Ahmed Elkhattam

Suivre À propos

Création d'un recommandeur basé sur le contenu à l'aide d'un algorithme de similarité cosinus

Ahmed Elkhattam 5 déc.2020 · 9 min de lecture

Dans cette série en trois parties, nous vous apprendrons tout ce dont vous avez besoin pour créer et déployer votre propre Chatbot. Il s'agit de la première partie de la série, où le reste de la série couvrira les sujets suivants:

- Partie 0: Présentation et aperçu du projet
- Partie 1: Construire un recommandateur basé sur le contenu à l'aide d'un algorithme de similarité cosinus
- Partie 2: Déploiement du modèle sur la plate-forme informatique sans serveur AWS et création d'une application Web de chatbot interactif.
- Partie 3: Analyse du modèle et recommandations futures .

Nous utilisons un bloc-notes Jupyter Python 3 comme environnement de codage collaboratif pour ce projet, et d'autres bits de code pour le développement et le déploiement de l'application Web. Tout le code de cette série est disponible dans ce référentiel GitHub .

Avant d'expliquer notre processus de construction, essayez de tester notre Chatbot entièrement déployé - peut-être qu'il pourra vous recommander votre prochain film :)

qui vise à prédire la « notation » ou « préférence » un utilisateur donne à un élément. Ils sont largement utilisés dans de nombreuses applications commerciales telles que Netflix, Youtube et Amazon Prime. Un système de recommandation aide les utilisateurs à trouver des éléments pertinents en une fraction de temps et sans avoir besoin de rechercher l'ensemble de données.

Il existe différentes approches pour créer un système de recommandation de films:

1. **Simple Recommender:** cette approche classe tous les films en fonction de critères spécifiques: popularité, récompenses et / ou genre, puis suggère les meilleurs films aux utilisateurs sans tenir compte de leurs préférences individuelles. Un exemple pourrait être le «Top 10 aux États-Unis aujourd'hui» de Netflix.

2. **Recommender de filtrage collaboratif:** cette approche utilise le comportement passé d'un utilisateur pour prédire les éléments susceptibles de les intéresser. Elle prend en compte les films précédemment regardés par un utilisateur, les notes numériques attribuées à ces éléments et les films précédemment regardés par des utilisateurs similaires.

3. **Content-based Filtering Recommender:** cette approche utilise les propriétés et les métadonnées d'un élément particulier pour suggérer d'autres éléments présentant des caractéristiques similaires. Par exemple, un recommandateur peut analyser le genre et le réalisateur d'un film pour recommander des films supplémentaires avec des propriétés similaires.

Chaque recommandateur a ses avantages et ses limites. Le recommandeur simple fournit des recommandations générales qui peuvent correspondre aux goûts de l'utilisateur. Le filtrage collaboratif peut fournir des recommandations précises, mais il nécessite une grande quantité d'informations sur l'historique et l'attitude antérieurs d'un utilisateur. D'un autre côté, le filtrage basé sur le contenu a besoin de peu d'informations sur l'historique de l'utilisateur pour fournir une bonne recommandation, mais sa portée est limitée car il a toujours besoin d'un élément avec des caractéristiques pré-étiquetées pour construire ses recommandations.

Comme nous n'avons pas accès au comportement passé d'un utilisateur, nous allons intégrer un mécanisme de filtrage basé sur le contenu dans notre robot Movie.

juillet 2017 ou avant, hébergés sur Kaggle . L'ensemble de données contient des métadonnées pour plus de 45 000 films répertoriés dans l'ensemble de données Full MovieLens. Les points de données incluent les acteurs, l'équipe, l'intrigue, les langues, les genres, le décompte des votes TMDB, les moyennes des votes et d'autres détails.

Les fichiers utilisés dans nos projets sont:

movies_metadata.csv : Le principal fichier de métadonnées Movies. Contient des informations sur 45 000 films présentés dans l'ensemble de données Full MovieLens.

keywords.csv : contient les mots-clés de l'intrigue de film pour nos films MovieLens. Disponible sous la forme d'un objet JSON stringifié.

credits.csv : comprend les informations sur les acteurs et l'équipe pour tous nos films. Disponible sous la forme d'un objet JSON stringifié.

notes_small.csv : sous-ensemble de 100 000 évaluations de 700 utilisateurs sur 9 000 films. Les notes sont sur une échelle de 1 à 5 et ont été obtenues sur le site Web officiel de GroupLens.

Nous commençons par télécharger les données du site Web de Kaggle et les stocker en interne dans Google Drive. Pour charger les données sur le notebook, nous *montons* Google Drive et utilisons la fonction *pd.read_csv* pour importer et charger les données.

```
de google . colab importation voiture
importer des pandas en tant que pd
importer numpy comme np

#Mounting Google Drive pour accéder à l'ensemble de données
conduire . mount ( '/ content / lecteur' )

#Importer les ensembles de données pertinents à partir du Google Drive monté (modifiez le code
métadonnées = pd . read_csv ( "/content/drive/.../movies_metadata.csv" )
notes = pd . read_csv ( "/content/drive/.../ratings.csv" )
crédits = pd . read_csv ( "/content/drive/.../credits.csv" )
mots-clés = pd . read_csv ( "/content/drive/.../keywords.csv" )
```

import_data.py hébergé avec ♡ par GitHub voir brut

Liste 1. Importation de données depuis Google Drive

- **movies_metadata.csv** a une forme de (45466 lignes, 24 colonnes) et des colonnes comme ['adult', 'comes_to_collection', 'budget', 'genres', 'homepage', 'id', 'imdb_id', 'original_language', 'original_title', 'overview', 'popular', 'poster_path', 'production_companies', 'production_countries', 'release_date', 'revenue', 'runtime', 'speak_languages', 'status', 'tagline', 'title ',' vidéo ',' vote_average ',' vote_count ']
- **notes.csv** a une forme de (26024289 lignes, 4 colonnes) et des colonnes comme ['userId', 'movieId', 'rating', 'timestamp']
- **credits.csv** a la forme (45476 lignes, 3 colonnes) et les colonnes ['cast', 'crew', 'id']
- **keywords.cvs** a la forme (46419 lignes, 2 colonnes) et les colonnes ['id', 'keywords']

Comme nous pouvons le voir ci-dessus, les données ne sont pas encore parfaitement organisées pour être utilisées ensemble. Les tableaux ont des dimensions différentes, et il pourrait être assez déroutant de les fusionner pour notre moteur de recommandation. Heureusement, tous ont une colonne pour l'ID de film, qui est l'ID IMDB unique de chaque film qui nous permettra de les fusionner efficacement.

```
#Data Exploration
#Here we explore the data shape and the name of its coloumns

#check the shape of the DataFrame (rows, columns)
#check the data columns
print("metadata shape:",metadata.shape)
print("metadata columns name:", metadata.columns)

print("ratings shape:",ratings.shape)
print("ratings columns name:", list(ratings.columns))

print("credits shape:",credits.shape)
print("columns name:", list(credits.columns))

print("keywords shape:",keywords.shape)
print("columns name:", list(keywords.columns))
```

data_exploration.py hosted with ♡ by GitHub view raw

Listing 2. Exploration des données

formant le modèle sur le traitement du langage naturel pour comprendre et suggérer des films similaires à l'entrée d'un utilisateur. Nous calculerons les scores de similitude cosinus par paire pour tous les films en fonction de leur distribution, des mots clés, des réalisateurs et des genres, puis recommanderons les films avec les scores de similitude les plus élevés à l'entrée de l'utilisateur.

Choisir les bonnes fonctionnalités

Nous combinons toutes les fonctionnalités importantes dans une seule trame de données. Nous utilisons *astype ('init')* pour changer le type de données d'ID de films en un entier, puis nous l'utilisons comme clé de référence pour fusionner les trames de données «crédits» et «mots-clés» dans la trame de données «métadonnées». En conséquence, les «métadonnées» df prennent la forme de 27 colonnes.

```
#Cutting the data to reduce resources

metadata = pd.read_csv("/content/drive/Shareddrives/Tutorial_GProject /Pedro's Experimenting Da
metadata = metadata.iloc[0:10000,:]

# Convert IDs to int. Required for merging on id using pandas .merge command
keywords['id'] = keywords['id'].astype('int')
credits['id'] = credits['id'].astype('int')
metadata['id'] = metadata['id'].astype('int')

# Merge keywords and credits into your main metadata dataframe: this will look
#for candidates on the credits and keywords tables that have ids that match those
#in the metadata table, which we will use as our main data from now on.
metadata = metadata.merge(credits, on='id')
metadata = metadata.merge(keywords, on='id')
metadata.shape
```

combining_features.py hosted with ♡ by GitHub view raw

Listing 3. Combinaison de mots-clés et de crédits aux principales métadonnées

Jetons donc un coup d'œil aux caractéristiques importantes des métadonnées:

	title	cast	crew	keywords	genres
0	Toy Story	[{'cast_id': 14, 'character': 'Woody (voice)',...	[{'credit_id': '52fe4284c3a36847f8024f49', 'de...	[{'id': 931, 'name': 'jealousy'}, {'id': 4290,...	[{'id': 16, 'name': 'Animation'}, {'id': 35, '...
1	Jumanji	[{'cast_id': 1, 'character': 'Alan Parrish', '...	[{'credit_id': '52fe44bfc3a36847f80a7cd1', 'de...	[{'id': 10090, 'name': 'board game'}, {'id': 1...	[{'id': 12, 'name': 'Adventure'}, {'id': 14, '...
2	Grumpier Old Men	[{'cast_id': 2, 'character': 'Max	[{'credit_id':	[{'id': 1495, 'name': 'fishing'}, {'id':	[{'id': 10749, 'name': 'Romance'},

Part II	Banks', '...	'52fe44959251416c75039ed7', 'de...	1599, 'n...	

Tableau 1. Le tableau des métadonnées après avoir combiné les mots clés et les crédits

Comme certains films peuvent avoir des dizaines de mots-clés et d'acteurs, nous sélectionnerons les informations les plus pertinentes dans chaque fonctionnalité. Nous écrirons des fonctions pour nous aider à extraire uniquement le nom du réalisateur de la fonction «Crew» et les trois premières informations pertinentes des autres fonctionnalités.

Avant de faire cela, nous devons changer les types de données des listes «stringifiées» en leurs types de données Python d'origine pour éviter les erreurs avec nos fonctions.

```python
from ast import literal_eval
#Parse the stringified features into their corresponding python objects
features = ['cast', 'crew', 'keywords', 'genres']
for feature in features:
  metadata[feature] = metadata[feature].apply(literal_eval)
```

string_to_original_objects.py hosted with ♡ by GitHub view raw

Listing 4. Conversion du type de données des chaînes en ses objets d'origine

La fonction get_director (x) extrait le nom du directeur et renvoie NaN s'il n'est pas trouvé. *La fonction get_lists (x)* renvoie les trois premiers éléments de chaque fonctionnalité.

```python
'''the crew list for a particular movie has one dictionary object per crew member.
Each dictionary has a key called 'job' which tells us if that person was the director or not.
With that in mind we can create a function to extract the director:'''

def get_director(x):
    for i in x:
        if i['job'] == 'Director':
            return i['name']
    return np.nan



#Getting a list of the actors, keywords and genres
def get_list(x):
    if isinstance(x, list): #checking to see if the input is a list or not
```

```
        #Check if more than 3 elements exist. If yes, return only first three.
        #If no, return entire list. Too many elements would slow down our algorithm
        #too much, and three should be more than enough for good recommendations.
        if len(names) > 3:
            names = names[:3]
        return names

    #Return empty list in case of missing/malformed data
    return []



'''
Now that we have written functions to clean up our data into director
names and listswith only the relevant info for cast, keywords and genres,
we can apply those functions to our data and see the results:
'''

metadata['director'] = metadata['crew'].apply(get_director)

features = ['cast', 'keywords', 'genres']
for feature in features:
    metadata[feature] = metadata[feature].apply(get_list)

metadata[['title', 'cast', 'director', 'keywords', 'genres']].head()
```

En appliquant ces deux fonctions à nos fonctionnalités, nous extrayons et travaillons uniquement avec les données pertinentes sur notre système de recommandation:

	title	cast	director	keywords	genres
0	Toy Story	[Tom Hanks, Tim Allen, Don Rickles]	John Lasseter	[jealousy, toy, boy]	[Animation, Comedy, Family]
1	Jumanji	[Robin Williams, Jonathan Hyde, Kirsten Dunst]	Joe Johnston	[board game, disappearance, based on children'...	[Adventure, Fantasy, Family]
2	Grumpier Old Men	[Walter Matthau, Jack Lemmon, Ann-Margret]	Howard Deutch	[fishing, best friend, duringcreditsstinger]	[Romance, Comedy]
3	Waiting to Exhale	[Whitney Houston, Angela Bassett, Loretta Devine]	Forest Whitaker	[based on novel, interracial relationship, sin...	[Comedy, Drama, Romance]
4	Father of the Bride Part II	[Steve Martin, Diane Keaton, Martin Short]	Charles Shyer	[baby, midlife crisis, confidence]	[Comedy]

Tableau 2. Le tableau des métadonnées après l'extraction des informations pertinentes

texte, nous devons appliquer le traitement du langage naturel avant de créer le recommandeur principal.

Notre objectif est finalement d'avoir un gros mot soupe pour chaque film afin que nous puissions vectoriser ces soupes et ensuite calculer la similitude cosinus. Nous commençons par convertir tous les mots en minuscules et en supprimant tous les espaces entre eux. La suppression d'espaces entre les mots aide le modèle à différencier les noms récurrents. Nous voulons que les noms soient rassemblés sans espace pour nous assurer que lorsque nous vectorisons, nous ne stockons pas le «Robert» dans «Robert De Niro» avec la même variable que le «Robert» dans «Robert Downey Junior», car il serait arbitraire de dire que ces acteurs sont similaires simplement en raison de leur nom. Lorsque nous stockons des noms comme «robertdeniro» et «robertdowneyjunior» à la place, nous différencions les acteurs en créant des vectorisations séparées.

```
def clean_data(x):
    if isinstance(x, list):
        return [str.lower(i.replace(" ", "")) for i in x] #cleaning up spaces in the data
    else:
        #Check if director exists. If not, return empty string
        if isinstance(x, str):
            return str.lower(x.replace(" ", ""))
        else:
            return ''


# Apply clean_data function to your features.
features = ['cast', 'keywords', 'director', 'genres']

for feature in features:
  metadata[feature] = metadata[feature].apply(clean_data)

metadata.head()
```

clean_data.py hosted with ♡ by GitHub view raw

Listing 5. Nettoyage des données pour le processus de vectorisation

Ensuite, nous combinons toutes les fonctionnalités dans une seule liste «soupe de métadonnées» car elles ont le même poids dans le choix du film recommandé. La fonction itère sur les lignes de nos métadonnées et joint les colonnes de mots-clés, de

qu'il s'agit d'un mot particulier, à coder séparément et de manière unique.

```
# Cette fonction utilise la propriété de la fonction de similarité cosinus qui
# l'ordre et les types d'entrées n'ont pas d'importance, ce qui compte c'est la similitude
#entre différentes soupes de mots

def  create_soup ( x ):
    retour  '' . join ( x [ 'keywords' ]) +  ''  +  '' . join ( x [ 'cast' ]) +  ''  +  x [ 'dir

metadata [ 'soup' ] =  métadonnées . appliquer ( create_soup , axis = 1 )
metadata [[ 'title' , 'soup' , 'cast' , 'director' , 'keywords' , 'genres' ]]. tête ()
```

create_soup.py hébergé avec ♡ par GitHub voir brut

Listing 6. Créer une soupe de mots

	title	soup	cast	director	keywords	genres
0	Toy Story	jealousy toy boy tomhanks timallen donrickles ...	[tomhanks, timallen, donrickles]	johnlasseter	[jealousy, toy, boy]	[animation, comedy, family]
1	Jumanji	boardgame disappearance basedonchildren'sbook ...	[robinwilliams, jonathanhyde, kirstendunst]	joejohnston	[boardgame, disappearance, basedonchildren'sbook]	[adventure, fantasy, family]
2	Grumpier Old Men	fishing bestfriend duringcreditsstinger walter...	[waltermatthau, jacklemmon, ann-margret]	howarddeutch	[fishing, bestfriend, duringcreditsstinger]	[romance, comedy]
3	Waiting to Exhale	basedonnovel interracialrelationship singlemot...	[whitneyhouston, angelabassett, lorettadevine]	forestwhitaker	[basedonnovel, interracialrelationship, single...	[comedy, drama, romance]
4	Father of the Bride Part II	baby midlifecrisis confidence stevemartin dian...	[stevemartin, dianekeaton, martinshort]	charlesshyer	[baby, midlifecrisis, confidence]	[comedy]

Tableau 3. Le tableau des métadonnées après l'ajout de la colonne soupe

Maintenant que nous avons la soupe pour chaque film, nous voulons créer une nouvelle soupe chaque fois que notre recommandation est exécutée pour capturer l'entrée de l'utilisateur. Nous voulons collecter les entrées de l'utilisateur afin de pouvoir ensuite les vectoriser. Plus tard, nous calculerons la similitude cosinus par paire entre cette entrée et chaque film de notre base de données, et classerons les films les plus similaires à cette entrée.

```
# Obtenir la contribution de l'utilisateur pour le genre, les acteurs et les réalisateurs de sc

def  get_genres ():
  genres  =  input ( "Quel genre de film vous intéresse (s'il est multiple, veuillez les sépare
  genres  =  "" . join ([ "" . join ( n . split ()) pour  n  dans les  genres . lower (). split
  renvoyer les  genres

def  get_actors ():
  acteurs  =  input ( "Quels sont les acteurs du genre que vous aimez (s'il y en a plusieurs, v
```

```
def get_directors ():
  directeurs = entrée ( "Quels sont les réalisateurs du genre que vous aimez (s'il y en a plu
  directeurs = "" . rejoindre ([ "" . rejoindre ( n . fendu ()) pour n dans les administra
   directeurs de retour

def get_keywords ():
  mots-clés = entrée ( "Quels sont certains des mots-clés qui décrivent le film que vous souh
  mots-clés = "" . join ([ "" . join ( n . split ()) pour n dans les mots-clés . lower ().
  renvoyer des mots-clés

def get_searchTerms ():
  searchTerms = []
  genres = get_genres ()
  if genres ! = 'sauter' :
    searchTerms . ajouter ( genres )

  acteurs = get_actors ()
  si les acteurs ! = 'sauter' :
    searchTerms . ajouter ( acteurs )

  directeurs = get_directors ()
  si directeurs ! = 'sauter' :
    searchTerms . ajouter ( administrateurs )

  mots-clés = get_keywords ()
  if keywords ! = 'ignorer' :
    searchTerms . ajouter ( mots-clés )

  retourner la recherche
```

collecting_users_data.py accueilli avec ♡ par GitHub voir brut

Listing 7. Obtention des fonctions d'entrée de l'utilisateur

Modèle de recommandation basé sur le vectoriseur de comptage et la similitude du cosinus

Maintenant que les fonctionnalités sont propres, apprenons à notre modèle à les lire.

Comme mentionné ci-dessus, notre modèle de recommandation prend à la fois les fonctionnalités de film prétraitées et les préférences de l'utilisateur comme entrée. Ensuite, il soupifie l'entrée de l'utilisateur et l'ajoute sous forme de ligne aux métadonnées. Le modèle vectorise ensuite le mot soupe à l'aide de la fonction *CountVectorizer* de la bibliothèque python *scikit-learn* . CountVectorizer prend des

stockées dans une liste:

corpus = [

'Ceci est le premier document.',

"Ce document est le deuxième document.",

'Et ceci est le troisième.',

«Est-ce le premier document?»]

Si nous leur appliquons le CountVectorizer, nous obtiendrons le tableau suivant:

Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7	Word 8	Word 9
0	1	1	1	0	0	1	0	1
0	2	0	1	0	1	1	0	1
1	0	0	1	1	0	1	1	1
0	1	1	1	0	0	1	0	1

Le tableau reflète la fréquence de chaque mot de la phrase. Disons que Word2 est le mot «document», donc mot2 est apparu une fois dans la première phrase et deux fois dans la deuxième phrase.

Nous vectorisons les entrées de l'utilisateur en ajoutant la soupe de mot entrée à la table de métadonnées, comme dernière entrée, puis en exécutant le processus de vectorisation pour toutes les données. Bien que ce ne soit pas le moyen le plus efficace de s'y prendre, la fonction CountVectorize est très rapide à exécuter et consomme peu de ressources. Le plus gros problème auquel nous devons faire face est celui des calculs de similarité cosinus

Similarité cosinus

Notre modèle de recommandation utilise les propriétés de tous les films et les métadonnées pour calculer et trouver le film le plus similaire à l'entrée de l'utilisateur.

données.

La similarité cosinus est un calcul mathématique qui nous indique la similitude entre deux vecteurs A et B. En effet, nous calculons le cosinus de l'angle *thêta* entre ces deux vecteurs. La fonction renvoie une valeur comprise entre -1, indiquant des vecteurs opposés complets, à 1, indiquant le même vecteur. 0 indique un manque de corrélation entre les vecteurs et les valeurs intermédiaires indiquent des niveaux intermédiaires de similitude. Heureusement, nous n'avons pas besoin d'écrire une nouvelle fonction pour calculer l'équation car scikit-learn a déjà une fonction pré-construite appelée cosine_similarities (). La formule que nous utilisons pour calculer la similitude cosinus est la suivante:

$$\cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}, \text{ where } \|A\| = \sqrt{\sum_{i=1}^{n} A_i^2} \text{ and } \|B\| = \sqrt{\sum_{i=1}^{n} B_i^2}.$$

La fonction de similarité cosinus augmente linéairement en complexité à mesure que nous augmentons la taille de A et B (notez que A et B ont la même taille, n). Le produit scalaire de A et B nécessitera $n + t$ plus de calculs si nous ajoutons t plus de valeurs à A et B, et la magnitude de chacune d'elles augmentera également linéairement. Jusqu'à présent, aucun problème de complexité de calcul.

Cependant, notre algorithme effectue un calcul de similarité cosinus entre chaque paire possible de films. Si nous avons k films, alors nous devons effectuer des calculs de k^2. C'est la raison pour laquelle nous avons dû réduire le nombre de films dans notre ensemble de données de 45 000 à 10 000, car la différence de 35 000 films se traduit par des calculs de $1,925 * 10^9$.

Bien sûr, il existe des méthodes pour diminuer le nombre de calculs nécessaires et donc nous permettre d'utiliser l'ensemble de données, mais nous avons décidé de les laisser comme des améliorations potentielles en retard pour le moment.

Enfin, nous définissons notre système de recommandation comme une fonction qui prend les entrées de l'utilisateur (Genera, Actor et Director), effectue un score de similitude cosinus entre les termes de recherche et tous les films, puis recommande les 10 films les plus similaires selon les préférences de l'utilisateur.

```
def  make_recommendation ( metadata = metadata ):
  new_row  =  métadonnées . iloc [ - 1 ,: ]. copy () #création d'une copie de la dernière ligne
  #dataset, que nous utiliserons pour saisir l'entrée de l'utilisateur

  # saisir le nouveau groupe de mots de l'utilisateur
  searchTerms  =  get_searchTerms ()
  new_row . iloc [ - 1 ] =  "" . join ( searchTerms ) #ajout de l'entrée à notre nouvelle ligne

  #ajout de la nouvelle ligne à l'ensemble de données
  metadata  =  métadonnées . ajouter ( new_row )

  # Vectoriser la matrice entière comme décrit ci-dessus!
  count  =  CountVectorizer ( stop_words = 'anglais' )
  count_matrix  =  nombre . fit_transform ( métadonnées [ 'soupe' ])

  # exécuter la similarité cosinus par paire
  cosine_sim2  =  cosine_similarity ( count_matrix , count_matrix ) # obtenir une matrice de si

  # tri des similitudes cosinus du plus élevé au plus bas
  sim_scores  =  liste ( énumérer ( cosinus_sim2 [ - 1 ,: ]))
  sim_scores  =  trié ( sim_scores , clé = lambda  x : x [ 1 ], reverse = True )

  # correspondance des similitudes avec les titres et les identifiants des films
  rank_titles  = []
  pour  i  dans la  plage ( 1 , 11 ):
    indx  =  sim_scores [ i ] [ 0 ]
    rank_titles . append ([ metadata [ 'title' ]. iloc [ indx ], metadata [ 'imdb_id' ]. iloc |

  retourner les  titres classés

# essayons notre fonction de recommandation maintenant
make_recommendation ()
```


Listing 8. La fonction de recommandation finale avec des scores de similarité cosinus

Essayons notre recommandation:

```
What Movie Genre are you interested in (if multiple, please separate them with a comma)? [Type 'skip' to skip this question] terror
Who are some actors within the genre that you love (if multiple, please separate them with a comma)? [Type 'skip' to skip this question] Jack Nicholson
Who are some directors within the genre that you love (if multiple, please separate them with a comma)? [Type 'skip' to skip this question] Martin Scorcese
[["Goin' South", 'tt0077621'],
 ['The Two Jakes', 'tt0100828'],
 ['Ironweed', 'tt0093277'],
 ['Man Trouble', 'tt0104804'],
 ['Carnal Knowledge', 'tt0066892'],
 ['The Fortune', 'tt0073008'],
 ['The Missouri Breaks', 'tt0074906'],
 ['Wolf', 'tt0111742'],
```

Conclusion

Dans cette partie de la série, nous avons discuté de différents types d'algorithmes de recommandation et sélectionné l'approche de filtrage basée sur le contenu comme étant notre algorithme principal pour le chatbot. Nous avons expliqué en détail le mécanisme de construction d'un modèle de recommandation à l'aide du vectoriseur de comptage et des scores de similarité cosinus.

Dans la prochaine partie de cette série, nous illustrerons le processus de déploiement de notre modèle de recommandation sur une plate-forme sans serveur AWS et de création d'une application Web chatbot interactive, ici .

Ressources

(Tutoriel) Systèmes de recommandation en Python. (nd). Extrait le 6 décembre 2020 de https://www.datacamp.com/community/tutorials/recommender-systems-python

Système de recommandation. (2020, 29 novembre). Extrait le 6 décembre 2020 de https://en.wikipedia.org/wiki/Recommender_system

Chatbots Apprentissage automatique