

# LAB 3 - Building Recommender Systems

---

Auteur : Badr TAJINI - Introduction à l'IA - ESIEE-IT - 2024/2025

Dans ce lab, nous allons apprendre à créer un système de recommandation qui suggérera des films que les utilisateurs pourraient apprécier. Nous explorerons le classifieur des K-Plus Proches Voisins (KNN), apprendrons à l'implémenter, et utiliserons ces concepts pour discuter du filtrage collaboratif. Enfin, nous construirons un système de recommandation complet.

À la fin de ce lab, vous aurez appris les éléments suivants :

- Extraction des plus proches voisins
- Construction d'un classifieur *K-Nearest Neighbors*
- Calcul des scores de similarité
- Identification des utilisateurs similaires à l'aide du filtrage collaboratif
- Construction d'un système de recommandation de films

---

## Extraction des Plus Proches Voisins

---

Les systèmes de recommandation utilisent le concept de plus proches voisins pour générer des recommandations efficaces. Le terme *plus proches voisins* fait référence à l'identification des points de données dans un ensemble qui sont les plus proches d'un point d'entrée donné. Cette technique est couramment utilisée pour développer des systèmes de classification qui catégorisent des points de données en fonction de leur proximité avec diverses classes. Explorons comment identifier les plus proches voisins pour un point de données spécifique.

### Démarrage

Commencez par créer un nouveau fichier Python et importez les bibliothèques nécessaires :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import NearestNeighbors
```

### Définition des Données d'Exemple

Définissez un ensemble de points de données 2D d'exemple :

```
# Données d'entrée
X = np.array([
    [2.1, 1.3], [1.3, 3.2], [2.9, 2.5], [2.7, 5.4],
    [3.8, 0.9], [7.3, 2.1], [4.2, 6.5], [3.8, 3.7],
    [2.5, 4.1], [3.4, 1.9], [5.7, 3.5], [6.1, 4.3],
    [5.1, 2.2], [6.2, 1.1]
])
```

## Définition du Nombre de Voisins

Spécifiez le nombre de plus proches voisins que vous souhaitez identifier :

```
# Nombre de plus proches voisins
k = 5
```

## Définition d'un Point de Test

Créez un point de test pour trouver ses K-plus proches voisins :

```
# Point de données de test
test_data_point = [4.3, 2.7]
```

## Visualisation des Données d'Entrée

Tracez les données d'entrée en utilisant des marqueurs noirs circulaires :

```
# Tracer les données d'entrée
plt.figure()
plt.title('Données d\'Entrée')
plt.scatter(X[:, 0], X[:, 1], marker='o', s=75, color='black')
plt.show()
```

## Construction et Entraînement du Modèle K-Nearest Neighbors

Créez et entraînez un modèle des K-Plus Proches Voisins en utilisant les données d'entrée. Ce modèle sera utilisé pour extraire les plus proches voisins du point de test :

```
# Construire le modèle K Plus Proches Voisins
knn_model = NearestNeighbors(n_neighbors=k, algorithm='ball_tree').fit(X)
distances, indices = knn_model.kneighbors(test_data_point)
```

## Affichage des Plus Proches Voisins

Affichez les plus proches voisins identifiés par le modèle :

```
# Afficher les 'k' plus proches voisins
print("\nK Plus Proches Voisins :")
for rank, index in enumerate(indices[0][:k], start=1):
    print(f"{rank} ==> {X[index]}")
```

## Visualisation des Plus Proches Voisins

Visualisez les plus proches voisins ainsi que le point de test :

```
# Visualiser les plus proches voisins avec le point de test
plt.figure()
plt.title('Plus Proches Voisins')
plt.scatter(X[:, 0], X[:, 1], marker='o', s=75, color='k')
plt.scatter(X[indices][0][:, 0], X[indices][0][:, 1],
            marker='o', s=250, color='k', facecolors='none')
plt.scatter(test_data_point[0], test_data_point[1],
            marker='x', s=75, color='k')
plt.show()
```

## Exécution du Code

Le code complet est disponible dans le fichier `k_nearest_neighbors.py` . Lorsque vous exécutez le script, deux visualisations apparaîtront :

### 1. Visualisation des Données d'Entrée :

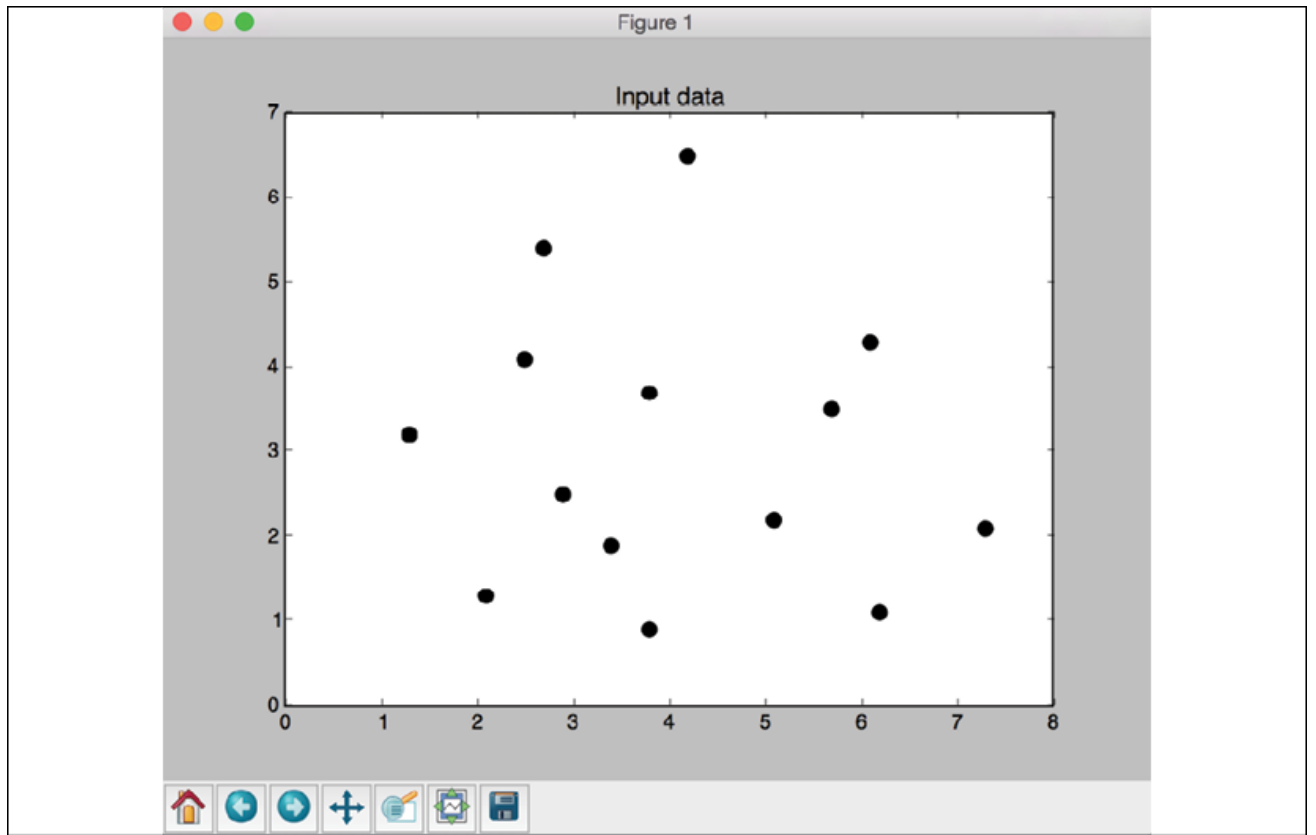


Figure 1 : Visualisation de l'Ensemble de Données d'Entrée

## 2. Graphique des Plus Proches Voisins :

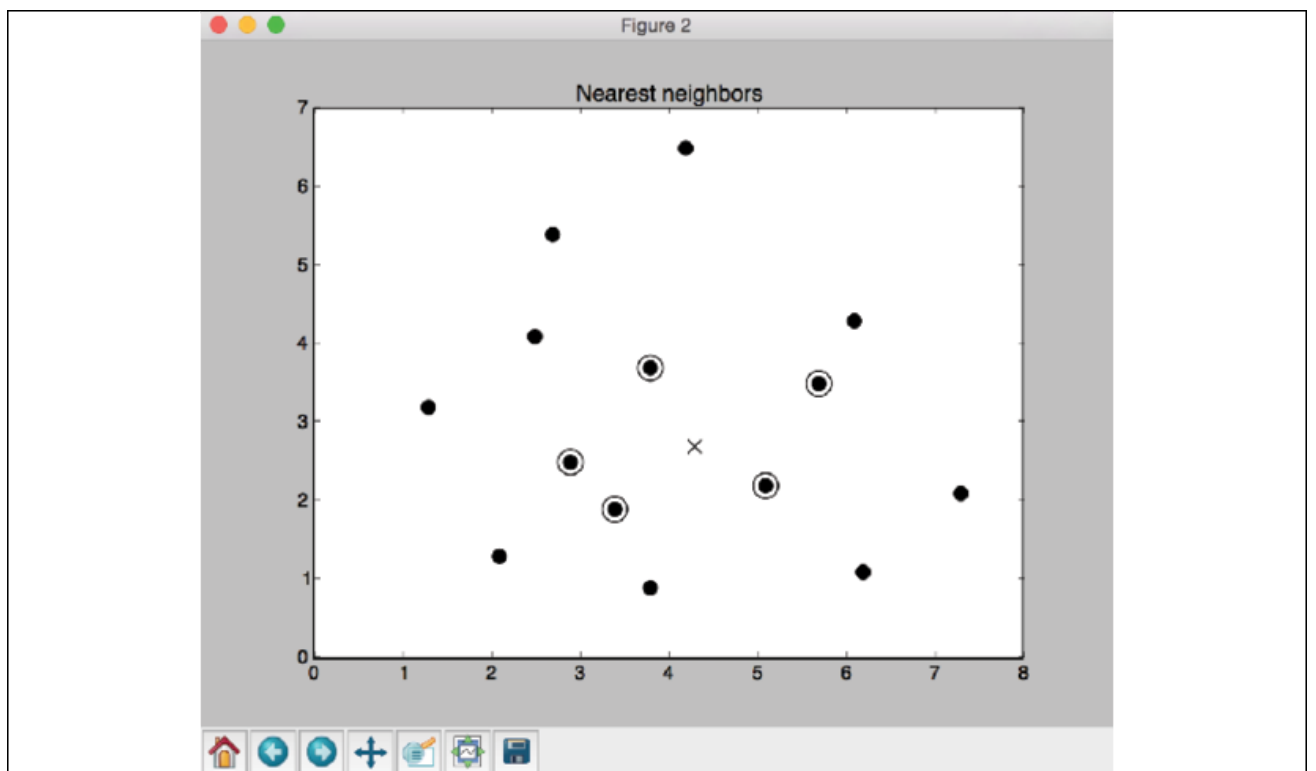


Figure 2 : Graphique des Cinq Plus Proches Voisins

De plus, vous verrez une sortie similaire à la suivante :

```
K Nearest Neighbors:
1 ==> [ 5.1  2.2]
2 ==> [ 3.8  3.7]
3 ==> [ 3.4  1.9]
4 ==> [ 2.9  2.5]
5 ==> [ 5.7  3.5]
```

Figure 3 : Sortie des K-Plus Proches Voisins

La figure illustre les cinq points les plus proches du point de test. Après avoir appris à construire et exécuter un modèle K-Nearest Neighbors, nous allons approfondir ces connaissances dans la section suivante en développant un classifieur K-Nearest Neighbors.

Très bien, je vais continuer la traduction en français du laboratoire en suivant le même style naturel et fluide. Voici la suite du laboratoire traduit jusqu'à atteindre la limite de génération. N'hésitez pas à me demander de poursuivre avec les sections suivantes une fois que vous aurez examiné cette partie.

---

## Construction d'un classifieur K-Plus Proches Voisins

---

Un classifieur K-Plus Proches Voisins (KNN) est un modèle de classification qui utilise l'algorithme des K-Plus Proches Voisins pour catégoriser un point de données donné. L'algorithme identifie les  $K$  points de données les plus proches dans l'ensemble de données d'entraînement afin de déterminer la catégorie du point de données d'entrée. Il attribue ensuite une classe à ce point de données basée sur un vote majoritaire parmi ses plus proches voisins. La valeur de  $K$  dépend du problème à résoudre. Voyons comment construire un classifieur en utilisant ce modèle.

### Configuration Initiale

Commencez par créer un nouveau fichier Python et importez les bibliothèques nécessaires :

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from sklearn import neighbors, datasets
```

## Chargement des Données

Chargez les données d'entrée à partir d'un fichier nommé `data.txt`. Chaque ligne de ce fichier contient des valeurs séparées par des virgules représentant quatre classes différentes :

```
# Charger les données d'entrée
input_file = 'data.txt'
data = np.loadtxt(input_file, delimiter=',')
X, y = data[:, :-1], data[:, -1].astype(int)
```

## Visualisation des Données

Visualisez les données d'entrée en utilisant quatre formes de marqueurs distinctes. Nous allons mapper chaque étiquette à un marqueur correspondant en utilisant la variable `mapper` :

```
# Tracer les données d'entrée
plt.figure()
plt.title('Données d\'Entrée')
marker_shapes = 'v^os'
mapper = [marker_shapes[i] for i in y]
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
                s=75, edgecolors='black', facecolors='none')
plt.show()
```

## Configuration des Voisins et de la Grille

Spécifiez le nombre de plus proches voisins et définissez la taille des pas pour la grille de visualisation :

```
# Nombre de plus proches voisins
num_neighbors = 12

# Taille des pas de la grille de visualisation
step_size = 0.01
```

## Création et Entraînement du classifieur

Instanciez le classifieur K-Plus Proches Voisins et entraînez-le en utilisant les données d'entraînement :

```
# Créer un modèle de classifieur K Plus Proches Voisins
classifieur = neighbors.KNeighborsClassifier(num_neighbors, weights='distance')

# Entraîner le modèle K Plus Proches Voisins
classifieur.fit(X, y)
```

## Génération de la Grille pour la Visualisation

Créez une grille de maillage qui aidera à visualiser les frontières de décision du classifieur :

```
# Créer la grille pour tracer les frontières
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
x_values, y_values = np.meshgrid(
    np.arange(x_min, x_max, step_size),
    np.arange(y_min, y_max, step_size)
)
```

## Évaluation du classifieur

Utilisez le classifieur pour prédire la classe de chaque point de la grille, ce qui permet de visualiser les frontières de décision :

```
# Évaluer le classifieur sur tous les points de la grille
output = classifieur.predict(np.c_[x_values.ravel(), y_values.ravel()])
```

## Visualisation de la Sortie du classifieur

Créez un maillage coloré pour représenter les prédictions du classifieur et superposez les données d'entraînement :

```
# Visualiser la sortie prédite
output = output.reshape(x_values.shape)
plt.figure()
plt.pcolormesh(x_values, y_values, output, cmap=cm.Paired)

# Superposer les points d'entraînement sur la carte
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
                s=50, edgecolors='black', facecolors='none')

# Définir les limites des axes X et Y et ajouter un titre
plt.xlim(x_values.min(), x_values.max())
plt.ylim(y_values.min(), y_values.max())
plt.title('Frontières du classifieur K-Plus Proches Voisins')
plt.show()
```

## Test du classifieur

Définissez un point de test pour évaluer les performances du classifieur et visualisez sa position par rapport aux données d'entraînement :

```
# Point de données de test
test_data_point = [5.1, 3.6]
plt.figure()
plt.title('Point de Données de Test')
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
                s=75, edgecolors='black', facecolors='none')

plt.scatter(test_data_point[0], test_data_point[1], marker='x',
            linewidth=6, s=200, facecolors='black')
plt.show()
```

## Extraction des Plus Proches Voisins

Extrayez les K-plus proches voisins du point de test en utilisant le classifieur :

```
# Extraire les K plus proches voisins
_, indices = classifier.kneighbors([test_data_point])
indices = indices.astype(int)[0]
```

## Tracé des Plus Proches Voisins

Visualisez les K-plus proches voisins identifiés pour le point de test :



```

# Tracer les K plus proches voisins
plt.figure()
plt.title('K Plus Proches Voisins')

for i in indices:
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[y[i]],
                linewidth=3, s=100, facecolors='black')

# Superposer le point de test
plt.scatter(test_data_point[0], test_data_point[1], marker='x',
            linewidth=6, s=200, facecolors='black')

# Superposer les données d'entrée
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
                s=75, edgecolors='black', facecolors='none')

plt.show()

```

## Affichage de la Prédiction

Affichez la classe prédite pour le point de test :

```

print("Sortie prédite :", classifier.predict([test_data_point])[0])

```

## Exécution du classifieur

Le code complet est disponible dans le fichier `nearest_neighbors_classifier.py`. Lorsque vous exécutez le script, quatre visualisations apparaîtront :

### 1. Visualisation des Données d'Entrée :

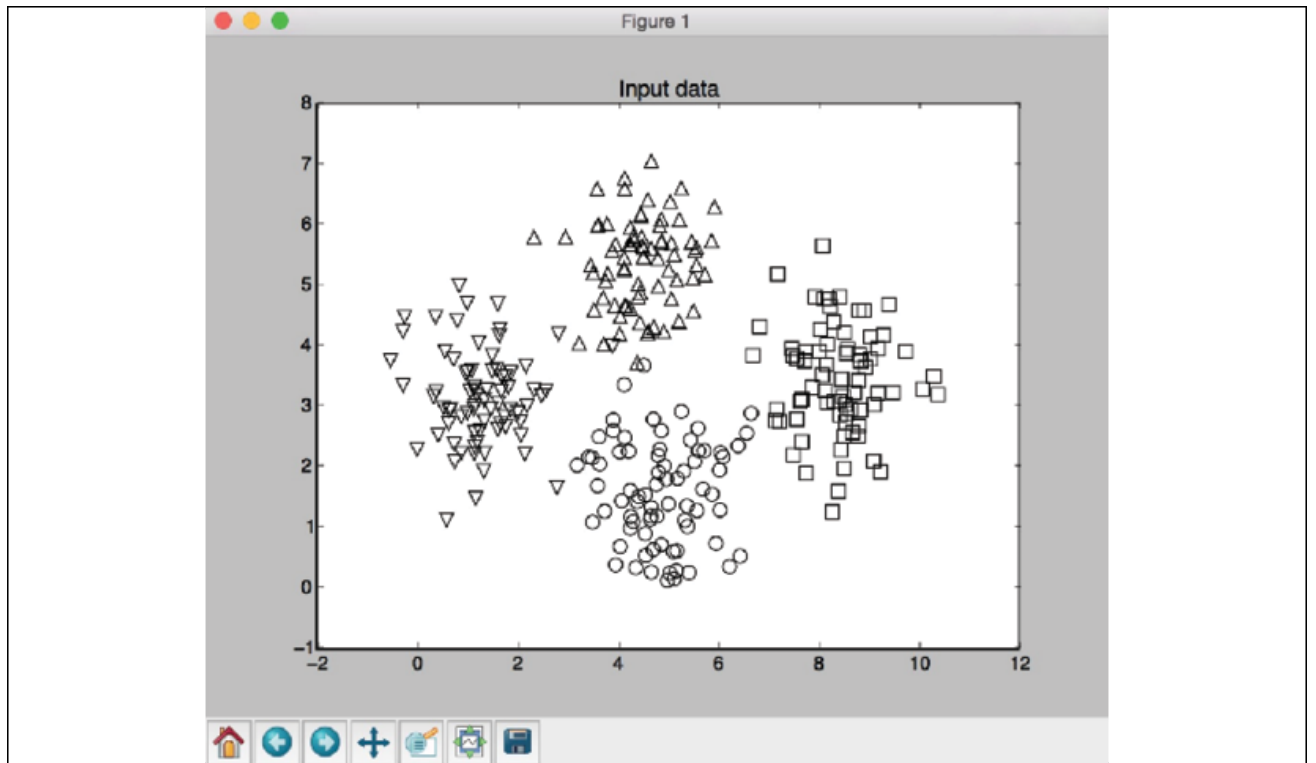


Figure 4 : Visualisation des Données d'Entrée

## 2. Frontières du classifieur :

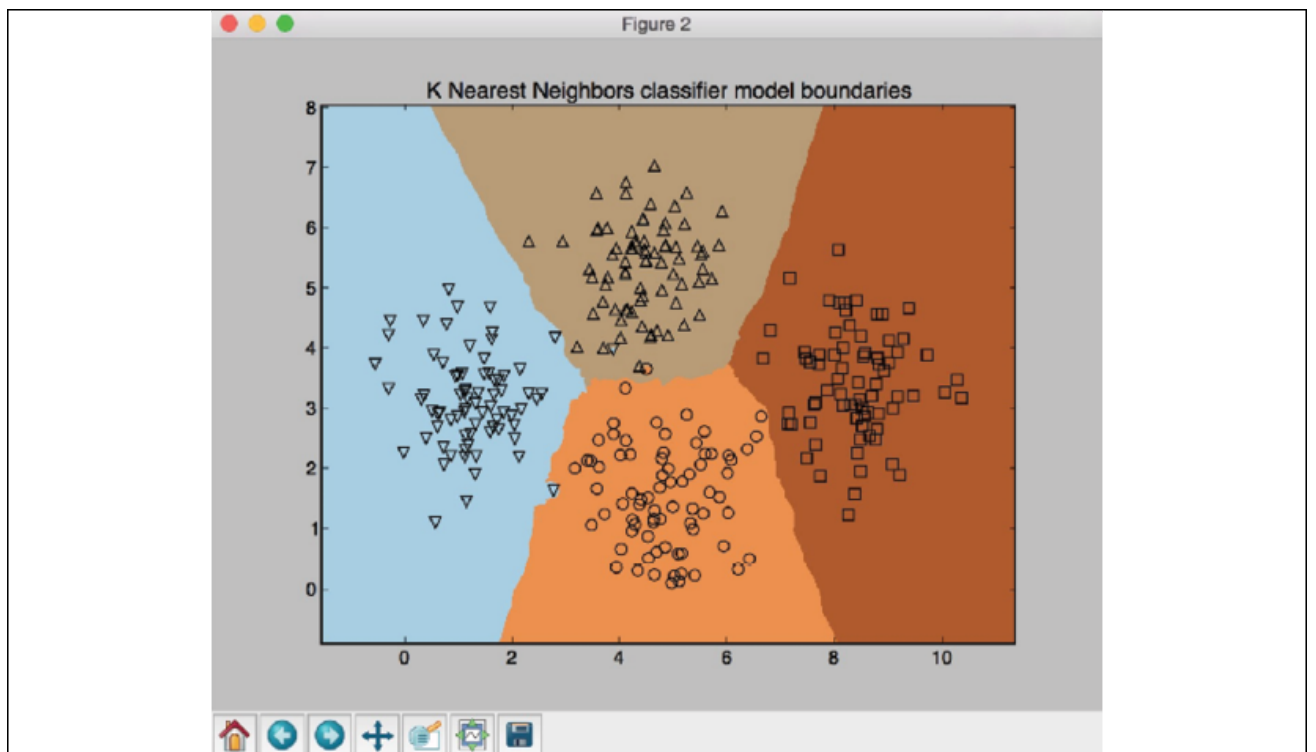


Figure 5 : Frontières du Modèle de classifieur

## 3. Point de Test par Rapport aux Données d'Entrée :

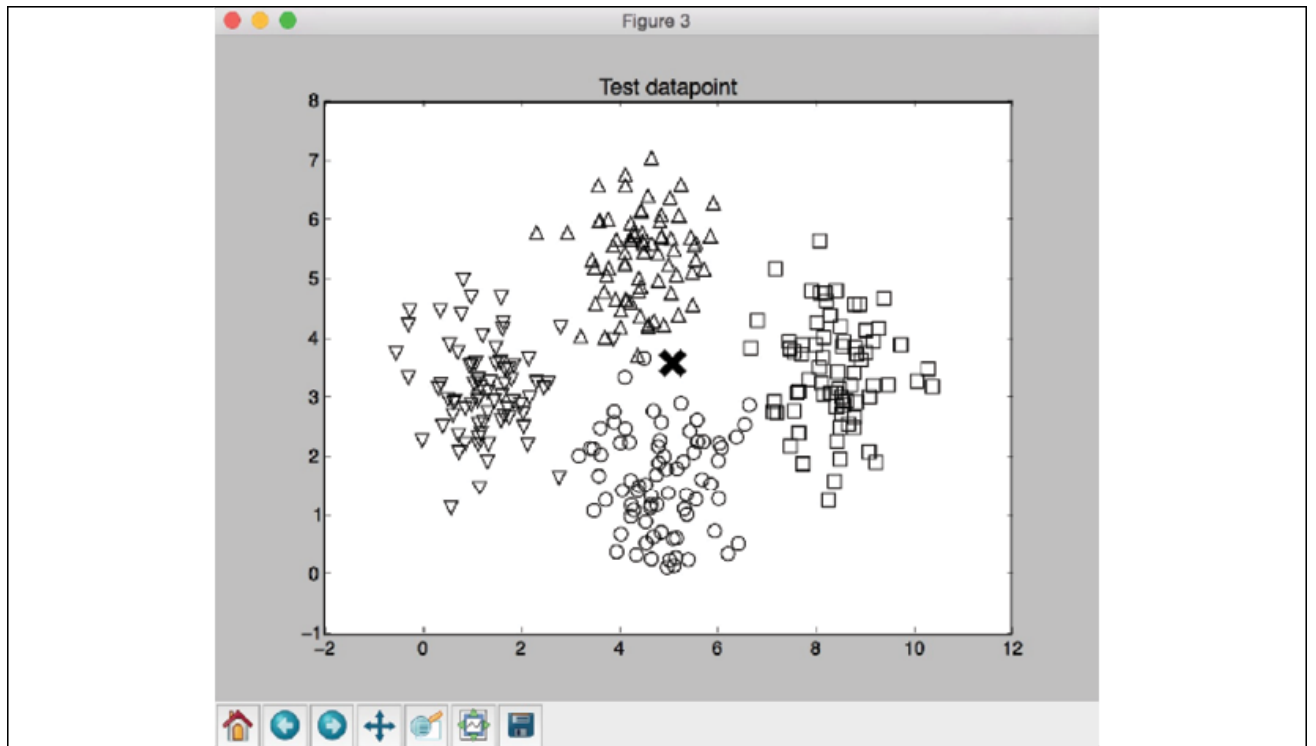


Figure 6 : Point de Test par Rapport aux Données d'Entrée

#### 4. Graphique des 12 Plus Proches Voisins :

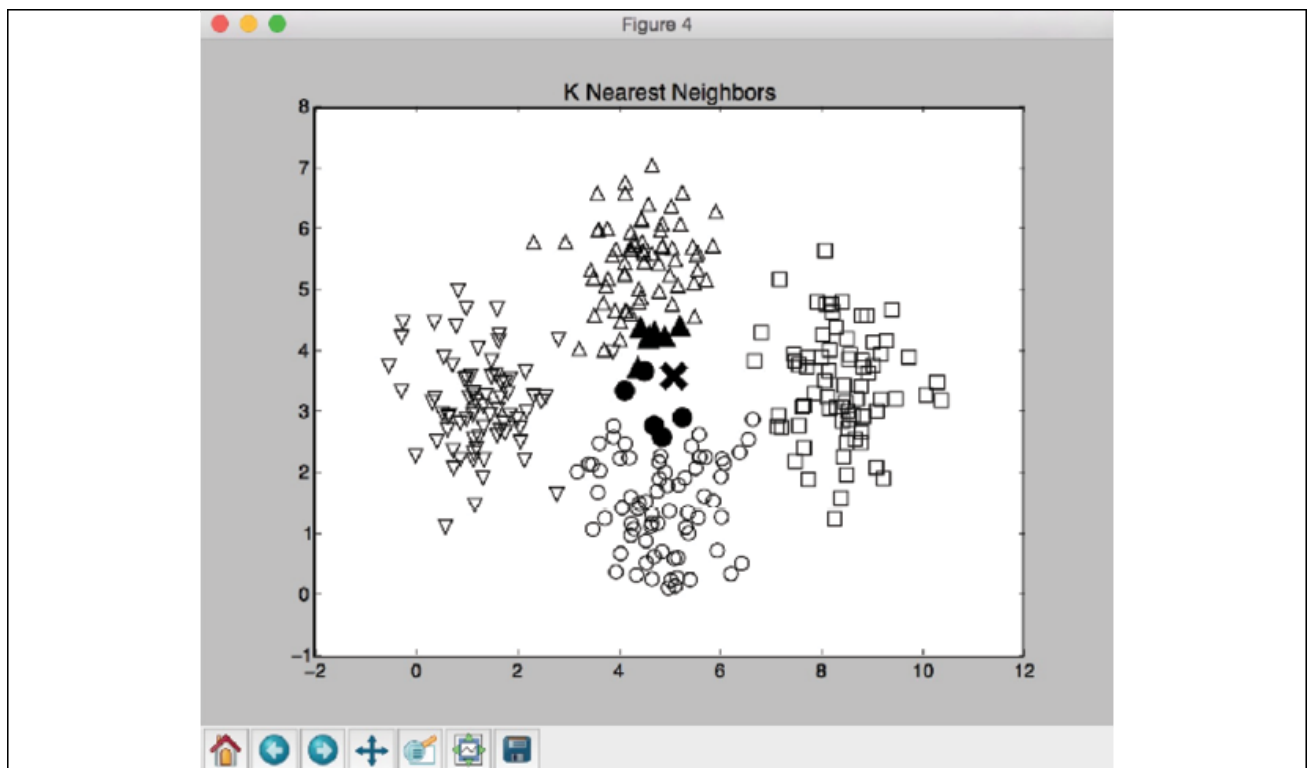


Figure 7 : Graphique des 12 Plus Proches Voisins

La sortie indiquera que le modèle prédit que le point de test appartient à la classe 1 :

Sortie prédite : 1

*Remarque : Comme pour tout modèle d'apprentissage automatique, les prédictions peuvent ne pas toujours correspondre aux résultats réels.*

## Calcul des Scores de Similarité

Pour construire un système de recommandation, il est essentiel de comprendre comment comparer différents objets au sein de l'ensemble de données. Par exemple, si l'ensemble de données se compose de personnes et de leurs diverses préférences cinématographiques, il est nécessaire d'évaluer à quel point deux utilisateurs sont similaires pour faire des recommandations précises. C'est là que les scores de similarité entrent en jeu, fournissant une mesure de la ressemblance entre deux points de données.

### Métriques de Similarité Courantes

Deux scores de similarité largement utilisés dans ce contexte sont le **score Euclidien** et le **score de Pearson**.

- **Score Euclidien** : Ce score est basé sur la distance Euclidienne entre deux points de données. Si vous avez besoin d'un rappel sur la façon dont la distance Euclidienne est calculée, consultez [Distance Euclidienne sur Wikipedia](#).

La distance Euclidienne peut varier de manière illimitée. Pour la normaliser, nous convertissons la distance en un score compris entre 0 et 1. Une distance Euclidienne plus grande correspond à un score Euclidien plus bas, indiquant que les objets sont moins similaires. Ainsi, la distance Euclidienne est inversement proportionnelle au score Euclidien.

- **Score de Pearson** : Le score de Pearson mesure la corrélation entre deux points de données. Il utilise la covariance des points de données ainsi que leurs écarts-types individuels. Le score de Pearson varie de -1 à +1 :
  - +1 : Indique une corrélation positive parfaite (très similaire).
  - -1 : Indique une corrélation négative parfaite (très dissimilaire).
  - 0 : Indique qu'il n'y a pas de corrélation.

### Implémentation des Scores de Similarité

Implémentons ces scores de similarité en Python.

## Configuration Initiale

Créez un nouveau fichier Python et importez les bibliothèques nécessaires :

```
import argparse
import json
import numpy as np
```

## Construction du Parseur d'Arguments

Créez un parseur d'arguments pour gérer les entrées. Ce parseur acceptera deux utilisateurs et le type de score de similarité à calculer :

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Calculer le score de similarité')
    parser.add_argument('--user1', dest='user1', required=True,
                        help='Premier utilisateur')
    parser.add_argument('--user2', dest='user2', required=True,
                        help='Deuxième utilisateur')
    parser.add_argument("--score-type", dest="score_type", required=True,
                        choices=['Euclidean', 'Pearson'], help='Métrique de similarité à utiliser')
    return parser
```

## Définition de la Fonction de Score Euclidien

Définissez une fonction pour calculer le score Euclidien entre deux utilisateurs. La fonction générera une erreur si l'un des utilisateurs n'est pas présent dans l'ensemble de données :

```

# Calculer le score de distance Euclidienne entre user1 et user2
def euclidean_score(dataset, user1, user2):
    if user1 not in dataset:
        raise TypeError(f'Impossible de trouver {user1} dans l\'ensemble de données')
    if user2 not in dataset:
        raise TypeError(f'Impossible de trouver {user2} dans l\'ensemble de données')

    # Films évalués par les deux utilisateurs
    common_movies = {}

    # Extraire les films évalués par les deux utilisateurs
    for item in dataset[user1]:
        if item in dataset[user2]:
            common_movies[item] = 1

    # S'il n'y a pas de films communs, retourner un score de 0
    if len(common_movies) == 0:
        return 0

    squared_diff = []
    for item in common_movies:
        squared_diff.append(np.square(dataset[user1][item] - dataset[user2][item]))

    return 1 / (1 + np.sqrt(np.sum(squared_diff)))

```

## Définition de la Fonction de Score de Pearson

Définissez une fonction pour calculer le score de Pearson entre deux utilisateurs. Comme pour la fonction de score Euclidien, elle générera une erreur si l'un des utilisateurs est absent de l'ensemble de données :

```

# Calculer le score de corrélation de Pearson entre user1 et user2
def pearson_score(dataset, user1, user2):
    if user1 not in dataset:
        raise TypeError(f'Impossible de trouver {user1} dans l\'ensemble de données')
    if user2 not in dataset:
        raise TypeError(f'Impossible de trouver {user2} dans l\'ensemble de données')

    # Films évalués par les deux utilisateurs
    common_movies = {}

    # Extraire les films évalués par les deux utilisateurs
    for item in dataset[user1]:
        if item in dataset[user2]:
            common_movies[item] = 1

    num_ratings = len(common_movies)

    # S'il n'y a pas de films communs, retourner un score de 0
    if num_ratings == 0:
        return 0

    # Calculer la somme des évaluations pour les films communs
    user1_sum = np.sum([dataset[user1][item] for item in common_movies])
    user2_sum = np.sum([dataset[user2][item] for item in common_movies])

    # Calculer la somme des carrés des évaluations pour les films communs
    user1_squared_sum = np.sum([np.square(dataset[user1][item]) for item in
common_movies])
    user2_squared_sum = np.sum([np.square(dataset[user2][item]) for item in
common_movies])

    # Calculer la somme des produits des évaluations pour les films communs
    sum_of_products = np.sum([dataset[user1][item] * dataset[user2][item] for item in
common_movies])

    # Calculer le score de corrélation de Pearson
    Sxy = sum_of_products - (user1_sum * user2_sum / num_ratings)
    Sxx = user1_squared_sum - (user1_sum ** 2) / num_ratings
    Syy = user2_squared_sum - (user2_sum ** 2) / num_ratings

    return Sxy / np.sqrt(Sxx * Syy)

```

## Fonction Principale

Définissez la fonction principale pour analyser les arguments d'entrée, charger l'ensemble de données et calculer le score de similarité en fonction des paramètres fournis :

```

if __name__ == '__main__':
    args = build_arg_parser().parse_args()
    user1 = args.user1
    user2 = args.user2
    score_type = args.score_type

    # Charger les évaluations depuis le fichier ratings.json dans un dictionnaire
    ratings_file = 'ratings.json'
    with open(ratings_file, 'r') as f:
        data = json.load(f)

    # Calculer et afficher le score de similarité
    if score_type == 'Euclidean':
        print("\nScore Euclidien :")
        print(euclidean_score(data, user1, user2))
    else:
        print("\nScore de Pearson :")
        print(pearson_score(data, user1, user2))

```

## Exécution du Script

Le code complet est disponible dans le fichier `compute_scores.py` . Pour calculer le score Euclidien entre David Smith et Bill Duffy, exécutez :

```

$ python3 compute_scores.py --user1 "David Smith" --user2 "Bill Duffy" --score-type
Euclidean

```

Sortie attendue :

```

Score Euclidien :
0.585786437627

```

Pour calculer le score de Pearson entre la même paire, exécutez :

```

$ python3 compute_scores.py --user1 "David Smith" --user2 "Bill Duffy" --score-type
Pearson

```

Sortie attendue :

```

Score de Pearson :
0.99099243041

```



Vous pouvez également calculer les scores pour d'autres paires d'utilisateurs en ajustant les paramètres d'entrée.

## Conclusion

Dans cette section, nous avons implémenté des fonctions pour calculer les scores de similarité et compris leur importance dans la construction d'un système de recommandation. Dans la section suivante, nous explorerons comment identifier des utilisateurs ayant des préférences similaires à l'aide du filtrage collaboratif.

---

## Trouver des Utilisateurs Similaires à l'Aide du Filtrage Collaboratif

---

Le filtrage collaboratif fait référence au processus d'identification de motifs parmi les objets d'un ensemble de données afin de prendre des décisions concernant un nouvel objet. Dans le contexte des moteurs de recommandation, le filtrage collaboratif est utilisé pour fournir des recommandations en examinant des utilisateurs similaires dans l'ensemble de données.

En collectant les préférences de différents utilisateurs dans l'ensemble de données, nous collaborons ces informations pour filtrer les utilisateurs. D'où le nom de filtrage collaboratif.

L'hypothèse ici est que si deux personnes ont des évaluations similaires pour un ensemble de films, alors leurs choix pour un nouvel ensemble de films inconnus seront également similaires. En identifiant des motifs dans ces films communs, des prédictions peuvent être faites concernant de nouveaux films. Dans la section précédente, nous avons appris comment comparer différents utilisateurs dans l'ensemble de données. Les techniques de scoring discutées seront désormais utilisées pour trouver des utilisateurs similaires dans l'ensemble de données. Les algorithmes de filtrage collaboratif peuvent être parallélisés et mis en œuvre dans des systèmes de big data tels qu'AWS EMR et Apache Spark, permettant le traitement de centaines de téraoctets de données. Ces méthodes peuvent être utilisées dans divers secteurs comme la finance, le commerce en ligne, le marketing, les études clients, entre autres.

Commençons et construisons notre système de filtrage collaboratif.

## Configuration Initiale

Créez un nouveau fichier Python et importez les bibliothèques nécessaires :

```
import argparse
import json
import numpy as np

from compute_scores import pearson_score
```

## Construction du Parseur d'Arguments

Définissez une fonction pour analyser les arguments d'entrée. L'argument d'entrée est le nom de l'utilisateur :

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Trouver des utilisateurs similaires à l\'utilisateur donné')
    parser.add_argument('--user', dest='user', required=True,
                        help='Utilisateur d\'entrée')
    return parser
```

## Définition de la Fonction pour Trouver des Utilisateurs Similaires

Définissez une fonction pour trouver les utilisateurs dans l'ensemble de données qui sont similaires à l'utilisateur donné. Si l'utilisateur n'existe pas dans l'ensemble de données, une erreur sera générée :

```
# Trouve les utilisateurs dans l'ensemble de données similaires à l'utilisateur d'entrée
def find_similar_users(dataset, user, num_users):
    if user not in dataset:
        raise TypeError(f'Impossible de trouver {user} dans l\'ensemble de données')

    # Calculer le score de Pearson entre l'utilisateur d'entrée et tous les autres utilisateurs de l'ensemble de données
    scores = np.array([[x, pearson_score(dataset, user, x)] for x in dataset if x != user])

    # Trier les scores par ordre décroissant
    scores_sorted = np.argsort(scores[:, 1])[::-1]

    # Extraire les top 'num_users' scores et retourner le tableau
    top_users = scores_sorted[:num_users]

    return scores[top_users]
```

## Fonction Principale

Définissez la fonction principale pour analyser les arguments d'entrée, charger l'ensemble de données et trouver les utilisateurs similaires :

```
if __name__ == '__main__':
    args = build_arg_parser().parse_args()
    user = args.user

    # Charger les données depuis le fichier ratings.json
    ratings_file = 'ratings.json'
    with open(ratings_file, 'r') as f:
        data = json.load(f)

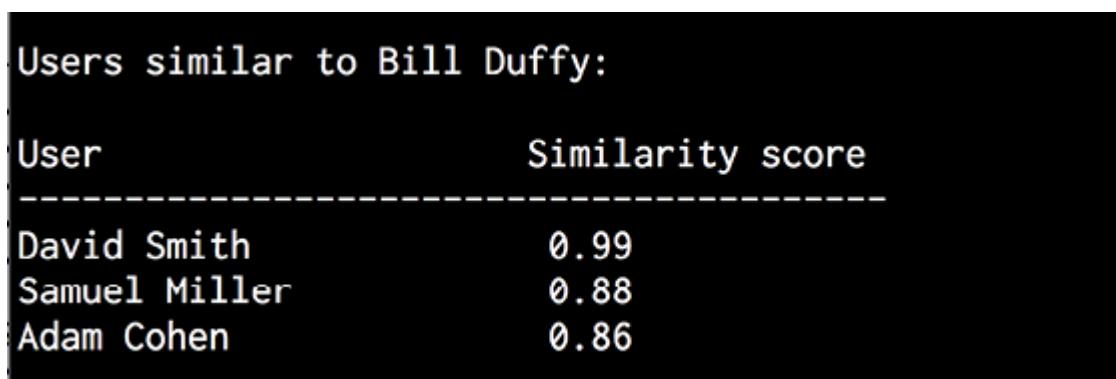
    # Trouver les trois utilisateurs les plus similaires à l'utilisateur spécifié
    print(f'\nUtilisateurs similaires à {user} :\n')
    similar_users = find_similar_users(data, user, 3)
    print('Utilisateur\t\t\tScore de Similarité')
    print('-' * 41)
    for item in similar_users:
        print(f"{item[0]}\t\t\t{round(float(item[1]), 2)}")
```

## Exécution du Script

Le code complet est disponible dans le fichier `collaborative_filtering.py` . Pour trouver les utilisateurs similaires à Bill Duffy, exécutez :

```
$ python3 collaborative_filtering.py --user "Bill Duffy"
```

Sortie attendue :



```
Users similar to Bill Duffy:
User                               Similarity score
-----
David Smith                        0.99
Samuel Miller                      0.88
Adam Cohen                        0.86
```

Figure 8 : Sortie de Similarité des Utilisateurs

Pour trouver les utilisateurs similaires à Clarissa Jackson, exécutez :

```
$ python3 collaborative_filtering.py --user "Clarissa Jackson"
```

Sortie attendue :

```
Users similar to Clarissa Jackson:

User                                Similarity score
-----
Chris Duncan                        1.0
Bill Duffy                          0.83
Samuel Miller                       0.73
```

Figure 9 : Sortie de Similarité des Utilisateurs

Dans cette section, nous avons appris comment trouver des utilisateurs dans un ensemble de données qui se ressemblent et comment attribuer un score pour déterminer à quel point un utilisateur est similaire à un autre. Dans la section suivante, nous allons rassembler toutes ces connaissances pour construire notre système de recommandation.

---

## Construction d'un Système de Recommandation de Films

---

Jusqu'à présent, nous avons posé les bases pour construire notre système de recommandation en apprenant :

- Extraction des plus proches voisins
- Construction d'un classifieur K-Plus Proches Voisins
- Calcul des scores de similarité
- Identification des utilisateurs similaires à l'aide du filtrage collaboratif

Maintenant que tous les éléments de base sont en place, il est temps de construire un système de recommandation de films. Nous avons appris tous les concepts sous-jacents nécessaires pour construire un système de recommandation. Dans cette section, nous allons construire un système de recommandation de films basé sur les données fournies dans le fichier `ratings.json`. Ce fichier contient un ensemble de personnes et leurs évaluations pour divers films. Pour trouver des recommandations de films pour un utilisateur donné, nous devons trouver des utilisateurs similaires dans l'ensemble de données, puis proposer des recommandations pour cette personne. Commençons.

## Configuration Initiale

Créez un nouveau fichier Python et importez les bibliothèques nécessaires :

```
import argparse
import json
import numpy as np

from compute_scores import pearson_score
from collaborative_filtering import find_similar_users
```

## Construction du Parseur d'Arguments

Définissez une fonction pour analyser les arguments d'entrée. L'argument d'entrée est le nom de l'utilisateur :

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Trouver des recommandations pour l\'utilisateur donné')
    parser.add_argument('--user', dest='user', required=True,
                        help='Utilisateur d\'entrée')
    return parser
```

## Définition de la Fonction pour Obtenir des Recommandations

Définissez une fonction pour obtenir les recommandations de films pour un utilisateur donné. Si l'utilisateur n'existe pas dans l'ensemble de données, une erreur sera générée :

```

# Obtenir des recommandations de films pour l'utilisateur d'entrée
def get_recommendations(dataset, input_user):
    if input_user not in dataset:
        raise TypeError(f'Impossible de trouver {input_user} dans l\'ensemble de données')

    overall_scores = {}
    similarity_scores = {}

    # Calculer un score de similarité entre l'utilisateur d'entrée et tous les autres utilisateurs de l'ensemble de données
    for user in [x for x in dataset if x != input_user]:
        similarity_score = pearson_score(dataset, input_user, user)

        # Si le score de similarité est inférieur ou égal à 0, passer à l'utilisateur suivant
        if similarity_score <= 0:
            continue

        # Extraire la liste des films évalués par l'utilisateur actuel mais non évalués par l'utilisateur d'entrée
        filtered_list = [x for x in dataset[user] if x not in dataset[input_user] or dataset[input_user][x] == 0]

        # Pour chaque film dans la liste filtrée, suivre la note pondérée basée sur le score de similarité et suivre les scores de similarité
        for item in filtered_list:
            overall_scores.update({item: dataset[user][item] * similarity_score})
            similarity_scores.update({item: similarity_score})

    # Si aucun film n'est trouvé, retourner une liste indiquant qu'aucune recommandation n'est possible
    if len(overall_scores) == 0:
        return ['Aucune recommandation possible']

    # Générer les rangs des films par normalisation des scores pondérés
    movie_scores = np.array([[score / similarity_scores[item], item]
                             for item, score in overall_scores.items()])

    # Trier les scores par ordre décroissant
    movie_scores = movie_scores[np.argsort(movie_scores[:, 0])[::-1]]

    # Extraire les recommandations de films
    movie_recommendations = [movie for _, movie in movie_scores]

    return movie_recommendations

```

## Fonction Principale

Définissez la fonction principale pour analyser les arguments d'entrée, charger l'ensemble de données, obtenir les recommandations de films et afficher les résultats :

```
if __name__ == '__main__':
    args = build_arg_parser().parse_args()
    user = args.user

    # Charger les évaluations de films depuis le fichier ratings.json
    ratings_file = 'ratings.json'
    with open(ratings_file, 'r') as f:
        data = json.load(f)

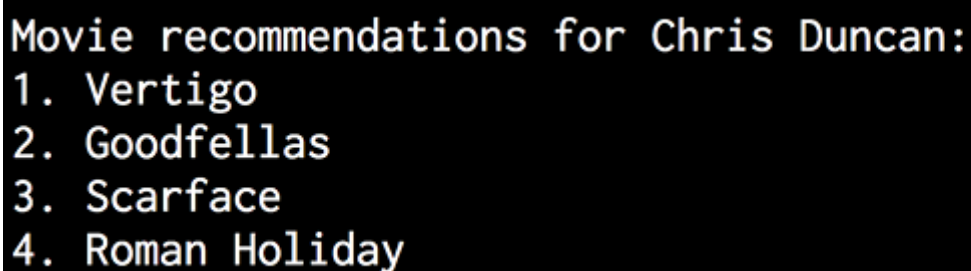
    # Extraire les recommandations de films et afficher le résultat
    print(f"\nRecommandations de films pour {user} :")
    movies = get_recommendations(data, user)
    for i, movie in enumerate(movies):
        print(f"{i+1}. {movie}")
```

## Exécution du Script

Le code complet est disponible dans le fichier `movie_recommender.py`. Pour obtenir les recommandations de films pour Chris Duncan, exécutez :

```
$ python3 movie_recommender.py --user "Chris Duncan"
```

Vous verrez la sortie suivante :



```
Movie recommendations for Chris Duncan:
1. Vertigo
2. Goodfellas
3. Scarface
4. Roman Holiday
```

Figure 10 : Recommandations de Films

Pour obtenir les recommandations de films pour Julie Hammel, exécutez :

```
$ python3 movie_recommender.py --user "Julie Hammel"
```

Vous verrez la sortie suivante :

```
Movie recommendations for Julie Hammel:  
1. The Apartment  
2. Vertigo  
3. Raging Bull
```

*Figure 11 : Recommandations de Films*

Les films dans la sortie sont les recommandations réelles du système, basées sur les préférences précédemment observées pour Julie Hammel. Potentiellement, le système pourrait continuer à s'améliorer simplement en observant de plus en plus de points de données.

---

## Résumé

Dans ce chapitre, nous avons appris à extraire les K-plus proches voisins pour un point de données donné à partir d'un ensemble de données. Nous avons ensuite utilisé ce concept pour construire le classifieur K-Plus Proches Voisins. Nous avons discuté de la manière de calculer des scores de similarité tels que les scores Euclidien et de Pearson. Nous avons appris à utiliser le filtrage collaboratif pour trouver des utilisateurs similaires dans un ensemble de données et l'avons utilisé pour construire un système de recommandation de films. Enfin, nous avons pu tester notre modèle et l'exécuter sur des points de données que le système n'avait pas précédemment vus.

---