COMPTE RENDU PROJET DATA MINING

ETAPE 1 : ACQUISITION DE DONNEES

L'objectif de cette première étape est d'obtenir un dossier « images » dans lequel se trouveront toutes les images que nous allons utiliser pour le projet.

Pour ce projet, j'utilise deux Datasets. Le premier est un Dataset composé d'images de pokemons en format png, c'est un dataset relativement petit. Le second dataset, plus consistent comporte des images de tous genres libres de droit. Pour l'instant on utilise le dataset pokemon pour tester si le code fonctionne. Ce dataset ayant un thème précise (pokemons), ça nous permet d'utiliser des tags connus (comme les types de pokemons), alors que dans le gros dataset, le choix de tags est plus complexe (voir partie tags)

En ayant cela en tête, J'ai donc téléchargé deux datasets sur Kaggle, puis les ai mis sur mon drive pour que n'importe quel utilisateur du projet puisse facilement accéder aux données.

Dans le code ci-dessous, j'utilise les librairies gdown et zipfiles pour pouvoir télécharger facilement les données et les décompresser ensuite.

```
import gdown
import zipfile
import os

# URL de téléchargement direct Google Drive
# pour le gros dataset (non tagged)
#url = 'https://drive.google.com/uc?id=16rYRrxUXpGyPWVq5uhgYzNlZd6hsGX7-'
# pour le petit dataset pokemon
url = 'https://drive.google.com/uc?id=1yfy6XXv0VikxR8xTuz-xQts_MmGWRDUy'
# Chemin de destination
output = 'dataset.zip'

# Télécharge le fichier depuis l'URL
gdown.download(url, output, quiet=False)

# Décompresse le fichier dans le dossier 'images'

with zipfile.ZipFile(output, 'r') as zip_ref:
    zip_ref.extractall('images')
# Supprime le fichier dataset.zip (pas besoin)
os.remove(output)
```

ETAPE 2 : EXTRACTION DES METADONNEES

L'objectif de cette partie est de créer un fichier JSON contenant toutes les informations utiles sur une image. Les informations qu' j'ai jugées pertinentes sont :

Le nom, la taille, le format (jpg, png,…), l'orientation (portrait, paysage, carré), la couleur dominante, les données exif (s'il y en a), les tags (qui seront ajoutés plus tard), et Favori (implémenté plus tard, n/a par défaut) .

Pour extraire ces métadonnées, j'utilise plusieurs fonctions :

-get_image_orientation pour avoir l'orientation en fonction de la longueur et largeur de l'image :

```
def get_image_orientation(img):
    if img.width > img.height:
        return "paysage"
    elif img.width < img.height:
        return "portrait"
    else:
        return "carre"
```

-get_exif_data pour avoir d'eventuelles données exif :

```
# Fonction pour extraire les données Exif
def get_exif_data(img):
    exif_data = {}
    raw_exif = img._getexif()
    if raw_exif:
        for tag, value in raw_exif.items():
            decoded_tag = ExifTags.TAGS.get(tag, tag)
            exif_data[decoded_tag] = value
    return exif_data
```

-find_dominant_color pour utilizer MiniBatchKMeans et trouver la couleur dominante :

```
def find_dominant_color(image_path, n_clusters=3):
    img = Image.open(image_path).convert('RGBA')  # Convertir en RGBA
    img = img.resize((50, 50))  # Redimensionner pour plus de vitesse

    # Convertir l'image RGBA en une array numpy
    numarray = np.array(img)
    # Ignorer les pixels transparents
    numarray = numarray[:, :, :3][numarray[:, :, 3] > 0]

    clusters = MiniBatchKMeans(n_clusters=n_clusters)
    clusters.fit(numarray)

    counts = np.bincount(clusters.labels_)
    most_frequent = clusters.cluster_centers_[counts.argmax()]

    return tuple(int(c) for c in most_frequent)
```

J'utilise MiniBatchKmeans, car comme vu dans le TP2, on arrive à trouver des résultats similaires au kmeans avec un temps d'exécution bien inférieur.

N_clusters peut être réduit ou augmenté, mais j'ai trouvé que le temps d'éxécution n'était pas ostensiblement supérieur, et la détection de couleur dominante était plus consistante avec 3 clusters.

Comme les images du dataset pokemon consistent en png, il fallait aussi gérer la transparence, car en restant en RGB, la couleur dominante était le noir car la transparence n'était pas prise en compte.

Cette fonction renvoie donc un tuple constitué des constituantes RGB de la couleur dominante de l'image.

Le programme principal ressemble donc a cela :

```python
for image_file in os.listdir(images_path):

    image_path = os.path.join(images_path, image_file)
    img = Image.open(image_path)
    # Extraction des données Exif
    exif_data = get_exif_data(img)
    # Trouver la couleur dominante
    dominant_color = find_dominant_color(image_path)

    print(exif_data)
    metadata.append({
        "nom": image_file,
        "taille": img.size,
        "format": img.format,
        "orientation": get_image_orientation(img),
        "exif": exif_data,
        "couleur_dominante": dominant_color,
        "tags": [],
        "favori": "n/a"
    })

if metadata:
    with open(metadata_file, 'w') as f:
        json.dump(metadata, f, indent=4)
    print("Métadonnées enregistrées.")
else:
    print("Aucune métadonnée à enregistrer.")
```

Après avoir obtenu et ajouté les métadonnées à la liste metadata, on les ajoute au fichier JSON, qui ressemble alors à cela :

```json
[
    {
        "nom": "abomasnow.png",
        "taille": [
            120,
            120
        ],
        "format": "PNG",
        "orientation": "carre ",
        "exif": {},
        "couleur_dominante": [
            247,
            249,
            250
        ],
        "tags": [],
        "favori": "n/a"
    }
]
```

Pour compléter les metadonnées, il ne nous manque plus qu'à ajouter des tags, et dire si l'utilisateur aime la photo.

ETAPE 3 : AJOUT DE TAGS

Dans cette partie, il s'agira d'ajouter des tags dans la liste « tags » du fichier JSON.

Pour le petit dataset pokemon (dataset de test) :

Les tags les plus judicieux sont surement le ou les types du pokemon en question (feu, plante, eau,…). Un pokemon peut en avoir un ou deux. Pour le dataset pokemon, on ne veut pas s'attarder sur les tags, car contrairement au gros dataset, ici l'utilisateur n'a pas à ajouter des tags. J'ai un fichier pokemon.csv qui contient les noms et types des pokemons :

```
Name,Type1,Type2
bulbasaur,Grass,Poison
ivysaur,Grass,Poison
venusaur,Grass,Poison
charmander,Fire
charmeleon,Fire
charizard,Fire,Flying
squirtle,Water
wartortle,Water
blastoise,Water
```

Pour ajouter des tags, il suffit donc d'ajouter les types dans la section tag du JSON :

```python
import pandas as pd
import json

# Chemin vers le fichier CSV
csv_path = 'pokemon.csv'
# Lire le fichier CSV
df = pd.read_csv(csv_path)

# Créer un dictionnaire pour mapper le nom de chaque Pokémon à ses types
pokemon_types = {row['Name'].lower(): [row['Type1'], row['Type2']] for index, row in df.iterrows()}

# Chemin vers le fichier JSON de métadonnées
metadata_path = 'image_metadata.json'

# Charger les métadonnées existantes
with open(metadata_path, 'r') as file:
    metadata = json.load(file)

for info in metadata:
    # Extraire le nom de l'image (le nom du fichier correspond au nom du Pokémon)
    pokemon_name = info['nom'].split('.')[0].lower()

    # Vérifier si le nom est dans le dictionnaire pokemon_types
    if pokemon_name in pokemon_types:
        # Récupérer la liste des types pour ce Pokémon
        types_list = pokemon_types[pokemon_name]

        # Filtrer les types car certains pokémons n'ont qu'un type
        true_types = []
        for t in types_list:
        # Regarder si la valeur n'est pas nulle et n'est pas une chaîne vide
            if pd.notna(t) and t != '':
                true_types.append(t)

        # Ajouter les types filtrés à la liste des tags pour ce Pokémon
        info['tags'].extend(true_types)

# Sauvegarder les métadonnées mises à jour dans le fichier JSON
with open(metadata_path, 'w') as file:
    json.dump(metadata, file, indent=4)
```

De cette manière, on obtient des tags correspondant aux types de chaque pokémon.

Pour le gros dataset d'images (dataset complet) :

Pour ce dataset, on n'a pas de tag par défaut comme « type » pour les pokémons. Il faut voir ce qu'il y a sur l'image et mettre un tag correspondant. Il y a plusieurs façons de faire :

•On peut demander à l'utilisateur, pour chaque image, d'entrer les tags qui pour lui correspondent à l'image , cela permet d'avoir des tags originaux comme « chien » en plus de « animal » par exemple. Cependant, le processus est TRES long, et l'utilisateur peut faire des erreurs de frappe par exemple.

• On peut présélectionner des tags globaux comme « nature », « humain », « technologie », « nourriture » etc.. et demander à l'utilisateur de choisir les images qui correspondent à un tag affiché. Cela permet d'obtenir certains tags, le processus est moins long que le premier car on peut afficher plusieurs images et demander de choisir toutes celles qui correspondent au tag « animal ». Mais le processus est toujours long, car on peut avoir des centaines de photos. De plus, on limite les tags à ceux que l'on propose à l'utilisateur. On a donc moins de tags possibles, ce qui entrainera une moins bonne précision dans les recommandations.

• On peut utiliser le Deep Learning pour automatiser la reconnaissance de tags. Cela permet d'avoir les tags que sait reconnaitre le modèle, et de manière automatisée, donc bien plus rapide. La mise en place peut être compliquée par rapport aux deux autres sélections de tags faites par l'utilisateur.

ETAPE 4 : FAVORI OU PAS ?

Après avoir donc ajouté des tags dans le fichier JSON, il ne nous reste plus qu'à demander à l'utilisateur s'il aime une image ou pas, et répéter le processus de nombreuses fois, afin de constituer une database d'entrainement.

Pour ce faire, il suffit de lui montrer l'image sur une interface graphique (tkinter) avec deux boutons « j'aime » ou « j'aime pas ». Pour une future update, on pourrait peut être nuancer le gout de l'utilisateur avec une note par exemple.

Dans ce code, on remplace le « n/a » par défaut de « favori » par « yes » ou « no »

Ce qui nous intéresse dans cette partie est l'implémentation dans le fichier JSON plus que l'interface graphique :

```
# Ajouter un tag à l'image actuelle et sauvegarder les métadonnées
def favori():
    image_name = image_files[current_image_index[0]]  # Nom de l'image actuelle

    # Chercher l'entrée correspondante dans les métadonnées grâce au nom de l'image
    # et ajouter le tag à la liste des tags
    for entry in metadata:
        if entry["nom"] == image_name:
            entry["favori"] = "yes"
            break
   # Sauvegarder les métadonnées mises à jour dans le fichier JSON
    with open(metadata_file, 'w') as f:
        json.dump(metadata, f, indent=5)
    next_image()
```

ETAPE 5 : CREER LES DATASETS D'ENTRAINEMENT ET DE TEST

Maintenant qu'une partie du dataset complet a des labels (yes ou no dans favori), on peut utiliser cette partie pour entrainer le modèle, et le tester pour mesurer sa précision. La partie du dataset complet qui n'a pas été évaluée par l'utilisateur sera le dataset sur lequel le modèle entrainé pourra faire ses prédictions et donc recommander certaines images non évaluées.

METTRE SCHEMA EXPLICATIF DATAFRAMES

Pour pouvoir rendre les données telles que les tags accessibles au modèle, nous allons utiliser des règles d'association, en séparant les tags en différentes colonnes, et en mettant 1 lorsque le tag est présent , ou 0 sinon grâce à MultiLabelBinarizer.

format_df = pd.get_dummies(dataframe['format'], prefix='format')

crée un nouveau dataframe format_df à partir de la colonne 'format' de dataframe. A partir de la colonne 'format' de dataframe, pour chaque valeur unique dans la colonne 'format', une nouvelle colonne est créée dans format_df. Si une image a un certain format, la colonne correspondante à ce format aura une valeur de 1 pour cette image, et 0 pour tous les autres formats.

L'argument prefix='format' ajoute le préfixe 'format_' à chaque nom de colonnegénéré pour clairement indiquer qu'il s'agit d'une colonne d'encodage dérivée du format de l'image.

X_train_eval, X_test_eval, y_train_eval, y_test_eval = train_test_split(X_train, y_train, test_size=0.2, random_state=50)

cette ligne divise les données évaluées en deux parties : un ensemble d'entraînement qui comprendra 80% des données et sera utilisé pour entraîner les modèles d'apprentissage automatique, et un ensemble de test qui comprendra les 20% restants des données et sera utilisé pour évaluer la performance des modèles.

```python
import pandas as pd
from sklearn.preprocessing import MultiLabelBinarizer
import json
import numpy as np

# Charger le dataset
with open("image_metadata.json", "r") as file:
    data = json.load(file)

# Créer le DataFrame
dataframe = pd.json_normalize(data)

# Calcul de l'aire de l'image à partir de la colonne 'taille'
dataframe['aire_image'] = dataframe['taille'].apply(lambda x: x[0] * x[1])

# Encodage des tags pour l'ensemble du dataset
mlb = MultiLabelBinarizer()
tags_encoded = mlb.fit_transform(dataframe['tags'])
tags_df = pd.DataFrame(tags_encoded, columns=mlb.classes_)

# Encodage pour 'format' et 'orientation' pour l'ensemble du dataset
format_df = pd.get_dummies(dataframe['format'], prefix='format')
orientation_df = pd.get_dummies(dataframe['orientation'], prefix='orientation')

# Concaténer les caractéristiques pour l'ensemble du dataset
all_final_df = pd.concat([tags_df, format_df, orientation_df, dataframe[['aire_image']]], axis=1)



# Séparer les images évaluées et non évaluées
dataframe_evalue = dataframe[dataframe['favori'] != 'n/a']
```

```
dataframe_non_evalue = dataframe[dataframe['favori'] == 'n/a']

# Préparation des labels pour le set évalué
labels_evalue = dataframe_evalue['favori'].map({'yes': 1, 'no': 0})

# Sélectionner les caractéristiques pour les images évaluées uniquement pour l'entraînement
X_train = all_final_df.loc[dataframe_evalue.index]
print("x train est: ",X_train)
y_train = labels_evalue

# Sélectionner les caractéristiques pour les images non évaluées pour la prédiction
X_test = all_final_df.loc[dataframe_non_evalue.index]
from sklearn.model_selection import train_test_split

# Diviser les données évaluées en ensembles d'entraînement et de test
X_train_eval, X_test_eval, y_train_eval, y_test_eval = train_test_split(
    X_train, y_train, test_size=0.2, random_state=50)
```

ETAPE 6: ENTRAINER LES MODELES

Après avoir séparé le datafrale evalué par l'utilisateur en une partie entrainement et une partie test, on peut essayer plusieurs modèles (ici SVC, perceptron et decisionTree), et regarder la précision qu'ils ont et leur score F1.

```
from sklearn.svm import SVC
from sklearn.linear_model import Perceptron
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report


# Initialisation des modèles
svc = SVC(probability=True)  # Pour SVC, activez la probabilité pour obtenir des scores de prédiction
perceptron = Perceptron()
decision_tree = DecisionTreeClassifier()

# Entraînement des modèles sur l'ensemble d'entraînement évalué
svc.fit(X_train_eval, y_train_eval)
perceptron.fit(X_train_eval, y_train_eval)
decision_tree.fit(X_train_eval, y_train_eval)



# Faire des prédictions sur l'ensemble de test évalué
predictions_svc_eval = svc.predict(X_test_eval)
predictions_perceptron_eval = perceptron.predict(X_test_eval)
predictions_tree_eval = decision_tree.predict(X_test_eval)

# Afficher la précision et d'autres métriques pour chaque modèle
print("Classification Report pour SVC:")
print(classification_report(y_test_eval, predictions_svc_eval))

print("Classification Report pour Perceptron:")
print(classification_report(y_test_eval, predictions_perceptron_eval))

print("Classification Report pour Decision Tree:")
print(classification_report(y_test_eval, predictions_tree_eval))
```

Pour ces trois modèles, voici les valeurs que j'obtiens :

```
              precision    recall  f1-score   support

           0       0.87      1.00      0.93        20
           1       0.00      0.00      0.00         3

    accuracy                           0.87        23
   macro avg       0.43      0.50      0.47        23
weighted avg       0.76      0.87      0.81        23

Classification Report pour Perceptron:
              precision    recall  f1-score   support

           0       0.86      0.90      0.88        20
           1       0.00      0.00      0.00         3

    accuracy                           0.78        23
   macro avg       0.43      0.45      0.44        23
weighted avg       0.75      0.78      0.76        23

Classification Report pour Decision Tree:
              precision    recall  f1-score   support

           0       0.86      0.95      0.90        20
           1       0.00      0.00      0.00         3

    accuracy                           0.83        23
   macro avg       0.43      0.47      0.45        23
weighted avg       0.75      0.83      0.79        23
```

ETAPE 7 : PREDICTIONS

Après une série de tests, SVC et decisionTree semblent être les deux plus performants, nous allons donc garder les deux pour faire des prédictions.

```
#################### recommandation par svc ############################

# Obtenir les probabilités de la classe positive (par exemple, "aimé" ou 1)
probabilities_svc = svc.predict_proba(X_test)[:, 1]
# Obtenir les indices des images triées par probabilité décroissante
indices_sorted_svc = np.argsort(probabilities_svc)[::-1]

# Sélectionner le top N images
top_n = 10
top_indices_svc = indices_sorted_svc[:top_n]

# Récupérer les noms des images pour les top-N recommandations
recommended_images_svc = dataframe_non_evalue.iloc[top_indices_svc]['nom'].values
print("Images recommandées par SVC :", recommended_images_svc)

#################### recommandation par decision tree ############################
# Faire des prédictions sur les données non évaluées
predictions_tree = decision_tree.predict(X_test)
# Trouver les indices des images prédites comme aimées ("yes")
indices_liked = np.where(predictions_tree == 1)[0]


top_n = 10  # Nombre d'images recommandées
recommended_indices = indices_liked[:top_n]

# Récupérer les noms des images recommandées
recommended_images_tree = dataframe_non_evalue.iloc[recommended_indices]['nom'].values

print("Images recommandées par Decision Tree :", recommended_images_tree)
```

Pour afficher les images, on réutilise tkinter pour afficher les recommandations toutes les 2 secondes :

```
import tkinter as tk
from PIL import Image, ImageTk
import time
import os
```

```python
# Fonction pour afficher une image dans la fenêtre Tkinter
def show_image(image_path):
    img = Image.open(image_path)
    img = img.resize((250, 250), Image.Resampling.LANCZOS)
    img_tk = ImageTk.PhotoImage(img)
    img_label.configure(image=img_tk)
    img_label.image = img_tk
    root.update_idletasks()
    root.update()

# Créer la fenêtre Tkinter
root = tk.Tk()
img_label = tk.Label(root)
img_label.pack()

# Définir le dossier contenant les images et les noms des images recommandées
image_folder = 'images/images'
recommended_images = recommended_images_tree  # Supposons que ceci est la liste des noms d'images
recommandées

# Afficher chaque image recommandée
for image_name in recommended_images:
    image_path = os.path.join(image_folder, image_name)
    show_image(image_path)
    time.sleep(2)  # Attendre 2 secondes avant de passer à l'image suivante

root.mainloop()
```

En choisissant comme favoris seulement des pokemons de type feu, les recommendations sont bien des pokemons de type feu, donc les recommendations sont bonnes. Maintenant il faut voir avec le gros dataset si les recommandations sont toujours correctes.