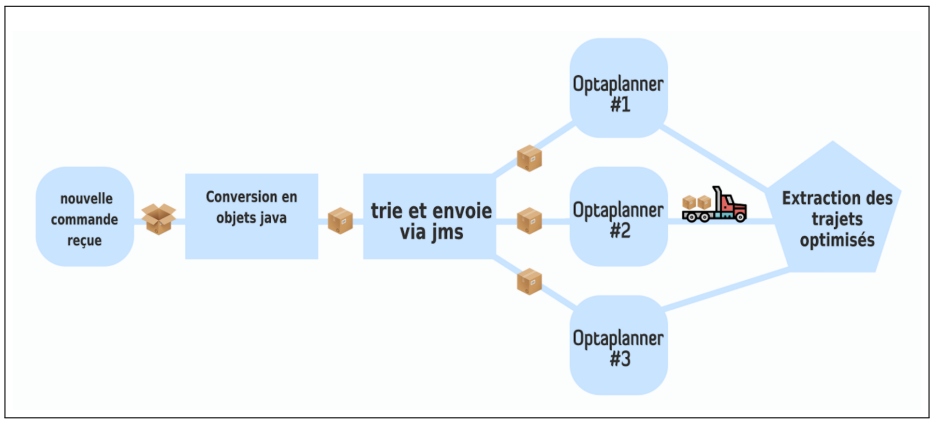


FIGURE 18 – Fonctionnement global Optaplanner

Tous les logs générés lors des étapes citées plus haut sont envoyés à **Logstash** qui est un pipeline côté serveur, qui structure ces informations puis les stocke dans **ElasticSearch**. ElasticSearch est un moteur de recherche permettant de stocker des grandes quantités de données auxquelles on peut accéder rapidement à travers des index. Ces données ont ensuite été utilisées pour mettre en place une interface graphique utilisable par les administrateurs afin de mieux monitorer le fonctionnement global de l'application, depuis l'enregistrement des drivers, la réception de commande, jusqu'à l'optimisation des trajets. Avant la mise en place de cette application, les administrateurs étaient obligés de rechercher manuellement dans la base de données les objets à l'origine des bugs.

En ce qui concerne plus spécifiquement les informations nécessaires pour la prédiction des délais, toutes les étapes concernant la livraison d'un colis ont été également loggées. Nous citerons non exhaustivement les informations des coursiers et des colis. À ce jour, Noé et moi travaillons sur l'ajout potentiel de plus de logs comme les informations sur la météo et d'avantage d'informations sur les coursiers/colis.

Après l'ajout de tous les logs nécessaires et la réception de données depuis la production, l'étape suivante serait d'extraire ces données avec ElasticSearch, les traiter, puis d'en faire une analyse descriptive des données. Cette analyse permettra de comprendre ces données et peut-être voir des corrélations évidentes par exemple la relation entre les conditions météorologiques et les retards. À la suite de cette analyse, nous définirons plus concrètement le problème de prédiction de délais sous la forme d'un problème de régression ou classification. Ensuite, nous évaluerons différents algorithmes sur ce jeu de données afin de décider si les résultats obtenus sont assez précis et crédibles dans le but de les exploiter.

Conclusion

Durant ces deux années au sein de Pymma Software, j'ai principalement travaillé sur deux projets liés à l'éducation et à la logistique. J'ai travaillé sur des projets concrets qui impliquaient des données réelles à travers lesquelles j'ai pu expérimenter des algorithmes de recommandation avancés, un domaine qui m'était étranger jusque là. J'ai eu l'occasion de mettre en œuvre mes connaissances acquises durant mon master, notamment sur les bonnes méthodes et pratiques à adopter. J'ai aussi eu à modéliser/collecter des données sur le projet *Onisep* et utiliser des outils de conteneurisation, des aspects non forcément abordés en détails durant ma formation. J'ai travaillé en collaboration avec Noé sur le projet *Almady*, sur lequel nous avons tous les deux loggé une grande partie de l'application et stocké ces informations. Des informations qui ont ensuite servi à mettre en place une application ayant beaucoup facilité la gestion par les administrateurs. Par cette même occasion, j'ai découvert ElasticSearch, Logstash et Kibana et d'autres frameworks Java comme Spring Boot et Spring Data. J'ai principalement travaillé en autonomie sur la première partie de mon alternance avec le projet *Onisep*. Cette partie est beaucoup moins technologique et plus formelle. En ce qui concerne la seconde partie de mon alternance sur le projet *Almady*, elle était un peu plus technologique et moins liée au machine learning proprement dit. Cependant, cette seconde partie était nécessaire afin de pouvoir collecter ultérieurement des données dans le but d'en extraire des connaissances intéressantes pour le client. Par la suite, l'utilisation de ces données avec du machine learning serviront à potentiellement prendre des décisions en temps réel.

Références

[1] Frédéric Guillou. *On Recommendation Systems in a Sequential Context. Machine Learning* [cs.LG]. Université Lille 3, 2016. English.

[2] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, Tat-Seng Chua. *Neural Collaborative Filtering* [cs.IR]. 2017.

[3] Jonathan Louëdec , Max Chevalier, Aurélien Garivier, Josiane Mothe. *Systèmes de recommandations : algorithmes de bandits et évaluation expérimentale* [cs.LG]. Université de Toulouse. 2015.

[4] Thomas N. Kipf, Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks* [cs.LG]. 2017.

[5] Jiancan Wu, Xiang Wang, Fuli Feng, Xiangnan He, Liang Chen, Jianxun Lian, and Xing Xie. *Self-supervised Graph Learning for Recommendation* [cs.IR]. 2021.