Réalisez une application de recommandation de contenu

Joëlle JEAN BAPTISTE - Décembre 2025

Introduction

Contexte, problématique & contraintes du MVP

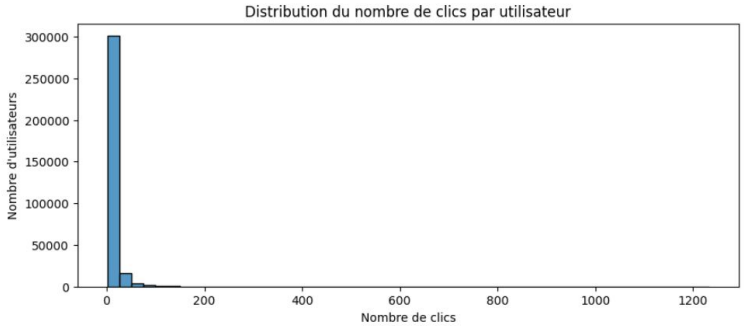
- My Content cherche un premier système de recommandation.
- Objectif : proposer 5 articles pertinents par utilisateur.
- Données limitées → gérer le cold-start.
- Architecture attendue : simple, scalable & serverless.
- Doit supporter l'arrivée de nouveaux utilisateurs / nouveaux articles.

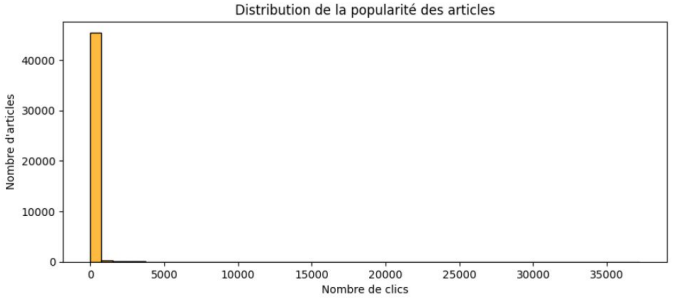
Données & Préparation

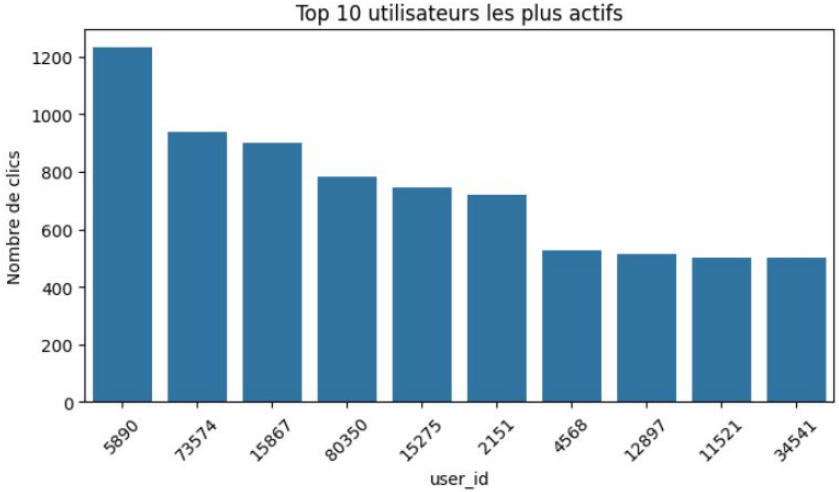
Présentation du dataset utilisateur

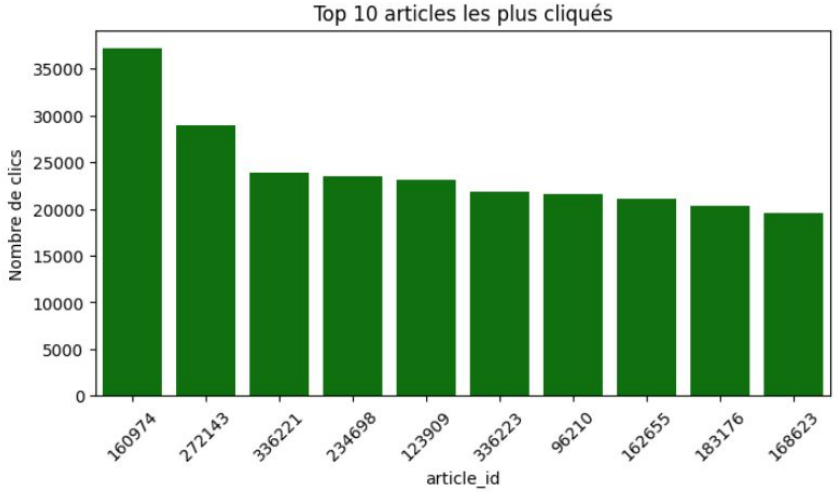
```
Taille du dataset :
Lignes : 2988181
Colonnes : 12

Colonnes et types :
user_id : object
session_id : object
session_start : object
session_size : object
click_article_id : object
click_timestamp : object
click_environment : object
click_deviceGroup : object
click_os : object
click_country : object
click_region : object
click_referrer_type : object
```






Top 10 utilisateurs les plus actifs
Nombre de clics
0
200
400
600
800
1000
1200
5890
73574
15867
80350
15275
2151
4568
12897
11521
34541
user_id


Top 10 articles les plus cliqués
Nombre de clics
0
5000
10000
15000
20000
25000
30000
35000
160974
272143
336221
234698
123909
336223
96210
162655
183176
168623
article_id

Construction du dataset final pour les modèles

- Deux datasets :
  - user–item (KNN / CF)
  - embeddings articles (FAISS / content-based)
- Alignement des identifiants
- Sous-ensemble du dataset utilisé pour KNN (limites mémoire)
- Structures finales : matrice CF + index FAISS

Stratégie Cold Start

- Utilisateurs sans historique → content-based
- Recommandation via embeddings articles
- Fallback : articles les plus populaires
- Pas besoin de réentraînement → réponse immédiate

Modèles Collaboratifs

KNN Basic (Collaborative Filtering)

- Basé sur la similarité entre utilisateurs
- Recommande les articles cliqués par les “voisins”
- Fonctionne sur un sous-échantillon du dataset
- Modèle simple → bon point de comparaison

KNN With Means

- Prédiction basée sur les catégories, pas sur les articles
- Utilise un anti-testset pour scorer toutes les catégories non vues
- Sélection des articles populaires de la catégorie prédite
- Nécessite un sous-échantillon (limites mémoire Surprise)

Limites des approches collaborative

- Dépend fortement du volume d’interactions
- Inefficace pour les nouveaux utilisateurs / nouvelles catégories
- Coûts mémoire importants (OOM → sous-échantillonnage)
- Ne prédit que des catégories, pas des articles
- Scalabilité limitée sur de grands datasets

Content-Based Filtering

Modèle Content-Based

- Basé sur les préférences catégorie de l'utilisateur
- Calcul d'une corrélation entre catégories (matrice user–category)
- Sélection des catégories les plus similaires
- Recommandation des articles populaires dans ces catégories

Modèle FAISS (Similarité vectorielle)

- Index FAISS créé sur 360k embeddings articles
- Profil utilisateur = moyenne des embeddings vus
- Recherche des articles les plus proches dans l'index
- Exclusion des articles déjà consultés
- Fallback : recommandations globales si aucun historique

Modèle Hybride (CF + FAISS)

- Combine KNN With Means (CF) + FAISS (embeddings)
- Normalisation des scores CF & CB
- Fusion
- Prise en compte de la catégorie de l’article
- Tri + sélection du Top-K articles
- Fallback : CF ou FAISS seul si données manquantes

Comparaison

Comparaison et faiblesses

Méthode

- **Train/Test split (80/20)** sur les interactions
- **Échantillon d'utilisateurs** pour accélérer les calculs
- **Top-K recommandations** générées pour chaque modèle
- **4 métriques standard** :
  - Hit Rate
  - Precision@K
  - Recall@K
  - nDCG@K
- **Format de sortie uniformisé** pour comparer tous les modèles

Résultat global

- **FAISS** : seul modèle avec des scores significatifs
- **CF (KNN)** : ≈ 0 → limité car basé sur **catégories**, dataset réduit
- **Hybride** : tiré vers le bas par la faiblesse de la partie CF

```
{'knn_baseline': {'hit_rate': 0.0,
  'precision': 0.0,
  'recall': 0.0,
  'ndcg': 0.0},
 'knn_means': {'hit_rate': 0.0, 'precision': 0.0, 'recall': 0.0, 'ndcg': 0.0},
 'cb_category': {'hit_rate': 0.0,
  'precision': 0.0,
  'recall': 0.0,
  'ndcg': 0.0},
 'cb_faiss': {'hit_rate': 0.0121580547112462,
  'precision': 0.0028368794326241137,
  'recall': 0.003174507929111898,
  'ndcg': 0.0034890989899190607},
 'hybrid': {'hit_rate': 0.0, 'precision': 0.0, 'recall': 0.0, 'ndcg': 0.0}}
```

Ce qu'on pourrait mettre en place

- **1. Split de la population**
  - Groupe A → Modèle 1
  - Groupe B → Modèle 2
    (répartition aléatoire, équilibrée)
- **2. Observation des comportements**
  - clics
  - temps passé
  - conversions
  - taux d'engagement global
- **3. Analyse statistique**
  - comparaison des métriques clés
  - test de significativité (p-value, uplift)
- **4. Décision**
  → Le modèle qui génère **le plus d'interactions positives** est retenu pour le déploiement complet.

Architecture & Backend

Architecture du système de recommandation

- L'utilisateur interagit avec un **front Streamlit local** qui envoie une requête vers l'**Azure Function**.
- L'Azure Function récupère les données nécessaires dans le **Table Storage** (utilisateurs, articles) et le **Blob Storage** (embeddings).
- Les différents modèles ML disponibles (FAISS, Content-Based, Collaborative Filtering) sont chargés côté backend.
- Selon le cas :
    - **Cold-start → modèle Content-Based**
    - **Utilisateur actif → FAISS (similarité d'embeddings)**
- Le backend renvoie un **Top-5 articles recommandés**, affiché dans Streamlit.

Azure Functions : fonctionnement serverless

Pas de serveur à gérer
→ Microsoft Azure alloue automatiquement les ressources nécessaires.

Activation à la demande
→ La fonction ne s'exécute **que lorsqu'elle reçoit une requête** (ex. /recommend_faiss?user_id=X).

Scalabilité automatique
→ Azure crée autant d'instances que nécessaire en cas de forte charge.

Coût à l'usage
→ Tu ne paies que le **temps d'exécution** de la fonction, idéal pour un MVP.

Déploiement simple
→ Un dossier contenant ton code + un fichier function.json suffit pour publier la fonction.

Stockage des données

Table Storage – Utilisateurs
→ Liste des user_id autorisés pour FAISS (utilisateurs ayant au moins 1 clic).
→ Utilisé pour vérifier si l'utilisateur existe dans l'historique.

Table Storage – Articles
→ Mapping *article_id* ↔ *faiss_idx* + catégorie.
→ Permet de relier embeddings, FAISS et résultats CF.

Blob Storage – Embeddings
→ Stockage du fichier embeddings.npy (≈ 360k vecteurs).
→ Chargé par la fonction Azure au démarrage.

Gestion des nouveaux utilisateurs

Cas 1 : utilisateur connu
→ Historique de clics disponible → calcul du *profil moyen* → recommandations FAISS/CB/Hybrid.

Cas 2 : nouvel utilisateur (cold-start)
→ Aucun clic → impossible d'estimer un embedding moyen.
→ Retour automatique d'une **recommandation Content-Based** via catégorie *globale* (les articles les plus populaires au niveau métier).

Pipeline de mise à jour : nouveaux articles & embeddings

1. Ajout des nouveaux articles

- Mise à jour du fichier *articles*.

2. Génération des embeddings

- Nouveau texte → nouveaux vecteurs.

3. Mise à jour FAISS

- Ajout des embeddings dans l'index.
- Mise à jour des métadonnées.

4. Stockage (Azure)

- Upload des embeddings + metadata.

5. Rechargement automatique

- Azure Function recharge l'index au prochain appel.

Optimisation (PCA, modèle compressé, FAISS)

PCA

- Réduction de dimension des embeddings.
- ↓ Taille mémoire, ↑ vitesse de recherche.

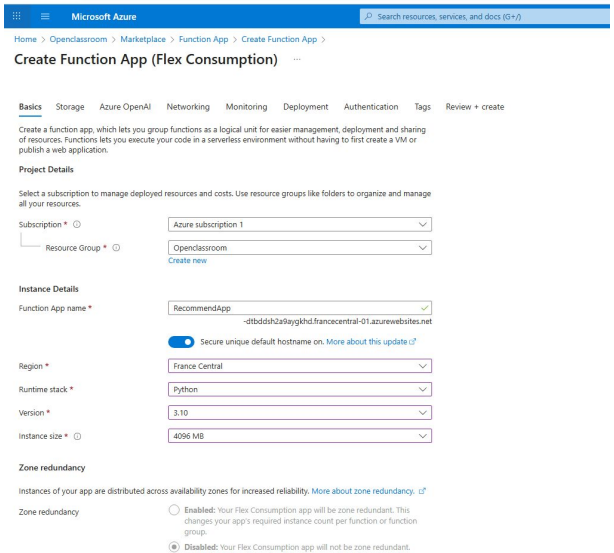
Modèle compressé

- Embeddings plus légers → stockage et chargement plus rapides.
- Moins de risques d'Out-Of-Memory.

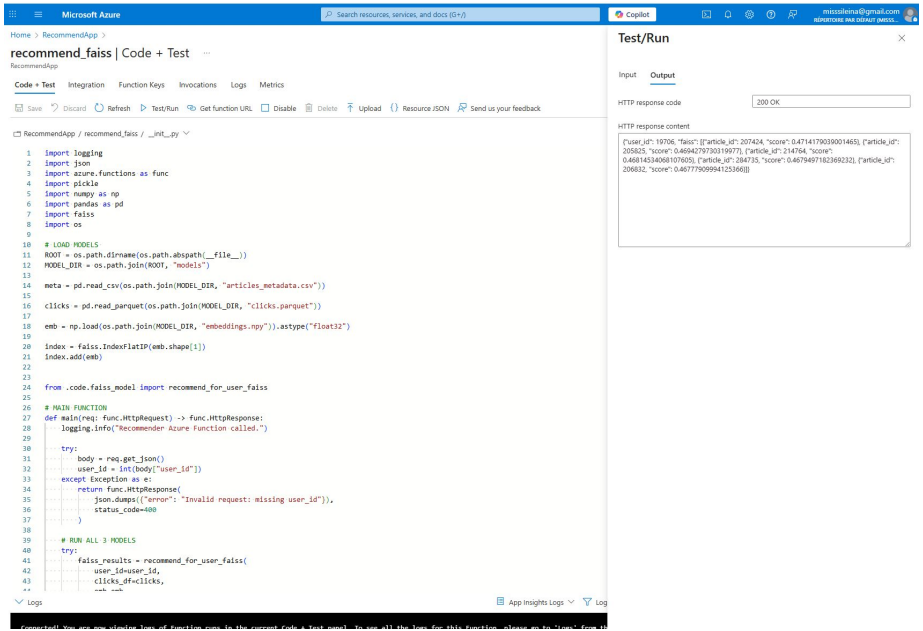
FAISS

- Index vectoriel optimisé (GPU/CPU).
- Recherche ultra-rapide parmi >300k articles.
- Supporte facilement la montée en charge.

Application & Démonstration




```
root@DESKTOP-T1RQA91:/home/my-reco-functions# func azure functionapp publish RecommendApp
'local.settings.json' found in root directory (/home/my-reco-functions).
Resolving worker runtime to 'python'.
Local python version '3.12.3' is different from the version expected for your deployed Function App. This may result in 'ModuleNotF
ound' errors in Azure Functions. Please create a Python Function App for version 3.12 or change the virtual environment on your loc
al machine to match '3.10'.
Getting site publishing info...
[2025-12-10T10:02:08.598Z] Starting the function app deployment...
[2025-12-10T10:02:08.602Z] Creating archive for current directory...
Performing remote build for functions project.
Uploading 691.23 MB [##############################################################################]
Deployment in progress, please wait...
Starting deployment pipeline.
[Kudu-SourcePackageUriDownloadStep] Skipping download. Zip package is present at /tmp/zipdeploy/93303c64-8cfe-4587-bf58-2cf0fa67480
c.zip
[Kudu-ValidationStep] starting.
[Kudu-ValidationStep] completed.
[Kudu-ExtractZipStep] starting.
[Kudu-ExtractZipStep] completed.
[Kudu-ContentValidationStep] starting.
[Kudu-ContentValidationStep] completed.
[Kudu-PreBuildValidationStep] starting.
[Kudu-PreBuildValidationStep] completed.
[Kudu-OryxBuildStep] starting.
[Kudu-OryxBuildStep] completed.
[Kudu-PostBuildValidationStep] starting.
[Kudu-PostBuildValidationStep] completed.
[Kudu-PackageZipStep] starting.
[Kudu-PackageZipStep] completed.
[Kudu-UploadPackageStep] starting.
[Kudu-UploadPackageStep] completed. Uploaded package to storage successfully.
[Kudu-RemoveWorkersStep] starting.
[Kudu-RemoveWorkersStep] completed.
[Kudu-SyncTriggerStep] starting.
[Kudu-CleanUpStep] starting.
[Kudu-CleanUpStep] completed.
Finished deployment pipeline.
Checking the app health...Host status endpoint: https://recommendapp-dtbddsh2a9aygkhd.francecentral-01.azurewebsites.net/admin/host
/status
. done
Host status: {"id":"3c39222feba1f6889014fafe0f0a9e08","state":"Running","version":"4.1044.400.0","versionDetails":"4.1044.400-dev.0
```


Microsoft Azure
Search resources, services, and docs (G+/)
Copilot
missslelna@gmail.com
Home > RecommendApp >
recommend_faiss | Code + Test
RecommendApp
Code + Test
Integration
Function Keys
Invocations
Logs
Metrics
Save
Discard
Refresh
Test/Run
Get function URL
Disable
Delete
Upload
Resource JSON
Send us your feedback
RecommendApp / recommend_faiss / __init__.py
import logging
import json
import azure.functions as func
import pickle
import numpy as np
import pandas as pd
import faiss
import os
# LOAD MODELS
ROOT = os.path.dirname(os.path.abspath(__file__))
MODEL_DIR = os.path.join(ROOT, "models")
meta = pd.read_csv(os.path.join(MODEL_DIR, "articles_metadata.csv"))
clicks = pd.read_parquet(os.path.join(MODEL_DIR, "clicks.parquet"))
emb = np.load(os.path.join(MODEL_DIR, "embeddings.npy")).astype("float32")
index = faiss.IndexFlatIP(emb.shape[1])
index.add(emb)
from .code.faiss_model import recommend_for_user_faiss
# MAIN FUNCTION
def main(req: func.HttpRequest) -> func.HttpResponse:
logging.info("Recommender Azure Function called.")
try:
body = req.get_json()
user_id = int(body["user_id"])
except Exception as e:
return func.HttpResponse(
json.dumps({"error": "Invalid request: missing user_id"}),
status_code=400
)
# RUN ALL 3 MODELS
try:
faiss_results = recommend_for_user_faiss(
user_id=user_id,
clicks_df=clicks,
Logs
App Insights Logs
Test/Run
Input
Output
HTTP response code
200 OK
HTTP response content
{"user_id": 19706, "faiss": [{"article_id": 207424, "score": 0.4714179039001465}, {"article_id": 205825, "score": 0.4694279730319977}, {"article_id": 214764, "score": 0.4681453406810765}, {"article_id": 284735, "score": 0.4679497182369232}, {"article_id": 206832, "score": 0.4677790999425366}]}

Interface Streamlit : présentation générale

Local Recommendations — ALL MODELS (KNNMeans / FAISS / Hybrid)

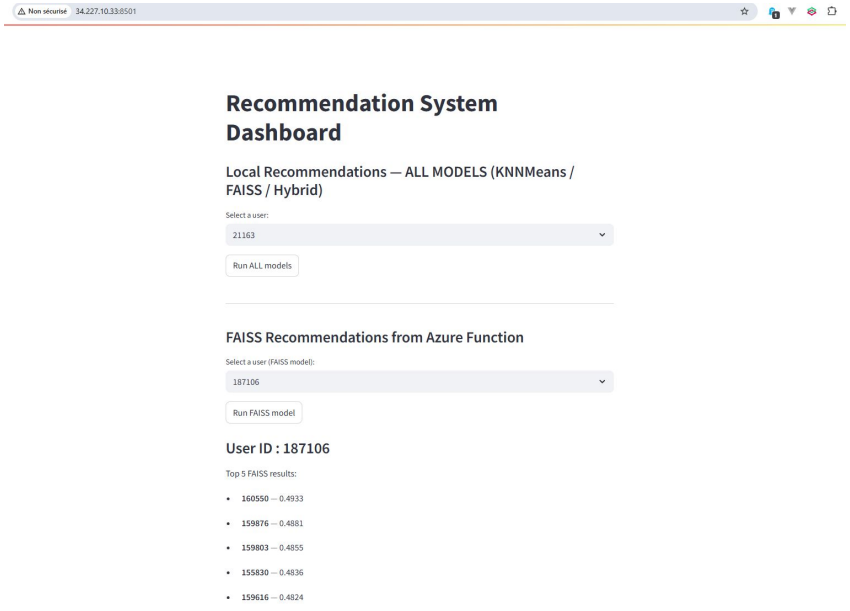
Select a user:

13006

Run ALL models

User ID : 13006

KNNMeans	FAISS	Hybrid
160974 — 37213.0000	224278 — 0.5575	160974 — 0.5000
234698 — 23499.0000	225036 — 0.5566	224278 — 0.5000
88914 — 66.0000	224956 — 0.5554	225036 — 0.4449
226240 — 15.0000	225058 — 0.5538	224956 — 0.3764
78432 — 3.0000	222307 — 0.5491	234698 — 0.3157



Conclusion