**Data Mining - Rapport de projet**

## But du projet

Dans le cadre du module « Data Mining – Machine Learning » du cursus Science du Numérique à CPE LYON, nous avons moose en place un système de recommandation d’images.

L’objectif était de nous faire manipuler les différents concepts étudiés pendant le module, dans des situations pratique. Ce document contient un résumé de ce projet.

## Cahier des Charges

Pour ce projet des contraintes ont été définie dans le cahier des charges. La première est sur le langage à utiliser, à savoir du Python.

Notre solution devra être composé de 3 parties, la première est la phase de collecte des données. Celle-ci permettra la récupération d’une centaine d’images, de manière automatique et libre de droit. Dans un deuxième temps, il y aura une phase d’étiquetage et d’analyse des images. Différentes données comme la couleur, les Exifs ou le format seront à renseigner dans un fichier JSON. Enfin, la solution devra fournir une fonction pour visualiser les images sauvées et une autre pour le système de recommandation.

## Choix de l’équipe

Le cahier des charges n’ayant pas définie de manière de visualiser et interagir avec l’utilisateur. Nous avons fait le choix de partir sur l’utilisation d’un serveur TCP qui utilise des sockets http (à l’aide du module *http.server*).

Le problème est que nous avons rencontrés des problèmes de compatibilité entre les bibliothèques. Ainsi, nous avons le choix d’utiliser une fonction *Main* dans le terminal qui va créer un serveur web et attendre les actions de l’utilisateurs. Celui-ci ouvrira des pages web générées en Python mais avec une surcouche de HTML pour la mise en forme.

## Structure du Projet

Pour ce projet, nous avons développé deux types de script. Les premiers que l’on pourrait catégoriser comme « Back-end » sont des fonctions qui nécessite un appel dans le terminal. Les fonctions vont s’exécutées et ne nécessite pas d’entrées de l’utilisateur. Dans cette catégorie, nous avons les scripts de gestion du serveur http, de récupération des images, etc.

Le second type de script, est celui que l’on peut appeler « front-end ». Ils vont être exécuté via le navigateur de l’utilisateur et attende des entrées de l’utilisateur via des formulaires HTML.

## Partie Collecte des données Source

### Des données, Des images et leurs licences

Notre projet se base sur l’utilisation de la base de connaissance libre Wikidata[[1]](#footnote-1). Les données et images issue de cette base, sont sous la licence CC0[[2]](#footnote-2) donc libres de droit (cf. condition d’utilisation[[3]](#footnote-3)).

Wikidata met à disposition ses données via une API qui utilise le langage de requête SPARQL[[4]](#footnote-4). Celui-ci va nous contraindre à utiliser la bibliothèque *qwikidata.sparql* et plus spécifiquement la *fonction return\_sparql\_query\_results*. Par ailleurs, la notion de base collaborative va créer des disparités dans les informations. Par exemple, les Exif Tags des images sont hétérogènes donc cela amène une complexité supplémentaire avec un traitement supplémentaire pour les harmoniser.

Nous avons par ailleurs choisi de cibler plusieurs grands thèmes afin de garantir l’existence d’images. Ces thèmes sont : chat, montagne, plante, peinture, manga, homme, chien, sport, film, art.

### Des images trop volumineuses

Quand le script récupère les images via Wikidata, leurs poids va varier entre des centaines de kilo-octets et plusieurs méga-octets. Sur un échantillon de quelques images, l’algorithme de calcul des couleurs via la méthode des K-Means va pouvoir gérer le très grand nombre de pixels pour un élément. Le problème est qu’en l’appliquant sur des centaines d’images, il plantait. Dans la phase de pré-traitement, nous avons dû appliquer une compression et redimensionnement des images. Les deux fonctions (*resize* et *save*) sont issues de la bibliothèque PIL, nous avons choisie un redimensionnement de ¼ et une compression de 30%. Ces valeurs obtenues empiriquement permettent de réduire de presque 90% le poids d’une image sans une perte de qualité « critique ».

### Récupération des données

Ce script est composé d’une boucle principale qui va aléatoirement (utilisation de la bibliothèque *random*) prendre une paire de clés-valeur. Exemple : ["chat","Q146"], une liste avec *chat*, le thème des images et *Q146* le paramètre SPARQL.

La requête va être exécuté et l’API retourne un tableau contenant un lien pour télécharger les différentes images.

Le code va télécharger les images avec la bibliothèque *urllib*, avant de l’analyser. Si elle contient les informations nécessaires à la suite, elle est conservée dans un dictionnaire (détaillez partie 5.4).

A la fin de l’exécution, le dictionnaire est enregistré dans un fichier *data.json.*

Remarque, les anciennes images sont supprimées à l’exécution du script avec la bibliothèque *os*.

### Pré-Traitement

La phase de pré-traitement va être composé de deux étapes. La première est l’extraction des informations contenues directement dans l’image : les Exif Tags.

Pour notre part, nous avons donc choisi de nous focaliser sur certains, qui nous paraissaient pertinents pour notre projet :

* 270 : la description de l’image (et donc son titre, entres autres)
* 305 : le logiciel de traitement de l’image
* 306 : date et heure de la prise de vue
* 40962 : la largeur en pixel de l’image
* 40963 : la hauteur en pixels de l’image

Ceux en rouge sont « obligatoire », et en cas de manque : l’image n’est pas conservée. Les autres sont optionnels.

La seconde phase est l’ajout de tags supplémentaires. Une première fonction a pour but de déterminer la couleur prédominante de l’image via la méthode des K-means. La deuxième va déterminer le type d’images : une icône, une grande ou une très grande image et son orientation. Enfin, le nom et le thème sont ajouté pour que l’on obtienne un dictionnaire du type :

{"nom": "Images/plante1.jpg", "40962": "1359", "40963": "1494", "306": 2012, "theme": "plante", "orientation": "portrait", "couleur": " black", "type": "tres grande image"}

Le dictionnaire est ensuite ajouté à une liste globale contenant toutes les images.

### Informations concernant les préférences utilisateurs

Pour que notre système de recommandation fonctionne, il va devoir avoir en entrée des premières entrées.

Ainsi, un script va proposer un ensemble de 5 images que l’utilisateurs peut « *liker* » et demander son nom. Une fonction va récupérer les likes de l’utilisateurs pour ajouter les noms des images dans deux listes (like et unlike) en fonction de son choix.

Par ailleurs, d’autres données sont analysées, comme son thème ou son orientation préférée. Le tout va être enregistrer dans un fichier profil\_*username*.json, pour conserver ses préférences.

Une des options permet l’affichage des informations utilisateurs comme son thème, sa couleur ou son orientation préférée. La page affiche tous des listes des images like ou unlike.

### Modèles d’exploration de données et d’apprentissage machine utilisés

Notre système de recommandation va utiliser un classificateur binaire. Celui-ci se base sur le module *sklearn* et particulièrement *tree*. Ce script va attendre en entrée le profil d’un utilisateur et en particulier les listes des images likés ou non.

La fonction va créer deux nouvelles listes, la première qui contient toutes les images vues par l’utilisateur et la seconde le résultat (‘*Favorite’* ou ‘*NotFavorite’*).

Nous allons utiliser la technique d’étiquetage pour extraire 6 critères pour juger de la pertinence d’une image. Parmi eux, nous retrouvons des critères comme la date, le thème, la couleur, etc.

A noter que nous avons eu des problèmes avec les labels. En effet, ceux-ci n’acceptent pas de nouveaux labels, une fois convertie avec la fonction *transform*. Notre solution consiste à récupérer l’intégralité des labels des données (dont ceux pas encore traités). Cette solution a pour limite de faire charger de nombreux labels, jamais utilisés par l’algorithme.

Un arbre de décision est initialisé avec en entrée la liste des images et celle des résultats. Le script va sélectionner des images non-traité. Quand l’arbre estime qu’une image va avoir la classe « *Favoris* », elle est affichée à l’utilisateur sur une page HTML.

L’utilisateur va pouvoir confirmer ou non sa préférence pour l’image. Les listes *like* et *unlike* seront mise à jour en fonction.

Notre système ne possède pas de moyen d’apprentissage mais va s’améliorer avec les itérations. En effet, plus le nombre d’image traitée est important et plus l’arbre de décision possède de données pour « comprendre » les goûts de l’utilisateur.

Cette solution possède des cas favorables comme des listes qui « like » qu’un thème. Mais certains critères sont mal exploités comme les couleurs. L’algorithme va faire une forte distinction entre les nuances et deux couleurs proches ont des labels différents. Un cas défavorable est quand une des listes est vide car l’algorithme ne possèdera pas de points de comparaisons dans un cas.

## Visualisation des données

Nous avons décidé d’utiliser l’interface graphique permise par le HTML pour permettre à l’utilisateur d’afficher et de visualiser les données.

Nous avons dans un premier temps choisi de laisser la plus grande liberté possible à l’utilisateur : les différentes fonctions étaient préprogrammées et uniquement appelées lorsque l’utilisateur choisissait de visualiser une donnée précise.

En front-end, la proposition pour l’utilisateur était faite de telle manière à ce que chaque image soit proposée, et que, pour chaque image, l’utilisateur choisisse entre plusieurs options : **AJOUTER TOUTES LES OPTIONS.**

Les calculs et la mise en forme des données de sorte à pouvoir les visualiser se faisait alors au moment où l’utilisateur cliquait sur « Visualiser mes données ».

Cependant, cette stratégie a rencontré un problème qui faisait que l’appel avec le module *cgi* (en mode *sudo*) rencontrait une erreur au moment d’importer les différentes bibliothèques. Conscients que cela impliquait à l’utilisateur trop de dispositions à prendre de son côté, nous avons décidé de lui simplifier la tâche, au détriment cependant du temps d’extraction des images en amont.

En effet, nous avons donc, au moment où les images sont récupérées par un script sur Wikidata, mis en place un second script qui analyse toutes les images et sort différentes visualisations possibles grâce aux fonctions existantes. Le seul souci présent ici est que, pour une photo, les images permettant la visualisation prennent 30sc chacune à être générées : cela nous donne alors 50mn de traitement pour 100 images.

Ainsi, lorsque l’utilisateur voudra visualiser une donnée, elle sera déjà présente et stockées dans un dossier, et le script de frontend ne fera qu’aller chercher dans ce dossier pour afficher l’image relative aux demandes de l’utilisateur.

1. Source : <https://fr.wikipedia.org/wiki/Wikidata> [↑](#footnote-ref-1)
2. Source : <https://fr.wikipedia.org/wiki/Licence_CC0> [↑](#footnote-ref-2)
3. Source : <https://www.wikidata.org/wiki/Wikidata:Licensing> [↑](#footnote-ref-3)
4. Source : <https://fr.wikipedia.org/wiki/SPARQL> [↑](#footnote-ref-4)