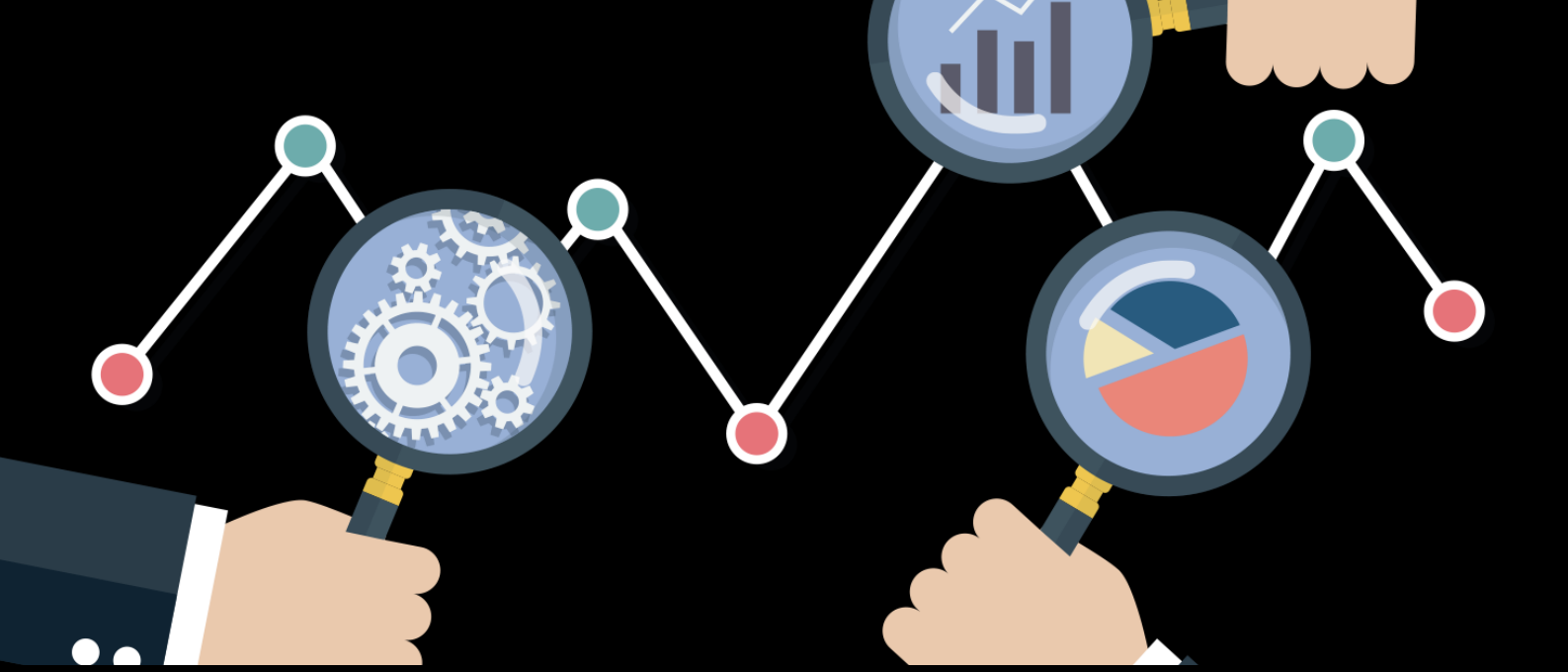


UNIVERSITÉ DE TECHNOLOGIE DE BELFORT-MONTBÉLIARD

Construction d'un moteur de recommandation E-commerce

DS50 – Projet Data Science

BELMONTE Elian
BOUNNIT Zakaria
GHARBI Aziz
MIGNEROT Grégori
PLANCHE Antoine
STACH Benjamin

Professeur responsable

Serge IOVLEFF

utbm
université de technologie
Belfort-Montbéliard

Introduction

Dans le cadre du bloc métier Data Science à l'UTBM, nous avons réalisé un projet de développement complet sous l'UV DS50. Notre projet vise à la création d'un site web de type e-commerce, comprenant un moteur de recommandation.

Pour cela, nous sommes partis d'un jeu de données venant du site web de recommandation de livres ***Goodreads***, et avons ainsi créé un site web de vente de livres. Ce dataset est en effet large, relativement complet, et porte sur des objets propices à ce type d'applications.

Nous avons ainsi réalisé un projet de data science complet, du prétraitement des données, au système de recommandation, un back-end et un front-end pour notre site. Celui-ci se trouve actuellement à l'adresse ***https://ds50-bookstore.mdbgo.io/***. Ce rapport décrit le travail derrière cette réalisation, et comment le tout a été conçu et implémenté.

Les codes développés dans le cadre de ce projet peuvent être retrouvés dans les dépôts GitHub suivants :

- Préprocessing et notebooks : ***https://github.com/Benjouzz/DS50_backend/***
- API : ***https://github.com/elianb11/flask_web_api***
- Front-end : ***https://github.com/ZackBNT/DS50_Frontend***

Table des matières

Prétraitement

Le dataset

Le *dataset d'origine* [1] [2] est un ensemble de données sur les **livres et les interactions des utilisateurs** avec eux, tiré du site web *Goodreads*. Il est constitué de plusieurs tables. Deux formats différents sont utilisés pour ces fichiers : CSV pour les interactions et les fichiers associés, et des fichiers avec un objet JSON par ligne pour les autres :

- `goodreads_books.json` : informations sur les livres et les « étagères » sur lesquelles les utilisateurs les ont mis (listes de lecture nommées, que nous utilisons comme des « tags »)
- `goodreads_book_authors.json` : informations sur les auteurs des livres
- `goodreads_book_series.json` : informations sur les séries de livres
- `goodreads_reviews_dedup.json` : commentaires des utilisateurs sur les livres
- `goodreads_interactions.csv` : interactions des utilisateurs avec les livres : indique qui a mis quel livre sur une de ses étagères, a lu ou commenté quel livre.

Nous avons préféré catégoriser nous-mêmes les tags sans utiliser le fichier `goodreads_book_genres_initial.json`. En effet, ce dernier donne une bonne catégorisation mais dans seulement une dizaine de genres, qui sont donc beaucoup trop génériques pour être vraiment pertinents dans nos recommandations.

Le pipeline de préprocessing

Le prétraitement est réalisé sous forme d'un pipeline organisant différentes étapes de façon cohérente. Celui-ci est mené par un programme Python qui gère ces étapes de façon à les exécuter en une commande, et de la façon la plus efficace possible, parallélisant au maximum les opérations.

Échantillonnage cohérent

Le dataset de base fait près de 30 Go, une quantité bien trop grande pour que nous puissions l'utiliser efficacement avec nos ressources.

Les premières étapes construisent donc un **échantillon** de ce dataset : au final, notre site utilise des échantillons d'environ 2,5 Go. Cependant, il n'est pas suffisant de prendre un certain nombre de lignes aléatoires de chaque table, car l'échantillon doit être **cohérent** : nous devons sélectionner des livres, des utilisateurs, les séries auxquelles appartiennent les livres, les autres livres de ces mêmes séries, les auteurs des livres, les interactions entre les utilisateurs et ces livres, etc. L'objectif est ainsi d'avoir un échantillon **entièrement cohérent**, sans aucune ligne « orpheline » qui pointerait sur un objet qui n'est pas dans l'échantillon ou qui ne serait pas liée au reste.

Pour cela, nous conditionnons la taille du dataset aux deux paramètres les plus importants et qui ne dépendent d'aucun autre : le nombre de livres, et le nombre d'utilisateurs à garder.

Échantillonnage des livres

Le nombre de livres initial est un paramètre du pipeline, que l'on a fixé à 30 000 (sur environ 2 360 000). Ces livres sont sélectionnés aléatoirement par un échantillonnage de type **roulette**, où les poids sont calculés à partir du nombre de commentaires et de notes accompagnant le livre. Ce type de sélection revient à choisir chaque livre à sélectionner en fonction d'un poids : plus le poids d'un élément est élevé, plus il a de chances d'être sélectionné.

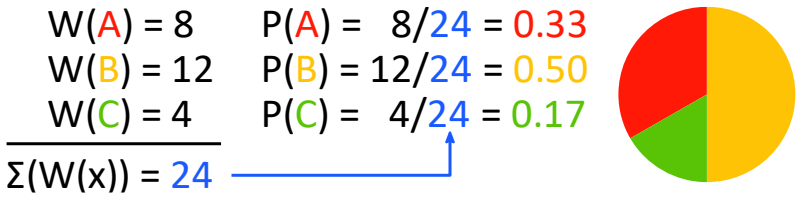


Figure 1 : Principe de la sélection aléatoire de type roulette

Ce type de sélection offre plusieurs avantages. Premièrement, l'échantillon est aléatoire, donc les données finales ne sont pas figées. On a ainsi plus de chances de repérer les éventuels problèmes. En outre, la sélection de type roulette permet d'avoir une meilleure répartition des données. En effet, une sélection purement aléatoire favoriserait une grande quantité de livres avec très peu d'interactions ; la roulette permet de recentrer la répartition de nos données afin de garder une majorité de livres « intéressants » :

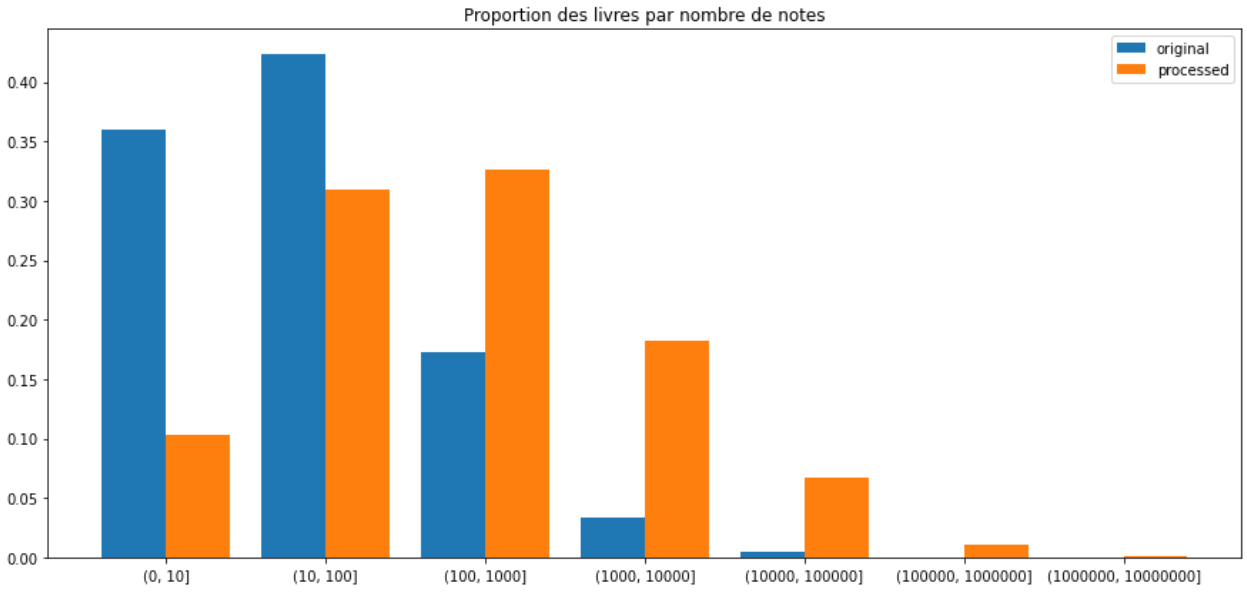


Figure 2 : Proportion de livres par nombre de notes avant et après échantillonnage

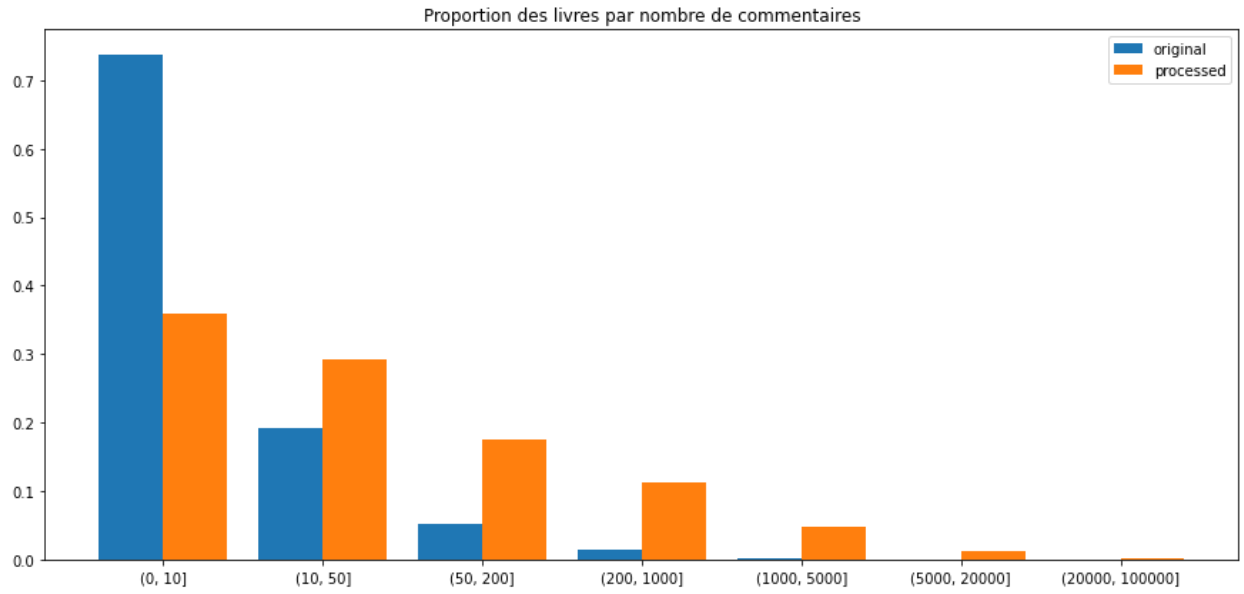


Figure 3 : Proportion de livres par nombre de commentaires avant et après échantillonnage

En plus de ces 30 000 livres, on ajoute tous les livres qui sont dans les mêmes séries, afin d'obtenir les séries complètes, ce qui double approximativement le nombre de livres.

Cette étape élimine également un certain nombre de livres qui pollueraient nos données pour la suite. Nous éliminons ainsi les livres sans interactions, sans commentaires ou sans tags, qui n'ont que peu d'intérêt pour nous ; et nous avons choisi de ne garder que les livres en anglais, pour faciliter le travail d'exploitation par la suite : comme il n'y a que relativement peu de livres et des ressources plus limitées dans d'autres langues, faire fonctionner nos modèles sur les livres dans chaque autre langue serait très consommateur de ressources pour des résultats de toute façon médiocres.

Échantillonnage des utilisateurs.

De la même façon, le **nombre d'utilisateurs** à garder est un paramètre du pipeline, que l'on a fixé à 50 000 pour le résultat final. On sélectionne également les utilisateurs par **roulette** en se basant sur le nombre d'interactions et de commentaires associés à l'utilisateur, afin de recentrer l'échantillon et d'éviter d'avoir une majorité d'utilisateurs présentant peu d'intérêt pour le projet.

Sélection des autres informations

Avec un échantillon de livres et d'utilisateurs, on peut sélectionner les lignes utiles des autres tables, nommément les **interactions**, les **commentaires**, les **auteurs** et les **séries**. On ne garde ainsi que les interactions et les commentaires des livres et utilisateurs sélectionnés, et les auteurs et séries correspondants aux livres conservés.

À cette étape, nous avons également choisi de ne conserver que les commentaires en anglais, pour la même raison que nous ne gardons que les livres en anglais : le peu de données dans chaque autre langue individuellement alourdirait considérablement l'exploitation pour des résultats médiocres.

Reconstitution des notes à partir des reviews

Une partie non-négligeable des commentaires n'étaient pas accompagnés de notes, ce qui pouvait créer des vides dans certaines situations (par exemple, certains livres n'avait quasiment que ce type de commentaires donc une notation numérique de faible qualité). Nous avons pour cela employé de l'analyse sentimentale afin de reconstituer les notes qui seraient associées à ces commentaires seuls. Cette partie est détaillée dans la ***section suivante***.

Catégorisation des tags utilisateur

Dans notre jeu de données, les livres sont « catégorisés » par les « étagères » sur lesquelles les utilisateurs les placent. Ces étagères sont essentiellement définies par les utilisateurs individuellement, elles sont donc très disparates, avec de nombreuses (plusieurs dizaines de milliers) étagères redondantes ou inutiles. Le dataset propose aussi un fichier `goodreads_book_genres_initial.json` donnant déjà une catégorisation en une dizaine de genres très vagues — trop vagues pour faire des recommandations pertinentes. Nous avons donc besoin d'un juste milieu, avec un nombre exploitable de tags pertinents.

Préfiltrage

La première étape du traitement charge les données en mémoire et préfiltre les étagères afin d'en éliminer certaines que l'on sait inutiles, comme celles reprenant le nom de l'auteur ou de la série, les collections du type « à lire », « lu en 2014 » ou « mes favoris ». On élimine également les tags qui ne sont pas en alphabet latin : ici, certains mots dans d'autres langues sont récupérables s'ils sont suffisamment proches du mot anglais (*historic*, *historique*, *historisch*, *histórico*, etc.), mais ce ne sera pas possible pour des noms dans d'autres systèmes d'écriture. On réalise aussi un seuillage sur le nombre d'utilisateurs qui ont mis le livre sur cette étagère par rapport au nombre total de placements, afin d'éliminer un grand nombre d'étagères très mineures et souvent extrêmement spécifiques ou personnelles qui pollueraient nos résultats plus qu'autre chose.

Les mots restants sont ensuite passés par une étape de normalisation grâce à ***NLTK*** [3], où on essaie de lemmatiser (retrouver le lemme, c'est-à-dire la forme fondamentale du mot) les mots qui peuvent l'être, et de retrouver quelle est la nature de chaque mot afin de réduire au maximum l'ambiguïté de sens.

Catégorisation

Cette partie s'est avérée complexe. En effet, il est relativement difficile de regrouper ces tags par des modèles de machine learning, car ils ne représentent que des points d'information isolés (d'un à quelques mots au plus), et nous n'avons pas de jeu d'entraînement possible.

Pour réaliser notre objectif, nous avons choisi d'utiliser une hiérarchie de tags prédéfinis, et de catégoriser chaque nom d'étagère par rapport à ces tags. Chaque étagère en entrée sera comparée avec ces tags pour en déduire celui ou ceux qui s'en rapprochent le plus. L'objectif est d'en tirer potentiellement deux possibilités : l'étagère peut soit correspondre à un tag en particulier, soit être un

composé de plusieurs tags. Par exemple, *dystopian-society* correspond exactement à *dystopia*, mais *dystopian-scifi* est un composé de nos tags *dystopia* et *science-fiction*.

Pour cela, on calcule la similarité entre les mots de l'étagère et du tag à comparer, dont on fait une matrice de similarité. On applique alors une mesure comparable à un coefficient de similarité de Jaccard. On commence par un seuillage afin d'éliminer les couples de mots insuffisamment similaires, puis on récupère la similarité maximum sur chaque ligne (quel mot de l'étagère correspond le mieux au mot du tag), et la similarité maximale sur chaque colonne (quel mot du tag correspond le mieux au mot de l'étagère).

Similarité	shelf1	shelf2	shelf3	
tag1	0,87	0,53	0,2	0,87
tag2	0,1	0,71	0,54	0,71
tag3	0,67	0,2	0,44	0,67
	0,87	0,71	0,54	
Seuillage	shelf1	shelf2	shelf3	
tag1	0,87	0	0	0,87
tag2	0	0,71	0	0,71
tag3	0,67	0	0	0,67
	0,87	0,71	0	
	1	1	0	
maxsum		3,83		
similarity		0,63833333		
subclass_denominator		5		
subclass		0,766		

Figure 4 : Calcul de la similarité entre groupes de mots

De ces valeurs, on tire nos deux mesures :

- Similarité totale : somme de tous les maximums (lignes et colonnes), divisée par la somme du nombre de lignes et de colonnes,
- Composition : somme de tous les maximums, divisée par le nombre de colonnes contenant un mot retenu (maximum > 0) auquel on ajoute le nombre de lignes — ainsi, on ne compte pas les éventuels mots de l'étagère qui ne seraient pas dans le tag, donnant une valeur plus élevée aux étagères avec des mots correspondant au tag ainsi que des éléments supplémentaires.

Avec un seuillage sur ces mesures, on trouve lequel des deux cas est le plus probable, et on associe ainsi l'étagère aux éventuels tags qui y correspondent suffisamment.

Calcul de similarité

Le calcul de la similarité entre deux mots a été le principal problème de cette partie. En effet il existe de nombreuses possibilités, aucune n'étant pleinement satisfaisante. Au final, nous avons retenu une combinaison de ces possibilités, afin de traiter autant de cas que possible :

- Premièrement, certains éléments sont des nombres. Il suffit alors de les comparer.
- De même, la similarité de deux mots identiques est triviale,
- Ensuite, on utilise *WordNet* [4] [5], de l'université de Princeton, via le module Python *nltk* [3], duquel on tire la similarité de Wu-Palmer entre les deux mots, qui se base sur la position relative des deux mots dans la taxonomie de WordNet. WordNet donne des relations particulièrement fiables entre les mots, mais ne donne que des relations purement formelles (synonymes, antonymes, hyperonymes, hyponymes, etc.). De fait, les mots de natures différentes n'auront jamais aucune relation. Par exemple, `realism` et `reality`, qui sont deux noms, seront très proches (similarité de `0.857` environ) ; mais `realism` et `realist`, quoique directement liés, sont un nom et un adjectif et n'ont donc aucune relation d'après WordNet.
- Pour résoudre ce problème, on utilise le *Wiktionnaire*, un dictionnaire moins complet, sous une forme plus complexe, mais qui contient des relations plus « vagues » qui nous permettraient d'atteindre cet objectif, notamment quels mots sont dérivés de quels autres. Nous avons ainsi réalisé un script permettant de parser un dump du Wiktionnaire sous un format exploitable pour notre cas d'utilisation. De là, par de simples distances dans le graphe de relations, nous pouvons éventuellement trouver une similarité, même entre mots de natures différentes.
- Enfin, si tout a échoué parce qu'au moins l'un des deux mots n'est dans aucun des deux dictionnaires (typiquement, un nom propre, une faute d'orthographe ou un mot dans une autre langue), on se rabat sur la comparaison des chaînes de caractères par une distance de Damerau-Levenshtein, qui compte le nombre d'opérations caractère par caractère nécessaires pour passer d'un mot à l'autre (substitution, insertion, délétion, transposition).

En normalisant correctement toutes ces mesures, on obtient des valeurs comparables entre elles qui nous permettent de calculer la similarité entre les mots d'une façon aussi fiable que possible.

Génération des utilisateurs

Le jeu de données est totalement anonymisé : nous avons des ID d'utilisateurs sur les interactions et les commentaires, mais aucune information sur les utilisateurs eux-mêmes : afin de compléter nos données, nous avons donc généré les informations sur les utilisateurs de façon relativement cohérente avec leur activité (en particulier les tags favoris).

Découpage et correction de la cohérence des données

Afin de simplifier l'import dans la base de données et de rétablir la cohérence de toutes les informations, les données passent par une dernière étape de prétraitement. Celle-ci a plusieurs objectifs :

- Rétablir la cohérence des champs calculés, comme le nombre et la moyenne des notes par livre et par auteur, les nombres de commentaires, etc., qui jusque-là étaient restés les valeurs originales avant échantillonnage,
- Fusionner les interactions et les commentaires, qui jusque-là sont deux tables différentes avec la même clé et des informations redondantes,

- Ajouter des valeurs calculées utiles, comme le nombre total de catégorisations par les utilisateurs sur un livre,
- Construire les tables de relation plusieurs-à-plusieurs (livre – auteur, livre – série, livre – tag) dans des fichiers séparés, pour simplifier le script d'import en base de données et ne pas devoir relire plusieurs fois les fichiers complets.

Import en base de données

Le pipeline de préprocessing se solde par l'import des données traitées dans notre base de données MySQL à destination des autres éléments du projet, qui est décrite ci-après.

Structuration du pipeline

Afin d'améliorer les performances, le pipeline de prétraitement a été mis sous la forme de tâches distinctes, lancées par un script en fonction d'un graphe de dépendances :

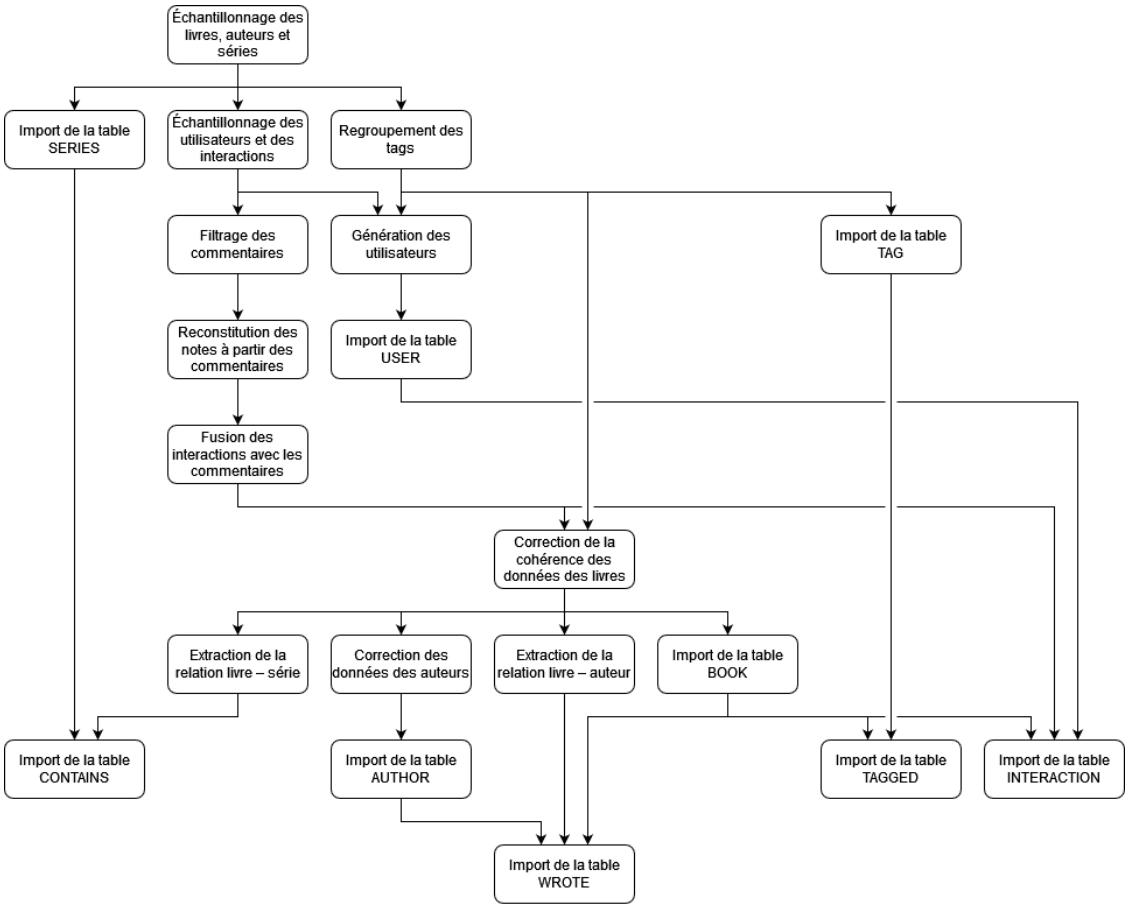


Figure 5 : Graphe de dépendances du pipeline de preprocessing

Ainsi, certaines étapes peuvent être réalisées en parallèle. Cela permet également d'améliorer la création de journaux, et de grandement faciliter l'utilisation du pipeline : on passe ainsi de 6 scripts séparés ayant chacun ses paramètres à une commande unique et un fichier de configuration.

Pour l'exécution du pipeline complet, du premier filtrage à l'import dans la base de données, on passe ainsi de près de 2h30 à environ 1h10.

La base de données

Après le prétraitement, les données sont importées en base de données pour les rendre accessibles aux autres composantes du projet.

Nous avons pris la décision d'utiliser un système de base de données relationnel, nommément MySQL, face à d'autres modèles comme MongoDB. Aucun des deux modèles n'était ici fondamentalement plus avantageux que l'autre, nous avions pour la plupart plus d'expérience avec les bases de données relationnelles, et l'hébergement d'une base MySQL était plus accessible pour nous. Nous avons ainsi hébergé une base MySQL sur le service cloud *DigitalOcean*, grâce aux *pack étudiant de GitHub*.

Nous avons ainsi transposé la structure de nos données en modèle de base de données relationnel :

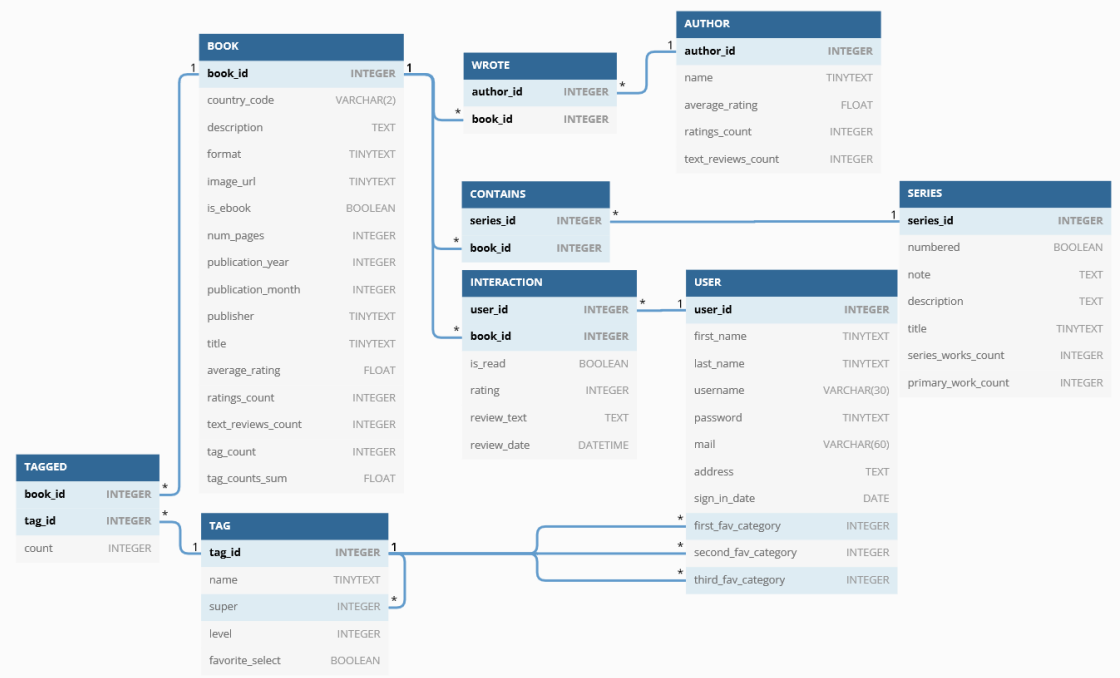
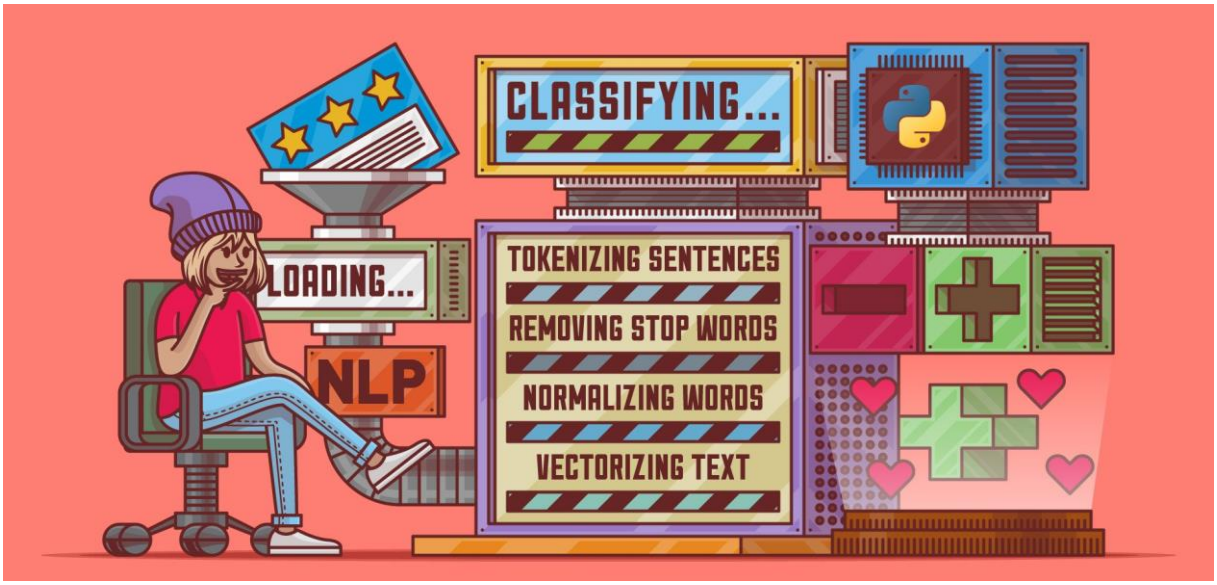


Figure 6 : Schéma de base de données final

Restauration des notes à partir des commentaires



Introduction

Les ordinateurs et les machines sont très doués pour travailler avec des données formelles. Cependant, en tant qu'êtres humains, nous communiquons généralement par des mots et des phrases, et non sous la forme de tableaux. La plupart des informations que les humains parlent ou écrivent sont structurées de manière extrêmement libre complexe. Il n'est donc pas facile de les faire interpréter par un programme. L'objectif du **Natural Language Processing** est de faire en sorte que les ordinateurs comprennent le texte non structuré et en extraient des éléments d'information significatifs. Le traitement du langage naturel (NLP) est un sous-domaine de l'intelligence artificielle, dont la profondeur implique les interactions entre les ordinateurs et les hommes.

Dans notre projet, nous avons une base de données avec une table qui recense l'ensemble des interactions entre les utilisateurs et les livres. Les utilisateurs peuvent proposer des commentaires et noter des livres. Certains utilisateurs ont mis un commentaire sans noter le livre. Les notes données par les utilisateurs pour chaque livre sont une donnée cruciale de notre algorithme de recommandation collaboratif. Nous avons donc utilisé le **Natural Language Processing** pour essayer, dans ces cas-là, de donner une note au livre à partir du commentaire mis par l'utilisateur.

Prétraitement des données textuelles

Une fois les données chargées, il est nécessaire d'implémenter un pipeline de traitement du langage naturel avant de pouvoir faire efficacement quelque chose d'utile de nos données.

Les étapes nécessaires comprennent (mais ne sont pas limitées à) ce qui suit :

- **Tokenizing** pour décomposer le texte en phrases, mots ou autres unités.
- **Suppression** des mots d'arrêt comme « si », « mais », « ou », etc.

- **Normalisation** des mots en condensant toutes les formes d'un mot en une seule forme
- **Vectoriser** le texte en le transformant en une représentation numérique destinée à être utilisée par le classificateur.

Toutes ces étapes servent à réduire le bruit inhérent à tout texte naturel et à améliorer la précision des résultats de notre classificateur. Nous avons, pour ce faire, utilisé la librairie Python ***NLTK*** [3] (Natural Language Toolkit)

Tokenisation

La tokénisation est le processus de décomposition de texte en morceaux plus élémentaires. NLTK est fourni avec un pipeline de traitement par défaut qui commence par la tokénisation, rendant ce processus rapide. Dans NLTK, on peut tokeniser soit par phrases, soit par mots. Ici, on découpe par mot :

```
#On tokenise
data['review_text'] = [word_tokenize(entry) for entry in data['review_text']]
```

Figure 7 : Tokenisation du texte des reviews

Suppression des mots vides

Les « mots vides » (*stopwords*) sont des mots extrêmement fréquents qui sont importants dans la communication humaine mais qui ne portent pas d'information utile pour le traitement que l'on en fait ici. NLTK est fourni avec une liste de stopwords par défaut, que l'on élimine de nos textes :

```
if word not in stopwords.words('english') and word.isalpha():
```

Figure 8 : Suppression des mots vides

Normalisation des mots

La normalisation est un peu plus complexe que la tokenisation. Elle consiste à condenser toutes les formes d'un mot en une seule représentation de ce mot. Par exemple, « watched », « watching » et « watches » sont des formes différentes du seul verbe « watch ». Il existe deux méthodes principales de normalisation.

Avec la **troncature**, un mot est coupé au niveau de sa racine, la plus petite unité de ce mot à partir de laquelle vous pouvez créer les mots descendants. Vous venez d'en voir un exemple ci-dessus avec « watch ». L'étymologie ne fait que tronquer la chaîne de caractères en utilisant les éléments communs, de sorte que la relation entre « feel » et « felt », par exemple, ne sera pas prise en compte.

La **lemmatisation** cherche à résoudre ce problème. Ce processus utilise une structure de données qui relie toutes les formes d'un mot à sa forme la plus simple, ou **lemme**. Cette stratégie est plus complexe, mais aussi plus puissante. C'est donc celle que nous avons choisi.

```python
#On met tout en minuscule
data['review_text'] = [entry.lower() for entry in data['review_text']]

#On tokenize
data['review_text'] = [word_tokenize(entry) for entry in data['review_text']]

tag_map = defaultdict(lambda : wn.NOUN)
tag_map['J'] = wn.ADJ
tag_map['V'] = wn.VERB
tag_map['R'] = wn.ADV

for index,entry in enumerate(data['review_text']):
    # On déclare une liste vide pour stocker les mots qui suivent les règles de cette étape
    Final_words = []
    # On initialise WordNetLemmatizer()
    word_Lemmatized = WordNetLemmatizer()
    # La fonction pos_tag ci-dessous fournira le 'tag', c'est-à-dire si le mot est Nom (N) ou Verbe (V) ou autre chose.
    for word, tag in pos_tag(entry):
        # La condition ci-dessous est de vérifier les mots vides et de ne considérer que les alphabets
        if word not in stopwords.words('english') and word.isalpha():
            word_Final = word_Lemmatized.lemmatize(word,tag_map[tag[0]])
            Final_words.append(word_Final)
    # L'ensemble de mots traité final pour chaque itération sera stocké dans 'text_final'
    data.loc[index,'review_text'] = str(Final_words)
```

Figure 9 : Lemmatisation des mots du texte

Vectorisation de texte

La vectorisation est un processus qui transforme un token en un vecteur, ou un tableau numérique qui, dans le contexte du traitement automatique des langues, est unique et représente diverses caractéristiques d'un token. Les vecteurs sont utilisés pour trouver des similitudes entre les mots, classer le texte et effectuer d'autres opérations NLP.

Cette représentation particulière est un tableau dense, dans lequel il existe des valeurs définies pour chaque espace du tableau. Cela s'oppose aux méthodes précédentes qui utilisaient des tableaux épars, dans lesquels la plupart des espaces sont vides.

```python
#création du jeu de donnée sans les zéros
datasans0 = data[data['rating'] != 0]

Train_X, Test_X, Train_Y, Test_Y = model_selection.train_test_split(datasans0['review_text'],datasans0['rating'],test_size=0.3)

#On encode les textes
Encoder = LabelEncoder()
Train_Y = Encoder.fit_transform(Train_Y)
Test_Y = Encoder.fit_transform(Test_Y)

#On vectorise
Tfidf_vect = TfidfVectorizer(max_features=5000)
Tfidf_vect.fit(data['review_text'])
Train_X_Tfidf = Tfidf_vect.transform(Train_X)
Test_X_Tfidf = Tfidf_vect.transform(Test_X)
```

Figure 10 : Vectorisation du texte

Utilisation de classificateurs d'apprentissage automatique pour prédire le sentiment

Les commentaires non notés sont maintenant dans une forme compréhensible par des modèles de machine learning, nous pouvons ainsi commencer à en retirer les « sentiments » pour les catégoriser par note de 1 à 5.

Outils de machine learning utilisés

Scikit-learn est à l'opposé de TensorFlow et PyTorch. Il est de plus haut niveau et permet d'utiliser des algorithmes d'apprentissage automatique prêts à l'emploi plutôt que de les créer explicitement. Ce qui lui manque en termes de personnalisation est largement compensé par sa facilité d'utilisation, qui permet d'entraîner rapidement des classificateurs en quelques lignes de code.

Nous avons implémenté un modèle **SVM** (*Support Vector Machine*), pour classifier nos commentaires (leur donner une note de 1 à 5).

```
#création, entrainement du modèle et prediction sur le jeu test
SVM = svm.SVC(C=1.0, kernel='linear', degree=3, gamma='auto')
SVM.fit(Train_X_Tfidf,Train_Y)
predictions_SVM = SVM.predict(Test_X_Tfidf)
```

Figure 11 : Classification des commentaires par note

Nous avons finalement intégré le NLP au pipeline de préprocessing des données.

Analyse des résultats.

Le modèle mis en production arrive à environ 90% de catégorisations « acceptables » (avec une marge de tolérance de 1 dans la note proposée). Voici la matrice de confusion entre la note déterminée par notre algorithme et la vraie note :

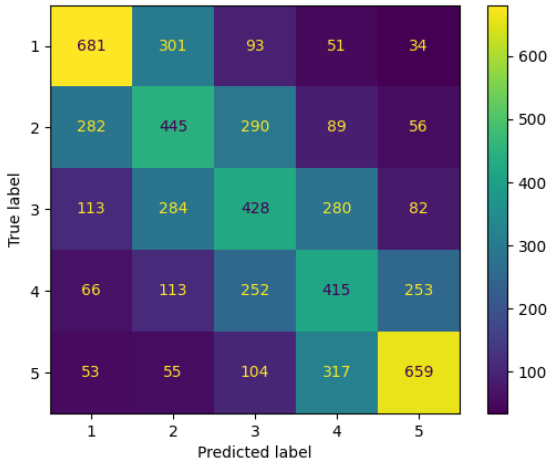


Figure 12 : Matrice de confusion des notes déterminées par notre modèle

On observe que plus l'écart entre les deux notes est élevé, moins les cas où ça arrive sont fréquents. Les notes moyennes sont aussi plus difficiles à noter avec précision du simple fait de leur plus grande rareté.

Cette étape de préprocessing va nous permettre d'améliorer la quantité de données dont l'on dispose pour l'algorithme de recommandation collaboratif et donc améliorer ses recommandations.

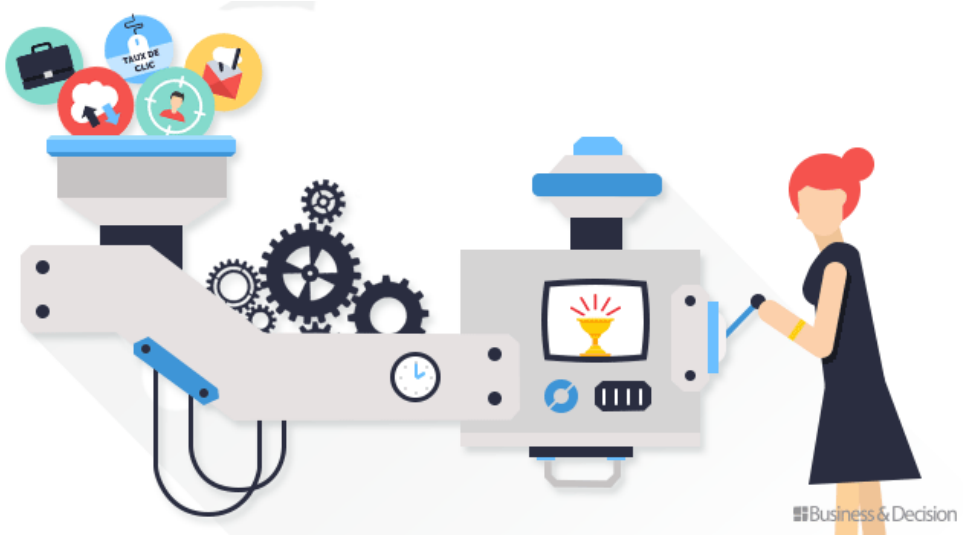
Moteur de recommandation

Un « moteur de recommandation », aussi appelé « système de recommandation » est un algorithme qui identifie et fournit des produits, des contenus ou éléments numériques recommandés et adaptés aux différents utilisateurs. L'objectif principal d'un moteur de recommandations est de guider les utilisateurs dans leurs recherches afin de leur proposer les produits qu'ils sont le plus susceptibles d'acheter, ou bien les contenus qu'ils sont les plus susceptible de consommer. Pour aboutir à des résultats pertinents, un système de recommandation doit être capable d'analyser la relation entre les utilisateurs et les éléments avec lesquels ils interagissent (produits, services, informations, etc.).

Les moteurs de recommandation sont omniprésents dans les applications que nous pouvons utiliser au quotidien tel que YouTube, Netflix, Spotify et bien d'autres. Aujourd'hui, ils sont également et surtout considérés comme des outils indispensables de la vente en ligne. En pleine ère du e-commerce, les enjeux sont de taille pour les géants du secteur comme Amazon, Cdiscount ou AliExpress mais également pour de plus petites entreprises émergentes. En effet, plus les catalogues de produits ou contenus proposés sont grands, plus il sera primordial d'implémenter un système de recommandations efficace.

Les moteurs de recommandations permettent aux sites de e-commerce de :

- Proposer une meilleure expérience utilisateur.
- Faciliter la navigation et la recherche d'article.
- Assurer la satisfaction des utilisateurs.
- Renforcer la fidélisation.
- Augmenter la taille des paniers d'articles des consommateurs.



Gestion des recommandations

Les moteurs de recommandations s'appuient sur différents types d'algorithmes dont l'utilisation et l'efficacité varient suivant le contexte. Il n'existe pas de solution prédéfinie et optimale à tout type de problème, un moteur de recommandations adapté à un site marchand ne le sera pas forcément pour un autre. L'utilisation d'un algorithme de recommandations dépend avant tout des données à disposition (utilisateurs, produits, transaction, etc.), c'est pourquoi nous avons dû prendre en compte ce paramètre lors du choix de notre jeu de données.

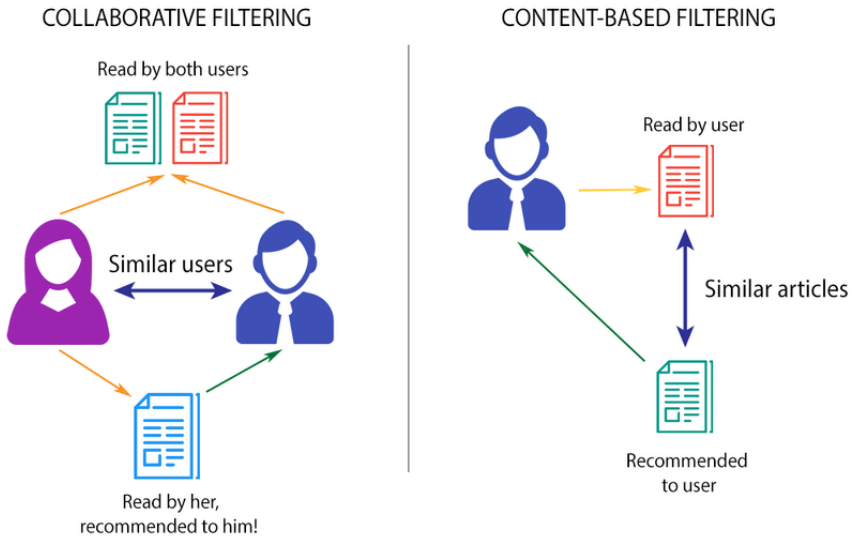
Nous souhaitions initialement être en mesure d'implémenter un algorithme pour chacune des deux grandes familles d'algorithme de recommandations, à savoir le filtrage basé sur le contenu (en anglais, « *content based algorithm* ») et le filtrage collaboratif (« *collaborative filtering* »).

Dans un premier temps, le filtrage basé sur le contenu consiste à recommander des produits similaires à un produit initial (page d'article, panier…). Cela implique que :

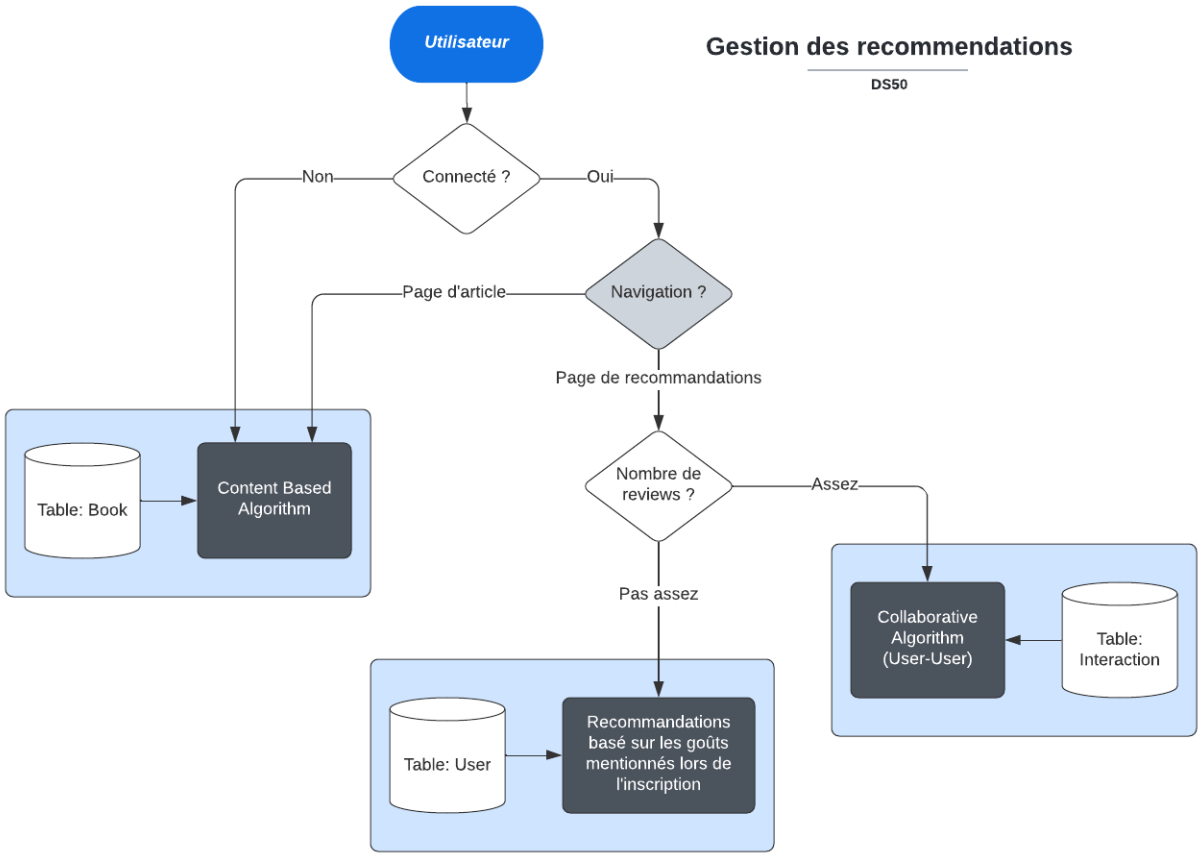
- Seules les informations sur les articles sont nécessaires.
- L'algorithme peut être utilisé sans que l'utilisateur soit connecté.
- Des points de comparaison doivent être définis.
- Plus les produits sont décrits, plus les comparaisons sont précises.

Dans un second temps, le filtrage collaboratif suit la logique selon laquelle un utilisateur est susceptible d'apprécier les produits que d'autres utilisateurs aux profils similaires au sien ont également apprécié. Ainsi, ce filtrage consiste à comparer des utilisateurs (dans le cas d'une approche centrée sur l'utilisateur) pour établir des degrés de similarité et pouvoir ensuite recommander les produits appréciés par des utilisateurs similaires. Cela implique :

- Un besoin de données sur les utilisateurs et leur interactions (notes) avec les produits,
- Que l'algorithme nécessite que l'utilisateur soit connecté,
- Que l'utilisateur connecté doit avoir un minimum d'interactions avec les produits.



Ainsi, nous avons décidé d'appliquer la logique de recommandation suivante :



Dans un premier temps, on distinguera les cas où l'utilisateur est connecté et les cas où il ne l'est pas. Si l'utilisateur n'est pas connecté, nous n'aurons pas accès aux informations relatives à son profil, son historique de commande, les interactions avec les produits, c'est pourquoi nous utiliserons alors avant tout l'algorithme de filtrage basé sur le contenu suivant sa navigation (page d'article, panier etc.).

Dans le cas contraire, nous pouvons ajouter à cela l'utilisation de l'algorithme de filtrage collaboratif. Cependant, il est nécessaire que l'utilisateur en question ait déjà noté quelques produits pour que nous ayons un minimum d'information sur ses goûts et préférences. Ainsi, en-dessous d'un certain nombre de produits notés, nous proposons des recommandations basées sur des thèmes appréciés par l'utilisateur.

En effet, lors de l'inscription, nous demandons à l'utilisateur de choisir ses 3 thèmes de livre (tags) favoris. Ainsi, de la même manière que certaines plateformes de contenu comme Netflix ou bien Spotify sondent leurs nouveaux utilisateurs lors de l'inscription, cela nous permet d'avoir une base d'information qui nous aidera à émettre les premières recommandations. Selon les thèmes cités, nous recommanderons à l'utilisateur quelques-uns des livres les plus populaires dans ces catégories. Ces recommandations sont temporaires et seront ensuite plus personnalisés lorsque l'utilisateur aura noté assez de livres pour que nous puissions utiliser le filtrage collaboratif de manière efficace.

Filtrage basé sur le contenu

Conception

Durant la phase de conception, la principale question à laquelle nous devions trouver une réponse était la suivante : Quelles seront les points de comparaison entre nos produits ? Dans notre cas, les produits proposés sont des livres. Les données proposées par les datasets fournis par Goodreads sont les suivantes :

```
['isbn', 'text_reviews_count', 'series', 'country_code', 'language_code',
'popular_shelves', 'asin', 'is_ebook', 'average_rating', 'kindle_asin',
'similar_books', 'description', 'format', 'link', 'authors', 'publisher',
'num_pages', 'publication_day', 'isbn13', 'publication_month',
'edition_information', 'publication_year', 'url', 'image_url', 'book_id',
'ratings_count', 'work_id', 'title', 'title_without_series']
```

Il s'agissait ici de sélectionner dans un premier temps les données qui décrivent et caractérisent réellement un livre. Par exemple, nous pouvions déjà mettre de côté les colonnes *isbn*, *isbn13* et *book_id* qui ne sont que des identifiants qui ne caractérisent pas de manière concrète un livre. De même, les colonnes *text_reviews_count* ou bien *ratings_count* ont été écartées car elles communiquent une information technique plutôt qu'une caractéristique du livre. Dans cette logique, nous avons écarté plusieurs colonnes pour finalement garder :

```
['series', 'country_code', 'language_code', 'popular_shelves', 'is_ebook',
'description', 'format', 'authors', 'publisher', 'num_pages',
'publication_day', 'publication_month', 'edition_information',
'publication_year', 'title', 'title_without_series']
```

Ensuite, l'étape suivante consistait à effectuer un deuxième tri des données. Cette fois, il s'agissait de se poser les questions suivantes : Est-il cohérent de comparer des livres sur un tel critère ? Comment numériser et exploiter efficacement les informations apportées ? Par exemple, la date de publication semble être un critère assez cohérent lors du choix d'un livre. Il se peut qu'une personne soit plus attiré par le moderne ou inversement, c'est pourquoi nous avons décidé de conserver la colonne *publication_year* afin de nous baser sur l'année de publication du livre. *publication_day* et *publication_month* ont été écartés car cela rajoute un degré de précision qui n'est pas nécessaire.

Selon nous, la colonne *popular_shelves* est la donnée la plus importante du dataset car elle caractérise les livres par des noms de thèmes (des tags). Etant donné que ce sont les utilisateurs qui étiquettent les livres de cette manière, cela caractérise de manière assez précise les livres. Nous avons donc choisi d'implémenter notre filtrage basé sur le contenu principalement autour de ces données, préalablement recatégorisées et prétraitées pour faciliter et améliorer le travail de cet algorithme.

Concernant les colonnes *title* et *description* qui sont du texte naturel, nous avons choisi de ne pas les utiliser pour ne pas trop complexifier l'algorithme. Cependant, nous aurions pu tenter une autre approche consistant à relever les similarités présentes dans les textes en les vectorisant.

Finalement, nous avons choisi de conserver les données suivantes :

['popular_shelves', 'format', 'authors', 'publisher', 'num_pages', 'publication_year']

Enfin, la dernière étape consistait à traiter les données, les numériser (si besoin) et les normaliser pour garder un certain équilibre lors de la comparaison. Par exemple, la colonne *popular_shelves* contient une liste de tags avec leur nom et nombre de mentions (ex : *[{name : "fiction", count : 123}, {name : "horror", count : 58}]*). D'après l'exemple précédent, nous pouvons en déduire que le livre appartient d'abord au genre fiction, puis au genre horreur. Pour numériser cette assertion, nous avons choisi de raisonner en proportion de genre et de créer une colonne pour chaque tag. Ainsi, l'exemple précédent donnerait :

$$Fiction = \frac{Nombre\ de\ mentions_{\ fiction}}{Nombre\ total\ de\ mentions} = \frac{123}{181} \approx 0.68$$

Et de même :

$$Horreur = \frac{Nombre\ de\ mentions_{\ horreur}}{Nombre\ total\ de\ mentions} = \frac{58}{181} \approx 0.32$$

En procédant ainsi, le genre du livre est défini de manière très précise. Chaque colonne de genre contiendra des valeurs entre 0 et 1, c'est pourquoi nous avons choisi d'utiliser une normalisation « *MinMax* » pour traiter certaines autres données.

La colonnes *num_pages* et *publication_year* contiennent déjà des valeurs numériques. Etant donné que certaines valeurs sont manquantes, nous avons choisi de les remplacer par les moyennes des échantillons avant de leur appliquer une normalisation *MinMax*. Concernant, la colonne *publication_year* nous avons pu constater la présence des certains outliers qui faussaient la normalisation. En effet, la normalisation *MinMax* est sensible aux outliers, c'est la raison pour laquelle nous ramènerons toutes les valeurs en dessous de 1960 (valeur déterminée lors de l'exploration des données) à cette valeur.

Concernant les colonnes restantes, il s'agit de données nominales. Cela implique qu'il ne faut pas créer de notion de distance entre les différentes valeurs lors de l'encodage. Par exemple, un « *Ebook* » ne ressemble pas plus à un « *Audiobook* » qu'un livre au format « *Standard* », ce sont juste des formats différents. Ainsi, nous avons décidé d'encoder directement ces colonnes par rapport à l'objet de la comparaison, c'est-à-dire qu'on affectera un 1 lorsque la valeur sera la même et un 0 lorsqu'elle sera différente.

Nous utiliserons la similarité cosinus pour évaluer nos recommandations :

$$similarity(A, B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Implémentation

Pour implémenter ce filtrage basé sur le contenu, la première des chose à faire est de requêter notre base de données. Nous utiliserons la requête suivante :

```
query = f"""
            SELECT
                B.*
                ,W.author_id
            FROM
                BOOK B INNER JOIN WROTE W
                    ON B.book_id = W.book_id
            WHERE
                B.book_id = {self.filter_base}
            UNION
            SELECT
                B.*
                ,W.author_id
            FROM
                BOOK B INNER JOIN WROTE W
                    ON B.book_id = W.book_id
            LIMIT {top}
        """
```

Figure 13 : Requête des informations relatives au livre initial

Cette requête renvoie les informations relatives au livre initial en filtrant la recherche grâce à l'ID *self.filter_base*. Elle renvoie également un nombre d'autres livres à recommander potentiellement (variable *top*). Ici, on ne requête pas encore les données des tags.

book_id	author_id	publisher	publication_year	format	num_pages
27421523	1077326	Pottermore Limited	2015.0	ebook	305.0
1	2927	Scholastic Inc.	2006.0	Paperback	652.0
2	2927	Scholastic Inc.	2004.0	Paperback	870.0
3	2927	Scholastic Inc	1997.0	Hardcover	320.0
4	1077326	Scholastic	2003.0	Hardcover	352.0
...	...	...	...	...	...
339045	3362	Henry Holt and Co. (BYR)	2007.0	Hardcover	32.0
339189	5098	Villard	2007.0	Paperback	302.0
339232	12546		1996.0		NaN
339340	19051	Avon	2001.0	Paperback	372.0
339410	3212	Houghton Mifflin Co.	1951.0	Hardcover	NaN

[6801 rows x 5 columns]

Figure 14 : Résultat de la requête d'information initiale sur le livre

Ensuite, nous passons par l'étape de « préprocessing » nécessaire pour obtenir une matrice exploitable. On retrouvera les différentes transformations de nettoyage, d'encodage et de normalisation citées précédemment. Par exemple, pour l'encodage des colonnes contenant des valeurs nominales :

```python
# Encodes processed_data that need to be encoded according one book
def encodeLabels(df, book_id, columns):
    for col in columns:
        df[col] = (df[col] == df.loc[book_id, col]).astype(float)
    return df
```

Figure 15 : Encodage des colonnes textuelles

Résultat :

book_id	author_id	publisher	publication_year	format	num_pages
27421523	1.0	1.0	0.964912	1.0	0.102867
1	0.0	0.0	0.807018	0.0	0.219899
2	0.0	0.0	0.771930	0.0	0.293423
3	0.0	0.0	0.649123	0.0	0.107926
4	1.0	0.0	0.754386	0.0	0.118718
...	...	...	...	...	...
339045	0.0	0.0	0.824561	0.0	0.010793
339189	0.0	0.0	0.824561	0.0	0.101855
339232	0.0	0.0	0.631579	1.0	0.116224
339340	0.0	0.0	0.719298	0.0	0.125464
339410	0.0	0.0	0.000000	0.0	0.116224

[6801 rows x 5 columns]

Figure 16 : Résultat de l'encodage des colonnes textuelles

Enfin, avant de pouvoir calculer les recommandations, on ajoute à la matrice les données relatives aux tags avec la requête suivante :

```
query = f"""
            SELECT
                TGD.book_id
                ,TAG.name
                ,TGD.count / SUM(TGD.count) OVER(PARTITION BY TGD.book_id) AS "perc"
            FROM
                TAGGED TGD INNER JOIN TAG TAG
                    ON TGD.tag_id = TAG.tag_id
            WHERE
                TGD.book_id IN {str(self.process.index.tolist()).replace('[','(').replace(']',')')}
        """
```

Figure 17 : Requête de proportion des tags associés au livre

Cette requête permet d'obtenir directement la proportion des genres de chacun des livres. On requête également les IDs des livres en question pour pouvoir rassembler les deux dataframes.

Résultat :

book_id	author_id	publisher	publication_year	format	num_pages	1:n+war:n+world:n	21:n+st:a+century:n	...
27421523	1.0	1.0	0.964912	1.0	0.102867	0.0	0.0000	...
1	0.0	0.0	0.807018	0.0	0.219899	0.0	0.0000	...
2	0.0	0.0	0.771930	0.0	0.293423	0.0	0.0000	...
3	0.0	0.0	0.649123	0.0	0.107926	0.0	0.0000	...
4	1.0	0.0	0.754386	0.0	0.118718	0.0	0.0000	...
...	...	...	...	...	...	...	...	...
339045	0.0	0.0	0.824561	0.0	0.010793	0.0	0.0112	...
339189	0.0	0.0	0.824561	0.0	0.101855	0.0	0.0000	...
339232	0.0	0.0	0.631579	1.0	0.116224	0.0	0.0000	...
339340	0.0	0.0	0.719298	0.0	0.125464	0.0	0.0000	...
339410	0.0	0.0	0.000000	0.0	0.116224	0.0	0.0000	...

[6787 rows x 135 columns]

Figure 18 : Résultat de l'insertion des tags dans la dataframe

Une fois cela fait, nous pouvons calculer une par une les notes de similarité du livre d'entrée avec l'ensemble des autres livres. Pour cela, nous ferons appel à la fonction *getSim* qui prendra deux vecteurs en entrée, ainsi que la méthode de calcul à employer (similarité cosinus, distance euclidienne ou corrélation de Pearson), et retourne la note.

```python
# Get similarity rates
def getSim(array1, array2, method='cos'):
    if method == 'cos':
        return sp_dist.cosine(array1, array2)
    if method == 'euc':
        return sp_dist.euclidean(array1, array2)
    if method == 'pea':
        num = sum([x1*x2 for x1, x2 in zip(array1, array2)])
        denom1 = 0
        denom2 = 0
        for x1, x2 in zip(array1, array2):
            if x1 != 0 and x2 != 0:
                denom1 += x1*x1
                denom2 += x2*x2
        denom = np.sqrt(denom1)*np.sqrt(denom2)
        if denom == 0:
            return 0
        else:
            return num/denom
```

Figure 19 : Calcul de similarité entre les livres

Les fonctions de calcul *cosine* et *euclidean* proviennent de la librairie Scipy, tandis que le calcul de la corrélation de Pearson a été implémenté manuellement.

Après avoir trié les livres par notes décroissantes, nous prendrons les dix premiers de la liste, soit les 10 livres les plus proches, comme recommandations.

Résultat final :

[49813, 224912, 330786, 49810, 227868, 4255, 121766, 47624, 4, 93124]

Filtrage collaboratif

Conception

Il existe plusieurs types de filtrage collaboratif :

- **Le filtrage collaboratif actif :** lorsque l'utilisateur donne directement son avis sur un produit ou un contenu. Cet avis prend généralement la forme d'une note sur une certaine échelle ou bien une mention j'aime/je n'aime pas.
- **Le filtrage collaboratif passif :** lorsque l'utilisateur ne donne pas un avis explicite mais qu'il est possible d'en donner une approximation grâce à d'autres données (navigation, temps passé sur certains produits ou thèmes, etc.)

- **Le filtrage mixte :** une version améliorée des deux méthodes citées.

Dans notre cas, étant donné que nous disposons d'un très grand nombre de données sur les notes (de 1 à 5) des utilisateurs, nous utiliserons un filtrage collaboratif actif. Depuis la table *INTERACTION* de notre base de données, nous aurons besoin de trois informations : *book_id* (identifiant du livre), *user_id* (identifiant de l'utilisateur) et *rating* (note donnée).

Cependant, nous pouvons nous poser la question suivante : Comment requêter la table *INTERACTION* afin de réduire au maximum notre matrice *user / book* ? En effet, un algorithme de filtrage collaboratif cherche à calculer des similarités entre les utilisateurs via leurs notations. Seulement, tous les utilisateurs n'ont sûrement pas lu et noté les mêmes livres, c'est pourquoi prendre des lignes au hasard dans la base de données risquerai d'ajouter des utilisateurs qui ne sont même pas comparable. Dans la même logique, nous ne pouvons pas affirmer avec le même taux de confiance que deux utilisateurs sont similaires lorsqu'ils ont noté un et un seul livre de la même manière que lorsqu'ils en ont plusieurs en commun. Ainsi, cela augmenterait inutilement le temps de calcul de l'algorithme et réduirait la pertinence des recommandations.

Afin de répondre à cette problématique, la solution optimale semblait être la suivante :

- Rechercher tous les livres notés par l'utilisateur initial.
- Compter, pour chaque autre utilisateur, le nombre de livres notés en commun avec l'utilisateur initial.
- Trier ces utilisateurs.
- Sélectionner uniquement les notations concernant les n premiers utilisateurs.
- Ne sélectionner AUCUN utilisateur n'ayant pas de notation en commun avec l'utilisateur initiale.

De cette manière, nous pouvons réduire au maximum la matrice *user / book* et le nombre de valeurs manquantes. Nous utiliserons la corrélation de Pearson pour calculer la similarité entre utilisateurs :

$$r = \frac{\sum(x - m_x)(y - m_y)}{\sqrt{\sum(x - m_x)^2 \ \sum\left(y - m_y\right)^2}}$$

Implémentation

D'abord, nous devons requêter les données relatives aux interactions des utilisateurs. Dans un premier temps, notre requête se limitait à sélectionner un grand nombre de lignes sans pré-filtrage. Afin d'optimiser l'algorithme, nous avons implémenter les requêtes suivantes :

```
query2=f"""
            SELECT
                I_OTHER.user_id
                ,COUNT(DISTINCT I_OTHER.book_id) "NB"
            FROM
                USER U INNER JOIN INTERACTION I_SELF
                    ON U.user_id = I_SELF.user_id
                INNER JOIN INTERACTION I_OTHER
                    ON I_SELF.book_id = I_OTHER.book_id
            WHERE
                I_SELF.rating != 0
            AND
                U.user_id = {self.filter_base}
            AND
                I_OTHER.user_id != {self.filter_base}
            GROUP BY
                I_OTHER.user_id
            HAVING
                NB > 0
            ORDER BY
                2 DESC
            LIMIT {top+1}
        """
```

```
query3=f"""
            SELECT
                user_id
                ,book_id
                ,rating
            FROM
                INTERACTION
            WHERE
                user_id = {self.filter_base}
            AND
                rating != 0
            UNION
            SELECT
                user_id
                ,book_id
                ,rating
            FROM
                INTERACTION
            WHERE
                user_id IN {str(same_read_df['user_id'].
tolist()).replace('[','(').replace(']',')')}
            AND
                rating != 0
        """
```

Figure 20 : Requêtes servant à la construction de la matrice user / book

La première requête cherche les utilisateurs ayant le plus de notations en commun avec l'utilisateur initiale.

Résultat :

	user_id	NB
0	439355	8
1	9880	8
2	299524	7
3	309155	7
4	320562	7
5	436968	7
6	322266	6
7	331075	6
8	396456	6
9	105106	6
10	119660	6

La seconde requête va extraire de la table *INTERACTION* toutes les notes sur les différents livres données par les utilisateurs sélectionnés précédemment.

Résultat :

	user_id	book_id	rating
0	101679	11	4
1	101679	13	3
2	101679	2865	2
3	101679	28876	5
4	101679	304481	3
...	...	...	...
24522	439355	28815474	5
24523	439355	29385546	4
24524	439355	31450633	4
24525	439355	32571395	5
24526	439355	33155777	4

[24527 rows x 3 columns]

Une fois les données rassemblées, nous pouvons les représenter sous la forme de matrice *user / book*. Avant cela, nous ajustons les notes de chaque utilisateur en leur soustrayant la moyenne (par utilisateur). Nous utiliserons ensuite la fonction *pivot_table* de Pandas pour créer notre matrice.

```
means = self.data.groupby(['user_id'], as_index=False, sort=False).mean().rename(columns={'rating': 'mean_rating'})
        means.drop(columns=['book_id'], inplace=True)
        self.data = self.data.merge(means, on='user_id', how='left', sort=False)
        self.data['adjusted_rating'] = self.data['rating'] - self.data['mean_rating']
        self.process = self.data.pivot_table(index='user_id', columns='book_id', values='adjusted_rating').fillna(0)
```

Figure 21 : Construction de la matrice user / book

Résultat :

book_id	1	2	3	4	5	6	11
user_id							
9880	1.217118	1.217118	1.217118	0.000000	0.217118	1.217118	1.217118
12891	-0.985149	-0.985149	1.014851	0.000000	0.014851	-0.985149	1.014851
31177	1.066000	1.066000	1.066000	0.000000	1.066000	1.066000	1.066000
33728	1.972727	1.972727	1.972727	0.000000	1.972727	1.972727	0.972727
68278	0.745763	0.745763	0.745763	0.000000	0.745763	0.745763	0.000000
98270	0.386973	0.386973	0.386973	0.000000	0.386973	0.386973	0.000000
101679	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.565217
105106	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
113131	1.641791	-0.358209	0.641791	0.000000	0.641791	0.641791	-0.358209
115644	1.213115	1.213115	-0.786885	0.000000	-0.786885	-0.786885	0.000000
119660	0.000000	0.000000	0.403756	0.000000	0.000000	0.000000	0.000000
123161	1.299145	1.299145	0.299145	0.000000	1.299145	1.299145	1.299145
126982	0.588966	0.588966	0.000000	0.000000	0.588966	0.588966	0.588966

Figure 22 : Matrice user / book résultante

Ensuite, on associe à chaque identifiant d'utilisateur une note de similarité que nous calculerons à l'aide de la méthode de corrélation de Pearson.

Résultat :

```
[(9880, 0.40596171365517897), (12891, -0.10505928383700161), (31177, 1.0), (33728, 0.5682615729104766), (6
8278, -1.0), (98270, -0.852292729232183), (105106, 0), (113131, 0.20167124211291493), (115644, -1.0), (119
660, 0), (123161, 1.0), (126982, -0.5292885578097453), (139072, -0.8152948840834626), (141311, 0), (145673
, 1.0), (162206, 1.0), (164249, 0.4136921682363761), (167352, -0.21038958142240116), (181216, 0.1913876462
5366358), (195992, 0), (200171, -0.9315592599838878), (203157, 0), (242270, -0.427914425671443), (258732,
0.9052369440730289), (264166, 0.12933918406776773), (280055, -0.525761274067058l), (299524, 0), (300826, 0
.8187607979294308), (302187, 0.20026331203762132), (309155, -0.503207119925535), (316762, -0.2503119768191
567), (316845, 0.3987261114144502), (319012, 1.0), (320562, 0), (322266, 0), (331075, 0.6160246114987641),
 (340318, 0), (348837, -0.6411423217685175), (353148, -1.0), (356693, 0), (359399, 0.48307616480401494), (
365778, 1.0), (368287, 0.6619827998077773), (371699, 0.6722363917225638), (374702, -1.0), (377986, 0.88540
09338283039), (380533, -0.8973291785064508), (396456, -1.0), (436968, -1.0), (439355, -1.0)]
```

Figure 23 : Association des utilisateurs à leurs similarités avec l'utilisateur initial

Enfin, il ne reste plus qu'à calculer la note finale à l'aide de la fonction suivante :

```python
def getBestRecommendations(self, df, top=10):
    reco = []
    df["sim"] /= df["sim"].sum()
    for book_id in df.columns[:-1]:
        reco.append((book_id, (df[book_id] * df["sim"]).sum()))
    reco.sort(key=lambda x: x[1], reverse=True)
    return reco[:top]
```

Figure 24 : Calcul des meilleures recommandations

Résultat final :

```
[1162543, 49041, 5907, 15931, 6900, 41865, 233093, 2865, 428263, 18135]
```

Construction de l'API

Choix de conception

Qu'est-ce qu'une API REST ?

Une **API** (*Application Programming Interface*) est une interface de programmation permettant d'accéder à un ou plusieurs services comme des données ou des fonctionnalités fournies par un système tiers.

REST (*Representational State Transfer*) est une architecture logicielle basée sur le HTTP (protocole de référence qui définit les communications sur le web), qui définit un ensemble de lignes directives architecturales à utiliser pour la création d'applications web. Les **six principes REST** qui guident la conception des API sont les suivants :

1. Découplage client-serveur : Ce principe consacre l'indépendance totale entre le client et le serveur. Ainsi, en séparant les préoccupations liées à l'interface utilisateur de celles liées au stockage des données, la portabilité sur différentes plateformes est améliorée et les possibilités d'évolution de l'application sont augmentées. Le client peut ainsi changer sans perturber le serveur, et inversement.

2. Interface uniforme : Tout type d'appareil client devrait interagir de manière uniforme avec le serveur.

3. Sans état (stateless) : L'API ne doit pas stocker des informations relatives au client. Par conséquent, toutes requêtes effectuées par le client doivent contenir toutes les données nécessaires pour les traiter.

4. Mise en cache : La réponse envoyée par le serveur doit indiquer si elle est « *cacheable* » ou non et pendant combien de temps les réponses peuvent être mises en cache côté client. Si la réponse est « *cacheable* », le client pourra récupérer cette réponse dans son propre système et il n'aura plus besoin d'envoyer de requêtes supplémentaires au serveur pour obtenir les mêmes données, ce qui évite des requêtes ultérieures. Ainsi, la latence du réseau est diminuée.

5. Architecture système en couches : L'architecture doit être composée de plusieurs couches qui n'interagissent respectivement qu'avec les couches voisines.

Communication via API REST

Le client envoie une requête HTTP en précisant la ressource, le serveur traite la requête en récupérant les informations demandées dans sa base de données et ensuite renvoie une représentation de la ressource.

Une requête API REST est généralement composée de :

1. Un point de terminaison : Une URI permettant au serveur et au client d'identifier la ressource,

2. Une méthode HTTP : Décrit le type de requête que le client envoie au serveur. On y retrouve les méthodes *GET*, *POST*, *PUT* et *DELETE*,

3. Un header : Les entêtes permettent de communiquer des informations utiles au client et au serveur, par exemple des données d'authentification,

4. Un corps : Le corps de la requête permet de fournir des données complémentaires pour le traitement de la requête.

L'API REST répond aux différentes requêtes avec des codes réponses HTTP comme « *200 OK* », « *201 Created* » ou « *400 Bad request* » qui indiquent le statut de la requête après traitement.

Pourquoi avoir choisi une API REST ?

Il existe d'autres alternatives d'architectures d'API comme GraphQL ou SOAP (*Simple Object Access Protocol*).

Contrairement à REST, une API GraphQL ne repose pas sur la création de plusieurs points de terminaison (endpoints). L'API GraphQL est privilégiée lorsque le service web est consommé par un grand nombre de services différents. Ainsi, chaque service peut fonctionner en consommant uniquement la donnée dont il a besoin. Cette architecture n'était donc pas à privilégier dans le cas de notre application.

Une API SOAP ne permet pas de renvoyer des ressources au client au format JSON mais uniquement au format XML. De plus, SOAP est un protocole à part entière et ne repose donc pas sur le protocole HTTP. Enfin, les services web SOAP sont plus volumineux qu'une API REST.

Choix d'implémentation et des outils

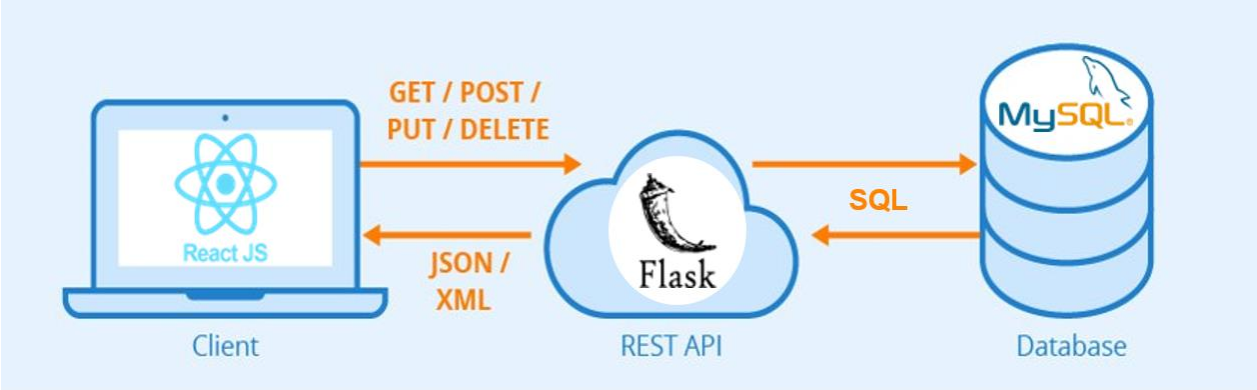


Figure 25 : Schéma de la communication Client-API-Serveur de notre projet

Pour développer notre API, nous avons choisi un framework basé sur le langage Python avant tout car le développement de nos algorithmes de recommandations et les tests se sont fait en Python via Jupyter Notebook.

Une fois le langage du framework backend identifié, il nous fallait choisir lequel était le meilleur pour notre projet et nos besoins parmi les frameworks Python proposés.

La problématique rencontrée par la suite s'est donc portée sur le choix entre le framework Django ou le micro-framework Flask. Ces deux frameworks Python sont au coude-à-coude ces derniers temps en termes de statistiques d'utilisation et d'activité open-source.

Plusieurs raisons nous ont conduit à ce choix, mais la principale reste que Flask est un micro-framework : il est très léger, ce qui en fait un des plus flexibles à l'installation et la configuration. Pour ce qui est de Django, il embarque de nombreux composants dont nous n'avions pas de réel besoin pour la mise en place de notre API. L'utilisation de Django aurait donc été à privilégier dans le cas d'un service web de plus grande envergure nécessitant davantage de fonctionnalités comme une interface d'administration (générée automatiquement avec Django) ou un générateur de templates HTML.

Développement de l'API

Librairie Flask-RESTPlus

Pour simplifier le développement de notre API, nous avons choisi d'utiliser la librairie Flask-RESTPlus.

Flask-RESTPlus est une extension pour Flask qui ajoute la prise en charge de la création rapide d'API REST. Flask-RESTPlus permet de mettre en place les meilleures pratiques avec une configuration minimale. Flask-RESTPlus est facile à prendre en main car il est similaire à Flask. Il fournit une collection cohérente de décorateurs et d'outils pour décrire une API et exposer correctement sa documentation (à l'aide de Swagger).

Swagger est un logiciel basé sur les technologies du web (HTML, Javascript, CSS) permettant de générer une documentation en utilisant les spécifications d'OpenAPI. Il fournit aussi une sandbox permettant de tester les appels API directement depuis la documentation générée.

La documentation Swagger de notre API se trouve à l'adresse *https://ds50api.fr/*

Création des routes et endpoints

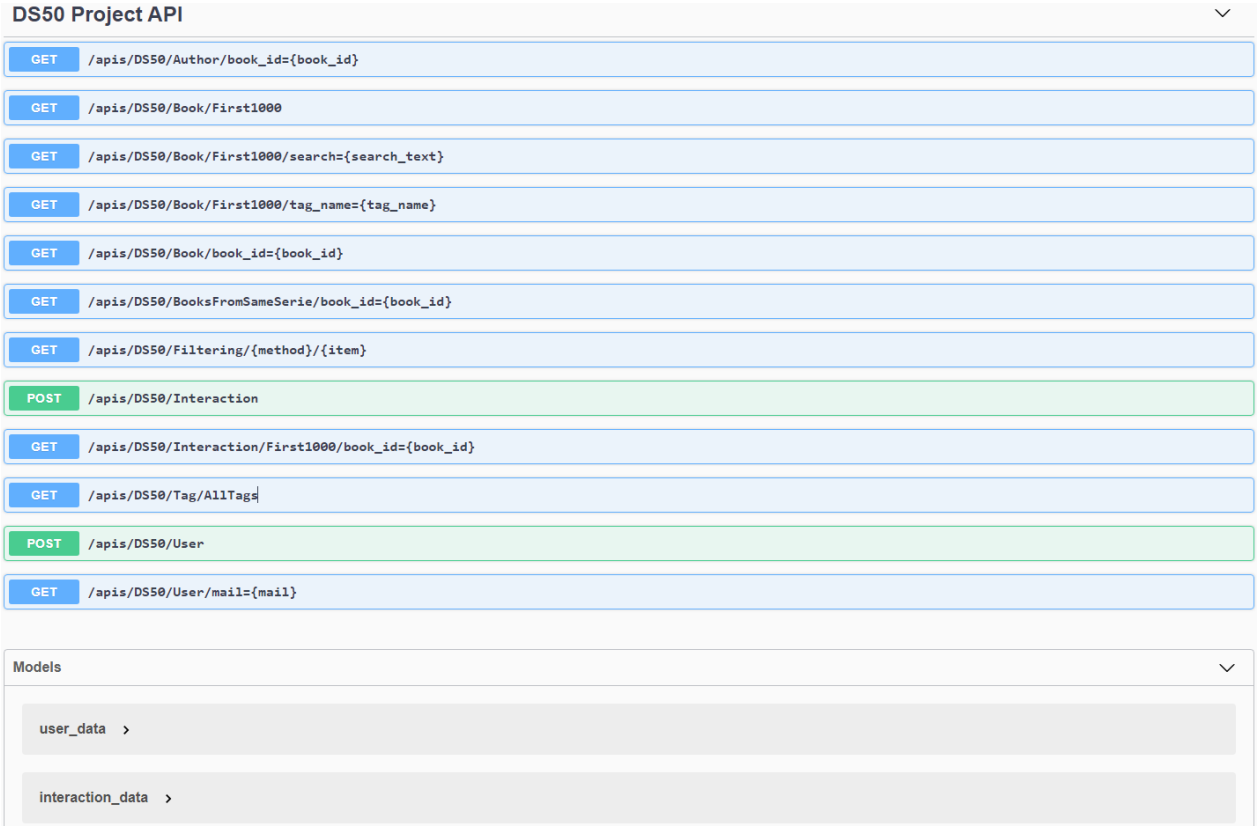


Figure 26 : Aperçu de la documentation présentant les requêtes implémentées et les modèles

Nous avons choisi d'implémenter uniquement les requêtes dont avait besoin le client bien que nous aurions pu implémenter beaucoup d'autres requêtes comme des requêtes d'*UPDATE* ou *DELETE*. Ainsi, toutes les requêtes ci-dessus sont utilisées dans l'application client.

Notre application Flask est initialisée et lancée depuis le fichier *__init__.py*.

C'est aussi dans ce fichier que la documentation est créée et que les routes et endpoints sont définis.

Voici par exemple comment est définie la route de notre requête GET permettant d'obtenir les 1000 premiers livres de notre base de données :

```
@api.route('/apis/DS50/Book/First1000')
class First1000Books(Resource):
    def get(self):
        return getFirst1000Books()
```

Figure 27 : Définition de la route d'une requête sans paramètre

C'est grâce à ce chemin concaténé à l'adresse IP ou nom de domaine de l'API que la requête pourra être appelée par le client. Voici l'URL complète de cette requête :

https://ds50api.fr/apis/DS50/Book/First1000

On peut aussi permettre au client de fournir un ou plusieurs paramètres à une requête lors de la définition de cette dernière.

```
@api.route('/apis/DS50/Book/First1000/search=<search_text>')
class First1000BooksBySearch(Resource):
    def get(self, search_text):
        return getFirst1000BooksBySearch(search_text)
```

Figure 28 : Définition de la route d'une requête avec paramètre

Interaction avec la base de données

Comme on peut le remarquer sur la figure précédente, lorsque l'on souhaite accéder à la route définie, une fonction Python est appelée. C'est cette fonction qui va contenir le code SQL permettant d'interagir avec la base de données et dans le cas précédent effectuer une requête de sélection.

Toutes nos fonctions Python permettant d'interagir avec notre base de données MySQL sont définies dans les fichiers suivants :

- *filtering.py* (sélection de données dans la base et traitements)
- *post_data.py* (insertions de données dans la base pour les requêtes *POST* côté client)
- *request_data.py* (sélection de données dans la base pour les requêtes *GET* côté client)
- *request_filter.py* (sélection de données dans la base et traitements)

Les fonctions présentes dans ces fichiers peuvent aussi bien effectuer d'autres opérations de traitement non liées à la base de données comme le pré-processing ou le filtrage des données dans le cas de nos fonctions liées à la recommandation.

Exemple pour une requête de sélection

Depuis une connexion à notre serveur de base de données, on définit la requête SQL grâce à laquelle nous pouvons extraire les données dans une dataframe Pandas.

```
def getFirst1000BooksBySearch(search_text):
    connection = getConnectionFromServer()
    df = pd.read_sql(
        """SELECT
            DISTINCT B.*
        FROM
            BOOK B INNER JOIN WROTE W
                ON B.BOOK_ID = W.BOOK_ID
            INNER JOIN AUTHOR A
                ON A.AUTHOR_ID = W.AUTHOR_ID
        WHERE
            UPPER(B.TITLE) LIKE '%"""+ search_text.upper() +"""%'
        OR
            UPPER(A.NAME) LIKE '%"""+ search_text.upper() +"""%'
        ORDER BY
            B.RATINGS_COUNT DESC
        LIMIT 1000;"""
        , connection)
    df = fillDataframeNulls(df)
    return df.to_dict('records')
```

Figure 29 : Fonction qui vient requêter des données dans la base

La fonction *fillDataFrameNulls()* permet de remplacer les champs nuls par des chaines vides ou des 0 dans le cas des colonnes d'entiers pour que les données soient plus facilement exploitables par le client.

Un dictionnaire ou tableau de dictionnaires est ensuite retourné à la fonction *get* du fichier *__init__.py*.

Exemple d'une requête d'insertion

Pour une requête d'insertion, la définition de la route est similaire. On définit ensuite le modèle de données attendu dans le *body* renseigné lors de la requête.

```python
@api.route('/apis/DS50/User')
class User(Resource):
    user_data = ns_user.model(
    "user_data",
    {
        "first_name": fields.String(required=True),
        "last_name": fields.String(required=True),
        "username": fields.String(required=True),
        "password": fields.String(required=True),
        "mail": fields.String(required=True),
        "address": fields.String(required=True),
        "first_fav_category": fields.String(required=True),
        "second_fav_category": fields.String(required=True),
        "third_fav_category": fields.String(required=True),
    },
)
    @ns_user.expect(user_data)
    def post(self):
        return postUser(request.get_json())
```

Figure 30 : Définition de la structure de données attendue et de la route pour une requête POST

On appelle ensuite la fonction (ici *`postUser()`*) qui va interagir avec la base de données en lui renseignant en paramètre (*`request.get_json()`*) les données à insérer qui sont présentes dans le corps de la requête.

```
def postUser(user):
    new_user_id = getDynamicNewUserID()
    connection = getConnectionFromServer()
    mycursor = connection.cursor()
    sql = """INSERT INTO
        USER
    VALUES (
        """+ new_user_id +""",
        '"""+ user['first_name'] +"""',
        '"""+ user['last_name'] +"""',
        '"""+ user['username'] +"""',
        '"""+ user['password'] +"""',
        '"""+ user['mail'] +"""',
        '"""+ user['address'] +"""',
        CURDATE(),
        """+ user['first_fav_category'] +""",
        """+ user['second_fav_category'] +""",
        """+ user['third_fav_category'] +"""
    )
    ;"""
    mycursor.execute(sql)
    connection.commit()
    return new_user_id
```

Figure 31 : Insertion des données reçues dans la base

Le champ *user_id* et le champ *sign_in_date* ne sont pas saisis par le client mais sont gérés via l'API pour simplifier la gestion côté client. Le champ *user_id* est calculé en requêtant la base et en recherchant le plus grand id connu ensuite incrémenté de 1.

Déploiement de l'API sur un serveur

Nous avons hébergé notre API sur un *droplet* de DigitalOcean en choisissant une configuration minimale en termes de ressources.

Les droplets DigitalOcean sont des machines virtuelles basées sur Linux qui s'exécutent sur du matériel virtualisé. Chaque droplet créé est un nouveau serveur que l'on peut utiliser, soit de manière autonome, soit dans le cadre d'une infrastructure cloud plus vaste.

Voici la configuration de notre Droplet :

Figure 32 : Configuration du Droplet

Nous avons utilisé une interface WSGI et un serveur Apache HTTP pour déployer notre application Flask. WSGI (Web Server Gateway Interface) est une interface entre les serveurs Web et les applications Web pour Python. Pour se connecter à notre serveur et le configurer, nous utilisons une communication SSH.

Voici les différentes étapes de la configuration :

1. Installer et activer *mod_wsgi* (module permettant à Apache de servir des applications Flask)
2. Cloner le repository Git avec les fichiers sources de l'API sur le serveur
3. Installer Flask et les librairies nécessaires dans l'environnement virtuel Python
4. Configurer et activer un nouvel hôte virtuel Apache

```
<VirtualHost *:80>
        ServerName ds50api.fr
        ServerAdmin admin@ds50api.com
        WSGIScriptAlias / /var/www/flask_web_api/flaskapp.wsgi
        WSGIApplicationGroup %{GLOBAL}
        <Directory /var/www/flask_web_api/>
                Order allow,deny
                Allow from all
        </Directory>
        Alias /static /var/www/flask_web_api/static
        <Directory /var/www/flask_web_api/static/>
                Order allow,deny
                Allow from all
        </Directory>
        ErrorLog ${APACHE_LOG_DIR}/error.log
        LogLevel warn
        CustomLog ${APACHE_LOG_DIR}/access.log combined
RewriteEngine on
RewriteCond %{SERVER_NAME} =ds50api.fr
RewriteRule ^ https://%{SERVER_NAME}%{REQUEST_URI} [END,NE,R=permanent]
</VirtualHost>
```

Figure 33 : Fichier de configuration de l'hôte virtuel

5. Créer le fichier de configuration WSGI

```
#!/usr/bin/env python3
import sys
import logging
logging.basicConfig(stream=sys.stderr)
sys.path.insert(0,"/var/www/flask_web_api/")
sys.path.insert(0,"/var/www/flask_web_api/env/lib/python3.8/site-packages")
from __init__ import app as application
application.secret_key = 'DS50project'
```

Figure 34 : Fichier de configuration flaskapp.wsgi

6. Redémarrer le serveur Apache

L'API est maintenant déployée et accessible par notre client.

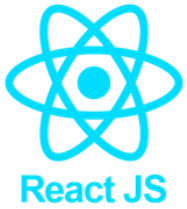
Pour la mettre à jour, il suffit de mettre à jour la branche Git et de redémarrer le serveur Apache avec la commande *sudo service apache2 restart*.

Front-End

Le frontend ou coté client est la partie du site avec laquelle les acheteurs interagissent directement. C'est la première chose qu'un visiteur remarque à propos d'un site web. La majorité des visiteurs attribuent la crédibilité d'un site web à son design plutôt qu'autre chose. L'UX design est donc un élément clé d'une importance cruciale pour un site web.

Outils et technologies

React.js



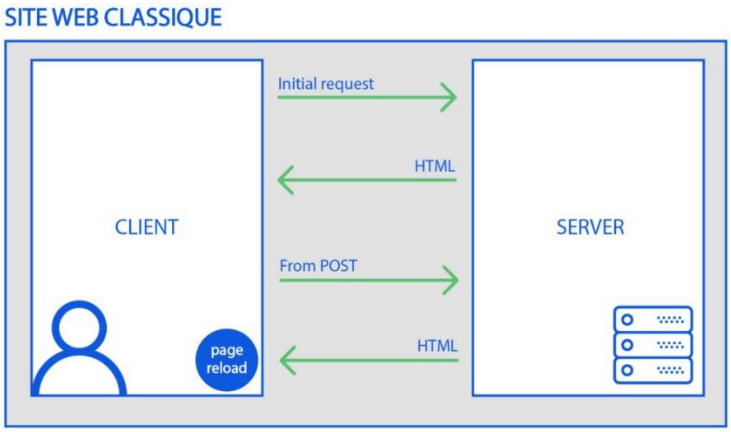
React est une bibliothèque JavaScript simple et performante pour créer des interfaces web. Nous avons choisi d'utiliser React pour ses nombreux avantages, à savoir :

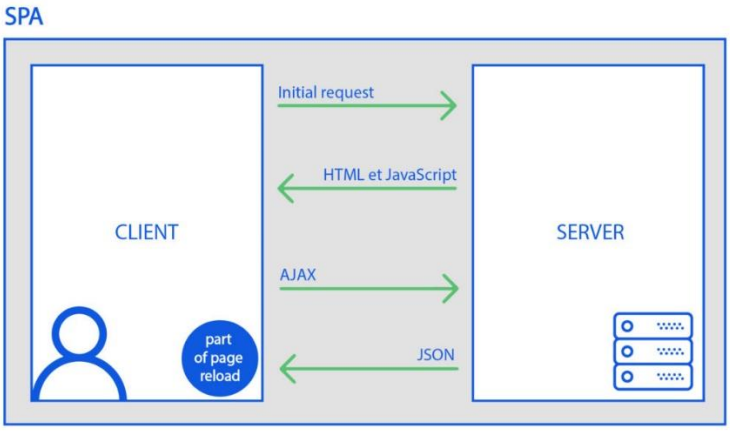
- C'est une plateforme très flexible et très performante,
- Elle facilite la création des interfaces utilisateurs en permettant de développer des composants réutilisables,
- Elle dispose d'une approche native qui permet de développer des applications mobiles pour n'importe quel système IOS, Android, etc., ce qui est d'un grand intérêt pour ce type de projets si on veut par la suite créer une application mobile pour notre site e-commerce et va permettre d'économiser des temps de développement considérables.

React permet également de construire des **SPA** (**S**ingle **P**age **A**pplication).

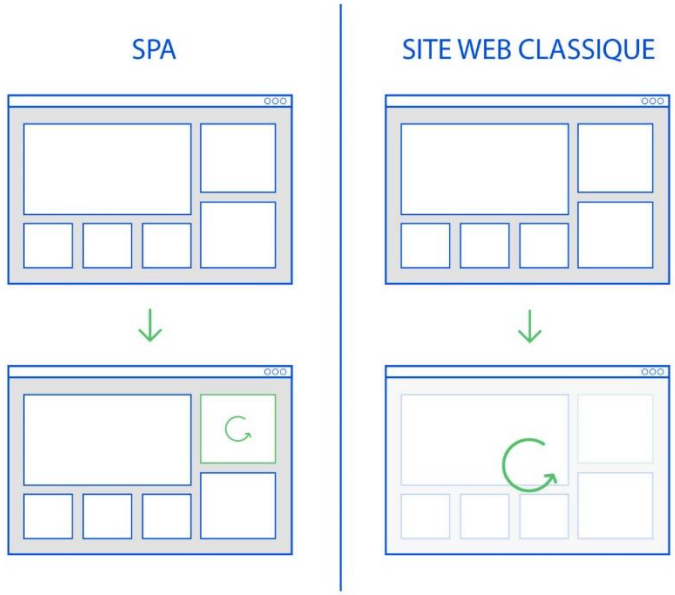
Un site web monopage **SPA** est une approche plus moderne du développement web. Elle est utilisée aujourd'hui par plusieurs grandes entreprises comme Google, Facebook, Twitter, etc. Les **SPA** fonctionnent à l'intérieur d'un navigateur et ne nécessite pas de rechargement de page pendant l'utilisation.

D'autre part, les sites web classiques **MPA** (**M**ultiple **P**age **A**pplication) sont le concept de base du développement web. Cependant, ce modèle de conception multipages nécessite un rafraîchissement complet de page même si une partie du contenu reste inchangée. C'est une option moins intéressante car la plupart des utilisateurs sont souvent déconcertés et agacés de devoir cliquer sur de nombreux liens dans un site traditionnel pour arriver là où ils veulent.





L'absence de rechargement de page permet une navigation plus rapide et fluide : il n'y a plus de flash blanc entre deux pages lors de rechargement, on peut donc mieux gérer l'affichage de contenus dynamiques ou asynchrones.



MDBootstrap

Nous avons également utilisé **MDB** (**M**aterial **D**esign for **B**ootstrap). Bootstrap est le Framework **CSS** le plus populaire au monde. Il permet de développer des sites web réactifs qui s'adapte à toutes les résolutions d'écran. Il utilise un système de grille flexible qui facilite le positionnement des composants React. Et grâce à **MDB**, nous avons pu déployer gratuitement notre site web sur internet en utilisant le **MDBgo**.

Conclusion

Pour conclure, nous sommes globalement satisfaits de notre production finale. L'objectif était de développer un site de e-commerce fonctionnel et doté d'un moteur de recommandation cohérent et efficace. Bien que le thème principal soit les moteurs de recommandation, nous ne souhaitions négliger aucun aspect du projet, c'est pourquoi la répartition des tâches aura été cruciale.

Concernant la partie Frontend, nous sommes très satisfaits du rendu visuel qui se rapproche bien de ceux des sites de commerce en ligne, mais aussi de toutes les fonctionnalités implémentées qui rendent le site plutôt complet.

Pour ce qui est du Backend, toute la partie concernant le prétraitement et la gestion de la base de données aura été assez longue et fastidieuse aux vu du dataset que nous avons choisi. Néanmoins, nous avons bien su gérer ces étapes du projet, que ce soit dans la conception ou bien dans la réalisation. Au niveau du moteur de recommandation, étant donné que nous travaillions avec un dataset très riche en informations, nous avons pu découvrir et expérimenter les principaux algorithmes de filtrage utilisés à l'heure actuelle. Même s'il est difficile d'évaluer rapidement l'efficacité d'un moteur de recommandation, les résultats obtenus nous semblent plutôt cohérents.

Aussi, nous sommes satisfaits d'avoir pu travailler sur le thème du NLP (Natural Langage Processing) afin de recouvrir quasiment toutes les UV de ce semestre en filière Data Science.

Enfin, ce projet nous aura permis de progresser au sein d'une équipe sur un projet de manière à nous rapprocher d'une expérience que nous aurions pu avoir dans un cadre professionnel. Par exemple, ce projet d'équipe nous aura amené à :

- Répartir les postes et tâches à réaliser en fonction des différents profils de l'équipe
- Progresser efficacement grâce à la méthode Agile
- Gérer des dépôts Github dans le cadre d'un projet d'équipe
- Déployer et héberger des web services
- Communiquer fréquemment sur l'avancement et se fixer des objectifs à court, moyen et long terme
- Documenter nos travaux

Ainsi, nous avons pu aborder tous ces aspects du travail d'équipe. Être acteur de la réalisation de ce projet aura été une très bonne expérience qui ne pourra que nous être bénéfique par la suite.

Bibliographie

[1] M. Wan et J. McAuley, «Item Recommandation on Monotonic Behaviour Chains,» *RecSys'18.*

[2] M. Wan, R. Misra, N. Nakashole et J. McAuley, «Fine-Grained Spoiler Detection from Large-Scale Review Corpora,» *ACL'19.*

[3] Bird, Steven, E. Loper et E. Klein, Natural Language Processing with Python, O'Reilly Media Inc., 2009.

[4] G. A. Miller, «WordNet: A lexical Database for English,» *Communications of the ACM,* vol. 38, n° %111, pp. 39-41.

[5] C. Fellbaum, WordNet: An Electronic Lexical Database, Cambridge, MA: MIT Press, 1998.

Table des illustrations

Mots clefs

Data Science – Web – Développement

BELMONTE Elian **MIGNEROT Grégori**
BOUNNIT Zakaria **PLANCHE Antoine**
GHARBI Aziz **STACH Benjamin**

DS50 – Rapport technique

Résumé

Dans le cadre du bloc métier Data Science de notre cursus d'ingénieur en informatique à l'UTBM, nous avons réalisé un projet général, la création d'un site web type e-commerce équipé d'un moteur de recommandation. Nous décrivons ici le travail réalisé et les détails de son implémentation

utbm
université de technologie
Belfort-Montbéliard