MyFlat

Plateforme immobilière et systèmes de recommandation

TRAVAIL DE BACHELOR

LOUIS CHAUBERT

Septembre 2018

Superviseur :

Prof. Dr. Jacques PASQUIER–ROCHA
Software Engineering Group

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Software Engineering Group
Department of Informatics
University of Fribourg
(Switzerland)

Abstract

Partant de l'idée d'un groupe d'étudiants de différents horizons, MyFlat est une plateforme immobilière souhaitant combler les déficiences des plateformes actuellement sur le marché.

Le rapport détaille son utilisation d'un point de vue utilisateur autant de la perspective d'un visiteur que d'un annonceur. La suite de la lecture conduit vers l'aspect technique du développement de la plateforme. Celle-ci a été développée grâce aux technologies *MeteorJS* pour le serveur, *ReactJS* du côté client et *MongoDB* pour la base de données. L'utilisation de ces technologies est détaillée pour une implémentation générale et accompagnée d'exemples tirés du projet MyFlat. La plateforme devant proposer des offres immobilières aux visiteurs, celle-ci se doit d'intégrer un système de recommandation le plus pertinent possible. C'est donc le sujet de la troisième partie de ce rapport, une étude sur les différents systèmes de recommandation existant avec leurs cas d'utilisation, avantages et inconvénients.

MyFlat intègre actuellement un système de recommandation très basique devant évoluer en un système hybride. Le futur du projet est de peaufiner les fonctionnalités existantes, intégrer celles manquantes et, surtout, développer et implémenter un système de recommandation plus pertinent.

Mots-clés : systèmes de recommandation, immobilier, meteorjs, reactjs, mongodb, MyFlat

Préambule

Préface

Le projet a originairement pris forme entre les mains de Kévin Currat. Travaillant alors pour une agence immobilière, il se rend compte des besoins du marché et décide de les combler. L'équipe actuelle, de quatre étudiants, est alors formée sur la base d'un objectif commun : MyFlat. Mon intérêt pour les sciences des données et le besoin d'un système de recommandation combiné aux nouvelles technologies utilisées pour le développement du projet, ont créé un terrain fertile à la rédaction de ce travail qui combine ces différents aspects.

Reconnaissances

Par ces quelques mots, j'aimerais exprimer une gratitude spéciale au Professeur Dr. Jacques Pasquier-Rocha que m'a donné l'opportunité de travailler sur ce projet passionnant dont les recherches sur les systèmes de recommandation m'ont apporté beaucoup de connaissances dans le domaine des sciences de données qui me passionne.

Mes remerciements vont également vers toutes les personnes m'ayant aidé et apporté des connaissances sur les différents thèmes du travail. Je pense notamment au Professeur Dr. Tobias Scheffer et son cours à propos de l'analyse intelligente de données.

L'équipe MyFlat, ma famille et mes amis reçoivent des remerciements particuliers pour m'avoir aidé à finaliser ce projet dans les limites de temps défini.

Notations et conventions

Les termes anglais faisant partie du vocabulaire nécessaire aux explications sont écrits en *italique*. Certains de ces termes, si une explication est jugée nécessaire, sont définis en note de bas de page mais, la plupart est considérée comme faisant partie intégrante du vocabulaire du sujet. En *italique* sont aussi les termes francophones faisant référence à un élément autre que leur signification première. Par exemple, *disponible le* fait référence au champ de l'interface dont le label s'appelle « disponible le ».

Certains concepts sont également considérés comme acquis et ne sont donc pas détaillés mais utilisés pour des explications plus approfondies. Par exemple, aucune explication n'est donnée sur ce qu'est un modèle de régression ou de classification mais ces concepts sont utilisés durant tout le chapitre sur les systèmes de recommandation.

Les portions de code incomplets servant aux explications sont affiché dans leurs totalités dans le chapitre Code Source.

Table des matières

Liste des figures

Liste des tableaux

Liste des sources de code

1 Introduction

1.1 But du projet

La plateforme web est développée dans le cadre d'un projet de start-up nommé MyFlat. Ce projet est mis sur pied par 4 étudiants, issus de l'EIA-FR, de la HEG-FR et de l'Université de Fribourg dont deux s'occupent du développement de la plateforme tandis que les seconds du marketing et de la gestion.

MyFlat a pour vocation d'offrir un service d'annonces immobilières gratuites combinées avec des outils jusqu'à lors manquants sur le marché. Il existe actuellement diverses plateformes web immobilières telles que *immoscout* ou *homegate* mais celles-ci ont toujours le désavantage d'être payantes pour les annonceurs. Beaucoup d'annonces se retrouvent donc sur des sites de petites annonces qui ne sont pas adaptés. Pour pallier le manque de revenus, MyFlat propose des services additionnels qui, eux, sont payants. Ces services sont inspirés du modèle des sites de petites annonces tels que la mise en évidence d'annonces ou le placement de l'annonce dans le haut de la liste.

La plateforme est développée au moyen du *framework Meteor* pour le serveur et utilise une base de données MongoDB. Le client est, lui, développé avec *ReactJS* dans l'objectif de pouvoir facilement migrer vers *React Native* afin de créer une application mobile même si, dès la première version, MyFlat sera *responsive design*[1]. Comme expliqué ci-dessus, nous sommes deux à nous occuper du développement, Thomas et moi-même. Dans les grandes lignes directrices, Thomas s'occupe de la partie authentification et gestion des services payants tandis que pour ma part, j'implémente tout ce qui est de la publication et visualisation des annonces.

Une seconde partie du travail consiste à faire de la recherche sur les services et algorithmes de recommandation afin de pouvoir proposer les annonces les plus pertinentes à l'utilisateur en ciblant ses intérêts et désirs. Ceci en vue d'une future implémentation dans MyFlat.

[1] *Responsive design* signifie que la page web s'adapte selon les caractéristiques de l'appareil utilisé.

1.2 Déroulement du projet

Le projet s'est déroulé en deux grandes phases principales. La première a été le développement de MyFlat suivi des recherches sur les systèmes de recommandation.

Durant le développement, de nombreux meetings ont été organisé entre les membres afin de discuter de la vision du projet, des différentes fonctionnalités et de quelles façons on souhaitait les implémenter. Après chaque meeting, une liste de tâches était définie et distribuée à chacun des membres. Pour les tâches liées au développement, Thomas et moi séparions d'un commun accord ce sur quoi chacun allait travailler. Les tâches principales et leurs répartitions durant le projet sont les suivantes :

Quoi ?	**Qui ?**
Recherche d'un thème (*template*)	Thomas
Mise en place du server	Thomas
Adaptation du thème *apartment* vers *ReactJS*	Louis
Identification et enregistrement	Thomas
Création d'annonces	Louis
Recherche d'annonces	Louis
Affichages des annonces	Louis
Modification des annonces	Louis
Modification des profils utilisateurs	Louis
Formulaires de contact (mail)	Thomas
Favoris	Louis

Tableau 1.1 : tâches et répartitions

L'un des grosses difficultés fût de faire fonctionner correctement les fichiers *Javascript* externe avec *ReactJS*, notamment l'*API* Google Maps. Ceci car *ReactJS* fonctionne de manière asynchrone et cela nous a posé quelques problèmes pour l'implémentation de certaines fonctionnalités. Pour résoudre ce type de problème, autant Thomas que moi essayions de chercher des solutions. Il a fallu également adapter à *ReactJS* le thème *apartment* acheté sur https://themeforest.net/.

La phase de recherche sur les systèmes de recommandation fût forte en apprentissage. J'ai notamment eu la chance de pouvoir participer au cours intitulé « *Intelligent Data Analysis* » du Professeur Scheffer de l'Université de Potsdam traitant la majorité des modèles de régression et classification utilisé dans le monde du *machine learning*[2] et de les utiliser dans des applications concrètes. Le *machine learning* établissant les fondamentaux des systèmes de recommandations, le livre « *Pattern recognition and machine learning* »[Bis06] de Christopher M. Bishop fût d'une grande aide. L'ouvrage de Charu C. Aggarwal intitulé « *Recommender*

[2] Le *machine learning* est un domaine de l'informatique utilisant les statistiques afin de donner à l'ordinateur la capacité d'« apprendre » avec des données, sans avoir été explicitement programmé.

Systems »[Agg16] m'a permis de cerner et comprendre les différences entre les différents systèmes de recommandation existants et dans quel cas utiliser lequel. D'autres sources non-négligeables m'ont également permis de comprendre plus en détail certains concepts associés aux thèmes principaux.

1.3 Plan du rapport

Le rapport est constitué de trois chapitres principaux accompagnés d'une introduction et d'une conclusion.

La première partie est une explication de MyFlat du point de vue des utilisateurs. Il s'agit donc d'une sorte de mode d'emploi des différentes fonctionnalités auxquels les utilisateurs ont accès. Le mode d'emploi est lui-même séparée en deux parties afin d'expliciter séparément les fonctions visiteurs et annonceurs.

La seconde partie offre un aperçu de MyFlat du point de vue des développeurs. Dans ce chapitre, les différentes technologies sont expliquées avec des exemples concrets d'implémentation dans MyFlat. Ce chapitre est décomposé en trois sous-chapitres faisant référence à chacune des technologies, c'est-à-dire : *MeteorJS, ReactJS* et *MongoDB.*

La troisième et dernière partie traite des systèmes de recommandations de manière générale. Les quatre principaux types de systèmes de recommandations ainsi que leurs différentes implémentations sont expliqués. Le premier type traité est celui des systèmes de recommandations collaboratifs, suivi de ceux basés sur le contenu puis de ceux basés sur la connaissance. Pour terminer, les systèmes de recommandations hybrides, dont ceux basés sur un ensemble de systèmes, terminent ce chapitre.

2

MyFlat : point de vue utilisateurs

Ce document décrit les différentes fonctionnalités de MyFlat. Dans un premier temps, les fonctionnalités des visiteurs, c'est-à-dire les personnes à la recherche d'un logement, sont décrites puis, suivent les fonctionnalités des annonceurs qui peuvent autant être des particuliers que des agences immobilières.

Le site est entièrement *responsive-design*. Il peut donc également être utilisé sur un mobile ou une tablette. Malgré cela, une application mobile est prévue afin d'optimiser encore plus l'expérience utilisateur.

L'en-tête (*header*) ainsi que le pied de page (*footer*) restent les mêmes sur toutes les pages, voir figure 2.1.

Dans le haut de page, l'utilisateur peut changer de langue ainsi que trouver quelques brèves informations de contact. Les langues qui seront disponibles sont le français, l'allemand et l'anglais. Actuellement, seule la traduction anglaise est complètement introduite mais le mécanisme est fonctionnel. L'icône en forme d'étoile permet d'afficher les favoris de l'utilisateur. Et le cadenas ouvre le pop-up de connexion d'où l'on peut se connecter, s'enregistrer ou récupérer son mot de passe en cas d'oubli. Plus d'informations sur ces sujets sont données dans les chapitres correspondants.

Dans le pied de page, différentes informations de contact sont disponibles ainsi que des liens vers les diverses pages du site. Une inscription à la newsletter permet également à l'utilisateur d'être au courant des dernières nouveautés.

La plateforme est actuellement disponible à l'adresse suivante : http://myflat.chaubi.ch

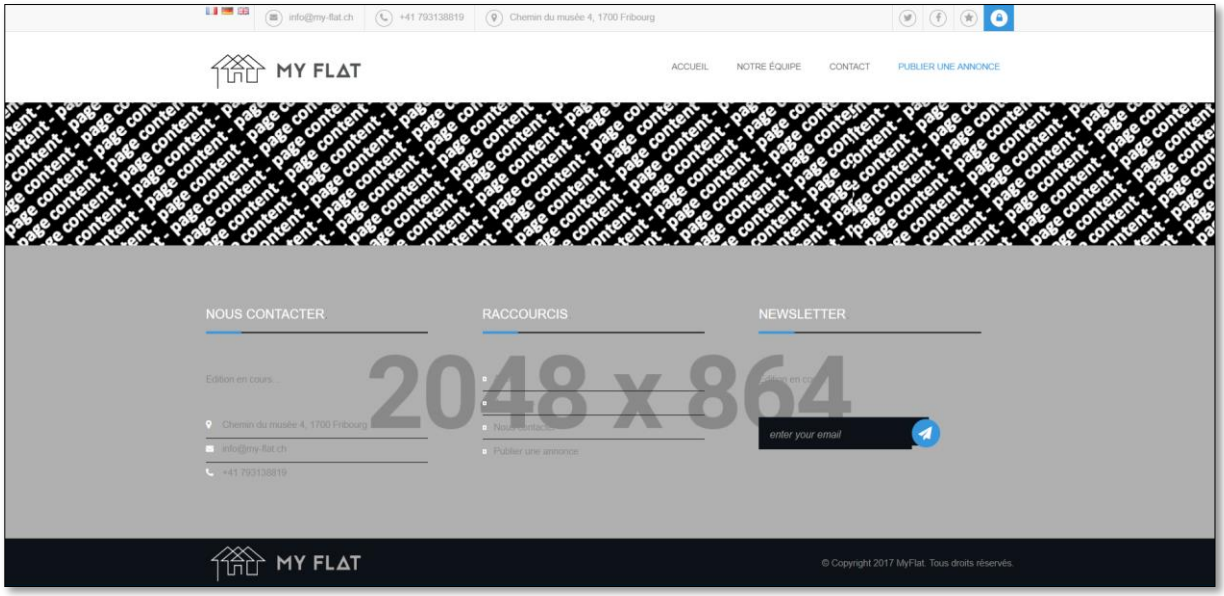

Figure 2.1: en-tête et pied de page

2.1 Visiteurs

2.1.1 Page d'accueil

La page d'accueil est principalement constituée de deux éléments. Le premier composant est le formulaire permettant la recherche de logements qui est accompagnée d'une carte affichant leur localisation. Le second élément est le carrousel d'offres fortement recommandées à l'utilisateur. Ce dernier est encore en cours d'élaboration. C'est lui qui aura pour rôle d'afficher les offres recommandées par le système de recommandation.

Le formulaire de recherche permet à l'utilisateur de rechercher selon trois catégories de *transaction* : location, colocation et vente. Cette information figurant dans la base de données, une quatrième catégorie peut aisément être ajoutée (ou supprimée/modifiée). Selon la

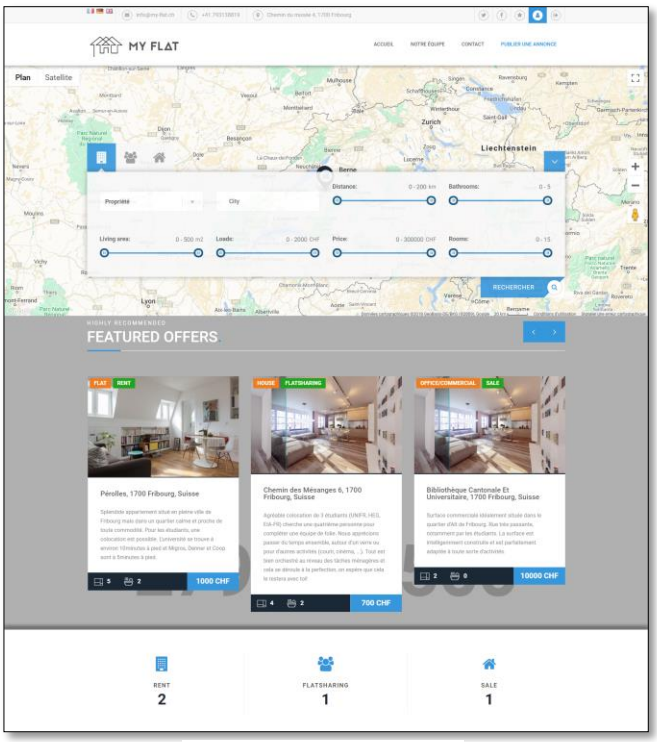

Figure 2.2: page d'accueil

transaction choisie, les champs de recherche sont adaptés. L'utilisateur peut alors sélectionner le type d'*estate* qu'il souhaite. Les différents types d'*estate* disponibles sont : les appartements, les bureaux/surfaces commerciales et les maisons. Selon l'*estate* choisie, les autres champs seront également adaptés en conséquence. Tous ces différents champs, appelés *properties*, proviennent de la base de données et peuvent donc également être modifiés selon la convenance.

La carte affichée en arrière-plan, derrière le formulaire de recherche, affiche tous les logements disponibles de la région. Si l'utilisateur a déjà effectué une recherche à l'aide du formulaire, la carte sera alors centrée sur cet endroit mais, si ce n'est pas le cas, l'utilisateur sera localisé grâce à son adresse IP afin d'afficher les logements proches de chez lui. Le carrousel d'offres fortement recommandées est, comme expliqué ci-dessus, encore en cours d'élaboration. Son fonctionnement actuel reste très sommaire. Il affiche, simplement, les logements les plus proches de la dernière recherche de l'utilisateur et les trie selon certains critères pré-définis.

2.1.2 Liste des offres

Lorsque l'utilisateur effectue une recherche à l'aide du formulaire de recherche depuis la page d'accueil, il est redirigé sur la page de la liste des offres. Cette page affiche les résultats selon la recherche de l'utilisateur. Les offres peuvent être triées selon des critères tels que la pertinence (par défaut), le plus proche et le prix croissant ou décroissant.

Accompagnant la liste, un second formulaire de recherche permet à l'utilisateur de modifier sa précédente recherche en précisant ses critères. Lorsque l'utilisateur passe au-dessus d'une offre avec la souris, la géolocalisation apparait et l'utilisateur peut accéder aux détails de l'offre.

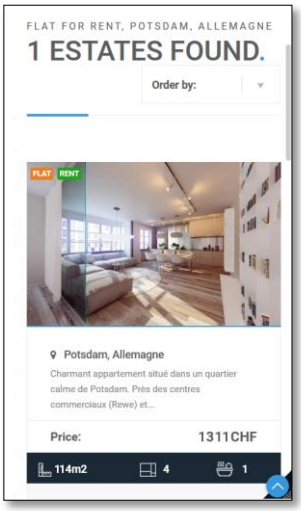

Figure 2.3: liste des offres

2.1.3 Détail de l'offre

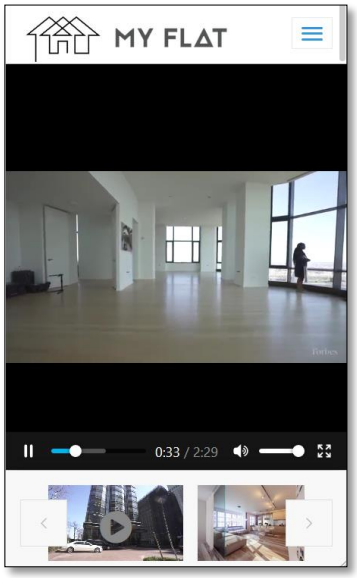

Figure 2.4: détail de l'offre

Le détail de l'offre commence par afficher tout le contenu multimédia (vidéos et photos) en mettant en avant les vidéos s'il y en a.

S'en suivent les différentes informations entrées par l'annonceur. Certaines informations, plus importantes, sont mises en évidence telles que le prix, la date de disponibilité, le nombre de pièces, etc…

L'utilisateur a également la possibilité d'ajouter l'annonce dans ses favoris afin de pouvoir la retrouver plus tard. Pour ce faire, il lui suffit de cliquer sur la grande étoile se trouvant à droite du titre de l'offre.

Après les informations relatives au logement, l'utilisateur peut repérer l'emplacement grâce à une carte et également voir l'aspect extérieur grâce à Google Street Views.

Le détail de l'offre se termine par le formulaire de contact de l'annonceur afin de le joindre en cas d'intérêt pour le logement.

2.1.4 Notre équipe

Cette page décrit l'équipe se trouvant derrière MyFlat. Afin d'humaniser la plateforme et d'expliquer les raisons et motivations qui nous ont poussé à lancer ce projet.

2.1.5 Contact

La page de contact est constituée d'un formulaire permettant d'envoyer un email à info@my-flat.ch. Quelques autres informations relatives à la prise de contact sont également disponibles telles qu'un numéro de téléphone, une adresse et des horaires.

2.1.6 Favoris

Pour accéder aux favoris, il faut cliquer sur l'étoile située dans l'en-tête du site. Les offres ajoutées aux favoris s'affichent dans un pop-up sous forme de liste. En cliquant sur une offre, on accède simplement à ses détails.

Les favoris sont autant ajoutés dans le *localStorage* que dans la base de données, si l'utilisateur est connecté. Il y a donc deux listes en parallèle. Celle affichée dans le pop-up est la fusion des deux. Le choix d'enregistrer les favoris autant dans le *localStorage* que dans la base de données vient du fait que l'on souhaite qu'un utilisateur puisse utiliser les favoris sans avoir à être connecté mais que s'il est connecté, il puisse également retrouver ses favoris sur un autre appareil.

2.2 Annonceurs

Pour accéder aux fonctionnalités des annonceurs, il faut préalablement s'enregistrer. Les fonctionnalités des annonceurs (et de toute personne connectée) sont la publication d'offre, la visualisation de ses annonces avec la possibilité de les modifier et de les supprimer, la modification de son profil.

D'autres fonctionnalités payantes viendront dans de prochaines versions. La première version de MyFlat est entièrement gratuite. Les futures fonctionnalités seront la mise en évidence des offres lors des recherches, une amélioration dans le classement de l'offre et, plutôt pour les agences, l'accès à des statistiques sur les visualisations de leurs offres (données démographiques, géographiques, etc…).

2.2.1 Connexion / Enregistrement

La connexion se fait depuis l'icône en forme de cadenas située dans l'en-tête. Il est possible de se connecter avec Google et Facebook, qui ne nécessitent pas d'enregistrement préalable, ou alors avec une adresse email et un mot de passe, qui nécessite un enregistrement préalable.

Pour s'enregistrer, il faut, dans un premier temps ouvrir le pop-up de connexion par le biais du cadenas et ensuite cliquer sur « S'ENREGISTRER » situé en bas de la pop-up. Le pop-up d'enregistrement s'ouvre et l'utilisateur est prié de remplir les différents champs. La connexion est automatique après l'enregistrement.

L'utilisateur peut également réinitialiser son mot de passe en cas d'oubli. Pour ce faire, il lui suffit de cliquer sur « Mot de passe oublié » situé sous le bouton de connexion.

Une fois connecté, le cadenas disparaît pour laisser place à une icône en forme de bonhomme et une autre avec une flèche. Ces derniers permettent, respectivement, d'être redirigé vers ses offres et de se déconnecter.

2.2.2 Publier une offre

Pour accéder à la page de publication des offres, l'utilisateur doit être connecté. Si ce n'est pas le cas, il sera invité à le faire.

Cette page est construite en plusieurs parties : différents champs obligatoires ou non que l'utilisateur est convié à remplir, une carte permettant à l'utilisateur de spécifier l'adresse exacte de son logement, un outil permettant à l'utilisateur de dessiner le plan du sol (en cours d'élaboration) et la galerie où l'utilisateur peut ajouter images et vidéos pour accompagner son offre.

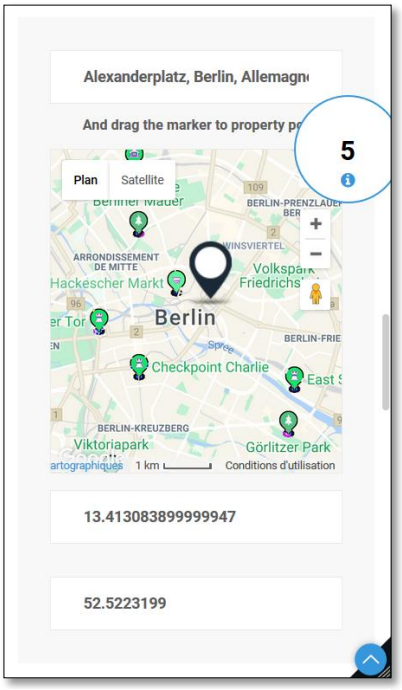

Figure 2.5: publier une offre: géolocalisation

Un système de points accompagne la publication des offres. Plus l'utilisateur remplit de champs, donne d'informations, télécharge d'images et de vidéos, plus il reçoit de points. Ces points vont faire que son offre sera mieux positionnée par rapport aux autres. Le but de ceci est d'encourager l'utilisateur à donner le maximum d'informations. Il s'agit pour l'instant d'un système de points rudimentaire qui doit être précisé quant aux valeurs des informations et de la visualisation du nombre de points acquis. Le nombre de points acquis est actuellement visible de façon brute mais une solution alternative avec des couleurs ou des messages d'encouragement selon le nombre de points est en réflexion. Ces solutions alternatives inciteraient moins à la triche que la solution brute. Ces points sont, pour l'instant, affichés dans la bulle située en haut à droite, voir figure 2.5.

Les différents champs tels que *disponible le, prix, référence* ainsi que les catégories telles que *Général* et *Environnement* sont tous enregistrés dans la base de données et sont donc très facilement modifiables.

Dans la partie carte, l'utilisateur saisit l'adresse du logement et peut ensuite déplacer le curseur si celui-ci n'est pas situé au bon endroit. La longitude et latitude sont calculés automatiquement.

Le *floorplan* est encore en cours d'élaboration. Une première version très simple et fonctionnelle a été développée mais a été mise de côté après la découverte d'une librairie *open-source* bien plus complète et dont le résultat est de très bonne qualité. L'intégration de cette librairie pose quelques problèmes d'affichage c'est pourquoi elle n'est pas encore en ligne.

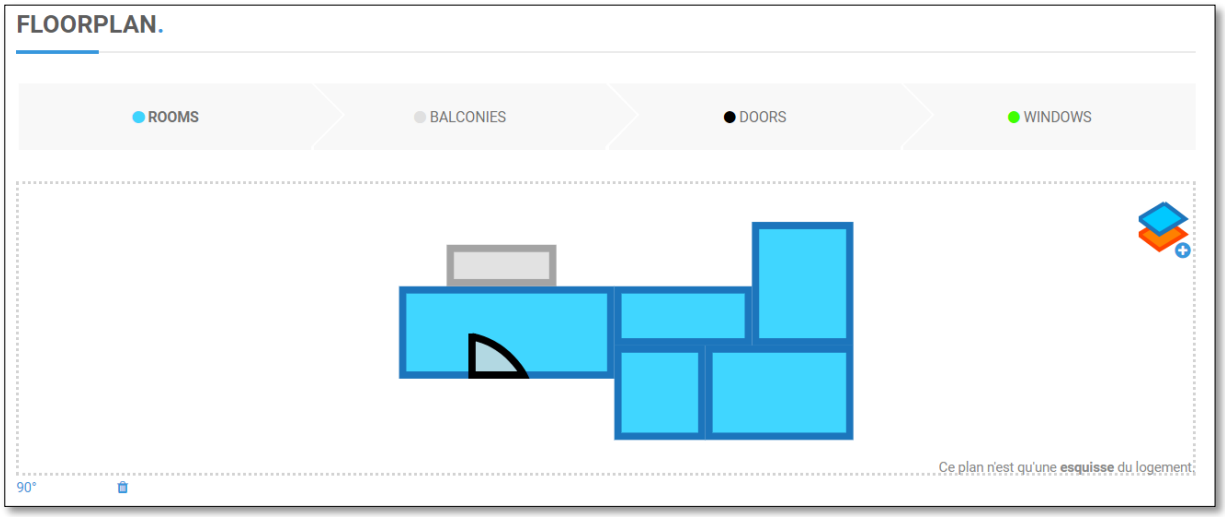

Figure 2.6: *floorplan*

La figure 2.6, ci-dessus, représente la première version du *floorplan*. Cette version est, pour l'instant, optimalisée pour un fonctionnement sur Google Chrome. L'utilisateur a la possibilité d'ajouter quatre éléments : des pièces, des balcons, des portes et des fenêtres. Ces éléments peuvent ensuite être déplacés, modelés (élargis, agrandis, rétrécis...), rotationnés et supprimés à la convenance. Il est également possible de dessiner le logement sur plusieurs étages et de naviger entre eux.

Avec la galerie, l'utilisateur peut très simplement ajouter vidéos et images à son offre.

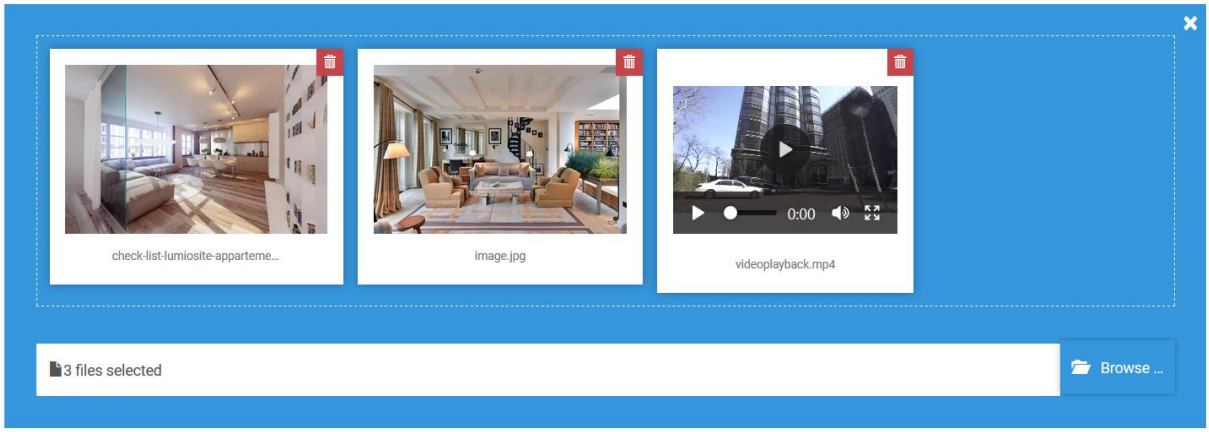

Figure 2.7: publier une offre : insérer des images

2.2.3 Mes offres

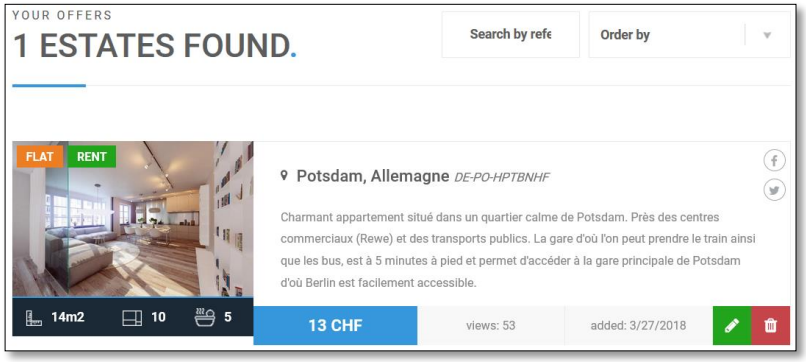

Figure 2.8: mes offres

Pour accéder à ses offres, un utilisateur connecté doit cliquer sur l'icône en forme de bonhomme situé dans l'en-tête. Ses offres sont affichées sous forme de liste qui peut être triée par date de création (par défaut), par nombre de vues ou par pertinence. Il est également possible de rechercher dans les offres par rapport au numéro de référence. Cette dernière fonctionnalité est principalement utile pour les agences.

Grâce aux boutons Facebook et Twitter, l'utilisateur peut très rapidement et facilement poster son annonce sur son propre compte Facebook ou Twitter.

Il est possible de modifier l'annonce. Ceci renvoie l'utilisateur sur la page de publication d'annonces mais, cette fois-ci, les champs sont préremplis. L'utilisateur peut également supprimer son annonce si le logement a trouvé preneur.

Dans cette même page, sur le côté gauche, un petit menu de navigation permet d'accéder directement aux différentes pages suivantes : *mes offres*, *mon profil* et *publier une annonce*.

2.2.4 Mon profil

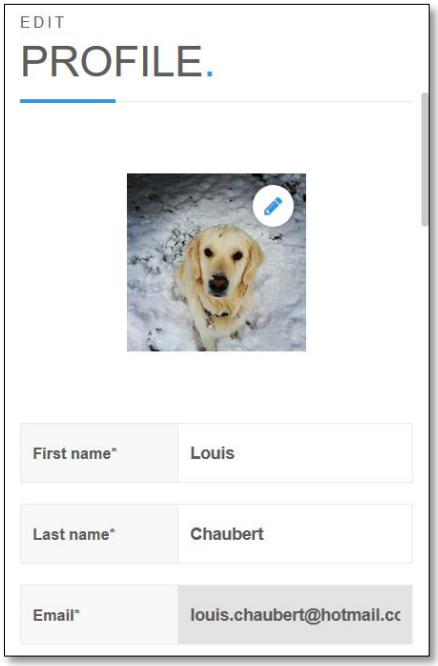

Figure 2.9: mon profil

L'accès à la page *mon profil* se fait, au travers du menu de navigation de la page *mes offres*, en cliquant sur le bouton *Mon profil*.

Depuis cette page, l'utilisateur peut mettre à jour ou compléter ses informations personnelles.

Si l'utilisateur s'est connecté avec Facebook ou Google, sa photo de profil sera automatiquement utilisée en tant qu'avatar. Mais il lui est bien évidemment possible de la changer.

Si l'utilisateur est connecté au moyen d'une adresse email et d'un mot de passe, il peut, depuis cette page, modifier son mot de passe.

Il est également possible de choisir de mettre son profil en mode publique ou privé. Dans le premier cas, plus d'informations seront affichées dans le formulaire de contact des offres (photos, nom, prénom) que si le profil est en mode privé (uniquement le formulaire sans informations personnelles).

3

MyFlat : point de vue développeurs

Autant la partie cliente que la partie serveur fonctionne grâce à des *frameworks* utilisant *Javascript ES6*[3] et toutes ses nouvelles fonctionnalités (*arrow functions, etc…*). La décision de choisir ces *frameworks* s'est faite selon un rapide pesage des intérêts et, surtout, dans l'idée d'en apprendre de nouveaux. Les grands avantages, autant de *ReactJS* que *MeteorJS*, sont qu'ils fonctionnent de manière asynchrone. Bien que cela implique une rigueur dans l'implémentation, du point de vue utilisateur, la rapidité des fonctionnalités est décuplée. *MongoDB*, la base de données *NoSQL*[4], offre également des performances améliorées (du moins en lecture) par rapport à une base de données *SQL*. Ceci en évitant des liens complexes entre les données.

La frontière entre le serveur et le client avec *MeteorJS* étant très relative, les thèmes seront abordés dans le chapitre dans lequel ils font le plus de sens même s'il ne s'agit pas, à proprement parlé, d'un élément du serveur ou du client (ces cas sont signalés).

3.1 Serveur : MeteorJS

MeteorJS est un *framework open-source* basé sur *NodeJS*. *MeteorJS* utilise le mécanisme du *Publish-subscribe*. Ce mécanisme rafraichit tous les clients qui sont abonnés à certaines

[3] *Javascript ES6* est la dernière version du langage *Javascript* et apporte son lot de nouveauté. Pour plus d'informations, voir http://es6-features.org.
[4] *NoSQL = Not Only SQL*

données lorsque celles-ci sont modifiées. Cela permet d'avoir des clients toujours à jour et, ceci, en temps réel. *MeteorJS* intègre par défaut les *frameworks* clients suivant : *Blaze*, *AngularJS* et *ReactJS*. Durant le développement, *MeteorJS* a aussi l'avantage de s'auto-compiler lors des modifications.

3.1.1 Installation

Selon le système d'exploitation, il suffit d'utiliser la commande appropriée :

OSX/LINUX :

```
curl https://install.meteor.com/ | sh
```

WINDOWS :

```
choco install meteor
```

3.1.2 Créer une application

Une fois *MeteorJS* installé, la variable d'environnement appelée *meteor* devrait être disponible.

```
meteor create MyFlat
```

L'architecture des dossiers et fichiers prend la forme suivante :

Base	MyFlat		Explication
client/main.js	client/main.js		Point d'entrée chargé par le client
client/main.html	client/main.html		Fichier *HTML* définissant les interfaces
client/main.css	-		Fichier *CSS* définissant les styles
-	client/compatibility	*dossier*	Dossier contenant les librairies
server/main.js	server/main.js		Point d'entrée chargé par le serveur
-	.deploy/mup.js		Fichier permettant le déploiement
-	.deploy/settings.json		Configurations (oAuth, smtp, …)
-	both/i18n	*dossier*	Dossier contenant les fichiers de traductions
-	imports/api	*dossier*	Dossier contenant les collections, leurs méthodes et publications
-	imports/startup	*dossier*	Dossier contenant les fichiers lancés aux démarrages du serveur/client
-	imports/ui	*dossier*	Dossier contenant les composants, conteneurs, *layouts*, et pages.

-	public	*dossier*	Dossier contenant les fichiers statiques associés au client (images, *CSS*, *JS*, …)
package.json	package.json		Fichier pour l'installation des paquets *NPM*
package-lock.json	package-lock.json		Fichier des dépendances *NPM*
.meteor	.meteor		Dossier contenant les fichiers internes à *MeteorJS*
.gitignore	.gitignore		Fichier de contrôle pour *GIT*

Tableau 3.2 : architecture d'un projet *MeteorJS*

3.1.3 Collections

Les collections sont les modèles de la base de données. La particularité de *MeteorJS* est que le client possède une copie réduite (selon ses besoins) de la base de données en local. Autant le client que le serveur peut donc accéder aux collections. Le client accède à la base de données cliente tandis que le serveur accède à la base de données réelle. La base de données cliente est automatiquement mise à jour grâce au système du *publish-subscribe* afin que le client ait toujours les données les plus actuelles. Ci-dessous, un exemple de la collection *offers* de MyFlat :

import/api/offers/offers.js

```
import { Mongo } from 'meteor/mongo';
import { Images } from './../images/images.js';
import SimpleSchema from 'simpl-schema';

export const Offers = new Mongo.Collection('offers');

Offers.schema = new SimpleSchema({
  estate: {type: Object, blackbox: true},
  transaction: {type: Object, blackbox: true},
  coordinates: [Number],
  description: {type: String, required:true},
  properties: {blackbox: true, type: Object},
  title: {type: String, required:true},
  online: {type: Boolean, defaultValue:true},
  views: {type: Number, defaultValue:0},
  createdAt: {type: Date, defaultValue: new Date()},
  owner: {type: String, required:true},
  point: {type: Number, defaultValue: 0}
});

Offers.attachSchema(Offers.schema)

Offers.helpers({
    image(){
      return Images.findOne({"meta.offer":this._id})
    }
});
```

Code 3.1 : *offers* collection

Uniquement deux lignes du code 3.1 sont réellement nécessaire à la création d'une collection. Il s'agit de l'importation de *Mongo* ainsi que l'exportation de la nouvelle collection *offers*. L'utilisation d'un *SimpleSchema* permet de forcer la collection à avoir une certaine forme. Sans cela, n'importe quel attribut pourrait être ajouté et aucun typage ne serait contrôlé. Les *helpers* permettent d'ajouter des méthodes aux collections.

Les appels aux collections se font à travers des méthodes et des publications. Globalement, les méthodes sont employées lors des ajouts, modifications ou suppressions d'enregistrement tandis que les publications permettent au client de s'abonner à une certaine portion de données afin d'afficher des données constamment à jour.

Méthodes

Chaque collection a son propre fichier avec ses propres méthodes. Ci-dessous, dans le code 3.2, les méthodes de la collection *offers* :

import/api/offers/methods.js

```
import { Meteor } from 'meteor/meteor';
import { Offers } from './offers.js'

Meteor.methods({
  'offers.addOffer'(offer) {
    Offers.insert(offer);
  },

  'offers.updateOffer'(offerId, offer) {
    Offers.update({"_id":offerId}, {$set: offer});
  },

  'offers.offlineOffer'(id){
    Offers.update({"_id" : id}, {$set: {"online":false}})
  },

  'offers.increaseViews'(id){
    Offers.update({"_id":id}, {$inc: {views:+1}})
  }

});
```

Code 3.2 : *offers* méthodes

Les méthodes appellent `update(...)`, `insert(...)` ou `delete(...)` de la collection qu'elles importent. La construction des requêtes vers la base de données est similaire à une requête *MongoDB* standard. Les mots-clés tels que `$set`, `$inc`, etc… peuvent être employés.

L'appel d'une méthode se fait de la manière suivante depuis une quelconque méthode du client:

```
Meteor.call('offers.increaseViews', offerId, (err, res) => {[...]})
```

Le premier argument de la fonction sert à diriger vers la méthode correspondante, le second transmet les données vers la méthode et, en tant que troisième argument, il est possible (facultatif) de récupérer le *callback* de la méthode.

Bien que les méthodes puissent très bien être utilisées afin de récupérer des données grâce à un `find(...)`, il est conseillé d'utiliser les publications afin d'utiliser l'avantage du *publish-subscribe*. Néanmoins, dans certains contextes, lorsque les données n'ont aucun intérêt à être à jour (voir des désavantages), il peut être intéressant d'utiliser une méthode pour lire les données.

Publications

Les publications sont utilisées pour ce qui est de l'affichage des données. Le code 3.3, ci-dessous, présente les publications associées à la collection *offers* :

import/api/offers/server/publications.js

```
import { Meteor } from 'meteor/meteor';
import { Counts } from 'meteor/tmeasday:publish-counts';
import { Offers } from './../offers.js'

const MAX_OFFERS = 150;

Meteor.publish('offers.allByXY', function (params, orderby, limit) {
  params["online"] = true
  return Offers.find(params, {sort: orderby, limit:Math.min(limit,
MAX_OFFERS)});
});

Meteor.publish('offers.allInArray', function (array) {
  return Offers.find({ _id : { $in : array }, online: true });
});

Meteor.publish('offers.one', function (id) {
  return Offers.find({_id:id, online: true});
});

Meteor.publish('offers.bounds', function (bounds) {
  return Offers.find({ "coordinates" : {
          "$geoWithin" : {
            "$box" : [
              [bounds.west, bounds.south],
              [bounds.east, bounds.north]
            ]
          }
        }, online:true });
});

Meteor.publish('countTransaction', function(transactionId){
  Counts.publish(this, 'count'+transactionId,
      Offers.find({"transaction._id" : transactionId, online:true}),
      {fastCount:true})
})
```

Code 3.3 : *offers* publications

À la différence des méthodes, les publications utilisent la méthode `find(...)` de la collection. Dans leurs constructions, les publications sont très similaires aux méthodes. La différence principale et que chaque publication est enregistrée grâce à un appel indépendant à la méthode `Meteor.publish(...)`.

Afin de s'abonner à une publication, le client utilise la méthode suivante :

```
Meteor.subscribe('offers.one', offerId);
```

Code 3.4 : *offers.one* abonnement

Cet abonnement aura pour conséquence de maintenir la base de données cliente à jour par rapport aux enregistrements et leurs attributs que l'abonnement prend en compte. Dans l'exemple de la publication *offers.one*, seul l'enregistrement dont l'*offer* correspondant à l'*offerId* et étant en ligne sera maintenu à jour et ceci, le temps de l'abonnement.

Puisque les données sont transmises au client de manière asynchrone et qu'elles peuvent recevoir une notification de mise à jour à tout moment, le client utilise un conteneur pour gérer ceci. Pour plus d'informations, voir le chapitre 3.2.2 sur les conteneurs.

3.2 Client : ReactJS

ReactJS est une librairie *Javascript* pour la création d'interface utilisateur. Elle est entretenue par Facebook et une communauté de développeurs. Il existe une variante, *React Native*, permettant le développement d'applications natives pour Android, iOS et UWP.

Si l'on souhaite utiliser *ReactJS* dans un projet *MeteorJS*, il faut en premier lieu installer son paquet. Uniquement *Blaze* est complètement intégré par défaut. Pour ce faire, il faut lancer la commande suivante à la racine du projet :

```
meteor npm install --save react react-dom
```

* --save permet d'ajouter la dépendance dans *package.json*.

Le point d'entrée de l'application se fait par le fichier *client/index.html* qui, dans MyFlat ressemble à ceci :

client/index.html

```
<head>[...]</head>
<body>
  <div class="loader-bg"></div>
  <div id="app" />
</body>
```

Code 3.5 : point entrée : index.html

L'élément important de ce fichier est la balise `<div />` et son `id="app"`. Ceux-ci sont utilisés dans le fichier *startup/client/index.jsx* pour y connecter tous les autres composants.

startup/client/index.jsx

```
import React from 'react';
import { render } from 'react-dom';
import { Meteor } from 'meteor/meteor';
import i18n from 'meteor/universe:i18n';

import [...]

import App from '../../ui/layouts/App.jsx';

Meteor.startup(() => {
    getLang = function() {
        return (
            sessionStorage.getItem("lang") ||
            navigator.languages && navigator.languages[0] ||
            navigator.language ||
            navigator.browserLanguage ||
            navigator.userLanguage ||
            'en'
        )
    };
    i18n.setLocale(this.getLang());
    render(<App />, document.getElementById('app'));
});
```

Code 3.6 : accrochage des composants : index.jsx

Ce fichier démarre l'application cliente grâce à l'appel de la méthode `Meteor.startup(...)` qui va inclure le composant *App* à l'intérieur du `<div id="app" />` précédemment défini. Le composant *App* est appelé *layout* car il englobe toutes les pages et autres composants de l'application. Mais, par définition, il s'agit tout de même d'un composant. C'est également à ce moment-là que l'on définit la langue dans laquelle l'application sera affichée. Ceci selon les cookies ou les paramètres du navigateur de l'utilisateur. Le paquet s'occupant du multilinguisme s'appelle *universe:i18n* et est particulièrement adapté à *ReactJS*.

3.2.1 Composants

Les composants s'imbriquent les uns dans les autres. Le composant au plus haut niveau est le *layout App* qui inclut les différentes pages. Les pages sont, elles, construites de différents composants. Il y a également une surcouche pour certaines pages qui est appelée « conteneur » ou *container* en anglais. Un conteneur s'occupe de collecter et, si nécessaire, retravailler les données dont la page a besoin. Les composants ont, pour la plupart, également un conteneur mais celui-ci est directement intégré.

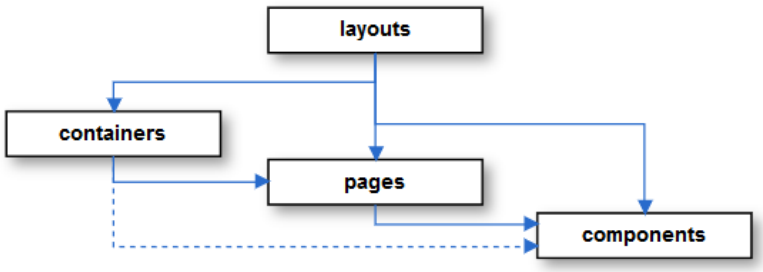

Figure 3.10: hiérarchie des composants

Les pages correspondent aux différentes vues utilisateurs telles que le détail d'une offre ou la liste des offres. Tandis que les composants sont des éléments visuels répétitifs dans ces différentes pages tels que le pied de page ou la barre de navigation.

Les composants *ReactJS* reçoivent deux paramètres lors de leurs créations : les *props* et les *states*. Les *props* sont toutes les données passées depuis le composant parent ou par le conteneur. Les *states* sont des variables qui, lorsqu'elles sont modifiées, forcent le composant à se rafraichir. Un exemple de composant est le code 3.7, ci-dessous :

import/ui/components/offers/InfoboxOffer.jsx

```
import React from 'react';
import [...]

export class infoboxOffer extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {

   return (
     <div>
      <a className='infobox-main' href={"estate?id="+this.props.offer._id}>
        <div className='infobox-image'>
           <img src={ this.props.image ?
                   this.props.image.link() :
```

```
                    'images/infobox-offer5.jpg'} alt={this.props.offer.title}
            />
          </div>
          <div className='infobox-text'>{this.props.offer.title}</div>
          <div className='infobox-price'>
             {
              this.props.offer.properties.price.value+
              " "+this.props.offer.properties.price.unit
             }
          </div>
       </a>
      </div>
   );
  }
}

export default withTracker((props) => {
  [...]
})(infoboxOffer);
```

Code 3.7 : *InfoboxOffer* composant

Il existe deux façons de déclarer un composant. Celle ci-dessus est la plus conventionnelle et permet d'accéder aux différentes méthodes facultatives. La seule méthode que cette implémentation est forcée d'avoir est `render(...)`. Cette dernière méthode retourne le code *HTML* du composant. Les *props* et *states* sont accessibles de la manière suivante : `this.props` et `this.states`. Seules les *states* sont modifiables une fois le composant créé. Pour modifier un *state*, il faut le faire de la manière suivante :

```
this.setState({myState : myNewValue})
```

Les composants ont certaines méthodes facultatives pouvant être utilisées à différentes fins. Il est également possible de définir et d'appeler ses propres méthodes au sein d'un composant.

Les méthodes par défaut sont les suivantes :

1) `constructor(props)` :
   Il s'agit du constructeur de la classe. Ceci signifie que cette méthode sera la première à être appelée lors de l'instanciation de l'objet. Si l'on souhaite assigner une valeur par défaut au *state*, il ne faut pas utiliser la méthode `setState(...)` mais lui assigner la valeur directement de la façon suivante : `this.state = { ... : ... } ;`

2) `componentDidMount()` :
   Cette méthode est appelée directement après que le composant soit monté.

3) `componentDidUpdate(prevProps, prevState)` :
   Celle-ci est appelée après qu'une actualisation du composant a eu lieu. Par exemple, si le *state* est modifié ou si des données d'un abonnement doivent être mises à jour.

4) `shouldComponentUpdate(nextProps, nextState)` :
   Cette méthode est appelée avant que le composant soit actualisé afin de savoir si l'actualisation a vraiment lieu d'être ou pas. Elle peut être utile afin d'éviter des actualisations inutiles.

5) `componentWillReceiveProps(nextProps)` :
   Cette méthode est appelée avant qu'un composant monté reçoive des nouveaux *props*. Ceci peut arriver lorsque le conteneur reçoit de nouvelles données d'un abonnement et qu'il les transmet au composant.

6) `componentWillUpdate(nextProps, nextState)` :
Semblable à `componentDidUpdate(...)` mais est exécuté avant que le composant soit actualisé.

Un composant déclaré selon la seconde façon s'appelle un *stateless functional component* (*SFC*) et prend la forme suivante :

import/ui/components/Alert.jsx

```
import React, { PropTypes } from 'react';

const Alert = ({type, title, text})  => (
    <div className={"alert alert-"+type+" alert-dismissible show"}
role="alert">
      <button type="button" className="close" data-dismiss="alert">
        <span aria-hidden="true">&times;</span>
      </button>
      <strong>{title}</strong> {text}
    </div>
);

export default Alert;
```

Code 3.8 : *Alert* SFC composant

Le code *HTML* peut être parsemé de données dynamiques grâce aux accolades `{...}`. Entre les accolades, il est autant possible d'appeler des variables que des méthodes et même d'utiliser des conditions et des boucles. Ci-dessous, dans le code 3.9, un exemple tiré du composant *FeaturedCarousel* :

```
<div className="featured-offers-container">
  <div className="owl-carousel" id="featured-offers-owl">
    {
     this.props.offers.map((offer) => {
        return (<div className="featured-offer-col" key={offer._id}>
                    <FeaturedOffer offer={offer}
                     isScriptLoaded={this.props.isScriptLoaded}
                     isScriptLoadSucceed={this.props.isScriptLoadSucceed}/>
                </div>);
     })
    }
  </div>
</div>
```

Code 3.9 : *data binding*

Dans l'exemple ci-dessus, on crée un nouveau composant *FeaturedOffer* pour chacune des offres de `this.props.offers`. Les attributs tels que `offer={offer}` et `isScriptLoaded={this.props.isScriptLoaded}` permettent de transmettre des *props* vers le composant enfant.

Les boucles sont généralement effectuées grâce à la méthode `map((...) => {return ...})`. Les conditions, elles, sont généralement écrites de la manière suivante (bien que le `if` existe) :

```
{condition ? executedIfTheConditionIsTrue : executedIfTheConditionIsFalse}
```

Une astuce à propos de l'export : en utilisant le mot-clé *default*, lorsque le fichier sera importé ailleurs, l'objet spécifié par défaut sera importé, sauf spécification supplémentaire au moyen des accolades. Exemple des différentes exportations :

```
export const uneConstante = ... ;
export default const uneAutreConstant = ... ;
```

Et exemple d'importation :

```
import Constant from ... ; // retourne uneAutreConstante
import { uneConstante } from ... ; // retourne uneConstante
```

3.2.2 Conteneurs

Le conteneur peut être vu comme une surcouche du composant. Il est utilisé afin de faire les abonnements aux données nécessaires au composant ainsi que de les traiter si besoin. Le conteneur est attentif aux données auxquelles il est abonné afin que, si une de ces données est modifiée, il puisse mettre à jour le composant.

Par convention et puisque les conteneurs des pages ont tendance à être plus conséquentes que ceux des composants, ceux-ci ont leurs propres fichiers. Tandis que les conteneurs des simples composants sont directement intégrés dans le même fichier. Ci-dessous, dans le code 3.10, un exemple de conteneur intégré au même fichier que le composant :

import/ui/components/offers/GridOffer.jsx

```
import { withTracker } from 'meteor/react-meteor-data';
import { Images } from '../../../api/images/images.js'
import [...]

export class GridOffer extends React.Component {
    [...]
}

export default withTracker((props) => {
    const subOfferImage = Meteor.subscribe('images.offerOne',
props.offer._id);

    const loadingImage = !subOfferImage.ready();

    let image = Images.findOne({"meta.offer":props.offer._id})

    return {
      loadingImage, image
    };
})(GridOffer);
```

Code 3.10 : *GridOffer* conteneur

Le conteneur est défini grâce au composant `withTracker` du paquet `meteor/react-meteor-data` et est exporté par défaut.

L'abonnement et la transmission des données du conteneur vers le composant se fait en quatre étapes :

1) L'abonnement à la publication grâce à `Meteor.subscribe(...)`
2) La création d'une variable informant sur l'état de chargement des données (facultatif)
3) La recherche des données d'intérêt dans la base de données cliente
4) Le retour des variables nécessaires

Et lorsqu'un composant souhaite intégrer le composant *GridOffer*, il lui suffit de faire l'importation de la manière suivante :

```
import GridOffer from '../components/offers/GridOffer.jsx';
```

À noter que l'on importe sans les accolades afin de faire appel au conteneur qui, lui, s'occupe d'appeler le composant *GridOffer*.

Les conteneurs ayant leurs propres fichiers sont situés dans le dossier *import/ui/containers/...* et, lorsque l'on souhaite importer une page (composant ayant le conteneur séparé), il faut importer son conteneur plutôt que directement le composant.

3.3 Base de données : MongoDB

MongoDB est une base de données *NoSQL*. La différence principale avec les bases de données *SQL* est que *MongoDB* essaie de minimiser les liens entre les données afin d'accroitre la rapidité des recherches. La contrepartie à payer est une redondance dans les données et que, par conséquence, lors de modifications ou suppressions, il faut veiller à la cohérence des données.

Une installation indépendante de *MongoDB* n'est pas nécessaire lorsque l'on installe *MeteorJS*, ce dernier s'occupe de l'installer.

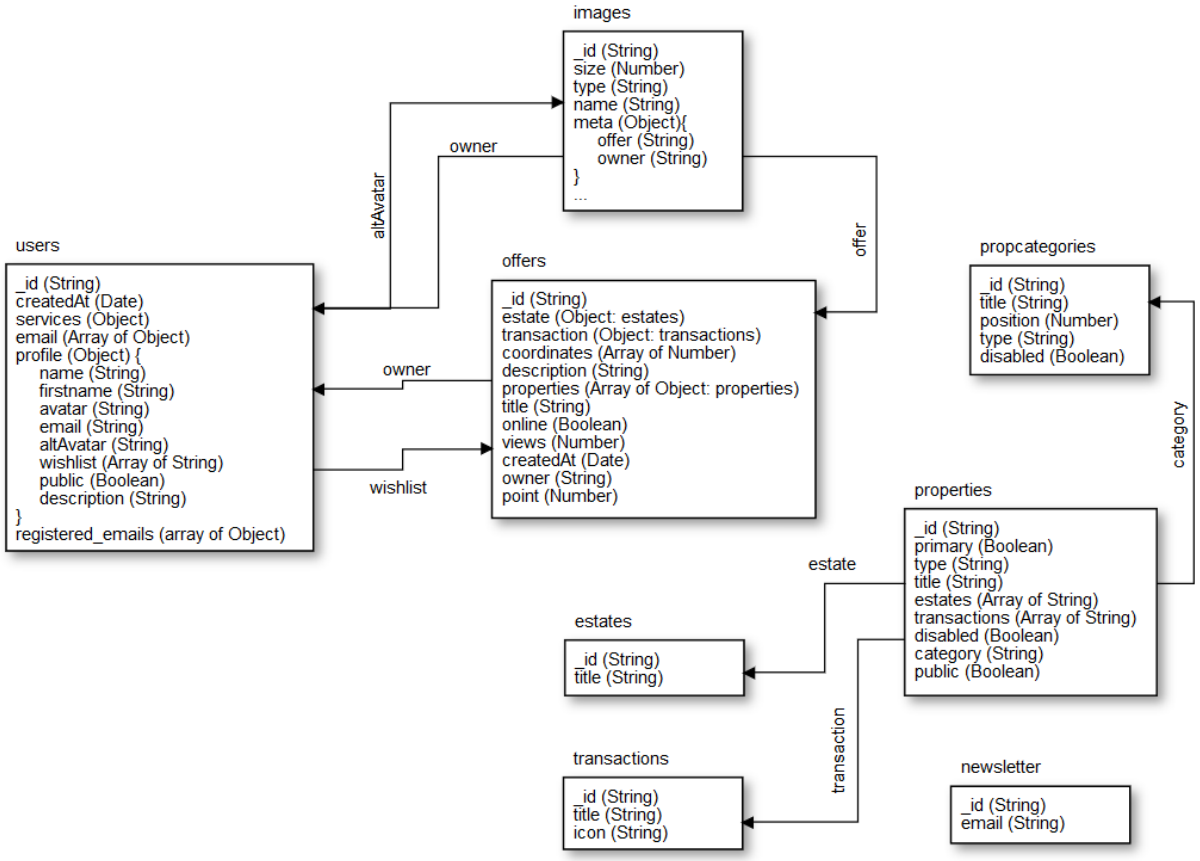

Figure 3.11: base de données de MyFlat

La Figure 3.11: base de données de MyFlat représente la dernière version schématique de la base de données de MyFlat. *MongoDB* ne s'occupe pas lui-même du typage et des attributs des enregistrements. Ce contrôle est fait grâce aux *SimpleSchema* des collections de *MeteorJS* (cf. chapitre 3.3.2).

Comme on peut le voir dans la Figure 3.11: base de données de MyFlat certains attributs sont de type *(Object : ...)*, ceci signifie qu'ils ont une copie d'un enregistrement d'une autre collection. Par exemple, dans le cas de l'attribut *properties (Array of Object : properties)* de la

collection *offers*, il s'agit d'une liste d'enregistrements copiés de la collection *properties*. D'autres attributs ont simplement le type *(Object)*, ceci signifie que cet attribut est simplement un objet *Javascript*, autrement dit un dictionnaire. La suppression totale de lien entre données n'est malheureusement pas possible, certains attributs contiennent donc l'identifiant unique d'un enregistrement d'une autre collection. Un tel cas peut être observé dans la collection *offers* où un enregistrement possède l'attribut *owner* correspondant à l'identifiant unique d'un utilisateur de la collection *users*.

3.3.1 CRUD

La forme des requêtes *MongoDB* est la même que celle utilisée par *MeteorJS* et ses collections. Connaître comment écrire une requête avec *MongoDB* suffit donc à écrire les requêtes du serveur.

Pour effectuer directement des requêtes sur la base de données utilisées par *MeteorJS*, il faut se rendre dans le répertoire du projet et exécuter la commande suivante :

```
meteor mongo
```

Puisque la collection *newsletter* est la plus simple, celle-ci sera utilisée en tant qu'exemple pour les ajouts, modifications et suppressions. Mais puisque *MongoDB* n'effectue aucun contrôle lui-même, on pourrait très bien utiliser n'importe quelle collection et y insérer des enregistrements ne respectant pas les modèles du serveur.

Ajouter

Requête depuis *MongoDB* :

```
db.newsletter.insert({email : 'unEmail@hebergeur.com'})
```

Requête depuis *MeteorJS* :

```
import { Newsletter } from 'import/api/newsletter/newsletter.js'
Newsletter.insert({'email' : 'unEmail@hebergeur.com'})
```

Cette requête ajoute un enregistrement dans la collection *Newsletter* dont l'attribut *email* a une valeur de « *unEmail@hebergeur.com* ».

Modifier

Requête depuis *MongoDB* :

```
db.newsletter.update({email : 'unEmail@hebergeur.com'},
                     {'$set' : {'email' : 'unNouvelEmail@hebergeur.com'}})
```

Requête depuis *MeteorJS* :

```
import { Newsletter } from 'import/api/newsletter/newsletter.js'
Newsletter.update({'email' : 'unEmail@hebergeur.com'},
                  {'$set' : {'email' : 'unNouvelEmail@hebergeur.com'}})
```

Cette requête modifie l'attribut *email* par la valeur « *unNouvelEmail@hebergeur.com* » de tous les enregistrements de la collection *Newsletter* dont l'attribut *email* a la valeur « *unEmail@hebergeur.com* ».

Supprimer

Requête depuis *MongoDB* :

```
db.newsletter.delete({email : 'unEmail@hebergeur.com'})
```

Requête depuis *MeteorJS* :

```
import { Newsletter } from 'import/api/newsletter/newsletter.js'
Newsletter.delete({'email' : 'unNouvelEmail@hebergeur.com'})
```

Cette requête supprime tous les enregistrements de la collection Newsletter dont l'attribut email a une valeur correspondante à « *unNouvelEmail@hebergeur.com* ».

3.3.2 Collections

En regardant le schéma de la Figure 3.11: base de données de MyFlat, on s'aperçoit facilement que les collections sont séparées en deux grandes catégories : celle des offres et utilisateurs et celle des propriétés et catégories. Il y a également la collection *newsletter* qui est laissée à part car elle ne contient qu'une liste d'adresse email.

Offres et utilisateurs

Cet ensemble de collection est constitué des collections *users*, *images* et *offers*.

La collection ***users*** contient toutes les données concernant les utilisateurs. Ceci comprend les informations personnelles et d'authentification grâce aux services auxiliaires tels que Google ou Facebook. Ces enregistrements peuvent (facultatif) contenir un avatar alternatif impliquant un lien vers la collection *images* qui contient les informations sur l'avatar. Ils ont également la possibilité d'avoir une *wishlist* contenant une liste d'offres pour lesquels ils sont intéressés. Ceci crée donc une dépendance vers la collection *offers*.

La collection ***images*** est générée automatiquement grâce au paquet `FilesCollection` de `meteor/ostrio:files` utilisée pour la gestion des fichiers à travers le projet. Celle-ci contient tout une série d'attributs par défaut des fichiers téléchargés dont: le nom, la taille et, surtout, le lien symbolique vers la ressource. Mais elle possède également l'attribut *meta* qui contient un lien vers la collection *users* afin de pouvoir identifier l'utilisateur ayant ajouté le fichier et un lien vers la collection *offers* dans le cas où il s'agit d'un fichier lié à une offre.

La collection ***offers*** est la collection la plus importante du projet. Elle contient toutes les offres publiées sur MyFlat. Celle-ci se doit donc d'être optimisée et de n'avoir que le strict minimum de dépendances à d'autres collections. Une offre contient l'attribut *estate* qui est une copie d'un enregistrement de la collection *estates* selon à quelle catégorie de logements à laquelle elle appartient. Similairement pour l'attribut *transaction* et le type de transaction de la collection *transactions* à laquelle l'offre appartient. L'attribut *properties* contient toute une liste d'enregistrements copiés de la collection *properties* dont la valeur de la *property* pour l'offre est ajoutée. Une *property* est, par exemple, la surface ou le prix. Cette collection contient malgré tout un lien vers la collection *users* avec l'attribut *owner* correspondant au créateur de l'offre.

Propriétés et catégories

Cet ensemble de collections est constitué des collections *estates, transactions, properties et propcategories*. La particularité de ces collections est qu'elles permettent la création d'offres dynamiques car les propriétés de celles-ci peuvent être modifiées à tout moment. Par exemple, si une certaine propriété telle que la surface habitable devait être manquante, il suffirait de l'ajouter à la collection *properties* et celle-ci serait disponible pour toutes les nouvelles publications d'offre.

La collection ***estates*** contient toutes les différentes catégories de logements qui peuvent être publiées et recherchées. Ces types sont, par exemple, les appartements ou les maisons. Et,

puisque cette information figure dans la base de données, on peut facilement ajouter une nouvelle catégorie telle que les surfaces administratives.

La collection ***transactions*** est similaire à *estates* mais contient les différentes transactions proposées telles que la location ou la vente.

La collection ***properties*** contient toutes les différentes propriétés que les offres peuvent utiliser pour être décrites. Puisque ces informations figurent dans la base de données, la description des offres peut être facilement améliorée à travers le temps en ajoutant, modifiant ou supprimant des propriétés. Chaque propriété peut avoir une liste de liens vers des enregistrements de la collection *estates* et *transactions* par le biais des attributs *estates* et *transactions*. Si l'une de ces listes n'existe pas, cela signifie que la propriété est disponible pour, respectivement, n'importe quelle catégorie de logement ou type de transactions. Si les listes des attributs *estates* ou *transactions* ne sont pas vides, la propriété correspondante sera uniquement disponible pour, respectivement, les catégories de logement ou types de transaction se trouvant dans ces listes. Cette collection contient également un lien vers la collection *propcategories* par l'attribut *category*.

La collection ***propcategories*** représente les différentes catégories de propriétés telles que celles liées aux dimensions ou à l'environnement.

4

Systèmes de recommandations

Le Web a fortement simplifié la façon dont un utilisateur peut donner son avis. Si l'on prend l'exemple de Netflix, l'utilisateur peut, en un clic, spécifier sur 5 étoiles s'il a apprécié ou non la prestation. Il existe d'autres formes d'avis, telles que la procédure d'achat ou la simple recherche, qui peuvent être interprétés comme un intérêt pour le produit ou le service.

L'idée de base du système de recommandation est d'utiliser ces différentes sources d'informations pour en déduire les intérêts des consommateurs. Les termes *utilisateurs* et *items* seront utilisés dans la suite du document pour spécifier, respectivement, les utilisateurs et les produits, services ou prestations. Le principe de base est qu'il existe des interdépendances entre les utilisateurs et les *items*. Par exemple, un utilisateur s'intéressant à un documentaire sur la Grèce antique aura plus de chance d'être intéressé par un documentaire historique ou un programme éducationnel que par un film d'action. Plus le nombre d'évaluations effectuées par un utilisateur est grand, plus il sera facile de faire une prédiction pertinente sur le comportement futur de ce même utilisateur.

L'augmentation des ventes est le but premier d'un système de recommandation. En recommandant aux utilisateurs des *items* soigneusement sélectionnés, le système de recommandation attire l'attention de l'utilisateur sur des *items* pouvant potentiellement l'intéresser. Il va donc être susceptible de consommer et, par conséquence, d'augmenter les ventes et le profit du commerçant. Le gain du système de recommandation n'est cependant pas toujours aussi évident que dans le cas d'Amazon. Parfois, il s'agit simplement d'améliorer l'expérience utilisateur afin qu'il revienne consommer au même endroit.

Les différents buts opérationnels et techniques des systèmes de recommandations sont les suivants :

La pertinence : le système se doit de recommander des *items* qui sont pertinents pour l'utilisateur. Les utilisateurs sont plus enclins à consommer un produit qui les intéresse. La pertinence est le but premier d'un système de recommandation.

Nouveauté : le système est réellement utile lorsqu'il propose un *item* dont l'utilisateur n'avait pas encore connaissance.

Heureux hasards : cette notion est liée à la précédente. Les *items* proposés peuvent parfois s'avérer inattendu pour l'utilisateur et ainsi créer un élément de surprise fructueux.

Diversité : si le système de recommandation ne propose qu'un *top-k* d'*items* semblables, il est probable que l'utilisateur ne soit intéressé par aucun d'entre eux. Mais si le *top-k* est varié, il est plus probable que l'utilisateur en aime au moins un. La diversité a l'atout de s'assurer que l'utilisateur ne se lasse pas de propositions toutes similaires.

Approche	**But du concept**	**Input**
Collaborative	Donne des recommandations basées sur une approche collaborative qui met en évidence les classements et actions faites par des pairs ou soi-même	Classements de l'utilisateur et classements de la communauté
Basé sur le contenu	Donne des recommandations basées sur le contenu (attributs) d'*items* que l'utilisateur a favorisé par ses classements ou actions passées	Classement de l'utilisateur et attributs des *items*.
Basé sur la connaissance	Donne des recommandations basées sur une spécification explicite du type de contenu (attributs) que l'utilisateur souhaite	Spécification de l'utilisateur, attributs des *items* et domaine de connaissance

Tableau 4.3 : comparaison des systèmes de recommandation

4.1 Systèmes de recommandation collaboratifs

Les systèmes collaboratifs utilisent la puissance collaborative des classements (*ratings*) fournis par de nombreux autres utilisateurs afin de produire des recommandations ciblées. Un de ses principal problème est que les matrices de classements sous-jacente $R_{m \times n}(m = users\ et\ n =$

items) sont souvent maigres. Par exemple, dans le cas de Netflix, les utilisateurs n'ont visionné qu'une petite portion de l'ensemble disponible. La plupart des évaluations sont donc manquantes. Par conséquence, on discerne les classements observés et ceux non-observés.

L'idée de base des systèmes collaboratifs est que les classements non-observés peuvent être déduit des classements observés grâce aux corrélations entre les utilisateurs et entre les *items*. Il existe deux méthodes de collaborations communément utilisées dans ce type de système : celles basées sur la mémoire, *memory-based methods*, et celles basées sur le modèle, *model-based method*.

1. **Méthodes basées sur la mémoire :** également connu sous le nom de *neighborhood-based collaborative filtering algorithms*. Les voisinages, ou *neighbourhoods*, peuvent être soit basés sur l'utilisateur soit sur l'*item*.
    a. Dans le premier cas, celui des systèmes collaboratifs basés sur l'utilisateur, on utilise les classements d'un utilisateur *Y*, similaire à *X* afin de faire des recommandations à *X*. L'idée est de déterminer des similitudes entre utilisateurs afin de pouvoir partager les évaluations entre les utilisateurs pour combler les vides.
    b. Dans le cas des filtres collaboratifs basés sur les *items*, la première étape est de déterminer un ensemble d'*items* similaires à un certain *item B* que l'utilisateur *X* apprécie. Pour déterminer cet ensemble, on peut s'appuyer sur les classements effectués par *Y* sur des *items* aux caractéristiques semblables à *B*.
2. **Méthodes basées sur les modèles :** cette méthode utilise les technologies du *machine learning* dans le contexte des modèles de prédiction.

La distinction entre les deux modèles peut toutefois être difficile. Il a été prouvé qu'une combinaison des deux types de modèle donne de très bon résultats [Kor08].

4.1.1 Méthodes basées sur la mémoire

« When one neighbor helps another, we strengthen our communities. »

- Jennifer Pahlka

Cette méthode est également connue sous le nom de système collaboratif basé sur le voisinage. Ces algorithmes se basent sur le fait que des utilisateurs similaires partagent des classements similaires et que des *items* similaires reçoivent des classements similaires. Il existe deux grands types d'algorithmes basés sur le voisinage.

Le premier type se base sur l'utilisateur. Les classements d'utilisateurs similaires à *A* sont utilisés pour lui faire des recommandations. Le second type se base sur l'*item*. Un ensemble d'*item* similaire à *B* est déterminé afin de déterminer, par le biais de cet ensemble, quel serait le classement de l'utilisateur *A* pour l'*item B*. La distinction entre les deux cas provient de la source utilisée afin d'établir la prédiction.

Dans la suite du rapport, la matrice incomplète de classement utilisateur/*item* $R_{m \times n} = [r_{uj}]$ contenant m utilisateurs et n *items* sera employée afin de faciliter les explications. Uniquement un sous-ensemble de classement est observé par utilisateur sur les différents *items*. Les algorithmes collaboratifs peuvent être classés en deux catégories. D'un côté, il y a ceux permettant de prédire un classement manquant r_{uj} d'un utilisateur u pour un item j. Et de l'autre

côté, ceux déterminant un *top-k* d'*items* ou d'utilisateurs. Dans la plupart des cas, il est plus intéressant de prédire un *top-k* plutôt que la valeur spécifique à une combinaison utilisateur/*item*. Ces deux catégories sont tout de même fortement liées. Par exemple, afin de déterminer les *top-k* d'*items* pour un utilisateur en particulier, on peut simplement prédire le classement de l'utilisateur pour tous les *items* disponibles et sélectionner ceux avec le classement le plus positif.

Les propriétés clés des matrices de classements

Comme expliqué précédemment, uniquement un petit sous-ensemble des classements de R sont observés. Les entrées observées sont référées en tant que données d'entrainement tandis que celles non-observées sont les données de test.

Il existe plusieurs manières d'échelonner les classements. L'échelonnage le plus commun est celui basé sur un intervalle, *interval-based rating*, qui prend, par exemple, un intervalle à 5 points sous la forme suivante $\{-2, -1, 0, 1, 2\}$ ou $\{1, 2, 3, 4, 5\}$ représentant l'appréciation de l'utilisateur pour un certain *item*. La taille de l'intervalle et sa composition peuvent varier. Les tailles les plus communes sont à 5, 7 et 10 points. Lorsque le nombre de point est paire et que, par conséquent, il n'y a pas de vote neutre possible, on appelle ce type de système un système d'évaluation à choix forcé, *forced choice rating system*. Ces intervalles sont généralement représentés par une approche visuelle telle que des étoiles. Un autre type similaire est celui des classements continus qui spécifie un classement sur une échelle continue (p. ex. : entre -10 et 10). Il existe également les classements binaires où l'utilisateur exprime simplement un classement positif ou négatif. Un autre cas spécial est celui des classements unaires, *unary ratings*, où il est uniquement possible d'enregistrer une évaluation positive mais pas négative. Un exemple de classement unaire est celui où les évaluations sont déduites des actions de l'utilisateur, lors d'un achat par exemple, et les évaluations sont donc données implicitement et non pas explicitement comme dans les cas précédents.

La distribution des classements entre les *items* satisfait souvent la propriété économique appelée *long-tail property* et signifie qu'uniquement une fraction des *items* vont être fréquemment classés. Ces *items* sont appelés des *items* populaires. Par conséquent, la majorité des *items* ne va être que rarement classée par les utilisateurs. Les résultats ont donc tendance à être fortement biaisés en direction de ces classements. Une telle distribution a une grande implication dans le processus de recommandation. Dans beaucoup de cas, les *items* avec une fréquence élevée sont généralement des *items* ayant beaucoup de compétition (d'offre) sur le marché et donc un profit plus bas pour le vendeur. D'un autre côté, les *items* avec une fréquence basse offrent de plus forte marge. Dans ce cas, il est avantageux pour le vendeur de recommander en premier lieu des *items* ayant une fréquence basse. Dans les faits, beaucoup de compagnies, telles que Amazon, font la majorité de leur profit en vendant des *items* se trouvant dans la *long tail* (Anderson C., 2006). À cause de la rareté des classements pour les *items* se trouvant dans la *long tail*, il est souvent compliqué de fournir des prédictions robustes pour ceux-ci. Cela a également un impact négatif sur la diversité et, par conséquent, les utilisateurs peuvent s'ennuyer de se voir proposer toujours les mêmes articles ou services. Cette distribution implique que les *items* qui sont fréquemment

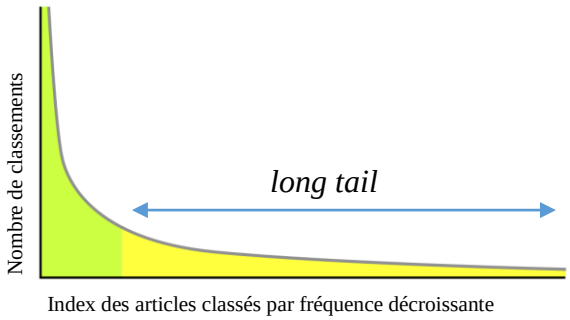

Figure 4.12: *long-tail property*

(source: wikipedia.org)

classés par les utilisateurs sont moins nombreux. Dans le cas des systèmes de recommandation basés sur le voisinage, cette propriété a un impact important car, les voisinages sont généralement définis par le biais de ces *items*. Dans beaucoup de cas, les classements de ces *items* à haute fréquence ne sont pas représentatifs des classements de ceux à basse fréquence, ceci à cause des différences inhérentes à ces deux classes d'*items*. Le processus de prédiction peut, par conséquent, mener à des résultats trompeurs. En ajustant l'algorithme de recommandation afin qu'il prenne en compte les propriétés du monde réel, telles que la *long tail*, peut conduire à des prédictions bien plus pertinentes.

Prédire les classements avec les méthodes basées sur le voisinage

L'idée de voisinage est d'utiliser des similarités, soit entre utilisateur, soit entre *items*, afin d'en faire des recommandations. Il existe deux principaux modèles basés sur le voisinage : celui basé sur l'utilisateur et celui basé sur l'*item*.

Puisqu'un problème collaboratif peut être vu comme une généralisation d'un modèle de régression ou de classification, les méthodes basés sur le voisinage peuvent être interprétées comme une généralisation d'un classificateur (*classifier*[5]) du plus proche voisin (*nearest neighbor classifier*).

Item-Id	**1**	**2**	**3**	**4**	**5**	**6**	$\boldsymbol{Sim(u,v)}$	
User-Id							$\boldsymbol{Cosinus(i,3)}$	$\boldsymbol{Pearson(i,3)}$
1	7	6	7	4	5	4	0,956	0,894
2	6	7	?	4	3	4	0,981	0,939
3	?	3	3	1	1	?	1,0	1,0
4	1	2	2	3	3	4	0,789	-1,0
5	1	?	1	2	3	3	0,645	-0,817

Tableau 4.4 : exemple de fonctions de similarité (cosinus & Pearson)

Le tableau 4.4 ci-dessus, représente les classements de 5 utilisateurs pour 6 items. Les points d'interrogation symbolisent le fait que l'utilisateur n'a pas évalué ledit item. Les deux colonnes de droites représentent les fonctions de similarité du cosinus et de Pearson entre l'utilisateur *i* de la ligne et l'utilisateur 3. La similarité entre l'utilisateur 3 et lui-même offre donc la valeur maximale de 1. Un exemple du calcul de similarité selon Pearson et des valeurs manquantes de l'utilisateur 3 figure dans le chapitre suivant.

Modèles de voisinage basés sur l'utilisateur

Ces modèles déterminent la similarité des utilisateurs par rapport à leur similarité dans leurs classements d'*items*. Si l'utilisateur *A* et *B* ont classé des *items* de façons similaires, le système va pouvoir utiliser les classements de l'un afin de prédire les classements non-observés de l'autre. L'utilisateur pour lequel on souhaite déterminer les classements d'*items* non-observés est appelé l'utilisateur cible (*target user*). Afin de déterminer le voisinage de l'utilisateur cible, on doit calculer sa similarité avec tous les autres utilisateurs. Il faut donc définir une fonction

[5] Un *classifier* est un modèle classant une entrée dans une liste finie de sortie.

de similarité entre les classements spécifiés par les utilisateurs. Le calcul d'une telle fonction peut s'avérer délicat car les utilisateurs peuvent, par exemple, ne pas avoir la même interprétation d'une échelle de grandeur (l'un peu être biaisé vers le haut tandis que l'autre vers le bas). De plus, tous les utilisateurs n'ont pas donné leurs avis sur les mêmes *items*. Le système doit donc être capable de gérer ces différents problèmes.

Pour une matrice de classement $R_{m \times n} = [r_{uj}]$ avec *m* utilisateurs et *n items*, I_u correspond à l'ensemble des index des *items* qui a été classé par l'utilisateur *u*. Par exemple, si l'utilisateur a donné son avis pour le premier, troisième et cinquième *item*, $I_u = \{1, 3, 5\}$. L'ensemble commun d'*items* classés par les utilisateurs *u* et *v* est défini par $I_u \cap I_v$. En reprenant l'exemple précédent, si l'utilisateur *v* a classé les quatre premiers *items*, $I_v = \{1, 2, 3, 4\}$ et donc $I_u \cap I_v = \{1, 3\}$. Il est possible, et même courant, que ce dernier ensemble soit vide car les classements des utilisateurs sont souvent dispersés. Cet ensemble représente les *items* mutuellement classés et va servir de base pour la fonction de similarité entre les deux utilisateurs.

Un exemple de mesure de similarité est le coefficient de corrélation de Pearson :

1) La première étape est de calculer le classement moyen μ_u pour chaque utilisateur *u* :

$$\mu_u = \frac{\sum_{k \in I_u} r_{uk}}{|I_u|}, \forall u \in \{1, \dots, m\}$$

2) Puis, le coefficient de la corrélation de Pearson entre les utilisateurs :

$$Sim(u, v) = Pearson(u, v) = \frac{\sum_{k \in I_u \cap I_v} (r_{uk} - \mu_u)(r_{vk} - \mu_v)}{\sqrt{\sum_{k \in I_u \cap I_v} (r_{uk} - \mu_u)^2} \sqrt{\sum_{k \in I_u \cap I_v} (r_{vk} - \mu_v)^2}}$$

Le coefficient de Pearson est calculé entre l'utilisateur cible et chacun des autres utilisateurs. On pourrait simplement sélectionner les utilisateurs ayant le coefficient le plus élevé afin de prédire les classements non-observés de l'utilisateur cible. Cependant, puisque les classements observés chez ce *top-k* d'utilisateur peuvent varier fortement par rapport à l'*item* que l'on souhaite classer, il faut déterminer un *top-k* d'utilisateur pour chaque *item*. On peut ensuite prendre la valeur de classement moyen du *top-k* afin de l'utiliser comme valeur du classement non-observé de l'utilisateur cible pour ledit *item*.

Un problème majeur de cette approche est que les utilisateurs vont avoir tendance à utiliser des échelles différentes de classement. Pour l'un, un 5/5 ne va quasiment jamais être donné car il signifie le saint-graal absolu tandis que pour l'autre, il signifie simplement que *l'item* est « bien ». Pour parer à ce problème, on transforme les valeurs brutes des classements de l'utilisateur en les recentrant sur la moyenne totale de ses classements. Le classement centré sur la moyenne s_{uj}d'un utilisateur *u* pour l'*item j* est défini en soustrayant le classement moyen μ_u au classement brut r_{uj}.

$$s_{uj} = r_{uj} - \mu_u, \forall \mathrm{u} \in \{1, \dots, \mathrm{m}\}$$

Comme auparavant, on utilise la valeur moyenne des classements centrés sur la moyenne du *top-k* d'utilisateur afin de prédire la valeur non-observée du classement de l'utilisateur cible.

$P_u(j)$ représente l'ensemble des utilisateurs du *top-k* par rapport à *u*. Cet ensemble peut parfois être filtré afin de retirer les utilisateurs ayant des corrélations vraiment très faibles voir négatives. La fonction générale de prédiction basée sur le voisinage ressemble donc à la suivante :

$$\hat{r}_{uj} = \mu_u + \frac{\sum_{v \in P_u(j)} Sim(u,v) * s_{vj}}{\sum_{v \in P_u(j)} |Sim(u,v)|} = \mu_u + \frac{\sum_{v \in P_u(j)} Sim(u,v) * (r_{vj} - \mu_v)}{\sum_{v \in P_u(j)} |Sim(u,v)|}$$

$\hat{r}_{uj}$ est couronné d'un chapeau (accent circonflexe) car il s'agit d'une prédiction du classement que l'utilisateur *u* aurait donné à l'*item j* et non pas sa vraie valeur qui est inconnue.

Exemple :

On cherche à prédire les valeurs $\hat{r}_{31}$ et $\hat{r}_{36}$ du tableau 4.4 ci-dessus. La première étape est de définir la similarité entre les utilisateurs afin d'en déterminer le *top-k*. Puis, lorsque le *top-k* est défini, on peut déterminer les prédictions de chacun des *items* non-observés pour l'utilisateur cible.

1) La similarité entre les utilisateurs et l'utilisateur cible peut avoir différentes valeurs selon la fonction de similarité que l'on choisit. Ici, l'exemple utilise le coefficient de Pearson entre l'utilisateur cible et l'utilisateur ayant *l'user-Id* numéro 1.

$$\mu_1 = \frac{6+7+4+5}{4} = 5{,}5, \mu_2 = \frac{3+3+1+1}{4} = 2$$

$$Pearson(1,3)$$
$$= \frac{(6-5{,}5)*(3-2)+(7-5{,}5)*(3-2)+(5-5{,}5)*(1-2)+(5-5{,}5)*(1-2)}{\sqrt{1{,}5^2+1{,}5^2+(-1{,}5)^2+(-0{,}5)^2} * \sqrt{1^2+1^2+(-1)^2+(1)^2}}$$
$$= 0{,}894$$

Il suffit ensuite de répéter la procédure pour chacun des autres utilisateurs. Les coefficients sont inscrits dans le tableau 4.4. Une fois les similarités définies entre l'utilisateur cible et les autres utilisateurs, on peut passer à l'étape suivante.

2) Autant le coefficient de Pearson que celui du cosinus définissent un top-2 des utilisateurs constitués des utilisateurs ayant les *user-ids* 1 et 2. Dans cet exemple, on utilise la méthode la prédiction centrée sur la moyenne. Il va donc falloir, dans un premier temps, déterminer μ_1 et μ_2 afin d'en déduire s_{11}, s_{16}, s_{21} et s_{26}

$$\mu_1 = \frac{7+6+7+4+5+4}{6} = 5{,}5, \mu_2 = \frac{6+7+4+3+4}{5} = 4{,}8$$

$$s_{11} = 7 - 5{,}5 = 1{,}5, s_{16} = 4 - 5{,}5 = -1{,}5$$
$$s_{21} = 6 - 4{,}8 = 1{,}2, s_{2,6} = 4 - 4{,}8 = -0{,}8$$

afin de pour pouvoir calculer les prédictions de classements des *items* 1 et 6 de l'utilisateur 3.

$$\hat{r}_{31} = \frac{1{,}5 * 0{,}894 + 1{,}2 * 0{,}939}{0{,}894 + 0{,}939} \approx 3{,}35$$

$$\hat{r}_{36} = \frac{-1{,}5 * 0{,}894 - 0{,}8 * 0{,}939}{0{,}894 + 0{,}939} \approx 0{,}86$$

Modèles de voisinage basés sur l'item

Dans ce type de modèle, la perspective est placée sur les classements qu'un utilisateur a donné à un certain type d'*item* afin d'en prédire les classements non-observés d'autres *items* proches dans leurs caractéristiques de ceux pour lesquels l'utilisateur a donné son avis. Par exemple, si l'utilisateur *A* a fortement apprécié les *items X* et *Y* et que *Z* est très semblable dans ses caractéristiques, le système va proposer l'*item Z* à l'utilisateur *A*.

La procédure est très similaire aux modèles de voisinage basés sur l'utilisateur. Dans un premier temps, on calcule le classement centré sur la moyenne s_{uj}. La seconde étape est de déterminer la similarité entre les *items i* et *j* au moyen de la fonction de similarité du *cosinus-ajusté* selon la formule suivante :

$$adjustedCosine(i,j) = \frac{\sum_{u \in U_i \cap U_j} s_{ui} * s_{uj}}{\sqrt{\sum_{u \in U_i \cap U_j} s_{ui}{}^2} \sqrt{\sum_{u \in U_i \cap U_j} s_{uj}{}^2}}$$

où U_i représente les index de l'ensemble des utilisateurs ayant évalué l'*item i*.

La fonction de similarité est appelée *cosinus-ajusté* car les classements sont centrés sur la moyenne avant le calcul de la valeur de similarité.

Lorsque la similarité entre l'*item* cible et les autres *items* est déterminée, on peut ensuite procéder à la création d'un *top-k* d'*items* similaires à l'*item* cible *t* pour l'utilisateur *u* nommé $Q_t(u)$. On utilise ensuite la valeur moyenne de ce *top-k* afin de prédire la valeur non-observée. La formule est la suivante :

$$\hat{r}_{ut} = \frac{\sum_{j \in Q_t(u)} adjustedCosine(j,t) * r_{uj}}{\sum_{j \in Q_t(u)} |adjustedCosine(j,t)|}$$

4.1.2 Méthodes basées sur les modèles

« Each person must live their life as a model for others. »

- *Rosa Parks*

Les méthodes basées sur les modèles génèrent un modèle à priori grâce à des méthodes de *machine learning*. La phase d'apprentissage (*training*) est donc clairement séparée de la phase de prédiction. Les méthodes traditionnelles en *machine learning* incluent les arbres de décision, les *Bayes classifiers*, les modèles de régression, les *support vector machines* et les réseaux neuronaux.

Dans un problème de classification, on a une matrice $m \times n$ pour laquelle les *n-1* premières colonnes sont les variables d'attributs (ou variables indépendantes), et la n^{ième} colonne est la variable de classe (ou variable dépendante). Toutes les variables indépendantes de chaque entrée sont spécifiées tandis que seul une partie des entrées n'ont pas de valeur pour la variable dépendante. L'ensemble des entrées complètes (indépendantes + dépendantes) forme les données d'entrainement (*training data*) tandis que l'ensemble des entrées où il manque la variable dépendante forme les données de test (*test data*). L'objectif est donc, en créant un modèle à partir des données d'entrainement, de prédire la valeur des variables dépendantes des données de test.

Mais contrairement aux problèmes de classification, dans notre cas, la matrice de classement est parsemée de valeurs manquantes autant sur les variables dépendantes qu'indépendantes. Il n'y a donc plus de claire distinction entre variables dépendantes et indépendantes. Par conséquence, il n'existe pas non plus de claire distinction entre les données d'entrainement et les données de test. De plus, une dernière différence est que dans le cas d'un problème de classification, chaque ligne représente une instance de données et chaque colonne un attribut tandis que dans notre cas, les colonnes ou les lignes représentent simplement la perspective de l'*item* ou de l'utilisateur. Malgré ces différences, le remplissage de la matrice de classement peut tout de même être vu tel une généralisation d'un problème de classification (ou de régression).

Les avantages principaux des systèmes de recommandation basés sur le modèle par rapport à ceux basés sur le voisinage sont les suivants :

1) ***Efficacité d'espace*** : la taille du modèle est fortement plus petit que la matrice de classement originale. L'espace nécessaire au stockage est donc souvent assez bas. Par exemple, dans le cas de la méthode du voisinage basée sur l'utilisateur, la complexité de l'espace (« *space complexity* ») est de $\boldsymbol{O(m^2)}$ où *m* est le nombre d'utilisateurs.
2) ***Rapidité d'apprentissage et de prédiction*** : les systèmes basés sur le modèle sont généralement beaucoup plus performants durant la face de pré-traitement par rapport à celui des méthodes basées sur le voisinage qui est quadratique, soit dans le nombre d'utilisateurs soit dans le nombre d'*items*.
3) ***Évite l'overfitting*** **:** l'*overfitting* est un sérieux problème en *machine learning*. Le risque est que le modèle soit trop influencé par des attributs aléatoires. Le but est donc d'avoir un modèle ne gardant que les attributs pertinents afin d'être pertinent autant bien sur des données de test que sur celles d'entrainement (sur lesquels il y a un risque d'être *overfitté*). Pour renforcer les modèles, des méthodes de régularisations (« *regularization terms* ») sont couramment employées.

Arbre de décision

Les arbres de décision sont conçus pour les cas où les variables dépendantes sont catégoriques tandis que les arbres de régression sont utilisés dans les cas de variables dépendantes numériques. L'arbre de décision est construit grâce à différents critères de partitions selon les attributs afin que les feuilles de l'arbre regroupent au mieux les catégories. Les critères de partitions peuvent séparer les données de façon plus ou moins optimales. Il existe différentes techniques permettant de choisir l'attribut selon lequel il faudrait partitionner. L'une des techniques s'appelle le gain d'information (*information Gain*) qui est calculé par rapport à l'entropie de la classe et de l'entropie conditionnelle par rapport à l'attribut. Formule de l'entropie :

$$H_L(y) = -\sum_{v=1}^{k} p(y = v) \log_2 p(y = v)$$

où $p(y = v)$ représente la quantité d'instances de classe *v* par rapport au total.

Il est ensuite possible de calculer l'entropie conditionnelle pour un sous-ensemble tel que $H_L(y \mid x_1 = 1)$. La formule du gain d'information est la suivante :

$$G_L(x_1) = H_L(y) - p_L(x_1 = 1)H_L(y \mid x_1 = 1) - p_L(x_1 = 0)H_L(y \mid x_1 = 0)$$

Il faut calculer ce gain d'information pour chaque attribut lorsque que l'on cherche un critère de partition. On partitionne ensuite par l'attribut donnant le plus haut gain d'information. Une

fois que l'arbre est établi, il peut parfois s'avérer utile d'appliquer ce que l'on appelle le *pruning* qui consiste à supprimer les critères de partitions séparant un sous-ensemble trop petit d'instances. On supprime donc les nœuds qui ont des feuilles avec moins de τ instances et on les combine avec la classe majoritaire.

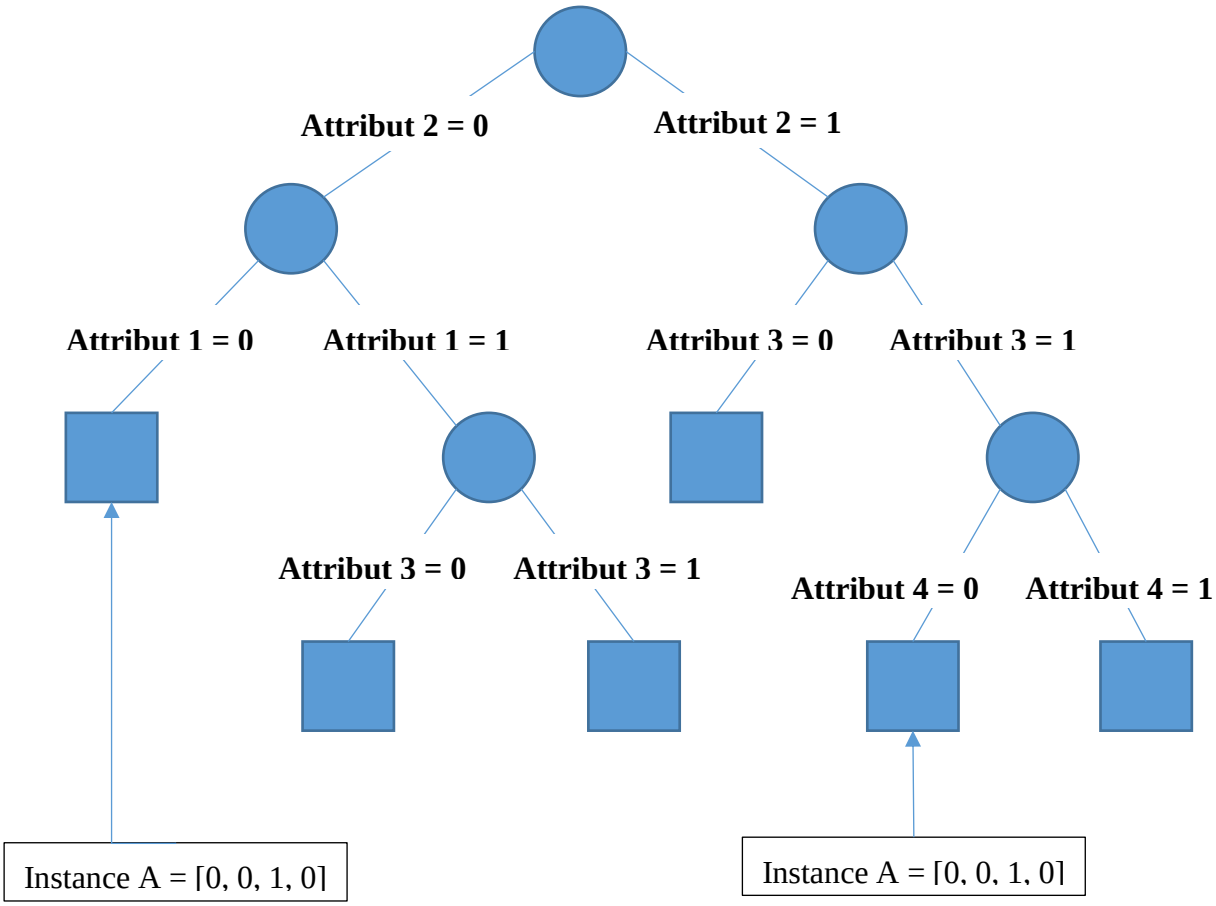

Figure 4.13: exemple d'arbre de décision

Afin d'étendre les arbres de décisions aux systèmes de recommandation collaboratifs, il faut, dans un premier temps, surmonter le problème que les entrées prédites et celles observées ne sont pas clairement séparées en tant que variables d'attributs et de classes. De plus, la matrice de classement n'est que faiblement remplie, la majorité des attributs des entrées est manquante.

Puisqu'il n'y a pas de claire démarcation entre les variables dépendantes et indépendantes, il n'est, à priori, pas possible de baser l'arbre sur un attribut spécifique. Pour parer à ce dernier problème, il est possible de créer un arbre de décision pour chacun des attributs afin que celui-ci soit la variable dépendante et les autres les variables indépendantes. Le nombre d'arbres de décision s'élève donc au nombre d'attributs, autrement dit, au nombre d'*items*. Lorsque l'on souhaite prédire le classement non-observé d'un certain *item* pour un utilisateur, on emploie l'arbre de décision correspondant à cet *item*.

De l'autre côté, parer au problème de variables indépendantes manquantes est plus compliqué. Si un attribut en particulier est utilisé en tant que critère de partition pour un certain *item*, tous les utilisateurs pour qui l'on souhaite prédire la valeur de classement de cet *item* vont devoir fournir cet attribut, sans quoi on ne pourra pas appliquer l'arbre de décision. Et, puisqu'une majorité des attributs est généralement manquante, il est probable que l'utilisateur, pour qui l'on souhaite faire la prédiction, n'ait pas cet attribut renseigné. Une manière de résoudre ce

problème serait de poursuivre en parallèle sur les deux chemins possibles qu'offre le critère de partition et d'en agréger les résultats. Cependant, une telle manière de faire peut possiblement conduire à des prédictions conflictuelles difficilement agrégeable. Une autre approche, plus raisonnable, est d'utiliser une matrice à dimension réduite. La matrice de dimension $m \times (n - 1)$, excluant la $j^{\text{ième}}$ colonne comprenant l'*item* qui a besoin d'être prédit, est convertie dans une matrice de dimension $m \times d$ avec $d \ll n - 1$ et dont tous les attributs sont spécifiés. Il faut ensuite générer une matrice de covariance entre chaque pair de classement d'*item* de la matrice $m \times (n - 1)$ pour ensuite y sélectionner le *top-d*, $\bar{e}_1, \dots, \bar{e}_d$, des vecteurs propres. On emploie ensuite la formule suivante :

$$a_{ui} = \frac{\sum_{j \in I_u} r_{uj} * e_{ji}}{|I_u|}$$

afin de projeter les classements de chaque utilisateur sur les vecteurs propres (le $j^{\text{ième}}$ *item* n'est pas inclus dans la partie droite de l'équation). On obtient un vecteur de classement de *d*-dimension pour chacun des utilisateurs dont les attributs sont complétement spécifiés. Cette matrice réduite est ensuite utilisée afin d'établir l'arbre de décision pour le $j^{\text{ième}}$ *item* tel n'importe quel autre problème de classification ou de régression. Cette approche est ensuite répétée en changeant la valeur de *j* (entre 1 et *n*) afin de construire *n* arbres de décision pour chacun des *items*. Afin de prédire le classement de l'*item j* pour l'utilisateur *i*, la $i^{\text{ième}}$ ligne de la matrice $m \times d$ est utilisée en tant qu'instance de test et le $j^{\text{ième}}$ arbre de décision est utilisé comme modèle afin de prédire la valeur du classement correspondante.

Modèle naïf de Bayes

Pour ce cas, on va considérer que les valeurs des classements sont sélectionnées dans un petit ensemble fini de valeurs distinctes tel que « *like* », « *neutral* » et « *dislike* ». On utilisera donc *l* valeurs distincts dénotées par $v_1, \dots, v_l$. Le classement d'un certain utilisateur *u* pour l'*item j*, r_{uj} prend donc une valeur dans l'ensemble $\{v_1, \dots, v_l\}$. Le modèle naïf de Bayes est communément employé pour les classifications. Les utilisateurs et les *items* de la matrice de classements seront utilisés, respectivement, en tant qu'instances et attributs. Comme pour le cas des arbres de décision, il va falloir faire face aux problèmes qu'il n'y a pas de fixe détermination de la variable dépendante et que certaines valeurs des variables indépendantes peuvent s'avérer manquantes. Cependant, ces problèmes peuvent être contournés par une adaptation du modèle naïf de Bayes. Si l'on considère qu'un utilisateur *u* a spécifié un certain ensemble d'*items* I_u et que l'on souhaite déterminer la valeur du classement r_{uj} où $j \notin I_u$ prenant sa valeur conditionnellement par rapport aux valeurs observées dans I_u. On va donc chercher à déterminer la probabilité que r_{uj} prenne l'une ou l'autre valeur de $s \in \{1, \dots, l\}$:

$$P(r_{uj} = v_s \mid classements\ observés\ dans\ I_u)$$

qui peut être écrite sous la forme générale $P(A \mid B)$ et ensuite traduite en règle de Bayes :

$$P(A \mid B) = \frac{P(A) * P(B \mid A)}{P(B)}$$

qui, pour chaque valeur de $s \in \{1, \dots, l\}$, donne la formule suivante :

$$P(r_{uj} = v_s \mid classements\ observés\ dans\ I_u) = \frac{P(r_{uj} = v_s) * P(classements\ observés\ dans\ I_u \mid r_{uj} = v_s)}{P(classements\ observés\ dans\ I_u)}$$

On cherche à déterminer la valeur de *s* pour laquelle la partie gauche de l'équation est la plus élevée possible. Le dénominateur de la partie droite de l'équation peut être retiré car il est indépendant de la valeur de *s*. Transformant ainsi l'équation de la façon suivante :

$$P(r_{uj} = v_s \mid classements\ observés\ dans\ I_u) = P(r_{uj} = v_s) * P(classements\ observés\ dans\ I_u \mid r_{uj} = v_s)$$

La valeur de $P(r_{uj} = v_s)$ est la probabilité préalable de r_{uj} et est communément appelé le *prior*. Le *prior* est estimé par la fraction d'utilisateurs ayant assigné le classement v_s à l'*item j*. $P(classements\ observés\ dans\ I_u \mid r_{uj} = v_s)$ est estimé en se basant sur l'indépendance conditionnelle des classements. Cette indépendance conditionnelle dit que les classements de l'utilisateur *u* pour les *items* dans I_u sont indépendant l'un de l'autre tout en sachant que r_{uj} a la valeur v_s. Il peut être exprimé mathématiquement de la manière suivante :

$$P(classements\ observés\ dans\ I_u \mid r_{uj} = v_s) = \prod_{k \in I_u} P(r_{uk} | r_{uj} = v_s)$$

$P(r_{uk} | r_{uj} = v_s)$ est la fraction des utilisateurs qui ont spécifié le classement r_{uk} pour le $k^{\text{ième}}$ *item* en sachant qu'ils ont donné la valeur v_s au $j^{\text{ième}}$ *item*. Il est désormais possible de calculer la probabilité postérieure du classement de l'utilisateur *u* pour l'*item j* de la manière suivante :

$$P(r_{uj} = v_s \mid classements\ observés\ dans\ I_u) = P(r_{uj} = v_s) * \prod_{k \in I_u} P(r_{uk} | r_{uj} = v_s)$$

L'estimation de la probabilité postérieure du classement r_{uj} peut être estimé de deux façons différentes :

1) En calculant toutes les valeurs pour chaque $s \in \{1, \dots, l\}$ et en choisissant celle donnant la plus grande probabilité, cette première méthode ignore tout ordre entre les classements et les traite en tant que simple catégorie. Elle est idéalement employée lorsque le nombre possible de classements est petit.

$$\widehat{r_{uj}} = argmax_{v_s} P(r_{uj} = v_s) * \prod_{k \in I_u} P(r_{uk} | r_{uj} = v_s)$$

2) Cette seconde approche est préférable dès que le nombre de classements possible est élevé. Celle-ci, au lieu de sélectionner la probabilité maximale, estime la valeur par une moyenne sur tous les classements, où le poids du classement est sa probabilité. La formule est la suivante :

$$\widehat{r_{uj}} = \frac{\sum_{s=1}^{l} v_s * P(r_{uj} = v_s) * \prod_{k \in I_u} P(r_{uk} | r_{uj} = v_s)}{\sum_{s=1}^{l} P(r_{uj} = v_s) * \prod_{k \in I_u} P(r_{uk} | r_{uj} = v_s)}$$

Puisque cette approche calcule la probabilité conditionnelle du classement basée sur les classements des autres *items*, cette approche est un modèle de Bayes basé sur l'*item*.

Réseau neuronal

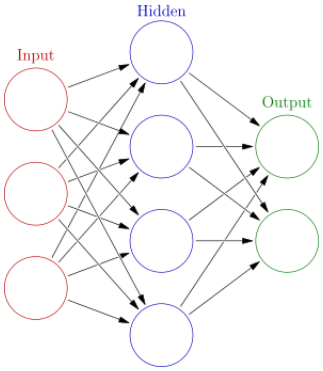

Figure 4.14: réseau neuronal
(source : wikipedia.org)

Les réseaux neuronaux sont une simulation d'un cerveau humain avec l'utilisation de neurones connectés l'un à l'autre grâce à des connections synaptiques. Dans un système biologique, l'apprentissage s'effectue en changeant la force des connections synaptiques lors de stimuli externes. Le cas du réseau neuronal artificiel est très similaire à celui du biologique. Les forces de connections synaptiques correspondent aux poids des attributs du modèle. L'unité de calcul de base est le neurone. Une des architectures de réseau neuronal la plus simple est celle du *perceptron* qui ne contient pas de couche cachée (*hidden layers*) mais uniquement celle d'entrée (*input layer*) et de sortie (*output layer*), autrement dit, un seul neurone. De manière résumé, chaque couche possède un modèle, composé de différents poids, qui va être appliqué au vecteur d'entrée puis passé dans une fonction d'activation dont le résultat sera ensuite transmis à la couche suivante.

Dans le cas du *perceptron*, et en utilisant la fonction d'activation linéaire du signe, fonctionnant bien pour les sorties binaires, on peut résumer le réseau neuronal par :

$$z_i = sign(\boldsymbol{\theta} * \boldsymbol{x_i})$$

où $\boldsymbol{\theta}$ correspond au modèle de la couche, $\boldsymbol{x_i}$ au vecteur d'entrée (autrement dit, les classements de l'utilisateur), z_i au classement estimé et $sign(\bullet)$ à la fonction d'activation dont le résultat est élément de $\{-1; 1\}$.

Lors de réseaux neuronaux à plusieurs couches, chaque couche possède son propre modèle $\boldsymbol{\theta}^i$ n'ayant pas nécessairement le même nombre de poids. Dans le cas des systèmes de recommandations, le vecteur d'entrée correspond aux $(n-1)$ *items* qui sont utilisés afin d'estimer la valeur de l'*item* manquant. y_i représente le classement observé pour l'*item* prédit. La valeur de z_i est la sortie prédite et l'erreur, $(z_i - y_i)^2$, permet de mettre à jour les poids du modèle $\boldsymbol{\theta}$. La fonction de mise à jour est la suivante :

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \alpha(y_i - z_i)\boldsymbol{x_i}$$

où α correspond au degré d'apprentissage et $\boldsymbol{\theta}^t$ est le vecteur des poids à la $t^{\text{ième}}$ itération.

Puisque, dans ce cas précis, on a défini que y_i est binaire, cette approche est conçue pour les matrices de classements binaires mais il est tout à fait possible de concevoir des réseaux neuronaux pour lesquels la sortie n'est pas binaire. En général, un réseau neuronal a plusieurs couches. Dans un tel cas, l'algorithme d'apprentissage afin d'établir les poids s'appelle l'algorithme de *back-propagation*. L'avantage principal des réseaux neuronaux est que l'architecture en couches permet le calcul de fonctions non-linéaires complexes qui ne sont pas facilement calculables avec d'autres méthodes de classification.

4.2 Systèmes de recommandation basés sur le contenu

« What really decides consumers to buy or not to buy is the content of your advertising, not its form. »

- *David Ogilvy*

Dans les systèmes de recommandations basés sur le contenu, les attributs décrivant l'*item* (description, genre, modèle, …) sont utilisés en combinaison avec les classements et le comportement de l'utilisateur dans le but de lui faire des recommandations. Le système utilise les caractéristiques d'*items* évalués par l'utilisateur afin de créer des données d'entrainement, *training data*, dans l'objectif de générer un modèle de régression. Le modèle de régression est spécifique à l'utilisateur. Ce modèle sera ensuite utilisé afin de prédire si l'utilisateur est susceptible d'apprécier un certain *item* quand bien même on ne possède aucune information sur la préférence de l'utilisateur pour ledit *item*.

L'un de principal avantage des systèmes basés sur le contenu est de pouvoir faire des recommandations pour un nouvel *item* n'ayant pas suffisamment de données de classement disponible. Malgré cet avantage considérable, deux désavantages sont à prendre en compte :

1) Les recommandations ont tendance à être évidentes. Ceci à cause de leurs similitudes dans les attributs. Par exemple, si l'utilisateur n'a jamais consommé d'*item* ayant un certain attribut, le système ne va jamais lui proposer d'*items* avec cet attribut (ou par hasard par le biais d'autres attributs)
2) Bien qu'efficace à fournir des recommandations pour de nouveaux *items*, ces systèmes ne sont pas efficaces pour les nouveaux utilisateurs. La cause en est qu'il n'y a aucune donnée d'entrainement afin de produire le modèle de régression. On ne peut donc pas fournir de recommandations pertinentes.

Pour contrer ces problèmes, certaines méthodes peuvent être employées. Par exemple, l'utilisateur peut spécifier des mots-clés pertinents à son profil. Ces profils sont ensuite utilisés afin de trouver des items intéressants et faire des recommandations à l'utilisateur. Une telle approche est utile lorsqu'on démarre sans (ou très peu) information sur l'utilisateur. Généralement, ces méthodes sont considérées comme une classe distincte de système de recommandation appelé les systèmes basés sur la connaissance ou *knowledge-based systems*.

Les trois étapes principales d'un système basé sur le contenu sont : la partie du prétraitement (hors-ligne), celle de l'apprentissage (hors-ligne) suivit de celle de la prédiction qui, elle, s'effectue en temps-réel. Les résultats des étapes hors-lignes sont ensuite utilisées lors de la génération des prédictions pour l'utilisateur. Les différentes phases d'un système basé sur le contenu sont les suivantes :

1) **Prétraitement et extraction des attributs** : généralement, les attributs sont extraits de la source (d'une description d'un produit par exemple) et convertie en un vecteur basé sur des mots-clés. Il s'agit de la première étape de n'importe quel système de recommandation basés sur le contenu et est fortement spécifique au domaine.
2) **Apprentissage des profils utilisateurs basés sur le contenu :** un modèle basé sur le contenu est spécifique à l'utilisateur. Un modèle est calculé pour chacun des utilisateurs afin de prédire ces intérêts vers les *items* selon, par exemple, son historique d'achats ou

de classements. Un modèle d'apprentissage est construit sur les données d'entrainement avec autant des données implicites qu'explicites sur l'utilisateur. Cette étape n'est pas vraiment très différente de la modélisation d'un modèle de classification ou de régression. Le modèle généré est appelé le *profil utilisateur*.

3) **Filtrage et recommandations** : cette étape utilise le modèle établi lors de la précédente étape afin de générer une recommandation sur les *items* pour un certain utilisateur. Cette étape se doit d'être efficiente car les prédictions sont faites en temps-réel.

4.2.1 Prétraitement et extraction des attributs

La première étape consiste à extraire les attributs discriminants afin de représenter les *items*. Les attributs discriminants sont ceux étant étroitement liés à l'intérêt de l'utilisateur. Cette étape sera suivie par celle de la représentation et du nettoyage des attributs précédemment extraits. En parallèle du traitement du contenu des *items*, il faut rassembler les données des préférences de l'utilisateur. La dernière étape consiste à faire une sélection dans les attributs et leur donner un certain poids.

Extraction des attributs

Cette phase extrait les attributs descriptifs des *items*. Bien qu'il soit possible d'utiliser différentes manières de les représenter, l'approche la plus commune est celle d'extraire des mots-clés des données sous-jacentes provenant, par exemple, du titre ou de la description de l'*item*. Mais dans certain cas, on est obligé de travailler avec des attributs multidimensionnels tels des quantités numériques (par exemple : prix) ou des ensembles finis (par exemple : couleurs).

Représentation des attributs et nettoyage

Cette étape est particulièrement importante car les données extraites des *items* ne sont pas structurées et peuvent contenir des informations impertinentes. Il existe trois phases principales afin de nettoyer ces mots-clés :

1) **Suppression des *stop-words*** : de nombreux mots-clés qui ont été extraits ne sont pas spécifiques à l'*item* mais sont partie intégrante du langage. De tels mots sont typiquement les articles, les prépositions, les conjonctions et les pronoms. Il existe des listes standardisées de *stop-words* pour différentes langues pouvant être utilisées afin de filtrer les mots-clés.

2) ***Stemming*** : cette procédure vise à unifier les différentes variations d'un même mot tel que les singuliers et pluriels. Généralement, la racine commune des mots est extraite et utilisée comme nouveau mot-clé. Par exemple : les mots « dansent », « danse » et « dansions » sont unifiés par le mot-clé « dans ». Bien sûr, le *stemming* a un effet néfaste car la racine commune peut avoir une autre signification tel qu'ici avec « dans ». Il existe de nombreux outils de *stemming* prêt à l'emploi.

3) **Extraction de phrase** : l'idée est de détecter des mots qui apparaissent fréquemment ensemble et qui, séparés l'un de l'autre, n'ont pas la même signification. Il existe des dictionnaires de ce type de mots ainsi que des outils automatiques pour les traiter.

Ces étapes exécutées, les mots-clés sont convertis en vecteurs. Chaque mot est référencé en tant que terme. Dans la représentation vectorielle, chaque terme est associé à sa fréquence d'apparition. Certains termes peuvent faire plus de sens dans le contexte que d'autres. Afin de ne garder que les termes les plus pertinents, on va utiliser la notion appelée *inverse document frequency*. Cette méthode donne une haute valeur aux termes ne se trouvant que dans une minorité des documents. Un document correspond aux descriptions d'un *item*. Plus la valeur est proche de 1, moins le terme est pertinent. La formule est la suivante :

$$idf_i = \log\left(\frac{n}{n_i}\right)$$

où idf_i est la fréquence de document inverse pour le $i^{\text{ème}}$ terme qui apparait dans n_i documents sur un total de n documents.

Il faut également faire attention à ne pas donner trop d'importance à un terme en particulier. Pour parer ce problème, on applique une fonction d'amortissement, telle que la racine carrée ou le logarithme, à la fréquence avant le calcul de similarité.

$$f(x_i) = \sqrt{x_i} \text{ ou } f(x_i) = \log(x_i)$$

Une fréquence normalisée $h(x_i)$ pour le $i^{\text{ième}}$ terme est défini de la manière suivante :

$$h(x_i) = f(x_i) * idf_i$$

Ce modèle est connu sous le nom de TF-IDF où TF représente la fréquence du terme tandis que IDF est la fréquence de document inverse.

Collecte des préférences de l'utilisateur

Parallèlement à l'analyse du contenu des *items*, il faut également collecter les préférences de l'utilisateur pour le processus de recommandation. Les préférences de l'utilisateur peuvent apparaitre sous la forme de classements, de feedbacks implicites (p. ex. : un achat ou une recherche), d'opinions sous forme de texte (p. ex. : des commentaires) ou de cas dans lesquels l'utilisateur spécifie des exemples d'*items* pour lesquels il a un intérêt.

Chaque type de préférence mentionné précédemment est ensuite traduit sous forme de classement unaire, binaire, basé sur un intervalle ou réel.

Sélection et pesage des attributs

L'objectif de cette étape est de peser et de garder uniquement les termes les plus pertinents. D'après les résultats de M. Pazzani et D. Billsus[Paz97], le nombre de termes devrait se situer entre 50 et 300. Retirer les termes les moins pertinents permet d'éviter d'*overfitter* le modèle. Cette phase possède deux aspects : celle de la sélection des attributs correspondant à la suppression de certains termes, et celle du pesage qui implique de donner différents poids selon les termes. La suppression des *stop-words* et l'utilisation de la fréquence de document inverse sont, respectivement, des exemples de sélection et de pesage des attributs. Cependant, ces deux méthodes ne prennent pas en compte les classements de l'utilisateur. Des méthodes communément utilisées pour peser les attributs sont : le coefficient de Gini, l'entropie, le χ^2 ou encore l'écart normalisée.

Coefficient de Gini

t est le nombre total de valeur possible du classement. Sur le total des documents contenant le mot w, $p_1(w), \dots, p_t(w)$ sont les fractions d'*items* classés pour chacune des valeurs t possible.

$$Gini(w) = 1 - \sum_{i=1}^{t} p_i(w)^2$$

Le coefficient de Gini prend toujours une valeur entre 0 et 1. Plus la valeur est petite, plus le pouvoir discriminant est grand. Par exemple, sur 100 *items* classés de 1 à 5 contenant le mot « vélo », 15 ont reçu la note de 1, 35 la note de 4 et 50 la note de 5 :

$$Gini(vélo) = 1 - \left(\frac{15}{100}\right)^2 - \left(\frac{35}{100}\right)^2 - \left(\frac{50}{100}\right)^2 = 0{,}605$$

Tandis que si tous les *items* ayant le mot « vélo » avaient la même note, le coefficient de Gini serait de 0.

Le coefficient de Gini indique le niveau de dispersion des classements entre les différentes valeurs possibles.

Entropie

L'entropie est très similaire au coefficient de Gini et le concept a déjà rapidement été introduit dans le chapitre sur les arbres de décision dans les systèmes de recommandation collaboratifs.

t est le nombre total de valeur possible du classement et $p_1(w), \dots, p_t(w)$ sont les fractions des documents contenant le mot w pour chacune des t valeurs possibles.

$$Entropy(w) = -\sum_{i=1}^{t} p_i(w) \log p_i(w)$$

La valeur de l'entropie se situe toujours entre 0 et 1. Les petites valeurs indiquent un grand pouvoir discriminant. Si l'on reprend l'exemple du coefficient de Gini avec le mot « vélo », l'entropie est la suivante :

$$Entropy(vélo) = -\frac{15}{100}\log\left(\frac{15}{100}\right) - \frac{35}{100}\log\left(\frac{35}{100}\right) - \frac{50}{100}\log\left(\frac{50}{100}\right) = 0{,}311$$

4.2.2 Apprentissage des profils utilisateurs basé sur le contenu

Chaque utilisateur possède son modèle qui lui est propre. Ce modèle est généré grâce à l'ensemble des documents D_L des *items* pour lesquels il a donné un avis. Un document correspond à la description d'un *item*. D_L est l'ensemble des documents d'entrainement, ou *training documents*, extrait durant la phase précédente. Il y a également l'ensemble des documents de test D_U qui correspond à la description des *items* pouvant potentiellement être recommandé à l'utilisateur. Le modèle entraîné sur D_L est ensuite utilisé afin de faire des recommandations entre les documents de D_U à l'utilisateur. Le modèle peut autant bien être utilisé afin de prédire une valeur de classement mais également générer une liste du *top-k* de recommandations. Il y a différentes méthodes d'apprentissage dont les plus communes sont : la classification selon le plus proche voisin (*nearest neighbor classification*), le modèle de

classification de Bayes (*Bayes classifiers*) ou encore le modèle de classification basé sur la règle (*Rule-based classifiers*).

Classification selon le plus proche voisin

La première étape est de choisir une fonction de similarité telle que celle du cosinus ou de Pearson. Soit $\boldsymbol{x}$ et $\boldsymbol{y}$ une paire de documents sous forme de vecteurs dont les fréquences normalisées des termes sont respectivement données par x_i et y_i. La fonction cosinus a la forme suivante :

$$cosinus(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{x} * \boldsymbol{y}}{\|\boldsymbol{x}\|\|\boldsymbol{y}\|} = \frac{\sum_{i=1}^{d} x_i y_i}{\sqrt{\sum_{i=1}^{d} {x_i}^2} * \sqrt{\sum_{i=1}^{d} {y_i}^2}}$$

Cette fonction de similarité est utile pour faire des prédictions sur des *items* pour lesquels les préférences de l'utilisateur sont inconnues. Pour chaque document $d \in D_U$, le top-*k* des voisins les plus proches dans D_L est déterminé grâce à la fonction de similarité. Afin de prédire la valeur du document (*item*) d, on effectue la moyenne des fréquences du top-*k* de d. Une fois toutes les prédictions calculées pour les documents de D_U, ils sont classés selon la valeur prédite puis proposés à l'utilisateur. Plus D_L et D_U contiennent de documents, plus la complexité de calcul sera grande. C'est pourquoi, afin d'améliorer les performances, une méthode couramment employée est celle du *clustering* telle que le *k-means*.

4.3 Systèmes de recommandation basés sur la connaissance

« A good decision is based on knowledge and not on numbers. »

- Plato

Les systèmes de recommandation basés sur la connaissance sont particulièrement utiles dans un contexte où les *items* sont rarement évalués par les utilisateurs. Des exemples de tels *items* sont : les biens immobiliers, les voitures, les services financiers ou encore les biens de luxe. Il s'agit précisément du cas de MyFlat. Puisque les *items* ne sont que rarement acheté/commandé, ou avec différents types d'options, il est difficile d'obtenir un nombre suffisant d'évaluations pour une certaine combinaison d'option d'un certain *item*. Un utilisateur peut être intéressé uniquement par certaines propriétés spécifiques à un *item* et crée ainsi une combinaison spécifique et complexe d'options difficile à gérer.

Dans le cas des systèmes basés sur la connaissance, les classements d'*items* par les utilisateurs ne sont pas utilisés car ils ne seraient pas pertinents. La procédure de recommandations est effectuée au moyen de similarité entre les exigences des clients et les attributs des *items* ou alors l'utilisation de contraintes spécifiant les exigences de l'utilisateur. Contrairement aux systèmes collaboratifs et ceux basés sur le contenu, ceux basés sur la connaissance sont uniques car ils permettent à l'utilisateur de spécifier explicitement ce qu'il veut. Ces types de systèmes peuvent être séparés en deux catégories selon l'interface utilisée pour obtenir les recommandations :

1) **Les systèmes de recommandations basés sur la contrainte** : l'utilisateur spécifie les exigences ou contraintes sur les attributs de l'*item*.

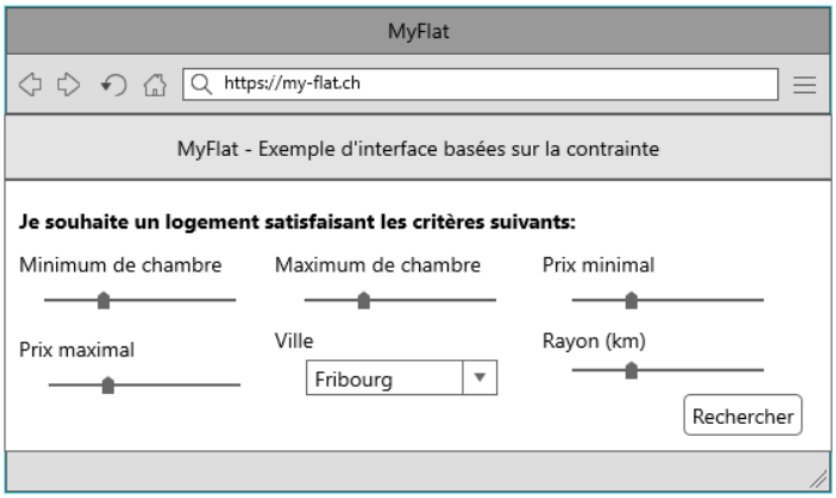

Figure 4.15: interface d'un système basé sur la contrainte

Les spécifications de l'utilisateur sont utilisées afin de retrouver les *items* correspondant à ses critères. Le système crée ensuite des règles selon les spécifications. Une règle sur les attributs d'un *item* peut être, par exemple, « *les voitures d'avant 1970 n'ont pas de régulateur de vitesse* ». Mais il peut également créer des règles entre des attributs de l'utilisateur et des attributs de l'*item* tel que « *les investisseurs âgés n'investissent pas dans les produits ultra-risqués* ». Mais dans ce dernier cas, les attributs de l'utilisateur doivent également être spécifié dans la procédure de recherche. Selon le nombre et le type de résultats retournés, l'utilisateur va pouvoir modifier sa requête originale en précisant ou relaxant certains de ses critères.

2) **Les systèmes de recommandation basé sur le cas** : des cas spécifiques sont déterminés par l'utilisateur en tant que cibles ou points d'ancrage. Des matrices de similarité sont ensuite définies par rapport aux attributs de l'*item* afin de retrouver d'autres *items* similaires à ce cas.

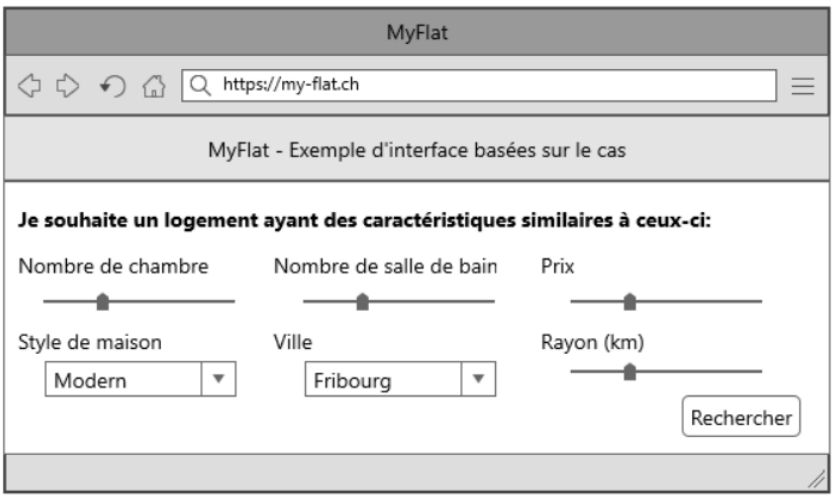

Figure 4.16: interface d'un système basé sur le cas

Le résultat retourné est souvent utilisé en tant que nouvelle cible avec quelques modifications par l'utilisateur. Par exemple, lorsqu'un utilisateur voit un résultat qui est proche de ce qu'il veut, il va pouvoir réutiliser une requête avec cette cible mais en modifiant quelques attributs par rapport à ses désirs. Ce procédé interactif est utilisé afin de guider l'utilisateur à travers l'univers des *items* d'intérêt.

Dans les deux cas, le système permet à l'utilisateur de changer et d'améliorer ses spécifications mais la différence réside dans la procédure. Il est important de noter que les systèmes basés sur la connaissance et ceux basés sur le contenu dépendent tous deux fortement des attributs des *items*. Par conséquent, les systèmes basés sur la connaissance héritent de certains des désavantages des systèmes basés sur le contenu. Par exemple, les recommandations peuvent parfois sembler évidentes car il n'y a pas d'utilisation de classements communautaires permettant de mettre certains *items* en évidence. La principale différence entre ces deux systèmes est que les systèmes basés sur le contenu apprennent du comportement passé de l'utilisateur tandis que ceux basés sur la connaissance font des recommandations par rapport aux spécifications actuelles des besoins et intérêts de l'utilisateur.

4.3.1 Systèmes de recommandation basés sur la contrainte

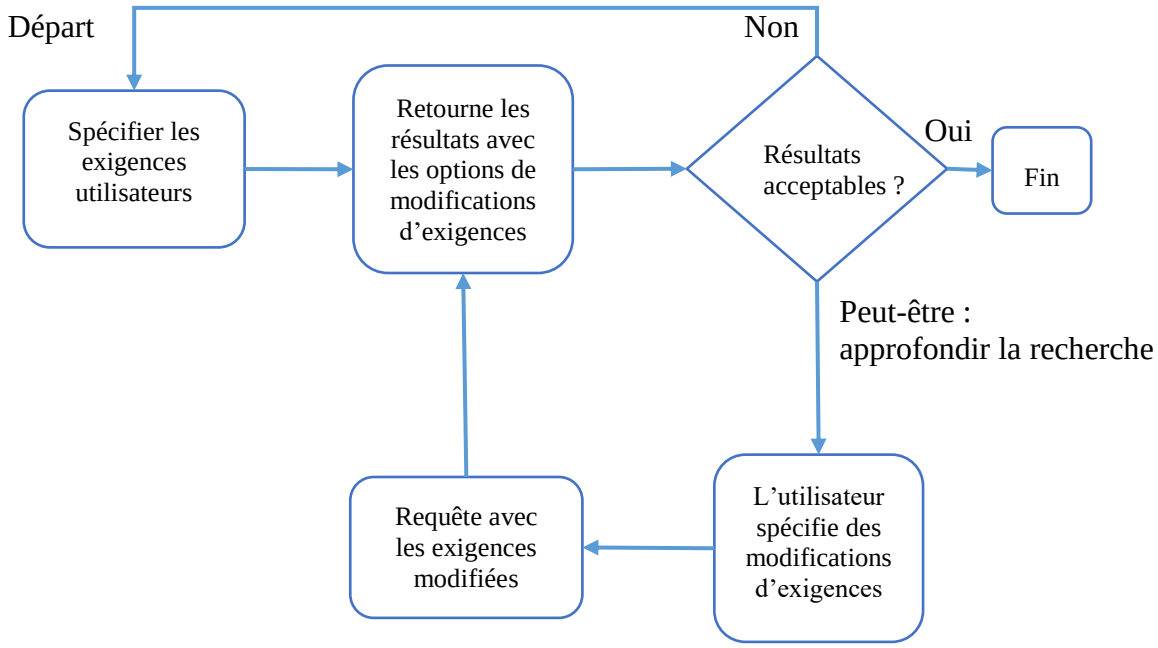

Figure 4.17: *use case* d'un système basé sur la contrainte [Agg16]

Les systèmes de recommandation basés sur la contrainte permettent aux utilisateurs de spécifier des exigences ou contraintes sur les attributs de l'*item*. En plus de cela, des règles seront appliquées afin de trouver des *items* correspondant aux attributs des exigences de l'utilisateur. Les attributs peuvent soit représenter des propriétés inhérentes aux consommateurs (par exemple, démographique) soit spécifier des exigences du consommateur pour le produit.

Par exemple : statut matrimonial (catégorie), taille de famille (numérique), banlieue (binaire), minimum de chambre (numérique), maximum de chambre (numérique), prix maximum (numérique), …

Certains attributs, tels que *le prix maximum* ou *le nombre de chambres*, sont plus faciles à faire correspondre aux produits que d'autres, tels que *banlieue* ou *familiale*, pouvant être plus compliqués à classer. Il faut donc être capable de lier ces critères utilisateurs aux attributs des produits afin de pouvoir filtrer les *items* à recommander. Ceci est fait grâce à l'utilisation de bases de connaissances contenant des règles supplémentaires permettant de relier les exigences de l'utilisateur aux attributs des produits.

Une base de connaissance est une collection de données à propos de produits, services, ou de différents sujets. Les données peuvent provenir de sources variées mais, le plus souvent, elles proviennent d'experts dans le domaine.

D'autres contraintes peuvent émerger de la relation des attributs de l'utilisateur avec ceux du produit : $statut\ marital = célibataire \Rightarrow\ maximum\ de\ chambre \leq 4$

Ce type de contraintes peut être le résultat d'expériences ou de l'extraction de données. Il existe trois principaux types de données pour les systèmes basés sur la contrainte :

1) Le premier type est représenté par les attributs décrivant les propriétés inhérentes au produit (p. ex. : *minimum de chambre*) ou à l'utilisateur (démographique, profil de risque). Certains de ces attributs sont plus évidents que d'autres, qui ne peuvent être révélés qu'à travers l'utilisation de bases de connaissance. Dans la plupart des cas, les exigences des utilisateurs ne sont valables que le temps d'une session. Contrairement à d'autres systèmes de recommandation où un historique est enregistré.

2) Le second type est représenté par les bases de connaissances qui relient les attributs et exigences de l'utilisateur aux attributs des produits. Ce procédé peut être fait de manière direct ou indirect :

   Directe : une règle directe relie une exigence de l'utilisateur, tel qu'*un minimum de 3 chambre*, à une condition sur l'attribut du produit, tel qu'*un prix plus élevé que 100'000 CHF*.

   $$minimum\ de\ chambre \geq 3\ \Rightarrow Prix \geq 100'000CHF$$
   $$attribut/exigence\ de\ l'utilisateur\ \Rightarrow\ attribut\ du\ produit$$

   Indirecte : les règles indirectes créent, en quelque sorte, une règle directe à partir d'un attribut ou exigence de l'utilisateur. Par exemple, un attribut d'*une taille de famille plus grande que 5* implique une exigence d'avoir au *minimum 3 chambres*. Cette dernière exigence impliquera par après la règle directe d'*un prix supérieur à 100'000 CHF*.

   $$taille\ de\ famille \geq 5\ \Rightarrow Minimum\ de\ chambre \geq 3$$
   $$attribut/exigence\ de\ l'utilisateur$$
   $$\Rightarrow\ attribut/exigence\ implicite\ de\ l'utilisateur$$

   Les bases de connaissances précédemment citées sont généralement dérivées d'informations publiques, d'experts du domaine, d'expériences passées ou de l'extraction de données.

3) Le dernier type de données est le catalogue des produits qui contient tous les *items* ainsi que leurs attributs correspondants.

Retourner un résultat pertinent

L'ensemble des règles et exigences peuvent être vues comme une tâche de filtrage de données sur le catalogue d'*items*. Toutes ces règles et exigences sont utilisées afin de construire une requête sélective vers la base de données. Cette requête se construit en trois étapes :

1) Pour chaque exigence ou attributs que spécifie l'utilisateur, le système vérifie que la spécification est autorisée par les règles de la base de connaissance. Par exemple, si l'utilisateur indique avoir *une famille de 6 membres*, les règles d'un *minimum de trois chambres et deux salles de bains* vont s'appliquer. Et à la suite de l'introduction de ces deux nouvelles règles, d'autres règles subséquentes peuvent apparaître tel qu'*un prix de plus de 100'000 CHF* pour un logement de plus de 3 chambres. Toutes ces règles doivent ensuite être appliquées aux attributs de l'*item*, c'est-à-dire, à l'attribut *nombre de chambres*, *nombre de salles de bain* et *prix* afin d'obtenir les conditions suivantes :
$$prix \geq 100'000, chambre \geq 3, salledebain \geq 2$$
2) Ces conditions sont ensuite appondues ($condition1\ and\ condition2\ and$ ...) afin de générer la requête de base de données.
3) Cette requête est ensuite utilisée afin de retourner les *items* pertinents par rapport aux exigences et attributs de l'utilisateurs.

Il est important de noter que la plupart des systèmes de recommandation basés sur la contrainte permettent la spécification des exigences et attributs de l'utilisateur durant la session même. Cette information n'est donc pas persistante. Si deux utilisateurs effectuent exactement la même requête, ils recevront tout deux les mêmes résultats.

Classement des *items*

L'approche la plus simple est d'autoriser l'utilisateur à spécifier un attribut numérique par rapport auquel le classement sera effectué. Par exemple, *le prix, la distance par rapport à un lieu, le nombre de chambre*, etc… L'utilisation d'un seul attribut a le désavantage de donner trop d'importance à cet attribut et trop peu aux autres. Une approche permettant de lutter contre ce problème est l'utilisation d'une fonction d'utilité. $\boldsymbol{v} = (v_1, \dots, v_d)$ étant un vecteur contenant les valeurs des attributs d'un *item*. La fonction d'utilité peut être définie telle une addition pondérée des utilités de chaque attribut. Chaque attribut a une pondération w_j permettant de leur donner plus ou moins de poids et également une fonction $f(v_j)$ dépendante de la valeur v_j de l'attribut. Cette fonction permet de donner plus ou moins d'importance à l'attribut en fonction de sa valeur. En effet, certains attributs peuvent avoir différentes importances selon leurs valeurs. Par exemple, une personne recherchant un logement avec un prix bas sera plus sensible à une augmentation de cet attribut qu'un utilisateur cherchant dans le haut de gamme.

$$U(\boldsymbol{v}) = \sum_{j=1}^{d} w_j * f(v_j)$$

La difficulté de cette approche est de déterminer w_j et $f(\bullet)$. Ceux-ci sont généralement déterminé par le biais de connaissances dans le domaine ou alors d'un apprentissage par rapport aux interactions passées avec les utilisateurs. Une manière répandue de récolter ces informations est de générer des données d'entrainement grâce à certains utilisateurs à qui on donne la tâche de classer des exemples d'*items*. Ces données d'entrainement sont ensuite

utilisées afin de générer un modèle de régression. Cette approche est méthodologiquement liée à l'analyse conjointe.

4.3.2 Systèmes de recommandations basés sur le cas

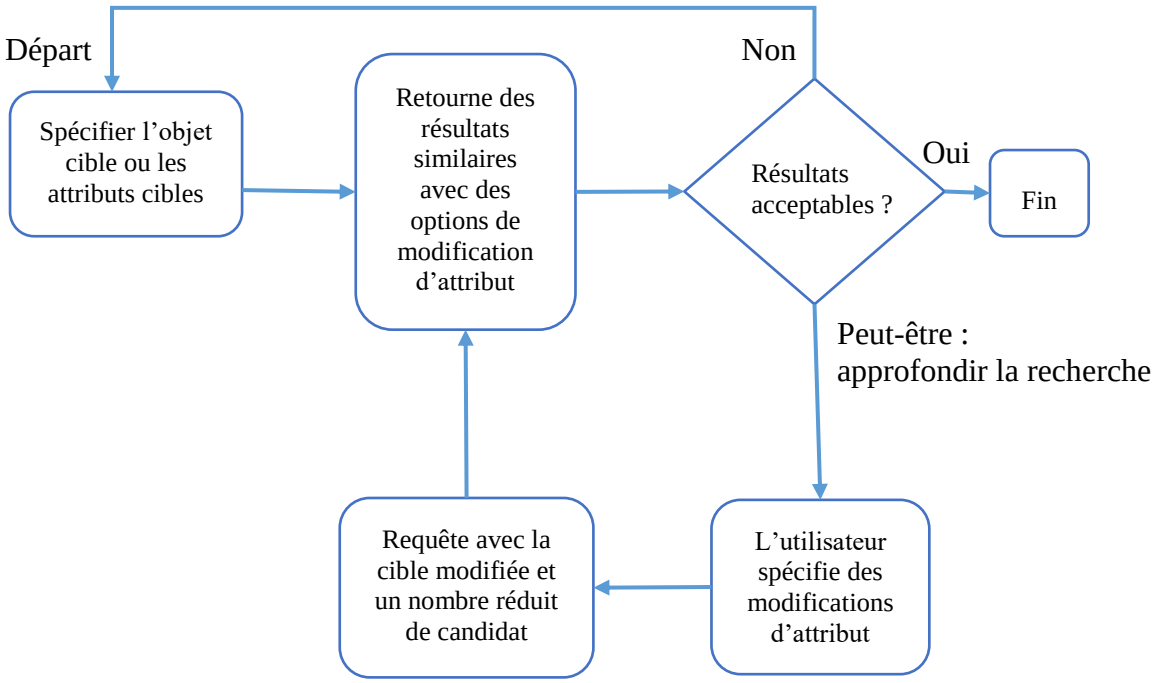

Figure 4.18: *use case* d'un système basé sur le cas [Agg16]

Les systèmes de recommandation basés sur le cas utilisent les mesures de similarité afin de déterminer des *items* similaires aux *items* cibles (aussi appelé cas). Généralement, l'utilisateur va commencer par spécifier les attributs désirés sur l'interface de départ. Une fonction de similarité est ensuite appelée afin de déterminer les *items* correspondant le mieux aux attributs désirés. Par exemple, si aucun logement ne correspond exactement aux critères de l'utilisateur (*3 chambres*, *deux salles de bain*, …), la fonction de similarité sera chargée de trouver et classer les *items* par rapport à leurs similarités aux critères de l'utilisateur.

Les premiers systèmes basés sur le cas, préconisaient la modification répétée des exigences utilisateurs jusqu'à ce qu'une solution appropriée soit trouvée. Par après, la méthode des critiques a été développée. L'idée est que l'utilisateur sélectionne un ou plusieurs résultats pertinents et spécifie les attributs qu'il souhaiterait changer. Ceci grâce à des requêtes approfondies telles que : « donne-moi plus d'*items* similaires à *X* mais différents pour l'attribut *Y* selon les indications *Z*. ». L'objectif principal des critiques est de permettre une recherche interactive dans l'espace des *items* durant laquelle l'utilisateur prend connaissance d'autres options disponibles. Il est possible que l'utilisateur obtienne un résultat pertinent d'*items* qu'il n'aurait pas atteint initialement. Le système utilise la requête cible en association avec des fonctions de similarité afin de déterminer les résultats correspondants. Lors de l'affichage des résultats, l'utilisateur peut trouver un *item* qu'il apprécie particulièrement à l'exception d'un (ou plusieurs) critère. Il va donc utiliser cet *item* comme point d'ancrage et spécifier l'attribut(s) qu'il souhaiterait changer. Le système procède ensuite à une nouvelle requête avec un ensemble réduit d'*items* candidats, qui sont le résultat de la précédente requête. Contrairement aux

systèmes basés sur la contrainte, le nombre de résultats est généralement décroissant et se réduit après chaque cycle. Il est toutefois possible d'effectuer chaque itération sur l'entier de la base de données mais cela est moins courant. L'avantage d'itérer sur l'entier de la base est de permettre d'obtenir un résultat plus distant de la requête originale. Mais le principal désavantage est que l'on peut rapidement, au travers des itérations, se retrouver avec des résultats non-pertinents.

Après plusieurs itérations, l'utilisateur peut parfois arriver à un résultat final légèrement différent des spécifications initiales. Ceci est dû au fait qu'il n'est pas toujours facile pour l'utilisateur d'articuler tous les attributs désirés dès le début. Afin que le système fonctionne de manière efficace, les mesures de similarité et les méthodes de critiques doivent impérativement être conçues efficacement.

Mesures de similarité

En règle générale, il est désirable de développer une fonction de similarité reflétant précisément la réalité. Les paramètres de cette fonction peuvent être déterminés soit par des experts dans le domaine soit par le biais d'une procédure d'apprentissage. Considérant un produit décrit par d attributs, on aimerait déterminer les similarités des valeurs entre deux vecteurs partiels d'attributs définis dans un sous-ensemble S d'un univers de d attributs.

$$|S| = s \leq d$$

$\boldsymbol{x} = (x_1, \dots, x_d) et\ \boldsymbol{t} = (t_1, \dots, t_d)$ sont deux vecteurs de dimension d qui peuvent être (et le sont probablement) que partiellement spécifiés. **t** représente l'*item* cible. Les deux vecteurs contiennent en tout cas les attributs du sous-ensemble $S \subseteq \{1, \dots, d\}$ qui sont spécifiés. On utilise généralement des vecteurs partiellement spécifiés car l'utilisateur va avoir tendance à spécifier uniquement les attributs qui lui sont important. La fonction de similarité $f(\boldsymbol{t}, \boldsymbol{x})$ est développée de la manière suivante :

$$f(\boldsymbol{t}, \boldsymbol{x}) = \frac{\sum_{i \in S} w_i * Sim(t_i, x_i)}{\sum_{i \in S} w_i}$$

$Sim(t_i, x_i)$ représente la similarité entre les valeurs t_i et x_i tandis que w_i représente le poids donné au $i^{ème}$ attribut. Comme pour les fonctions d'utilité, la complexité réside dans la détermination de la fonction de similarité $Sim(t_i, x_i)$ et du poids w_i.

Fonction de similarité

Les attributs peuvent être quantitatifs ou catégoriques, ce qui complexifie le système. De plus, les attributs peuvent être symétriques ou asymétriques d'un point de vue des valeurs supérieures et inférieures. C'est-à-dire que si la valeur de l'attribut, tel que *le prix*, d'un *item* est plus bas que celui de l'*item* comparé, le résultat sera beaucoup plus facilement accepté que si *le prix* est plus élevé. D'autres attributs peuvent avoir un comportement complétement symétrique. L'utilisateur souhaite donc une valeur équivalente à la valeur cible t_i. Exemple d'une mesure symétrique :

$$Sim(t_i, x_i) = 1 - \frac{|t_i - x_i|}{max_i - min_i}$$

Où max_i et min_i représente la valeur maximum et minimum de l'attribut i. Il est également possible d'utiliser l'écart-type σ_i, en se basant sur un historique de données collectées, afin de définir la fonction de similarité suivante :

$$Sim(t_i, x_i) = \max\left\{0{,}1 - \frac{|t_i - x_i|}{3\sigma_i}\right\}$$

Dans un cas symétrique, la similarité est entièrement définie par la différence entre les deux valeurs de l'attribut. Tandis que dans le cas asymétrique, on ajoute une « récompense » asymétrique supplémentaire qui se déclenche en fonction de la valeur de l'attribut cible.

Dans le cas où les valeurs larges sont préférées, la mesure asymétrique peut (il existe différentes manières de la calculer) ressembler à cela :

$$Sim(t_i, x_i) = 1 - \frac{|t_i - x_i|}{max_i - min_i} + \underbrace{\alpha_i * I(x_i > t_i) * \frac{|t_i - x_i|}{max_i - min_i}}_{\text{récompense asymétrique}}$$

Dans le cas contraire où les plus petites valeurs sont appréciables (p. ex. : *le prix*), la fonction est similaire sauf pour la fonction indicateur $I(condition)$:

$$Sim(t_i, x_i) = 1 - \frac{|t_i - x_i|}{max_i - min_i} + \alpha_i * I(x_i < t_i) * \frac{|t_i - x_i|}{max_i - min_i}$$

où $\alpha_i \geq 0$ est un paramètre défini par rapport à l'utilisateur et $I(condition)$ est une fonction d'indicateur qui prend la valeur 1 si la condition est respectée et, sinon, la valeur 0.

La récompense n'est donc activée que si la condition de la fonction d'indicateur est vérifiée. La valeur de α_i est déterminée de manière spécifique au domaine. Pour les valeurs de $\alpha_i > 1$, plus la distance par rapport à la valeur cible est grande, plus la « similarité » est grande. La similarité est mise entre guillemets car, dans certains cas, il est parfois préférable de voir la fonction de similarité en tant que fonction d'utilité. Si l'on prend l'exemple du prix, l'utilisateur préférera toujours un prix bas qu'un prix haut, les autres valeurs d'attributs restant constantes. Lorsque α_i est exactement 1, cela signifie que l'utilisateur est indifférent aux changements de valeurs dans l'une des directions. Si $\alpha_i \in (0, 1)$, l'utilisateur préfère une valeur proche de la cible. Il n'est pas facile de définir les mesures de similarité et cela implique beaucoup de travail en préalable par des experts dans le domaine.

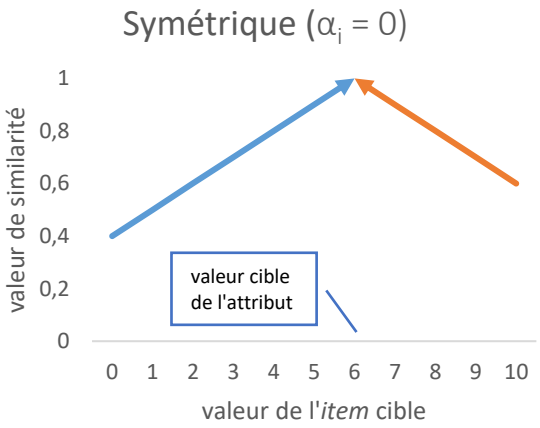

Figure 4.19: similarité symétrique (a=0)

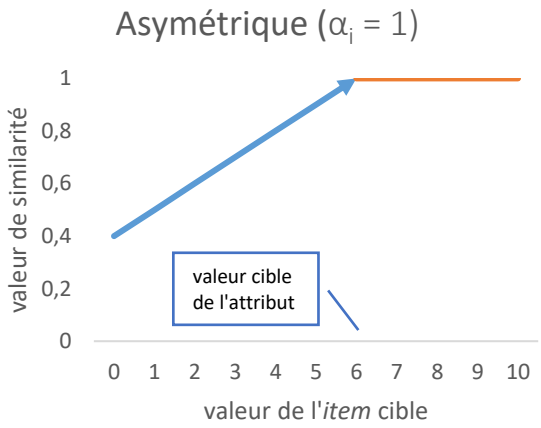

Figure 4.20: similarité asymétrique (a=1)

Le graphique 4.17 ci-dessus représente un exemple de fonction de similarité

Si, à partir d'un certain seuil, un changement de la valeur de l'attribut n'importe plus à

symétrique. On peut notamment observer que la similarité est linéairement dépendante de la distance par rapport à la valeur de l'*item* cible.

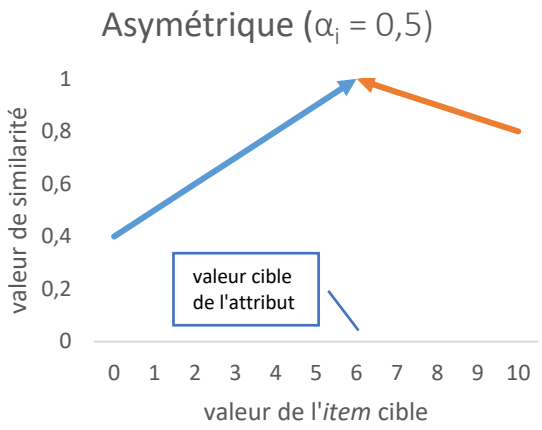

Figure 4.21: similarité asymétrique (a=0.5)

l'utilisateur, on utilise une fonction asymétrique telle que la figure 4.18 ci-dessus. Un tel exemple peut être la résolution d'une caméra (pour les novices).

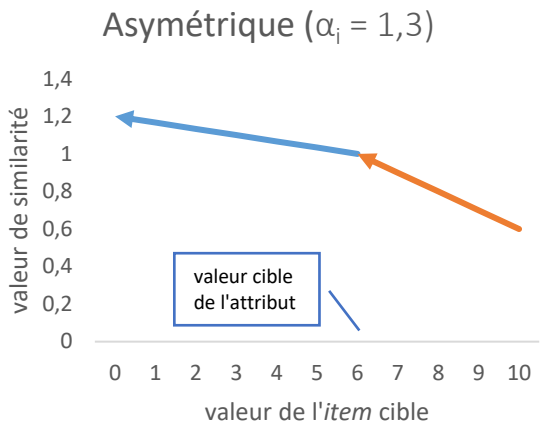

Figure 4.22: similarité asymétrique (a=1.3)

Dans certain cas, la fonction de similarité converge vers la valeur cible de l'attribut différemment selon si l'on se trouve en dessous ou au-dessus de valeur. Un exemple est celui du nombre de chevaux d'une voiture. L'utilisateur désire un certain nombre de chevaux mais ne souhaite pas non plus que la voiture consomme trop.

Le cas de la figure 4.20, ci-dessus, reflète l'exemple du prix où un prix plus bas sera toujours préféré. On parle donc ici plutôt d'une fonction d'utilité.

Dans le cas de données catégoriques, il est encore plus compliqué déterminer des similarités. Généralement, une hiérarchie du domaine est établie afin de déterminer les valeurs de similarité. Plus deux objets sont proches l'un de l'autre dans la hiérarchie, plus la similarité est considérée comme grande. Exemple de hiérarchie :

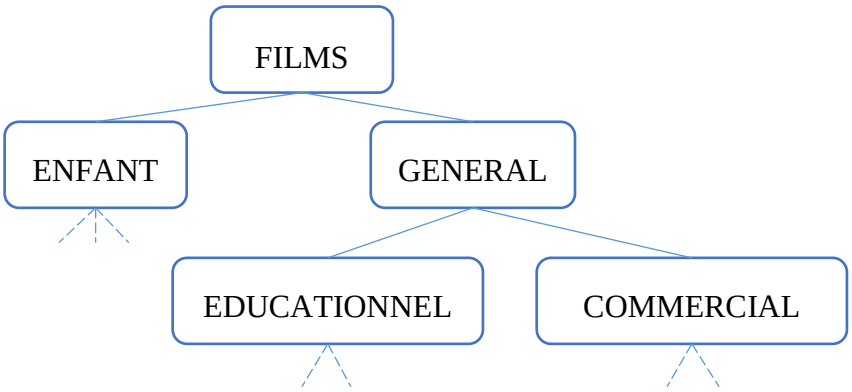

Figure 4.23: exemple de hiérarchie de catégories

Autant pour les fonctions de similarité que pour les hiérarchies, la mise en place de ces fonctions demande un effort initial conséquent dans le paramétrage des problèmes spécifiques au domaine. L'aide d'experts dans le domaine peut être précieux.

Poids de l'attribut

L'importance relative d'un attribut est régulée par le paramètre w_i. Une possibilité de déterminer ce poids est de faire appel à des experts dans le domaine. Il peut également être déterminé en faisant appel aux classements des utilisateurs. En leurs demandant, par exemple, de jauger la similarité entre deux *items*, pour ensuite, grâce à toutes ces données récoltées, déterminer un modèle de régression linéaire permettant la prédiction de w_i.

Méthode de critique

Les critiques sont motivées par le fait que les utilisateurs n'arrivent pas forcément à spécifier précisément et aisément leurs exigences dès le début. Après avoir retourné un résultat par rapport à la requête initiale, l'utilisateur peut fournir une critique quant à la perspicacité du résultat. L'utilisateur spécifie les attributs qu'il souhaiterait changer sur un certain nombre d'*items* qu'il apprécie. Les critiques peuvent être de type directionnel (p. ex. : *moins chère*) ou de remplacement (p. ex. : *une autre couleur*). Dans l'approche directionnelle, l'utilisateur peut spécifier une augmentation ou une diminution de la valeur d'un certain attribut. L'avantage de cette approche est que l'utilisateur peut naviguer à travers l'espace d'*item* sans avoir à spécifier précisément le changement de valeur. Un autre avantage est l'intuitivité et la facilité d'utilisation de l'interface. Il existe trois différents types de critiques selon le nombre d'attribut qui est changé par itération.

Critiques simples

Dans le cas des critiques simples, l'utilisateur a la possibilité de ne modifier qu'un seul attribut à chaque itération. Le désavantage principal de l'approche des critiques simples est sa navigation laborieuse. Si l'*item* recommandé contient beaucoup d'attributs que l'utilisateur aimerait changer, cela va le conduire à une longue suite de critiques simples. Ceci car il ne peut changer qu'un attribut à la fois. De plus, les attributs peuvent varier selon les *items* disponibles. Dans la plupart des cas, il n'est pas possible de garder la valeur des autres attributs exactement constante d'une itération à l'autre. Lorsque l'utilisateur aura changé quelques attributs, il risque de se rendre compte que de précédents attributs ne sont plus disponibles. Plus l'utilisateur effectuera d'itération, moins il aura d'accès aux attributs qui étaient présents dans les précédentes itérations. Par conséquent, de nombreuses itérations de critiques simples conduisent parfois à un résultat infructueux.

Critiques composées

Les critiques composées ont été développées afin de réduire le nombre d'itération. Dans ce cas, l'utilisateur peut spécifier plusieurs attributs qu'il souhaite modifier dans une même itération. À la place de laisser l'utilisateur spécifier exactement les attributs qu'il souhaite modifier et dans quelle proportion, les interfaces utilisent généralement les critiques de type conversationnelles. Celles-ci se présentent sous une forme semblable aux critiques bidirectionnelles à la différence que celles-ci peuvent combiner plusieurs attributs. Par exemple, une critique pourrait s'appeler *moins chère* et dirigerait uniquement *le prix* vers le bas mais il pourrait aussi y avoir la critique s'appelant *plus de pièces* qui modifierait les attributs *nombre de chambres* et *nombre de salles de bains* vers le haut. Le but recherché est de permettre à l'utilisateur d'exprimer plus facilement ses désirs en termes d'attributs. Ce processus itératif est conçu afin d'aider les utilisateurs à naviguer de façon intuitive dans un espace complexe d'*items*.

Critiques dynamiques

Bien que les critiques composées permettent des sauts plus grands à travers l'espace d'*item*, il y a toujours le problème que les options de critiques présentées à l'utilisateur sont statiques. Dans le sens qu'elles n'évoluent pas selon le résultat. Par exemple, si la recherche de l'utilisateur ressort déjà les *items* extrêmes dans certains attributs, l'utilisateur aura toujours la possibilité de demander des *items* avec une valeur encore plus extrême dans ces attributs. Ceci alors qu'aucun *item* ne sera disponible. Ce qui résultera en une recherche non-fructueuse.

Le but des critiques dynamiques est de contrer ce problème en explorant les *items* résultants de chaque itération afin de déterminer les directions de recherche les plus prometteuses. Les critiques dynamiques sont, par définition, des critiques composées car elles permettent à l'utilisateur de naviguer par le biais d'options combinant le changement de plusieurs attributs. La différence principale est qu'uniquement un sous-ensemble des options, les plus pertinentes, est présenté à l'utilisateur.

Afin de continuer, deux nouveaux concepts doivent être introduits. Celui des *transactions* et celui du *support*. Une *transaction* est un sous-ensemble d'*item* résultant d'une certaine combinaison d'options. L'ensemble de *transactions* T est composé de toutes les *transactions* possibles par rapport aux différentes combinaisons d'options. Le *support* d'un ensemble d'*item* $X \subseteq I$ (I étant l'ensemble de tous les *items*) est la fraction de *transactions* dans T dans lesquels le sous-ensemble X est présent. Si T est constitué d'un total de 10 *transactions* et que 4 d'entre elles contiennent le sous-ensemble X, le *support* est de $\frac{4}{10} = 40\%$.

Afin de déterminer l'ensemble d'options qui nous intéresse pour les critiques dynamiques, les *supports* des différentes options vont être comparés à un seuil de *support* minimum qu'ils doivent dépasser afin de pouvoir être présentés à l'utilisateur. Par exemple :

Option 1 : $plus\ de\ chambre\ et\ un\ prix\ plus\ élevé: support = 25\%$
Option 2 : $plus\ de\ chambres, plus\ de\ salles\ de\ bain\ et\ un\ prix\ plus\ élevé: support = 20\%$
Option 3 : $moins\ de\ chambre\ et\ un\ prix\ plus\ bas: support = 20\%$

L'option $plus\ de\ chambre\ et\ un\ prix\ plus\ bas$ sera probablement éliminée car son *support* risque d'être en dessous du seuil minimum. Les options ayant un faible *support* ne sont cependant pas inintéressantes. De nombreux systèmes de recommandations trient les options passant le seuil par ordre croissant du niveau de *support*. Ceci dans le but de mettre en avant des options, à priori, moins évidentes et ainsi, potentiellement, éliminer un ensemble plus grand d'*items* candidats. Une observation importante à propos des critiques dynamiques est que chaque *itération* nécessite plus de travail et donc de temps que dans le cas des critiques composées. Mais ceci est compensé par le fait que le nombre d'itérations est réduit grâce à la pertinence des options. Ainsi la quantité de travail et de temps total, sur l'entier de la recherche, est diminuée.

4.3.3 Persistance dans les systèmes basés sur la connaissance

Le désavantage principal des systèmes de recommandation basé sur la connaissance est que les préférences utilisateurs, les caractéristiques ou les attributs démographiques sont spécifiques à la session et ne sont pas persistants.

Le manque de persistance est une conséquence naturelle de la manière très limitée d'utiliser les données historiques de la part des systèmes basés sur la connaissance contrairement à ceux basés sur le contenu ou les systèmes collaboratifs. Il s'agit cependant d'un avantage lors des départs à froids, lorsqu'on a très peu (ou pas du tout) d'informations sur l'utilisateur. Les systèmes basés sur la connaissance sont principalement utilisés dans les cas d'*item* relativement cher, acheté occasionnellement ou qui peuvent être hautement personnalisés. Les données personnelles historiques doivent donc être utilisées prudemment.

Cependant, certains systèmes de recommandations basés sur la connaissance ont été conceptualisé afin d'inclure une certaine forme de personnalisation persistante. Les actions de l'utilisateur à travers les différentes sessions peuvent être utilisées afin d'établir un profil persistant selon ce qu'il aime et ce qu'il n'aime pas. Le processus résultant fonctionne en deux étapes : dans la première étape les résultats sont récupérés par rapport aux exigences de l'utilisateur, comme dans n'importe quel système basé sur la connaissance. Et, dans un second temps, les résultats sont triés selon leurs similitudes aux *items* que l'utilisateur a aimés.

Il est également possible d'inclure des informations collaboratives en identifiant des utilisateurs avec des profils similaires et en utilisant leurs informations de sessions. Des fonctions d'utilité ou de similarité par rapport aux attributs peuvent autant être utilisés dans les systèmes basés sur la contrainte, durant la phase de classement, ou dans les systèmes basés sur le cas, durant la phase de recherche. Lorsqu'un utilisateur a précédemment donné son avis, il est possible d'en déduire l'importance relative de certains attributs pour cet utilisateur. Les critiques dynamiques peuvent être personnalisées lorsque suffisamment d'informations sur l'utilisateur sont disponibles. Les options des critiques sont alors influencées par les données spécifiques à l'utilisateur et non pas sur l'entier des données. Bien qu'il y ait plusieurs manières d'incorporer une personnalisation à travers les sessions utilisateur, le problème principal réside dans l'indisponibilité d'une quantité suffisante d'informations sur l'utilisateur.

4.4 Systèmes de recommandation hybrides

« Within an ensemble, like in sports, if you do your little job, it makes everybody look good. »

- *Stephen Hunter*

Les trois systèmes précédemment étudiés utilisent différentes sources d'entrées et ont différentes forces et faiblesses. Dans beaucoup de cas, lorsqu'une large variété d'entrées est disponible, il est possible de combiner différents types de systèmes de recommandation afin d'arriver à une solution s'inspirant du meilleur de chacun et, ainsi, de ne pas perdre une source de données. Les systèmes de recommandation hybride sont liés de près à l'analyse d'ensemble dans lequel la puissance de plusieurs algorithmes de *machine learning* sont combinés afin de créer un modèle encore plus puissant.

Il existe trois grandes catégories de systèmes de recommandation hybride : la première est celle basée sur un ensemble de systèmes, la seconde correspond à celle basée sur un concept monolithique puis, la troisième et dernière catégorie fonctionne sur un concept mixte.

4.4.1 Ensemble

La conception d'un système de recommandation basé sur un ensemble consiste à développer plusieurs systèmes distincts dont les résultats sont agrégés afin d'obtenir le résultat final. Chaque système calcule les valeurs manquantes de la matrice de classement selon ses propres caractéristiques puis, en combinant les résultats de chacune des matrices, une matrice finale est calculée. Les méthodes pour combiner les résultats peuvent être divers et varient selon le type de valeur (unaire, binaire, numérique, …). Ces méthodes peuvent être, par exemple, la moyenne des valeurs ou l'utilisation de la valeur majoritaire.

4.4.2 Monolithique

Les systèmes monolithiques sont construits en adaptant les algorithmes de plusieurs systèmes afin de n'en former qu'un seul. Une claire distinction entre les algorithmes de base n'existe plus forcément et ceux-ci nécessite d'être modifiés par rapport à leurs formes originales.

4.4.3 Mixte

De la même manière que les systèmes de recommandation basés sur un ensemble, les systèmes mixtes utilisent différents algorithmes de recommandation distincts pour prédire les *items* à recommander. Mais, au contraire de ceux basés sur un ensemble, ceux-ci utilisent les recommandations faites par chacun des systèmes afin de créer une liste mixte d'*item* qui sont proposés à l'utilisateur. Pour résumer, les top-*k* d'*item* de chacun des systèmes de recommandation sélectionnés seront combiné afin de générer un top-*k* final.

4.5 Système de recommandation envisagé pour MyFlat

MyFlat étant une plateforme immobilière, ses *items* sont des publications d'offre de logements par les utilisateurs. Les utilisateurs peuvent être distingués en deux catégories : les visiteurs et les annonceurs. Les utilisateurs cible du système de recommandation sont ceux recherchant un logement, autrement dit, les visiteurs. Être un annonceur n'empêchant pas d'être également un visiteur, il est possible qu'un annonceur soit cible du système de recommandation par le biais de son caractère visiteur. D'après plusieurs enquêtes, dont celle du journal *le temps*, la fréquence de déménagement est en hausse en Suisse. Sur l'année 2015, 12% des résidents suisses ont changé de logement[6]. Malgré ces chiffres élevés, il va être difficile de collecter des informations de manière fréquente et importantes sur l'utilisateur.

Les visiteurs vont avoir tendance à se rendre régulièrement sur la plateforme durant une certaine période, celle de la recherche de leur nouveau logement, puis, ne plus la fréquenter durant une très longue période, jusqu'à leur prochain déménagement. Par conséquent, très peu d'informations sont disponibles sur le visiteur et, de plus, les informations récoltées d'un déménagement à l'autre risquent de perdre en pertinence. Il va donc falloir que le système

[6] Article du journal *le temps* : https://www.letemps.ch/economie/suisses-nont-jamais-autant-demenage

prenne en compte les périodes de fréquentations de l'utilisateur afin de donner plus de poids aux récentes informations qu'à celles des précédents déménagements.

Le peu d'information collectable sur l'utilisateur et les fortes différences dans les attributs des *items* ouvrent la porte aux systèmes de recommandation basés sur la connaissance. Entre un système basé sur la contrainte et un basé sur le cas, le choix d'un système de recommandation basé sur la contrainte semble plus adapté. La raison principale est qu'il s'agit du type de système le plus communément utilisé sur les autres plateforme web, telles que *hotel.com* ou *skyscanner.net*, et a donc l'avantage d'être connu de la plupart des utilisateurs, leur évitant ainsi un effort d'apprentissage. Ce système sera utilisé afin de fournir des résultats directement à l'utilisateur selon ces critères.

En plus d'afficher des résultats selon les critères de l'utilisateur, le souhait de MyFlat est de pouvoir fournir à l'utilisateur des offres alternatives auxquelles il n'aurait pas forcément pensé lors de ses recherches. Ces offres alternatives seront déterminées par le contenu des offres visualisées par l'utilisateur et ses éventuels favoris. MyFlat inclut donc un second système basé sur le contenu.

4.5.1 Système basé sur la contrainte

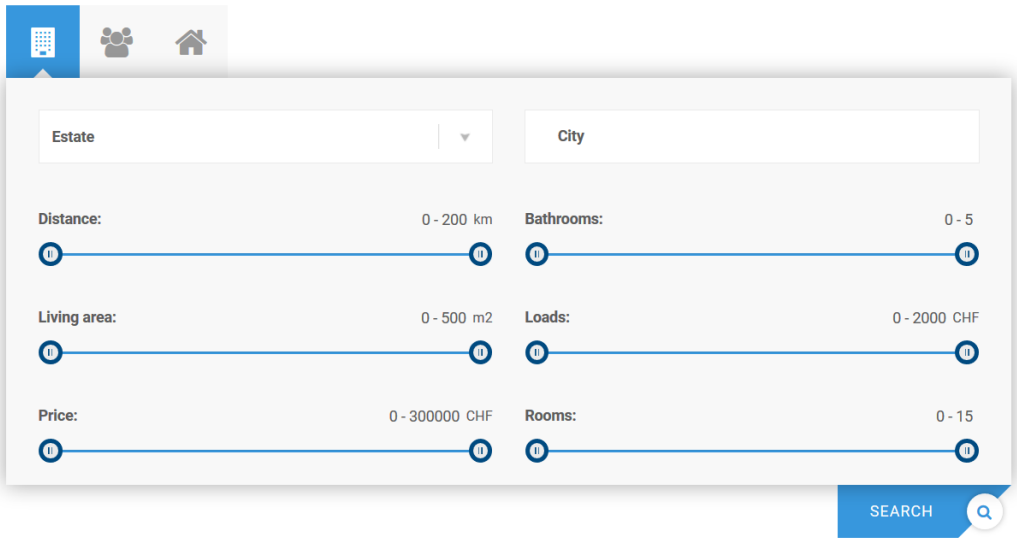

Figure 4.24: outil de recherche de MyFlat

Ce système permet à l'utilisateur de fournir les limites de ses différents critères sur le logement qu'il recherche. Par exemple, le minimum et le maximum de chambre ou la catégorie de prix. Cette partie du système est déjà implémentée et fonctionnelle sur MyFlat. La seconde étape, celle du classement des résultats, doit, elle, être améliorée afin d'augmenter la pertinence du tri. Lorsque l'utilisateur choisit de trier les offres par pertinence (tri par défaut), les *items* doivent être classés selon une fonction de similarité entre les critères de l'utilisateur et chacune des offres répondant à ces critères.

Recherche

Les attributs, appelées *properties* dans le cas de MyFlat, sont enregistrés dans la base de données. Cela a pour conséquence et avantage de pouvoir modeler l'outil de recherche (cf.

figure 4.24) très facilement. De plus, chaque offre a un attribut *properties* contenant tous les attributs dynamiques que l'annonceur lui a spécifié. Il ne reste donc qu'à appliquer une simple comparaison entre les valeurs des *properties* de l'outil de recherche et ceux des offres.

Le travail est majoritairement effectué dans le conteneur de la page affichant la liste des offres, *listingContainer*. L'outil de recherche ne fait que rediriger sur la page des offres en ajoutant les attributs de la recherche dans l'URL. Ci-dessous, un exemple d'URL :

```
http://myflat.chaubi.ch/listing?estate[_id]=flat&estate[tit
le]=Flat&title=Fribourg%2C%20Suisse&coordinates[0]=7.161971
900000026&coordinates[1]=46.8064773&properties[baths][min]=
0&properties[baths][max]=5&properties[livingarea][min]=0&pr
operties[livingarea][max]=500&properties[loads][min]=0&prop
erties[loads][max]=2000&properties[price][min]=0&properties
[price][max]=300000&properties[rooms][min]=0&properties[roo
ms][max]=15&distance[min]=0&distance[max]=200&transaction[_
    id]=rent&transaction[title]=Rent&orderby=relevance
```

Le conteneur récupère les données de l'URL et le transforme en JSON grâce au paquet *qs*. Puis, il construit les paramètres de la requête et la transmet à la publication *offers.allByXY*.

Le code, nous intéressant, du conteneur est le suivant (code 4.11) :

import/ui/containers/ListingContainer.jsx

```
const search = qs.parse(location.search.slice(1));
let params = {}

if(search.distance && search.coordinates && search.estate &&
search.transaction){
   params["$and"] = [
                     {"estate._id" : search.estate._id},
                     {"transaction._id" : search.transaction._id}
                ]
   params["coordinates"] = {
     "$near" : {
       "$geometry" : { "type" : "Point", "coordinates" :
[parseFloat(search.coordinates[0]), parseFloat(search.coordinates[1])]},
       "$minDistance" : parseInt(search.distance.min)*1000,
       "$maxDistance" : parseInt(search.distance.max)*1000
     }
   }
   Object.keys(search.properties).forEach((key) => {
     const property = search.properties[key];
     const constraint = '{"properties.'+key+'.value" : {"$gte" :
'+property.min+', "$lte" : '+property.max+'}}'
     params["$and"].push(JSON.parse(constraint))
   })
}

const subOffersWhereXY = Meteor.subscribe('offers.allByXY', params,
orderby.get(), requestedOffers.get());
```

Code 4.11 : recherche des offres

Le code 4.11 prépare les paramètres de la requête en deux étapes : la première s'occupe des attributs statiques tandis que la seconde de ceux dynamiques.

Les paramètres statiques sont l'*estate*, la *transaction* et la géolocalisation. Les deux premiers sont simplement combinés à la requête grâce au mot-clé *$and*. Ils doivent être traités indépendamment des attributs dynamiques car ils ne respectent pas le même format. La géolocalisation utilise un outil de *MongoDB* permettant d'ajouter une condition de distance par rapport à une coordonnée. Cet outil s'appelle *2dsphere*[7].

Les paramètres dynamiques sont préformatés pour qu'ils puissent tous être facilement traitable de la même manière. Ils sont, comme pour les paramètres *estate* et *transaction*, combinés à la requête grâce au mot-clé *$and*.

Lorsque les paramètres sont prêts, le conteneur s'inscrit à la publication *offers.allByXY*. Cette dernière effectue une simple requête *find(...)* vers la base de données. Le code de la publication est le suivant (code 4.12) :

import/api/offers/server/publications.jsx

```
const MAX_OFFERS = 150;

Meteor.publish('offers.allByXY', function (params, orderby, limit) {
  params["online"] = true
  return Offers.find(params, {sort: orderby, limit:Math.min(limit,
MAX_OFFERS)});
});
```

Code 4.12 : publication *offers.allByXY*

Tri

L'étape précédente, celle de la recherche, retourne une liste d'offres répondant aux critères de l'utilisateur. Or, celle-ci n'est pas triée. Il se peut que l'offre la plus pertinente se trouve tout en bas de la liste et, ainsi, l'utilisateur risque de la manquer s'il ne visualise pas toutes les offres. C'est pourquoi il faut trier les offres selon leur pertinence par rapport à la recherche de l'utilisateur.

Contrairement à l'étape précédente, celle-ci reste à implémenter.

Comme appris dans le chapitre 4.3.1, la manière de trier se fait par le biais d'une fonction d'utilité. Un second avantage d'avoir les *properties* (ou attributs des offres) enregistrés dans la base de données est de permettre d'attribuer un poids et une fonction à chaque attribut.

Ces poids et fonctions seront déterminés grâce aux résultats d'une l'analyse conjointe sur un ensemble d'utilisateurs volontaires. Cette étape demande un investissement mais ne devrait être effectuée que très rarement, lors d'ajout de nouveaux attributs. Afin d'harmoniser ces poids et fonctions, on définit qu'ils doivent, tous deux, avoir une valeur entre 0 et 1. Et la fonction doit être écrite en *Javascript*.

Les poids et fonctions des attributs vont être ajoutés à la collection *properties*. Cette dernière évoluera de la manière suivante (cf. figure 4.25) :

[7] Plus d'informations sur l'outil *2dsphere* à l'adresse suivante : https://docs.mongodb.com/manual/core/2dsphere/

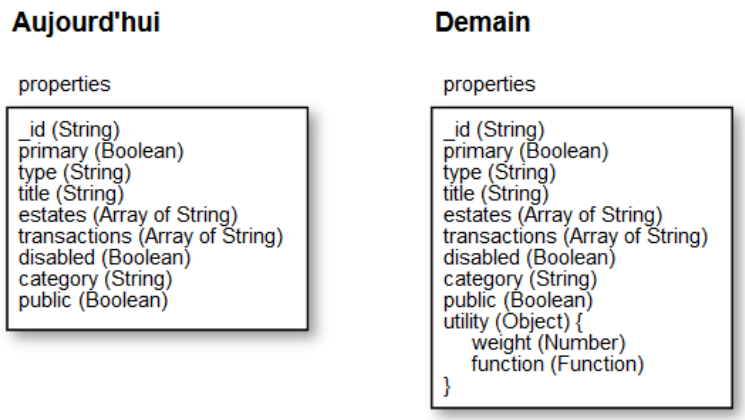

Figure 4.25: évolution de la collection *properties*

où la collection **Aujourd'hui** représente celle qui est actuellement utilisée par MyFlat et la collection **Demain** est celle contenant les modifications nécessaires au tri.

En prenant l'exemple de l'attribut prix, voici à quoi ressemble le document actuel (code 4.13) et à quoi pourrait ressembler le document une fois adapté (code 4.14) :

Aujourd'hui

```
{
    "_id" : "price",
    "primary" : true,
    "mainSearch" : true,
    "type" : "number",
    "min" : NumberInt(0),
    "max" : NumberInt(300000),
    "title" : "Price",
    "disabled" : false,
    "unit" : "CHF",
    "category" : "general"
}
```

Code 4.13 : *property price* aujourd'hui

Demain

```
{
    "_id" : "price",
    "primary" : true,
    "mainSearch" : true,
    "type" : "number",
    "min" : NumberInt(0),
    "max" : NumberInt(300000),
    "title" : "Price",
    "disabled" : false,
    "unit" : "CHF",
    "category" : "general"
    "utility" : {
       "weight" : NumberInt(1),
       "function" : function
         utility(value, max, min) {
            return (1-(value/(max - min)))
         }
    }
}
```

Code 4.14 : *property price* demain

Lorsque chaque attribut a son poids et sa fonction, il faut, après avoir obtenu la liste des offres respectant les critères utilisateurs, appliquer la formule suivante sur chaque offre de cette liste:

$$U(\boldsymbol{offer}) = \sum_{j=1}^{d} weight_j * utility(value_j)$$

où l'offre contient *d* attributs et la fonction d'utilité est écrite de façon générale (ne pas se référer à l'exemple 4.14).

Les résultats sont ensuite triés selon leur niveau d'utilité et présentés selon cet ordre à l'utilisateur.

4.5.2 Système basé sur le contenu

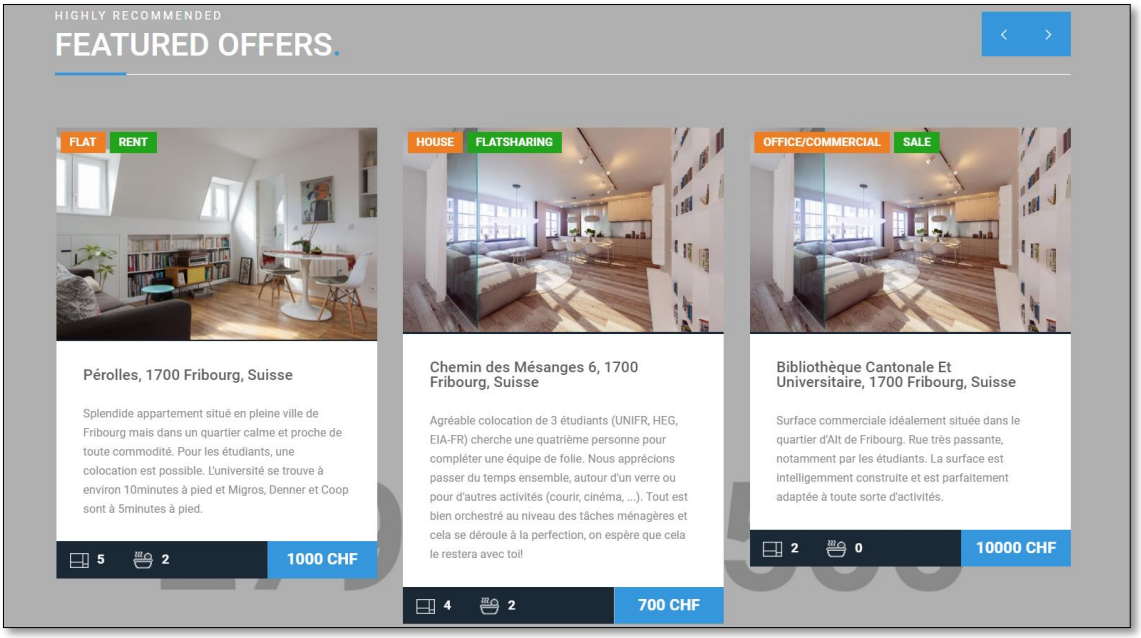

Figure 4.26: offres recommandées – page d'accueil

Le système basé sur le contenu affichera ces résultats sur la page d'accueil ainsi que dans la barre latérale figurant sur la page de la liste des offres. Ce système fonctionne, à ce jour, de façon très rudimentaire : il affiche les offres géolocalisées proches de la dernière recherche de l'utilisateur. L'idée est de collecter des informations sur l'utilisateur lorsqu'il visualise des offres ou en ajoute dans ses favoris.

De manière générale, les informations collectées seront enregistrées dans les cookies. Et, si cet utilisateur est connecté, elles seront également enregistrées dans la base de données. Ceci afin de leur offrir une meilleure capacité de survie et de persistance puisque l'utilisateur peut, à tout moment, vider ses cookies ou changer de navigateur ou d'ordinateur. Le cookie, ayant une durée de vie limitée, règle par lui-même le problème des périodes de fréquentations. Les informations collectées et enregistrées dans la base de données devront, quant à elles, être datées afin de ne pas les utiliser si elles datent d'il y a plus de 6 mois. Ces informations sont les identifiants uniques des offres. L'évaluation est collectée de manière implicite et est de type unaire puisque l'utilisateur ne peut signifier son intérêt que de manière positive.

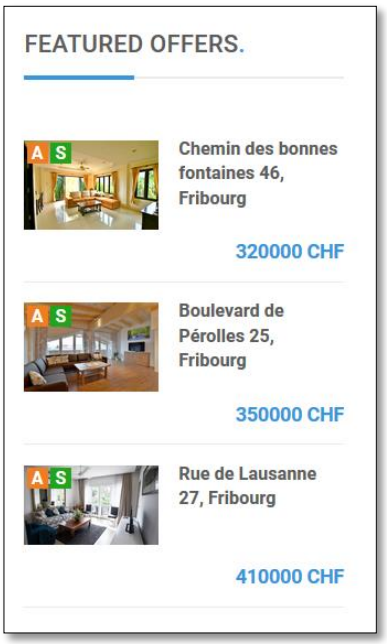

Figure 4.27: offres recommandées - barre latérale

Pour rappel, comme expliqué dans le chapitre 4.2 sur les systèmes de recommandation basés sur le contenu, le travail se fait en trois étapes : (1) le prétraitement et extraction des attributs, (2) l'apprentissage des profils utilisateurs basés sur le contenu, (3) le filtrage et les recommandations.

Il existe de nombreux paquets existants et fonctionnels qui peuvent être utilisés pour ce travail. Notamment le paquet *npm* intitulé *Content Based Recommender*[8] de Stanley Fok qui semble être parfaitement adapté à la structure de MyFlat et très facile d'utilisation.

Il suffit, dans un premier temps, d'entrainer le système de manière asynchrone sur toutes les offres. Le processus d'entrainement du paquet *Content Based Recommender* fonctionne en trois étapes qui sont, majoritairement, expliquées dans le chapitre 4.2.1 :

1. Prétraitement du contenu : par exemple, la suppression des tags *HTML*, des *stopwords* et le *stemming*.
2. Formation des vecteurs de document grâce à TF-IDF
3. Recherche des scores de similarité, avec la fonction du cosinus, entre tous les vecteurs de document

Une fois le système entrainé, il suffit de lui fournir l'identifiant d'une offre pour laquelle on souhaite des offres comparables, ainsi que le nombre d'offre que l'on souhaite recommander pour qu'il nous retourne les identifiants et les scores de chaque offre jugée similaire.

Puisque MyFlat collectera une liste d'offres par rapport à l'utilisateur (selon les offres qu'il visite et celles de ses favoris), il sera possible de mixer ces résultats de plusieurs manières et, ainsi, éviter de lui présenter toujours les mêmes offres. MyFlat va alors constituer un *top-k* d'offres similaires aux offres que l'utilisateur a visitées ou favorisées dont il affichera les résultats sur la page d'accueil et la barre latérale.

Illustration d'une recommandation sur MyFlat :

La figure 4.28, ci-contre, illustre l'ensemble des offres disponibles sur MyFlat. Les boxes vertes représentent les offres que l'utilisateur a favorisées et, les flèches, le *top-3* des offres similaires avec leur coefficient de similarité. Ici, le système a donc à disposition un *top-4* d'offres. Il est facilement imaginable que ce top s'agrandisse plus on obtient d'informations sur l'utilisateur ce qui permet de varier les offres affichées. Ce *top-4* va ensuite être trié et, si nécessaire, filtré selon leur coefficient puis, proposé à l'utilisateur. La figure 4.28, offrirait à l'utilisateur un top constitué des offres suivantes : (1) 44523, (2) 94834, (3) 39483 et (4) 20392.

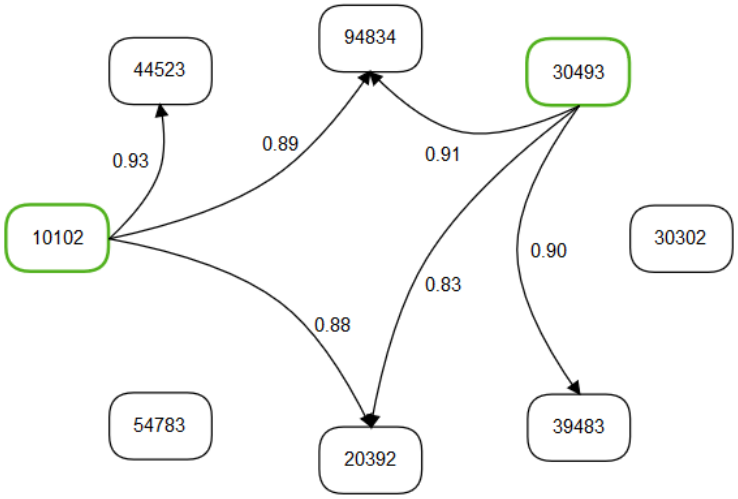

Figure 4.28: exemple de similarité entre offres

[8] Plus d'informations sur le paquet *Content Based Recommender* à l'adresse suivante : https://www.npmjs.com/package/content-based-recommender

5 Conclusion

La rédaction de ce rapport ainsi que le travail et les recherches qui l'ont précédé furent un grand enrichissement intellectuel et personnel.

La découverte des différents systèmes de recommandation a alimenté mon intérêt pour les sciences des données. Il est important de connaître tous ces systèmes, leurs cas d'utilisation ainsi que leurs avantages et inconvénients dans le but de pouvoir appliquer le système de recommandation le plus adapté à la situation. Il faut également garder en tête que chacun de ces systèmes n'est pas aussi rigide qu'il n'y parait et qu'il est parfois sage de combiner leurs forces afin d'augmenter la pertinence des résultats. De la perspective d'un utilisateur quasi-quotidien de systèmes de recommandation, il est également intéressant d'être capable de discerner pourquoi un tel système nous recommande un tel *item*.

Un second aspect intéressant découle du travail d'équipe. Une bonne organisation et une bonne communication est la clé d'un travail de groupe réussi. Ces derniers mois à travailler sur MyFlat nous a, à tous, permis de nous améliorer dans ce domaine. J'ai pu observer qu'il est important de comprendre et connaître les besoins, limites et contraintes de chacun afin de pouvoir s'y adapter et inclure dans le projet les visions de chacun. Le travail d'équipe est d'autant plus important qu'il s'agit de la réalité de tous les jours et un élément central dans la vie professionnelle.

MyFlat est un projet à long terme. Des nombreuses fonctionnalités et améliorations restent à implémenter ce qui rend le projet d'autant plus attrayant car on le voit s'améliorer et grandir en s'adaptant de mieux en mieux aux utilisateurs. Les prochaines grandes étapes seront les fonctionnalités payantes ainsi que l'amélioration du système de recommandation.

Grâce à ce travail, mon intérêt pour la science des données (*data science*) s'est confirmé. Je projette d'ailleurs d'en faire mon domaine d'étude pour le Master. La combinaison de l'informatique et des statistiques au travers des données offre d'extraordinaires applications potentielles autant dans le domaine du web que dans celui de la finance ou du médical.

A

Code Source

Le code source est disponible au téléchargement à l'adresse suivante :

https://drive.google.com/file/d/1GkJFidWTxnWzzmXMWE0R5KunKT1EbqM-/view?usp=sharing

Cette adresse sera accessible pour une durée minimale de trois mois à compter de la date de remise du travail final. Un accès ultérieur au code source peut être envisagé après une prise de contact personnel.

Le code source propre au projet MyFlat, contenu principalement dans les dossiers et fichiers suivants :
.deploy, both, client, imports, private, server, package.json, package-lock,json
a été analysé, à l'aide de l'outil CLOC, et fournit le résultat suivant :

Fichiers	Lignes vides	Lignes de commentaires	Lignes de code
111	1181	586	13862

Le fichier README.md, contenant les informations minimales sur le projet, est disponible, en anglais, à l'adresse suivante :

https://github.com/citronpower/myflat

B

Site web du projet

La plateforme est actuellement (le 20 août 2018) accessible à l'adresse suivante :

`http://myflat.chaubi.ch/`

Cette adresse n'étant pas l'adresse officielle (`http://my-flat.ch/`), il se peut que celle-ci ne soit pas toujours active ou souffre de certains problèmes (pas de certificat SSL par exemple).

Depuis le 16 juillet 2018, Google Maps a changé sa politique de facturation. Cela a une conséquence directe sur le fonctionnement de MyFlat qui en dépend fortement. Désormais, même les utilisateurs n'effectuant qu'un nombre très limité de requêtes vers l'*API* doivent payer. Une solution temporaire est en place depuis le 23 août 2018. Mais, il est toutefois possible que des problèmes liés à Google Maps surviennent.

Références

[Agg16]

Charu C. Aggarwal. *Recommender Systems: The Textbook*, Springer, 2016, ISBN 9783319296593.

[And03]

Chris Anderson. *The Long Tail: Why the Future of Business is Selling Less of More*. Hyperion, 2006, ISBN 9781401387259.

[Bis06]

Christopher M. Bishop. *Pattern recognition and machine learning*, Springer, 2006, ISBN 0387319738.

[Bou14]

Sarah Bouraga, Ivan Jurata, Stéphane Faulkner et Caroline Herssens. *Knowledge-based Recommendation Systems: A Survey*, University of Namur, 2014. [Obtenu le 04 mai 2018, de https://www.researchgate.net/profile/Sarah_Bouraga/publication/287729916_Knowledge-based_recommendation_systemsA_survey/links/568e2bda08ae78cc0514b8aa/Knowledge-based-recommendation-systemsA-survey.pdf?origin=publication_detail].

[Bur00]

Robin Burke. *Knowledge-based recommender systems*, University of California, 2000. [Obtenu le 04 mai 2018, de https://pdfs.semanticscholar.org/1b50/ff4e5d8420f3ffae7de2a060b5a6fd4b8023.pdf].

[Kor08]

Yehuda Koren. *Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model*, AT&T Labs, 2008, ISBN 9781605581934.

[Loz15]

Esther Lozano, Jorge Gracia, Diego Collarana, Oscar Corcho, Asunción Cómez-Pérez, Boris Villazón, Sander Latour et Jochem Liem. *Model-based and memory-based collaborative filtering algorithms for complex knowledge models*, DynaLearn, 2015. [Obtenu le 06 juin 2018, de https://www.researchgate.net/profile/Jochem_Liem/publicatio

n/265111575_Model-based_and_memory-based_collaborative_filtering_algorithms_for_complex_knowledge_models/links/540989820cf2718acd3d5242/Model-based-and-memory-based-collaborative-filtering-algorithms-for-complex-knowledge-models.pdf].

[Mur12]

Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*, The MIT Press, 2012, ISBN 0262018020.

[Paz97]

Micheal Pazzani et Daniel Billsus. *Learning and Revising User Profiles: The Identification of Interesting Web Sites*, University of California, 1997. [Obtenu le 12 juin 2018, de https://link.springer.com/content/pdf/10.1023%2FA%3A1007369909943.pdf].

[Xia17]

Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu et Tat-Seng Chua. *Neural Collaborative Filtering*, National University of Singapore, Columbia University, Shandong University, Texas A&M University, 2017. [Obtenu le 13 juillet 2018, de https://arxiv.org/pdf/1708.05031.pdf].

Ressources web

[1] Atlassian: *What's a knowledge base and why you need it* https://www.atlassian.com/it-unplugged/knowledge-management/what-is-a-knowledge-base (dernier accès, le 20 août 2018).

[2] IBM: *Introduction to approaches and algorithms*. https://www.ibm.com/developerworks/library/os-recommender1/ (dernier accès, le 20 août 2018).

[3] Meteor : https://www.meteor.com/developers (dernier accès, le 20 août 2018)

[4] MongoDB : https://www.mongodb.com/ (dernier accès, le 20 août 2018)

[5] ReactJS : https://reactjs.org/ (dernier accès, le 20 août 2018)

[6] Recommender Systems: http://recommender-systems.org/ (dernier accès, le 20 août 2018)

[7] Statsbot : *Recommendation System Algorithms.* https://blog.statsbot.co/recommendation-system-algorithms-ba67f39ac9a3 (dernier accès, le 20 août 2018)

[8] Toptal : *Predicting Likes : Inside A Simple Recommendation Engine's Algorithms*. https://www.toptal.com/algorithms/predicting-likes-inside-a-simple-recommendation-engine (dernier accès, le 20 août 2018).

Index

S

V