**ROMÂNIA**

**MINISTERUL APĂRĂRII NAŢIONALE**

**ACADEMIA TEHNICĂ MILITARĂ „FERDINAND I”**

**FACULTATEA DE SISTEME INFORMATICE ȘI SECURITATE CIBERNETICĂ**

## Specializarea: Calculatoare şi sisteme informatice pentru apărare şi securitate naţională

****

***SISTEM DE RECOMANDARE FOLOSIND TEHNICI DE MACHINE LEARNING***

CONDUCĂTOR ŞTIINŢIFIC:

# Lect. dr. ing. Stelian Spînu

ABSOLVENT:

## Banu Ion

Conţine file Inventariat sub nr. Poziţia din indicator: Termen de păstrare:

**BUCUREŞTI2025**

DECLARAŢIE PE PROPRIE RĂSPUNDERE

PRIVIND ORIGINALITATEA CONŢINUTULUI LUCRĂRII DE

DIPLOMĂ/DISERTAŢIE

Subsemnatul(a)................................................................................domiciliat(ă) în .........................................str............................................ nr. .. sc. ..... et. ..... ap. ..... jud. ............................., posesor al C.I. seria ........ nr. ............ C.N.P. .................... eliberat de ........................, autorul lucrării de diplomă/disertaţie cu titlul ............................................................................................................................

………………………………………………………………………………………. elaborată în vederea susţinerii examenului de finalizare a studiilor în sesiunea ........a anului universitar................ declar pe propria răspundere că această lucrare este rezultatul propriei activităţi intelectuale, nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Data, Semnătura,

# Abstract

In a setting where the availability of video content is constantly expanding, users eventually want to watch a movie more often than not. This reality has shaped the idea of creating a recommendation system that will help filter options and generate suggestions based on each person’s preferences.This thesis

describes the process of developing and implementing an intelligent movie recommendation system using automated learning algorithms and personalized approaches.

The system was designed to analyze user history, ratings, preferred keywords, and other pertinent features in order to identify preferred categories and generate relevant suggestions. For this purpose, techniques such as collaborative filtering and content analysis were used, integrating into a hybrid model that could dynamically adapt the received data. The application’s results are not limited to recommendations; it can also learn from user behavior to improve the accuracy of its recommendations over time.

The implementation was carried out in a user-friendly online environment where users can interact with the platform with ease and the results are presented in an understandable and intuitive manner. The main goal was for the system to be functional, effective, and simple to expand rather than relying solely on theoretical demonstration.

This work demonstrates how machine learning algorithms can be used in real-world applications that are accessible to any user who wants a more personalized viewing experience.

# Rezumat

Într-un mediu în care cantitatea de conținut video crește necontenit, utilizatorii doresc, de cele mai multe ori, să urmărească un film fără să piardă timpul cu căutarea lui. Pornind de la această realitate, am propus elaborarea unui sistem de recomandari care să simplifice procesul de selecție și să ofere sugestii relevante, adaptate preferințelor fiecărui utilizator.

Lucrarea de față descrie etapele de implementare a unui astfel de sistem inteligent de recomandare a filmelor, construit cu ajutorul algoritmilor de învățare automată și prin metode personalizate. Sistemul a fost proiectat să analizeze istoricul de vizionare, evaluările acordate, cuvintele-cheie preferate și alte caracteristici relevante, în scopul identificării celor mai potrivite recomandări.

Pentru a obține rezultate cât mai precise, am combinat tehnici de filtrare colaborativă cu analiza conținutului, într-un model hybrid activ, capabil să se adapteze dinamic în funcție de noile date primite. Sistemul nu furnizează doar recomandări statice, ci învață continuu pe baza comportamentului utilizatorilor, îmbunătățindu-și astfel precizia în timp.

Implementarea a fost realizată într-un mediu online intuitiv și prietenos, astfel încât platforma să poată fi utilizată de orice categorie de utilizatori, iar rezultatele să fie afișate într-un mod clar și ușor de înțeles. Principalul obiectiv al proiectului nu a fost doar demonstrarea teoretică a unor concepte, ci dezvoltarea unui sistem funcțional, eficient și ușor de extins în viitor.

Această lucrare demonstrează aplicabilitatea algoritmilor de învățare automată în crearea unor soluții reale, care oferă fiecărui utilizator o experiență de vizionare cu adevărat personalizată.

# Cuprins

[Listă de abrevieri 10](#_Toc202461026)

[Listă de figuri 12](#_Toc202461027)

[Listă de tabele 13](#_Toc202461028)

[1. Introducere 14](#_Toc202461029)

[1.1 Importanța temei 14](#_Toc202461030)

[1.2 Scopul lucrării 15](#_Toc202461031)

[1.3 Obiectivele proiectului 15](#_Toc202461032)

[1.3.1 Cerințe funcționale 16](#_Toc202461033)

[1.3.2 Cerințe nefuncționale 18](#_Toc202461034)

[1.4 Rezumatul lucrării pe capitole 19](#_Toc202461035)

[2. Colectare și preprocesare date 21](#_Toc202461036)

[2.1 Surse de date 21](#_Toc202461037)

[2.1.1 TMDb API 21](#_Toc202461038)

[2.2 Curățare și transformări 22](#_Toc202461039)

[2.2.1 Încărcarea și validarea datelor 22](#_Toc202461040)

[2.2.2 Tokenizare și curățare text (overview, keywords) 23](#_Toc202461041)

[2.2.3 Extracție de caracteristici: genuri (one-hot) și straturi textuale 24](#_Toc202461042)

[2.3 Construirea matricelor 26](#_Toc202461043)

[2.3.1 Matricea user–item 26](#_Toc202461044)

[2.3.2 Matricea document–termen 27](#_Toc202461045)

[2.3.3 Matricea click-based 28](#_Toc202461046)

[3. Metode de recomandare 31](#_Toc202461047)

[3.1 Filtrare Colaborativa 31](#_Toc202461048)

[3.1.1 CF-SVD cu biblioteca Surprise 31](#_Toc202461049)

[3.1.2 CF-Weighted Mean (media ponderată), user-based 34](#_Toc202461050)

[3.1.3 CF bazată pe click-uri (Click-Based Collaborative Filtering) 37](#_Toc202461051)

[3.1.4 CF-KNN cu biblioteca Surprise (K-Nearest Neighbors) 38](#_Toc202461052)

[3.1.5 CF prin modele Neurale (NN-CF, GMF+MLP, Autoencoder, GNN) 38](#_Toc202461053)

[3.2 Recomandări bazate pe conținut (Content-Based) 45](#_Toc202461054)

[3.2.1 Similitudinea cosinus 45](#_Toc202461055)

[3.2.2. Modele avansate bazate pe conținut(NMF,BERT) 46](#_Toc202461056)

[3.3 Model Hibrid 47](#_Toc202461057)

[4. Arhitectura Sistemului 49](#_Toc202461058)

[4.1 Viziune de ansamblu 49](#_Toc202461059)

[4.2 Front-end 52](#_Toc202461060)

[5. Back-end 57](#_Toc202461061)

[5.1 Rute principale 57](#_Toc202461062)

[5.1.1 Rute de înregistrare și autentificare 57](#_Toc202461063)

[5.1.2 Rute de activare și verificare 2FA 60](#_Toc202461064)

[5.1.3 Rute pentru gestionarea sesiunii și logout 61](#_Toc202461065)

[5.1.4 Rute pentru prietenii și mesaje (API JSON) 61](#_Toc202461066)

[5.1.5 Rute pentru crearea, citirea și ștergerea utilizatorilor (admin) 65](#_Toc202461067)

[5.1.6 Pagină principală și paginare de filme 67](#_Toc202461068)

[5.1.7 Pagină de profil 70](#_Toc202461069)

[5.1.8 Ruta de căutare filme 71](#_Toc202461070)

[5.1.9 Ruta detalii film 72](#_Toc202461071)

[5.1.10 Endpoints AJAX pentru voturi și recenzii 74](#_Toc202461072)

[5.1.11 Autocomplete și liste de titluri filme (JSON) 75](#_Toc202461073)

[5.1.12 Rute AJAX și webhook-uri diverse 76](#_Toc202461074)

[5.2 Servicii 78](#_Toc202461075)

[5.2.1 Rute de load/initializare a modelelor și embeddingurilor 78](#_Toc202461076)

[5.2.2 Funcții auxiliare (nu expuse direct ca rute) 79](#_Toc202461077)

[5.3 Mecanisme de Securitate 81](#_Toc202461078)

[5.3.1 Protecția autentificării și a parolelor 81](#_Toc202461079)

[5.3.2 Mecanisme de securitate HTTP și infrastructură 82](#_Toc202461080)

[6. Testare și rezultate 83](#_Toc202461081)

[6.1 Setup experimental 83](#_Toc202461082)

[6.2 Metrici 85](#_Toc202461083)

[6.3 Rezultate pentru Filtrare Colaborativa 86](#_Toc202461084)

[6.4 Rezultate Content Based 93](#_Toc202461085)

[6.5 Netflix Prize 93](#_Toc202461086)

[7. Concluzii și viitoare îmbunătățiri 95](#_Toc202461087)

[8. Bibliografie 97](#_Toc202461088)

# Listă de abrevieri

1. 2FA – **Two-Factor Authentication**
2. AJAX – Asynchronous JavaScript And XML
3. ALS – Alternating Least Squares
4. API – Application Programming Interface
5. BERT **– Bidirectional Encoder Representations from Transformers**
6. CB – Content Based
7. CF – Collaborative Filtering
8. CNN **–** Convolutional Neural Network
9. CRC **–**  Click-Through Rate
10. CRUD – Create,Read,Update,Delete
11. CSS – Cascading Style Sheets
12. **DB** – DataBase
13. FIFO – First In First Out
14. GMF – Generalized Matrix Factorization
15. GNN **–** Graph Neural Network
16. HTML – Hypertext Markup Language
17. HTTP – Hypertext Transfer Protocol
18. JSON – JavaScript Object Notation
19. KNN – K-Nearest Neighbors
20. MAE **–** Mean Absolute Error
21. ML – Machine Learning
22. NeuMF – Neural Matrix Factorization
23. NLP – Natural Language Processing
24. NMF – Non-negative Matrix Factorization
25. NN – Neuronal Networks
26. OCR **–** Optical Character Recognition
27. ORM – **Object-Relational Mapping**
28. OTP – **One-Time Password**
29. PC – Program Counter
30. PIL **–** Python Imaging Library
31. RMSE – Root Mean Squared Error
32. RNN – Recurrent Neural Network
33. SAE– Stacked AutoEncoder
34. SVD – Singular Value Decomposition
35. TF-IDF – Term Frequency - Inverse Document Frequency
36. TOTP – Time-based One-Time Password
37. UI – User Interface
38. URI – Uniform Resource Identifier
39. UX – User Experience
40. XSS – Cross-Site Scripting

# Listă de figure

[Fig. 2.1: Funcția update\_movie 1 21](#_Toc202469575)

[Fig. 3.1: Factorizarea matricei de scoruri utilizator-film prin SVD 32](#_Toc201954432)

[Fig. 3.2: Predicția scorurilor lipsă pentru utilizatorii parțial observați[15] 33](#_Toc201954433)

[Fig. 3.3: Predicția valorilor necunoscute[15] 34](#_Toc201954434)

[Fig. 3.4: Matrice de similaritate între utilizatori[1] 35](#_Toc201954435)

[Fig. 3.5: Model neural bazat pe embeddinguri 40](#_Toc201954436)

[Fig. 3.6: Distanța cosinus[1] 45](#_Toc201954437)

[Fig. 4.1: Arhitectura sistemului **Error! Bookmark not defined.**](#_Toc201954421)

[Fig. 4.2: Pagina home 53](#_Toc201954422)

[Fig. 4.3: Pagina login 54](#_Toc201954423)

[Fig. 4.4: Pagina movie detail 55](#_Toc201954424)

[Fig. 4.5: Pagina prieteni 56](#_Toc201954425)

[Fig. 5.1: Fluxul de înregistrare 58](#_Toc201954426)

[Fig. 5.2: Fluxul de autentificare 59](#_Toc201954427)

[Fig. 5.3: Fluxul de generare al recomandărilor colaborative 70](#_Toc201954428)

[Fig. 5.4: Fluxul de căutare film 72](#_Toc201954429)

[Fig. 5.5: Fluxul de generare filtrare bazat pe conținut 73](#_Toc201954430)

[Fig. 5.6: Fluxul de generare filtrări hibride 74](#_Toc201954431)

[Fig. 6.1: Evoluția Loss pentru MLP[5] 88](#_Toc201954438)

[Fig. 6.2: Evoluția Loss pentru MLP+GMF 89](#_Toc201954439)

[Fig. 6.3: Euromed University of Fez[2] 89](#_Toc201954440)

[Fig. 6.4: Evoluția Loss MLP+GMF[11] **Error! Bookmark not defined.**](#_Toc201954441)

[Fig. 6.5: Autoencoder[14] 92](#_Toc201954442)

[Fig. 6.6: Rezultat Euromed University of Fez(one epoch) 92](#_Toc201954443)

[Fig. 6.7: Rezultat Euromed University of Fez(two epoch) 92](#_Toc201954444)

# Listă de tabele

[Tabel 6.1: Metrica RMSE pe DB 86](#_Toc201959483)

[Tabel 6.2: Metrica RMSE pe DB pentru algoritmi neuronali 87](file:///C:\Users\BANU\Downloads\Lucrare_diploma_banu_ion%20(Autosaved)%20(3).docx#_Toc201958931)

[Tabel 6.3: Metrica RMSE pentru 70000 ratings 90](#_Toc201958932)

[Tabel 6.4: Metrica RMSE pentru 100000 ratings 90](#_Toc201958933)

[Tabel 6.5: Metrica RMSE pentru un million ratings 91](#_Toc201958934)

[Tabel 6.6: Metrica RMSE pentru 10 milioane ratings 92](#_Toc201958935)

# Introducere

## Importanța temei

Înainte de apariția platformelor de streaming, alegerea unui film sau a unui serial a însemnat un efort redus de navigare într-o bibliotecă fizică din incinta cazării sau de selectare a câtorva canale TV. Astăzi, însă, serviciile precum Netflix, Amazon Prime Video, Disney+ sau YouTube pun la dispoziție zeci de mii de titluri, iar oferta crește în permanență. De la producții clasice la lansări originale, varietatea copleșitoare poate genera un fenomen cunoscut în literatura de specialitate drept **„choice overload”**, adică un exces de opțiuni care, paradoxal, îngreunează decizia de cumpărare sau consum și poate duce la frustrare și abandon.

Din punctul de vedere al experienței utilizatorului (UX), căutarea manuală a unui titlu potrivit devine un proces îndelungat și ineficient: navigare pe conținut irelevant, clicuri repetate, vizionarea unui trailer pentru a convinge utilizatorul că merită sau citirea părerilor altora. Toate acestea pot genera o senzație de satisfacție redusă, scăzând timpul efectiv petrecut pe platformă și crescând rata de abandon.

Un sistem de recomandari bine pus la punct răspunde tocmai acestei nevoi: acționează ca un asistent personalizat care filtrează automat conținutul, învață din preferințele și comportamentele fiecărui utilizator și livrează sugestii într-un timp foarte scurt. Implicația directă este o navigare mai plăcută, descoperirea rapidă a conținutului relevant și creșterea nivelului de implicare (engagement), ceea ce, în timp, se traduce prin fidelizarea abonaților și scăderea ratelor de abandon. Prin urmare, sistemele de recomandari nu reprezintă doar un lux, ci o componentă esențială pentru orice platformă de streaming care dorește să ofere o experiență optimă utilizatorilor săi.

## Scopul lucrării

**Scopul principal al proiectului** este dezvoltarea unui sistem de recomandare de filme cap-la-cap, care să îmbine atât partea de backend (algoritmii de învăţare automată), cât şi cea de frontend (interfaţa HTML/JavaScript/CSS), într-o aplicaţie web complet funcţională. Se urmărește:

* Integrarea în main.py a diferitelor strategii (filtrare colaborativă, analiză de conţinut, reţele neuronale simple) şi compararea performanţelor lor pe un corpus de date.
* Crearea unui model hibrid care să combine punctele forte ale fiecărei metode şi să ofere cele mai relevante recomandări, evaluând calitatea prin metrici precum RMSE, precizie şi recall.
* Dezvoltarea unei interfeţe web prietenoase.
* Realizarea unui panou principal (home.html) unde utilizatorul se poate autentifica, poate vizualiza istoricul de vizionare şi poate descoperi filme recomandate, apoi poate accesa pagina de detaliu a fiecărui titlu (movie\_detail.html).
* Asigurarea unei experienţe fluide prin JavaScript pentru încărcarea dinamică a datelor (apeluri AJAX către backend) şi CSS responsiv, adaptabil la orice dimensiune de ecran.
* Validarea şi rafinarea soluţiei: testare cu grupuri de utilizatori pentru a colecta feedback privind uşurinţa în utilizare şi relevanţa recomandărilor, precum şi corectarea eventualelor erori.
* Documentarea completă a procesului, de la configurarea bazei de date şi a pipeline-ului de preprocesare, până la modul în care main.py integrează API-urile de recomandare cu rutele serverului web şi fişierele HTML/CSS.

Prin atingerea acestor obiective, proiectul demonstrează nu doar capacitatea de implementare a algoritmilor de învățare automată, ci şi livrarea unei aplicaţii funcţionale, pregătite pentru utilizare într-un context real de streaming video.

## Obiectivele proiectului

În primul rând, crearea unei „coloane vertebrale” stabile pentru sistemul nostru de recomandare: vom extrage din baza de date toate metadatele existente pentru filmele respective — titlu, genurile asociate, distribuție, descriere și rating-uri — și vom consemna, în paralel, comportamentul utilizatorilor (istoric de vizionare, note acordate sau click-uri pe „Îmi place”). În codul Python, vom preprocesa cu atenție aceste date, curățându-le și standardizându-le pentru a asigura coerența ulterioară.

Propunem implementarea eficientă a ambelor paradigme majore de recomandare: filtrarea colaborativă, în care estimăm similaritățile între utilizatori folosind matrici rare și tehnici de factorizare (SVD), și analiza conținutului, în care, prin TF-IDF și modele de învățare automată, identificăm caracteristicile cheie ale fiecărui film. Apoi, printr-un model hibrid, vom combina recomandările generate de cele două metode, echilibrându-le astfel încât să obținem sugestii cât mai relevante.

În paralel, vom dezvolta o interfață web care să afișeze în timp real rezultatele algoritmilor: paginile HTML și scripturile JavaScript vor face apeluri asincrone (AJAX) către serverul Python, solicitând recomandările și afișându-le imediat — în mai puțin de 200 ms, pentru a menține fluiditatea navigării.

Am introdus, de asemenea, un mecanism de feedback continuu, în care fiecare nouă apreciere sau click al userului declanșează reactualizarea modelelor din backend și o reînvățare periodică, pentru ca sugestiile să devină tot mai precise pe măsură ce sistemul „conștientizează” din ce în ce mai bine.

În cele din urmă, ne dorim ca întregul proiect să fie modular și ușor de extins: am despărțit logica algoritmică și componentele frontend și backend în fișiere și funcții separate, așa cum apare reflectat și în arhitectura codului Python și în structurile directoarelor HTML/JS/CSS, pentru ca fiecare să poată adăuga rapid o nouă filtrare, o nouă pagină sau un alt algoritm de recomandare. Astfel, ne asigurăm că nu dispunem de doar un prototip teoretic, dar de un produs software solid, potrivit de extensibil și gata pentru viitoare îmbunătățiri.

## Cerințe funcționale

 **Încărcarea și preprocesarea datelor de film**

Citirea fișierului movies.csv (sau echivalent) și extragerea metadatelor:

titlu, gen, descriere, rating.

Construirea textelor sub formă de matrice document–termen pentru analiza de conținut (TF-IDF).

 **Capturarea comportamentului utilizatorului**

Înregistrarea istoricului de vizionări și a rating-urilor date de utilizator (prin

apeluri Python din main.py).

Stocarea interacțiunilor (click-uri „Îmi place”, adăugări la favorit) într-o

structură de date accesibilă pentru model.

 **Recomandări bazate pe filtrare colaborativă**

Calcularea similarităților între utilizatori folosind factorizare matricială

(SVD) sau „vecini” KNN.

Generarea listei de top-N filme recomandate pentru fiecare utilizator activ.

 **Recomandări bazate pe analiză de conținut**

Extracția vectorilor TF-IDF din descrierile filmelor.

Determinarea similarității cosinus între filme pentru a sugera titluri

asemănătoare.

 **Combinarea metodelor într-un model hibrid**

Combinarea scorurilor colaborative și de conținut cu o pondere configurabilă

în cod (main.py).

Reglarea (tuning) ponderilor pe baza feedback-ului utilizatorilor.

 **Interfață web de prezentare a recomandărilor**

Pagina principală (home.html):

Afișarea unui slider cu „Recomandări pentru tine” încărcat din main.py

prin AJAX.

Bara de căutare live care filtrează titlurile cu JavaScript.

Pagina detaliu film (movie\_detail.html):

Informații extinse despre film (sinopsis, gen, rating mediu).

Recomandări „Filme similare” generate din modelul de conținut.

 **Mecanism de feedback și reantrenare**

La primirea unui nou rating sau click „Îmi place” din interfața HTML,

trimiterea unui request POST către endpoint-ul Python.

Recalcularea periodică a modelelor (batch retraining nightly) și reîncărcarea

lor în memorie.

 **Performanță și scalabilitate**

Asigurarea timpului de răspuns sub 200 ms pentru solicitările de

recomandare.

Structurare modulară a codului (main.py, recommender.py,

data\_loader.py) și a fișierelor front-end (\*.html, \*.js, \*.css)

pentru extindere facilă.

 **Securitate și acces**

Validarea intrărilor din formularul de căutare și rating pentru a preveni

injecțiile de cod.

Gestionarea sesiunilor de utilizator pentru a afișa recomandări personalizate

doar după autentificare (opțional).

## Cerințe nefuncționale

 **Performanţă**

Pregenerarea (caching) matricelor de similaritate şi a modelelor de filtrare

colaborativă la pornirea aplicaţiei, pentru a nu recalcula la fiecare solicitare.

 **Scalabilitate**

Componentele de recomandare (SVD, NMF, click-based, content-based) să

poată fi rulate în paralel sau mutate uşor pe un cluster distribuit.

Baza de date SQLite poate fi înlocuită cu un motor SQL (PostgreSQL,

MySQL) fără modificări majore de cod.

 **Disponibilitate & fiabilitate**

Mecanism de fallback la recomandări bazate pe fallback (baseline = rating

mediu 3.0) dacă modelul principal nu este încărcat sau nu are date suficiente.

Protecţie împotriva erorilor de reţea (TMDb API) prin tratare explicită a

codurilor de răspuns şi retry cu back-off.

 **Securitate**

Talisman + Content Security Policy pentru toate rutele HTTP (header-uri de

protecţie XSS).

Limitare de rată (Flask-Limiter) de 10 cereri/minut pentru endpoint-urile de

login şi API-urile critice.

Toate parolele stocate hashed cu

werkzeug.security.generate\_password\_hash, verificare cu

check\_password\_hash.

Autentificare cu 2FA (pyotp + QR code) pentru autentificare robustă.

 **Portabilitate**

Configurare TMDB\_API\_KEY, calea către model (model.h5,

tokenizer.pkl) şi secret\_key în variabile de mediu, nu hard-codate.

Aplicaţia rulează atât sub Windows, cât şi Linux, fără dependenţe native

imposibil de instalat (foloseşte doar biblioteci Python pure).

 **Extensibilitate & mentenabilitate**

Arhitectură modulară: funcţii separate pentru fiecare strategie de

recomandare (cf-user, cf-clicks, content, hybrid etc.)

Folosirea unui ORM minim (sqlite3 + pandas pentru citire) poate fi înlocuit

cu SQLAlchemy fără refactor major.

Structura proiectului clar delimitată în main.py, şabloane HTML, CSS şi

JS în /templates şi /static, pentru viitoare adăugiri de UI.

 **Usabilitate**

Front-end realizat cu HTML/CSS/JS simplu, formulare intuitive de

înregistrare şi login, pagini de detaliu film

Mesaje de feedback (flash) pentru utilizator la acţiuni critice (înregistrare,

login, erori de 2FA).

 **Observabilitate & debuggability**

Logging detaliat în fişierul rate\_limit\_logs.log pentru evenimente

de limitare rate-limit.

Print-uri de debug pentru distanţe, predicţii şi erori de API, pentru audit şi

tuning.

Adăugarea unui dashboard de monitorizare a performanţei (ex.

Prometheus/Grafana) apelând punctele de hook din cod.

## Rezumatul lucrării pe capitole

1. **Capitolul 2 (Colectare și preprocesare date)** descrie modul în care extragem din TMDb API informațiile esențiale despre filme (titlu, genuri, descriere, actori) și modul în care le stocăm local, în SQLite. Descriem pașii de curățare (eliminarea duplicatelor, completarea valorilor lipsă) şi tehnicile de transformare a textelor (tokenizare, TF-IDF/CountVectorizer) pentru a construi matricile user–item, document–termen și click-based necesare algoritmilor.
2. **Capitolul 3 (Metode de recomandare)** descrie algoritmii pe care îi folosim: filtrarea colaborativă (SVD şi KNNBasic), recomandările content-based bazate pe TF-IDF şi similaritate cosinus, precum și abordarea click-based ce acordă greutate click-urilor recente. Prezentăm în final strategia hibridă prin care reunim top-urile fiecărei metode şi le reordonăm pentru a obține un rezultat echilibrat între acuratețe și diversitate.
3. **Capitolul 4 (Arhitectura sistemului)** descrie viziunea de ansamblu a soluției: front-end-ul HTML/CSS/JavaScript care oferă funcții de căutare, vizualizare recomandări şi tracking de click-uri, şi back-end-ul Flask din main.py care expune rute pentru filme, recomandări și securitate. Evidențiem modul în care modulele de recomandare (Surprise, scikit-learn, PyTorch) interacționează cu baza de date și stochează în cache răspunsurile TMDb pentru performanță.
4. **Capitolul 5 (Back-end)** explicăm cum scriptul Python inițializează datele, antrenează modelele şi salvează obiectele serializate; apoi cum serviciile Flask privesc cererile de recomandare, încarcă modelele potrivite și returnează un JSON cu top-N filme. Mai arătăm integrarea cheilor TMDb, mecanismul de caching și măsurile de securitate—rate limiting, hashing parole şi 2FA—pentru a asigura fiabilitate și protecție.
5. **Capitolul 6 (Testare rezultate)** detaliem modul în care am împărțit setul de date în train/test și cum am aplicat validare încrucișată pe modelele colaborative. Prezentăm metricile folosite (RMSE, MAE) şi comparăm performanțele fiecărei metode, ilustrând prin grafice și tabele avantajele şi limitările lor. Concluziile analizei arată combinațiile optime şi impactul ponderilor temporale.
6. **Capitolul 7** (**Concluzii și perspective**) pentru încheiere recapitulăm realizările proiectului în raport cu obiectivele inițiale (prototip funcțional, comparație de modele, interfață ușor de utilizat) și discutăm extinderile posibile: recomandări în timp real, explicații automatizate („de ce ți-am sugerat acest film?”), suport mobile și A/B testing pentru optimizarea continuă. Ce a funcționat bine și ce limitări am întâmpinat (date insuficiente pentru antrenamente NN, rate limits TMDb). Propuneri de îmbunătățire: integrarea de feedback în timp real, recomandări secvențiale (RNN/Transformer), deploy în cloud cu scalare automată, extinderea pe seriale sau videoclipuri scurte.

# Colectare și preprocesare date

## Surse de date

Pentru a construi un set de date complex, capabil să alimenteze atât algoritmi de filtrare colaborativă, cât şi modele de analiză de conţinut, dar și pentru cei de invățare profundă am pornit de la metadatele brute extrase prin TMDb API: identificator unic (movie\_id), titlu, descriere (overview), genuri (genre\_ids), data lansării, lista de actori şi membri ai echipei („cast” şi „crew”), rating-ul IMDB, popularitate şi cuvinte-cheie asociate. În paralel, am îmbinat aceste informaţii cu date sintetizate despre comportamentul utilizatorilor, istoricul de vizionare, evaluările acordate şi click-urile.

### **2.1.1 TMDb API**

The Movie Database (TMDb) oferă un API public complex, care ne-a permis să populăm catalogul de filme cu informații actualizate și detaliate. Am folosit patru endpoint-uri principale:

* /movie/popular pentru a prelua liste paginabile de filme populare, împreună cu titlul, descrierea, data lansării și genurile asociate;
* /movie/{movie\_id} pentru a obține, la cerere, date complete despre un anumit film;
* /movie/{movie\_id}/videos pentru a identifica trailer-ele aferente;
* /movie/{movie\_id}/credits pentru a extrage primii cinci actori.

Fiecare răspuns JSON a fost mapat și stocat local în tabela movies, asigurându-ne astfel că sistemul nostru poate funcționa rapid, fără dependențe directe de rețeaua externă după faza inițială de populare.

În plus faţă de popularea iniţială a tabelei movies, am inclus şi o rutină de actualizare periodică care menține catalogul sincronizat cu ultimele apariții de pe TMDb. Funcția cheie în acest sens este:

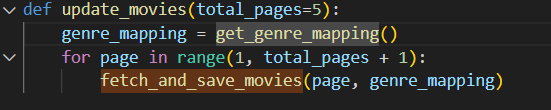


Fig. 2.1: Funcția update\_movie

Fig. 2.1: Funcția update\_movie

Funcția update\_movies orchestrează preluarea și salvarea batch-urilor de filme

din API-ul TMDb.

La început, apelăm get\_genre\_mapping() pentru a obține un dicționar care leagă fiecare genre\_id TMDb de numele său uman-citibil. În acest fel, când aducem lista de genuri asociate fiecărui film, vom putea stoca direct denumirile și nu doar id-urile numerice.  
 Parametrul total\_pages controlează câte pagini de rezultate descărcăm (implicit 5).   
În interiorul fetch\_and\_save\_movies, fiecare răspuns JSON este parsat, se înlocuiesc genre\_ids cu numele genurilor din genre\_mapping, iar datele (titlu, overview, poster, data lansării ș.a.) se inserează sau actualizează în tabela movies din SQLite.

În acest mod, update\_movies asigură că datele despre filme rămân actualizate și complete, printr-un proces simplu de paginare și salvare automată.

## Curățare și transformări

Orice sistem de recomandare eficient pornește de la date curate și bine structurate. În cadrul lucrării de față, datele brute provin din două surse principale: catalogul de filme preluat prin API-ul TMDb și interacțiunile utilizatorilor (rating-uri și click-uri) stocate local în baza SQLite. Înainte ca aceste date să poată fi folosite cu adevărat de algoritmii de recomandare (filtrare colaborativă, content-based sau click-based), ele trebuie curățate de eventualele erori, duplicate și valori lipsă, iar câmpurile de text trebuie transformate în forme numerice utile pentru modele. În continuare, vom parcurge în ordine pașii esențiali de preprocesare, ilustrând modul în care funcțiile din codul Python (în special **compute\_similarity()** și **preprocess\_clicks()**) realizează aceste transformări.

### **2.2.1 Încărcarea și validarea datelor**

Primul pas este citirea datelor din baza de date SQLite direct în Pandas DataFrame-uri. În funcția **compute\_similarity()**,se încarcă toate filmele cu câmpurile relevante.

În mod similar, pentru click-uri, funcția preprocess\_clicks() pornește prin interogarea întregii tabele clicks.

Orice set de date poate conține erori de import, valori lipsă sau duplicate. După ce citim tabelele în DataFrame-uri, urmăm câțiva pași fundamentali de validare:

**Verificarea existenței datelor**: dacă DataFrame-ul este gol (de exemplu, nu există înregistrări de click), funcția returnează imediat un DataFrame gol şi afișează un mesaj de atenționare.

**Conversia tipurilor**: coloana timestamp este convertită din șir de caractere în obiect datetime. Orice valoare neconformă (care generează NaT) este eliminată ulterior:

**Eliminarea valorilor lipsă**: în cazul filmelor, coloanele de text precum overview şi genres pot conține NULL. Pentru a nu avea probleme la vectorizarea textului, înlocuim toate valorile lipsă cu șiruri goale.

Astfel, dacă un film nu are descriere sau gen, îl tratăm ca având pur și simplu un șir gol, fără a introduce erori în etapele următoare.

**Eliminarea duplicatelor**: deși tabela movies din baza TMDb este construită astfel încât nu ar trebui să existe filme duplicate, în mod normal am aplica totuși un filtru suplimentar pentru a ne asigura că niciun titlu nu se repeat.

Pentru click-uri, deși timestamp-ul este capturat la nivel de secundă, se poate adăuga un filtru care să elimine înregistrările identice (user\_id, movie\_id, timestamp). În cod, nu există o linie explicită de drop\_duplicates în preprocess\_clicks(), însă logică similară poate fi integrată rapid dacă apare cazul.

În urma acestor verificări și operațiuni, ne asigurăm că intrăm în etapa de transformare doar cu date valide, fără rânduri goale, șiruri NaN sau duplicate de genul celor care ar putea afecta calitatea modelelor de recomandare.

### **2.2.2 Tokenizare și curățare text (overview, keywords)**

O componentă esențială a sistemului de recomandare bazat pe conținut este reprezentarea descrierii filmelor și a cuvintelor-cheie („overview” și „genres”) sub formă numerică. În cod, această sarcină este îndeplinită pas cu pas în funcția compute\_similarity():

**Construirea câmpului „tags”**. Îmbinăm descrierea (overview) cu lista de genuri (genres) într-un șir de text unic, pentru a simplifica procesul de vectorizare. În acest mod, nu avem nevoie să tratăm separat textul și genurile, ci le considerăm similare din punct de vedere al tokenizării.

Prin această concatenare, un film cu overview = "A space adventurer travels the galaxy" și genres = "Action, Adventure, Sci-Fi" va primi un câmp tags = "A space adventurer travels the galaxy Action, Adventure, Sci-Fi". Evident, vor urma aceleași pași de curățare (lowercase, eliminare semne de punctuație, stop-words) atunci când vectorizatorul descompune șirul în termeni.

**Eliminarea caracterelor nedorite**. Biblioteca CountVectorizer este configurată implicit să transforme totul în lowercase și să ignore semnele de punctuație şi „stop-words” în limba engleză. Dacă însă dorim o curățare și mai strictă (de exemplu, eliminarea cifrelor sau a accentelor), am fi putut introduce o funcție de tip clean\_text(), dar în proiectul actual nu a fost considerat necesar, deoarece CountVectorizer(stop\_words='english') se descurcă foarte bine cu cele mai frecvente situații.

**Vectorizarea textului prin CountVectorizer**. Odată ce avem coloana tags completată, instanțiem un CountVectorizer cu parametri adecvați:

 max\_features=5000 limitează numărul maxim de termeni la 5.000, ceea ce ne ajută să ținem sub control dimensiunea matricei (echilibrul între spațiu și acuratețea semantică).

 În urma acestei apelări, count\_matrix este o matrice rară (sparse) de forma (numar\_filme × 5000), în care fiecare coloană reprezintă frecvența unui anumit termen în șirul tags al fiecărui film.

**(Opțional) TF-IDF în loc de simplu Count**. Dacă am fi dorit să penalizăm termenii extrem de frecvenți și să scoatem în evidență cuvintele distinctive, am fi schimbat CountVectorizer cu TfidfVectorizer.

În lucrarea actuală, am rămas la CountVectorizer pentru simplitate, dar menționăm acest pas ca o posibilă îmbunătățire viitoare.

### **2.2.3 Extracție de caracteristici: genuri (one-hot) și straturi textuale**

Pentru a crea un sistem hibrid de recomandare, avem nevoie să combinăm mai multe tipuri de informații:

**Genurile** (Action, Comedy, Drama etc.) pot fi transformate într-un vector binar (one-hot) prin următorul procedeu:

În primul rând, despărțim șirul genres într-o listă de genuri individuale apoi instanțiem un MultiLabelBinarizer pentru a primi un array de dimensiune (numar\_filme × numar\_genuri\_posibile), cu valori 1 acolo unde filmul corespunde genului respective.

După acest pas, fiecare film are coloane binare suplimentare, de exemplu Action=1 dacă filmul aparține genului „Action”, și 0 în caz contrar. Această reprezentare one-hot poate fi folosită într-un model hibrid care combină metadatele genurilor cu vectorizarea textului.

**Straturi textuale**: rezultatul CountVectorizer (sau TfidfVectorizer) este o matrice rară de termeni. Dacă ne uităm strict la codul din compute\_similarity(), vedem că a doua linie calculează matricea de similaritate cosinus pentru toate filmele, pe baza apariției cuvintelor din tags. Rezultatul similarity este o matrice pătrată de forma (numar\_filme × numar\_filme), unde elementul [i, j] reprezintă cât de asemănătoare sunt filmul i și filmul j la nivel de conținut textual și genuri.

În concluzie, **“Curățare și transformări”** în codul Python cuprinde:

**Înlocuirea manuală a valorilor lipsă** pentru overview și genres cu string-uri goale.

**Eliminarea oricăror duplicate** potențiale (deși, în mod implicit, tabela movies nu conține duplicate de id).

**Crearea coloanei tags** care îmbină descrierea și genurile.

**Tokenizarea și vectorizarea textului** prin CountVectorizer(stop\_words='english', max\_features=5000).

**(Opțional) One-hot encoding pentru genuri**, deși în fluxul actual ele sunt tratate ca parte a textului în tags.

**Preprocesarea click-urilor** prin conversia timestamp-urilor, filtrarea interacțiunilor mai vechi de 30 de zile și calcularea unei ponderi finale (final\_weight) care combină vechimea și frecvența click-urilor.

Acest set de operațiuni transformă datele brute într-un set de varia­­bile complet numeric-evaluate, gata de a fi “înmânate” algoritmilor de recomandare: filtrare colaborativă (bazată pe matricea user–item), content-based (bazată pe similaritatea cosinus a textului) și click-based (bazată pe ponderile calculate). În capitolele următoare vom vedea cum folosim aceste structuri pentru a antrena efectiv modelele, cum integrăm rezultatele într-un backend Flask și cum afișăm propunerile finale pe interfața web.

## Construirea matricelor

Pentru ca fiecare dintre cele trei categorii de algoritmi (filtrare colaborativă, recomandare bazată pe conținut şi recomandare bazată pe click-uri) să funcţioneze eficient, trebuie să creăm câte o structură matricială care să reflecte informaţiile relevante din datele brute (rating-uri, descrieri de filme, interacţiuni cu click-uri). În continuare vom prezenta modul în care, în codul de licenţă, fiecare dintre aceste matrici este construită, astfel încât să poată fi utilizată ulterior de algoritmi precum SVD din biblioteca Surprise, de calculul similitudinii cosinus pentru conținut, respectiv de antrenarea unui model SVD pe date preprocesate de click-uri.

### **2.3.1 Matricea user–item**

**Scopul**: Filtrarea colaborativă are nevoie, în esenţă, de o reprezentare a utilizatorilor şi a filmelor într-o matrice „user × movie” în care fiecare celulă conţine rating-ul acordat de un utilizator u filmului i. Dincolo de această matrice strict numerică, biblioteca Surprise aşteaptă să primească datele sub formă de triplete (user\_id, movie\_id, rating), urmate de construirea unui „trainset” propriu.

**Cum se construieşte, pas cu pas, în codul main.py:**

* **Încărcarea rating-urilor din baza de date:**

 Se creează un DataFrame ratings\_df cu coloanele user\_id, movie\_id și rating.

 Dacă nu există nicio înregistrare în tabela ratings, modelul nu se poate antrena (în cod există un verificator care returnează imediat dacă ratings\_df.empty).

* **Pivotarea datelor într‐o matrice cu valori numerice**

 Folosim metoda Pandas pivot\_table pentru a genera un obiect de tip DataFrame în care indexul este user\_id, coloanele sunt movie\_id, iar valorile efective sunt rating-urile acordate.

* **Construirea „trainset”-ului pentru Surprise**

Înainte de a antrena efectiv un model SVD, codul creează un obiect Reader(rating\_scale=(1, 5)) și folosește Dataset.load\_from\_df pentru a introduce direct tripletele (user\_id, movie\_id, rating) în formatul intern al Surprise.

Astfel, datele de rating trec de la structura „long” (fiecare rând = un rating) la structura acceptată de Surprise, iar trainset devine un obiect intern care conține toate rating-urile ca matrice sparse.

* **Antrenarea modelului SVD**

După ce trainset este creat, se antrenează modelul SVD ,apoi in acest moment, algo reţine factorizarea matricii user–item în două matrici de factori latenti (user × latent\_dimension, respectiv movie × latent\_dimension), care pot fi ulterior folosite pentru a prezice rating-uri pentru perechi (user, movie) nevizitate.

* **Utilizarea serviciului de recomandare**

get\_collaborative\_filtering\_recommendations(user\_id,top\_n=10) construieşte un „anti-testset” (toate filmele pe care utilizatorul nu le-a notat) și rulează algo.test(anti\_testset) pentru a obține estimări de rating.

Se sortează predicţiile după estimarea est în ordine descrescătoare și se extrag primele filme.

În concluzie, matricea „user–item” este creată temporar cu Pandas pentru vizualizare și analiză, dar efectiv codul trimite direct tripletele de rating în Surprise, care intern construiește propria reprezentare sparse. Acest mecanism asigură o implementare scalabilă și rapidă, fără să fie nevoie să stocăm efectiv întreaga matrice dense în memorie.

### **2.3.2 Matricea document–termen**

**Scopul**: În recomandarea bazată pe conținut, fiecare film este caracterizat printr-un vector numeric care reflectă textul asociat (descriere, genuri, cuvinte-cheie etc.). Acest vector (adesea numit vector de „features” sau de „tags”) este obținut prin tehnici de procesare a limbajului natural (NLP), cum ar fi CountVectorizer sau TF-IDF pe textul concatenat al metadatele filmului. Apoi, similaritatea dintre filme se măsoară cu cosinusul unghiului dintre acești vectori.

**Modul concret de implementare în cod:**

* **Încărcarea datelor de la filme din SQLite într-un DataFrame**

Pentru a construi document–term, codul primește în funcţia compute\_similarity() un DataFrame, movies\_df, care conține cel puțin coloanele: title, overview, genres, poster.

* **Preprocesarea textului (combinarea şi curățarea)**

Se completează orice NaN din coloanele overview şi genres cu șirul gol.

Apoi se creează o nouă coloană numită tags, care reprezintă concatenarea dintre overview și genres.

* **Vectorizarea textului—CountVectorizer**

Se inițializează un obiect CountVectorizer din scikit-learn, cu un număr maxim de caracteristici (max\_features) și cu stop-words englezeşti:

cv = CountVectorizer(max\_features=5000, stop\_words='english')

Convertim textul din coloana tags la un vector numeric sparse, apelând cv.fit\_transform(...). Rezultatul se numește count-matrix.

Unde, count\_matrix este o matrice de dimensiune (numar\_filme × numar\_vectori), unde fiecare rând corespunde unui film, iar fiecare coloană unei „features” (cuvânt frecvent din descriere sau gen).

* **Calculul similitudinii cosinus**

Odată ce avem count\_matrix (format sparse), definim matricea de similitudine drept: similarity = cosine\_similarity(count\_matrix)

Rezultatul similarity este o matrice de dimensiune (numar\_filme × numar\_filme) în care fiecare intrare (i, j) este scorul de similitudine cosinus dintre rândul i şi j. Practic, filmele asemănătoare din punct de vedere al textului vor avea valoare aproape de 1, iar filme foarte diferite aproape de 0.

Această structură ne ajută apoi să extragem rapid „Top N” filme similare cu un film de referinţă: dacă avem movie\_index, putem selecta linia similarity[movie\_index] şi sortăm descrescător.

* **Accesarea listelor de recomandări**

Funcția get\_content\_based\_recommendations(movie\_id, top\_n=10) primește movie\_id, găsește indexul filmului în movies\_df (prin movies\_df.index[movies\_df['id']==movie\_id]), apoi sortează perechile (index, similarity\_score) și returnează ID-urile top\_n filme cu cele mai mari scoruri.

În esență, **matricea document–term** (count\_matrix) este creată prin vectorizarea textului combinat (descriere + genuri), iar similitudinea cosinus dintre aceşti vectori devine „matricea de similaritate”.

### **2.3.3 Matricea click-based**

**Scopul**: În recomandarea bazată pe click-uri, ideea este să folosim simpla acțiune „user a apăsat pe filmul X” (în loc de un rating numeric) ca indicator al relevanţei. Deoarece click-urile pot apărea frecvent, chiar fără ca utilizatorul să acorde o notă efectivă, le vom combina cu un mecanism de ponderare bazat pe recență și frecvență, apoi vom antrena un model de filtrare colaborativă (SVD) asupra acestor ponderi normalizate.

Cum se construiește, pas cu pas, în codul main.py:

* **Încărcarea click-urilor din baza de date**

Obținem un DataFrame clicks\_df cu coloanele user\_id, movie\_id și timestamp. Orice rând reprezintă faptul că utilizatorul user\_id a vizualizat-apăsat filmul movie\_id la momentul timestamp.

* **Convertirea timestamp-urilor și eliminarea valorilor lipsă**

Convertim coloana timestamp la tipul datetime. Orice conversie eșuată devine NaT, iar apoi eliminăm rândurile nevalide (fără un timestamp clar).

* **Calculul ponderii bazate pe recență**

Pentru fiecare click, calculăm numărul de zile trecute de la interacțiune până în prezent (now). În ultimele 30 zile, greutatea variază liniar de la 1 (click foarte recent) la 0.0 (dacă au trecut 30+ zile). Însă, prin max(0.1, …), chiar și un click vechi mai mult de 30 zile primește o greutate minimă de 0.1 (nu vrem greutăți zero absolute).

* **Calculul frecvenței de interacțiuni**

Pentru fiecare pereche (user\_id, movie\_id), determinăm de câte ori acea combinaţie apare în ultimul interval. Practic, dacă un utilizator a dat clic pe același film de trei ori în ultimele 30 de zile, interaction = 3.

Folosim Pandas groupby + transform('count'), astfel încât să obținem o nouă coloană interaction care conţine, pe fiecare rând, numărul de rânduri (click-uri) cu același (user\_id, movie\_id).

* **Greutatea finală a fiecărui rând (= click)**

Calculăm greutatea combinată prin produsul dintre greutatea temporală (weight) şi frecvența de interacțiune (interaction).

De exemplu, dacă un click are weight = 0.8 (foarte recent) și interaction = 3 (s-a dat click de trei ori), atunci final\_weight = 0.8 × 3 = 2.4. Această valoare indică un semnal puternic: utilizatorul a accesat filmul de 3 ori și cel puțin una dintre acele interacțiuni a fost foarte recentă.

* **Filtrarea click-urilor mai vechi de 30 de zile**

Eliminăm orice click înregistrat cu mai mult de 30 de zile în urmă, pentru că ne interesează doar interacțiunile recente care arată interes actual.

În final, clicks\_df conține doar rânduri (user\_id, movie\_id, timestamp, weight, interaction, final\_weight) pentru ultimele 30 de zile.

* **Utilizarea datelor de click în Surprise**

Funcția initialize\_collaborative\_filtering\_clicks() construiește un set de

date Surprise pornind de la tripletele (user\_id, movie\_id, final\_weight).

În practică, Surprise normalizează valorile într-un interval intern (automatically), iar eu am ales SVD cu rating\_scale=(0,1) pentru a reflecta semnalele de click.

* **Generarea recomandărilor click-based**

Funcția get\_collaborative\_click\_recommendations(user\_id, top\_n=10)

construiește un anti-testset pentru toate filmele la care userul din context nu a dat click.

Apoi sortează predicțiile după valoarea estimată a ponderii de click și extrage primele top\_n.

În concluzie, **matricea click-based** nu este stocată explicit ca Pandas DataFrame cu toţi utilizatorii pe toate filmele (matrice densă), ci codul realizează un DataFrame de triplete (user\_id, movie\_id, final\_weight) și îl transformă prin Surprise într-o matrice latentă internă, apoi folosește SVD pentru recomandări.

# Metode de recomandare

În acest capitol vom prezenta, pas cu pas, principalele metode de recomandare implementate în cadrul aplicației, așa cum reies din codul sursă. Vom explica atât filtrarea colaborativă (CF), cât și recomandările bazate pe conținut (CB), urmând apoi să descriem modelele avansate și abordarea hibridă, care combină cele două paradigme pentru rezultate superioare. Pentru fiecare metodă, vom menționa funcțiile și clasele din cod, fluxul de date și raționamentul matematic din spatele algoritmilor.

## 3.1 Filtrare Colaborativa

Filtrarea colaborativă(CF) se bazează pe ideea că utilizatori asemănători vor avea gusturi asemănătoare. În aplicație există mai multe variante de CF, implementate sub forma unor funcții diferite. Le vom prezenta în ordine:

* CF bazată pe SVD (Singular Value Decomposition) cu biblioteca Surprise
* CF bazată pe Weighted Mean (media ponderată), user-based
* CF bazată pe Weighted Mean filtrate după atribute (sex, sex + ocupație)
* CF bazată pe Click-uri (click-based CF)
* CF bazată pe KNN (K-Nearest Neighbors) cu Surprise
* CF prin modele Neurale (NN-CF, GMF+MLP, Autoencoder, GNN)

### **CF-SVD cu biblioteca Surprise**

**Încărcarea datelor**

* Tabelul ratings este citit din baza SQLite (tmdb\_movies.db) și transformat într-un DataFrame pandas (ratings\_df).
* Se definește un Reader(rating\_scale=(1,5)), apoi obținem un obiect Surprise trainset care conține toate tripletele (utilizator, film, rating).

**Antrenarea modelului SVD**

* Se creează un obiect SVD() (din Surprise) și se apelează algo = SVD(); algo.fit(trainset).
* Intern, SVD descompune matricea incompletă R (utilizatori × filme) în trei matrice: Users,Sigma,V transpus

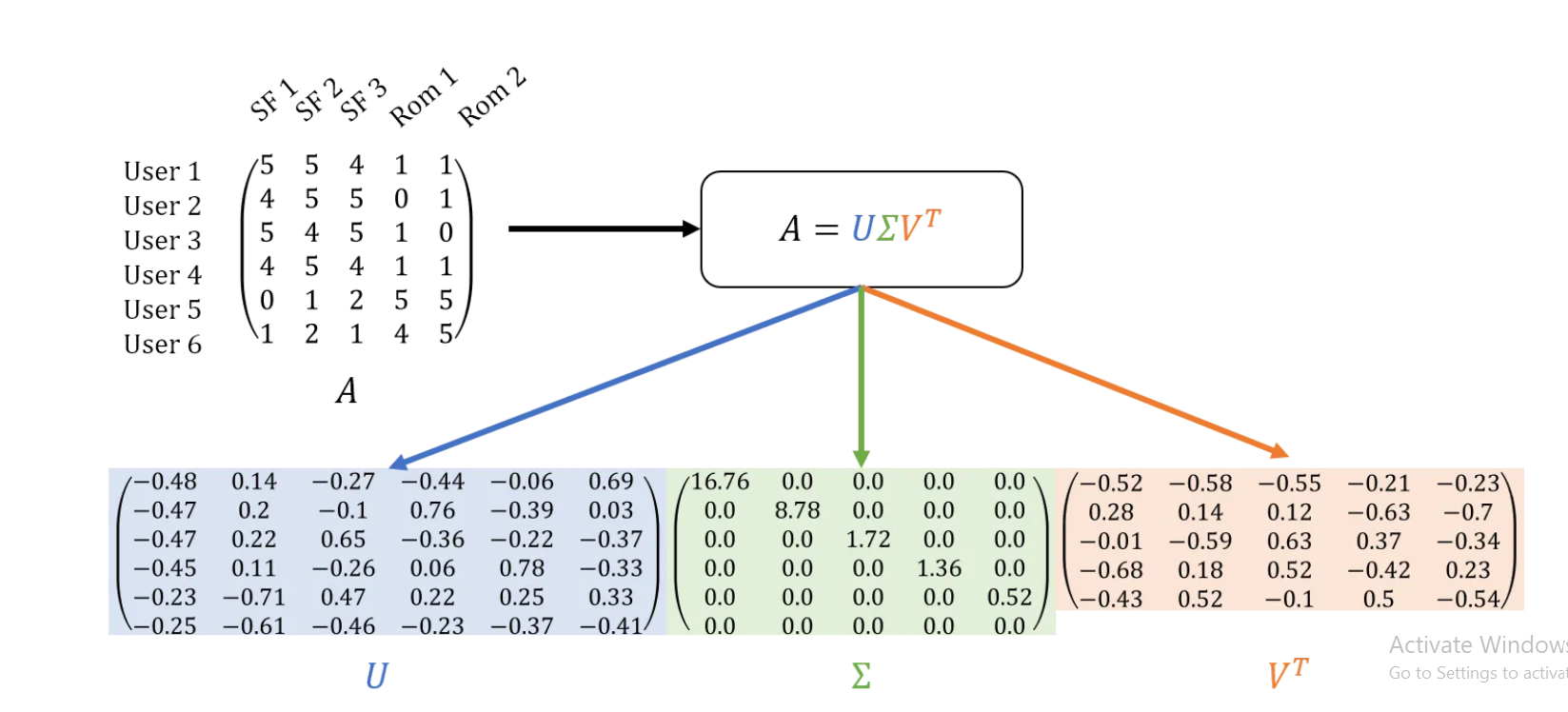


Fig. 3.1: Factorizarea matricei de scoruri utilizator-film prin SVD

unde:

* U este matricea de factori latenti a utilizatorilor (dim: n\_users × k),
* Σ(Sigma) este matrice (k × k) cu valorile singulare,
* V^T dim k × n\_movies este matricea de factori latenti a filmelor.

Nu există implementarea internă a SVD (o face biblioteca Surprise), ci e folosită pur și simplu algo.fit(...).

**Generarea recomandărilor**

* Pentru a obține recomandări pentru un user\_id dat, se construiește testset = trainset.build\_anti\_testset(), care conține toate perechile (u,i,0) pentru care utilizatorul nu a dat încă rating.
* Se filtrează doar perechile (user\_id,film\_neg).
* predictions este o listă de obiecte Prediction(user\_id, movie\_id, true\_rrating, est, details). Sortăm această listă descrescător după est (ratingul estimat).
* Extragem primele top\_n filme. [9]

**Avantaje:**

* Capturarea directă a relației utilizator × film bazate pe istoricul de ratinguri.
* SVD reduce zgomotul și extrage factori latenti care captează preferințe generale (ex. genuri).
* Dă rezultate bune chiar și cu date rare, datorită proprietății de low-rank factorization.

**Dezavantaje:**

* Necesită suficiente date de rating (dacă datele sunt prea rare, modelul nu converge bine).
* Când apare un nou film sau nou user (cold-start), nu putem prezice direct fără reantrenare.

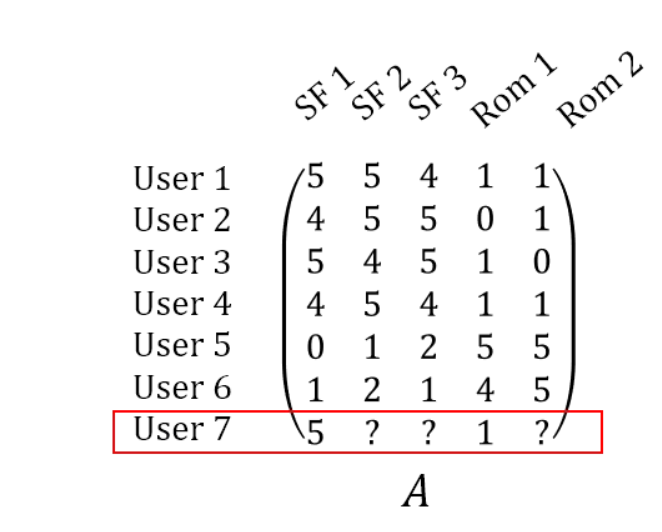


Fig. 3.2: Predicția scorurilor lipsă pentru utilizatorii parțial observați[15]

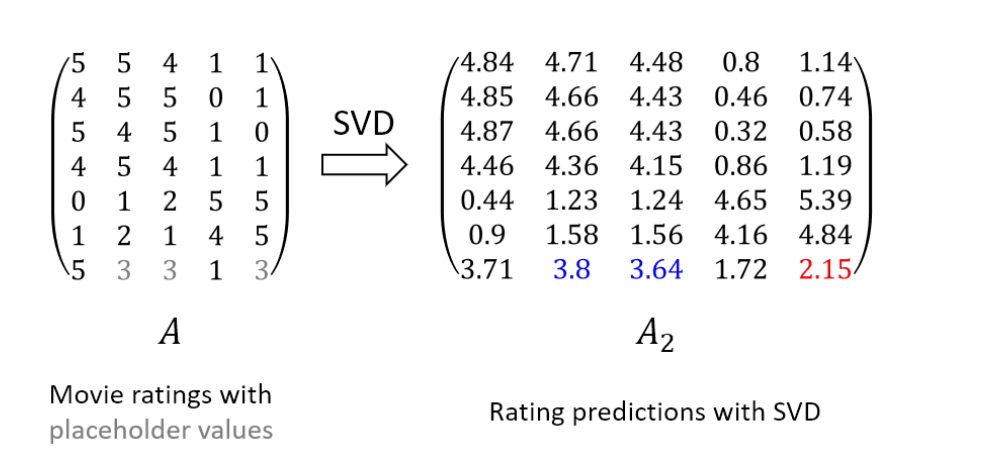


Fig. 3.3: Predicția valorilor necunoscute[15]

### **3.1.2 CF-Weighted Mean (media ponderată), user-based**

În implementare este inclusă o metodă de tip user-based weighted mean, care calculează scorurile de recomandare prin media ponderată a ratingurilor oferite de utilizatori similari, utilizând coeficienți de similaritate drept ponderi.Practic, predicția pentru un (user,film) ne-evaluat se face ca medie ponderată a ratingurilor pe care alți useri asemănători (din punct de vedere al profilului lor de rating) le-au acordat acelui film.

**Construirea matricei de rating și a similarității user–user**

* În funcția initialize\_user\_based\_matrices():
* r\_matrix rămâne un DataFrame (user\_id × movie\_id) cu NaN în loc de ratinguri lipsă.
* cosine\_sim este DataFrame-ul (user\_id × user\_id) cu valorile de similaritate cosinus calculată pe vectorii compleți (zero-fill) ai fiecărui user.

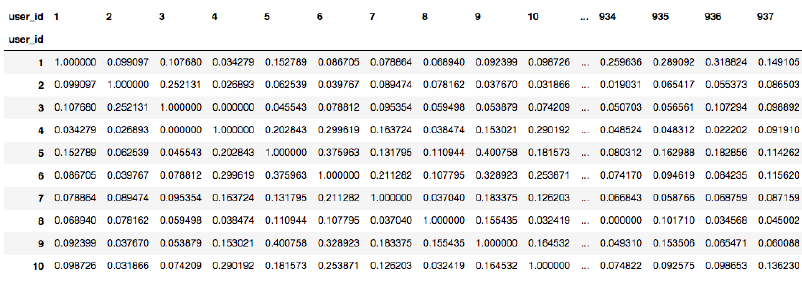


Fig. 3.4: Matrice de similaritate între utilizatori[1]

**Formula mediei ponderate**

Pentru a prezice ratingul:

* + - Similaritățile cu alți utilizatori: sim\_scores = cosine\_sim[user\_id] (Serie cu valori ∈ [−1,1]).
    - Se extrag ratingurile tuturor celorlalți utilizatori pentru filmul m.
    - Astfel, common\_users sunt toți cei care au dat note la filmul m și pentru care există și un sim\_scores[user].
    - Se calculează media ponderată,unde r reprezinta ratingul dat de utilizatorul u fimului m:



Dacă nu există utilizatori care să fi notat filmul (common\_users.empty), se face fallback la 3.0.

**Avantaje:**

* Nu cere factorizare matricială.
* Funcționează bine când datele nu sunt foarte rare și există mulți useri pentru fiecare film.

**Dezavantaje:**

* Dacă există doar puțini useri care au notat filmul, estimarea poate fi instabilă.
* Calculul similarităților cosinus la nivel de toată matricea (n\_users × n\_users) poate fi costisitor dacă numărul de useri este mare.

***CF după sex(*cf\_user\_wmean\_gender*)***

Flux:

* Preia sim\_scores față de toți userii.
* Scoate din sim\_scores toți userii care nu au același sex (songs\_df['sex'] != target\_gender).
* Continua exact ca la Weighted Mean, dar doar cu subsetul de useri de același sex.

**CF după sex + ocupație (*cf\_user\_wmean\_gen\_occ*)**

Flux:

* Verifică întâi dacă movie\_id există în coloanele r\_matrix.
* Preia sexul și ocupația target‐userului.
* Din sim\_scores se elimină toți utilizatorii care nu au ambele atribute egale (sex și ocupație).
* Aplică Weighted Mean pe subset.

**Avantaje:**

* Adaugă filtrare suplimentară (de exemplu, recomandări de la useri de același sex și aceeași ocupație). Aceasta poate îmbunătăți calitatea predicției atunci când există pattern-uri de preferințe distincte după aceste demografice.

**Dezavantaje:**

* Subsetul de utilizatori similari poate fi mic (dacă prea puțini utilizatori împărtășesc aceleași atribute), iar predicția devine imprecisă.

### **3.1.3 CF bazată pe click-uri (Click-Based Collaborative Filtering)**

În plus față de ratinguri, aplicația înregistrează și click-uri: ori de câte ori un user vede detaliile unui film, se stochează o înregistrare (user\_id,movie\_id,timestamp) în tabela clicks. Apoi se construiește un CF separat care învață un model SVD pe baza „greutăților” generate din click-uri.

* Citește toate click-urile din ultimele 30 zile.
* Creează coloana weight liniar descrescător cu vârsta click-ului (de la 1 la 0.1).
* interaction este numărul de click-uri (frecvența), calculat cu groupby(...).transform('count').
* final\_weight = weight × interaction.
* Se filtrează click-urile mai vechi de 30 zile. Rezultă un DataFrame în care fiecare rând e (user,movie,finalweight)(user, movie, final\_weight)(user,movie,finalw​eight).
* Se preprocesează clickurile și se obține un DataFrame (user,movie,finalweight)(user,movie,final\_weight)(user,movie,finalw​eight).
* Se definește Reader(rating\_scale=(0,1)) pentru a înțelege că valorile sunt în [0,1].
* Se antrenează un model SVD() (Surprise) pe acest nou set de date.

**Avantaje:**

* Dacă userii nu au oferit ratinguri (sau rareori), clickurile oferă un proxy al preferințelor lor.
* Recomandarea se face chiar și dacă nu s-au dat ratinguri, dar s-au făcut click-uri.

**Dezavantaje:**

* Clickurile nu semnalează neapărat un grad de apreciere. Se face click la un film din curiozitate, nu neapărat că ți-a plăcut.

### **3.1.4 CF-KNN cu biblioteca Surprise (K-Nearest Neighbors)**

În cod se construiește un model KNN (User-based) folosind KNNBasic (din Surprise) în funcția initialize\_knn\_model.

**Antrenarea modelului:**

sim\_options = {'name':'pearson', 'user\_based':True} înseamnă ca

* Folosim similaritate Pearson între useri (puteam alege şi 'cosine').
* Modelul este user-based (comparam profilurile userilor).

**Generarea recomandărilor KNN**

**Flux:** analog cu SVD, dar knn\_algo face estimarea directă prin media ratingurilor celor mai apropiați useri, unde proximitatea = similaritate Pearson (suplimentar codului intern al KNNBasic).

**Avantaje:**

* Nu implică factorizare matricială.
* Modelul este interpretabil, deoarece se poate identifica exact care utilizatori au influențat recomandarea generată.

**Dezavantaje:**

* Dacă datele sunt sparse, găsirea celor mai apropiați k useri poate fi inexactă.
* Scalabilitatea scade odată cu creșterea numărului de useri și filme.

### **3.1.5 CF prin modele Neurale (NN-CF, GMF+MLP, Autoencoder, GNN)**

Aplicația include și mai multe variante de CF bazate pe rețele neuronale, implementate în mai multe etape:

* Model NN-CF simplu (MovieRecommender)
* GMF + MLP (NeuMF)
* Autoencoder (SAE)
* Graph Neural Network (GNN)

Paralel, am construit un tabel de rating-uri care conține peste 52.000 de utilizatori unici, unde fiecare înregistrare asociază un user\_id, movie\_id și valoarea rating-ului (1–5). Datele brute de la TMDB au fost normalizate (tipărire JSON → tabele SQL), iar pentru modelele neuronale am duplicat această structură într-o bază separată (tmdb\_movies\_for\_nn.db). Astfel s-au obținut seturile de antrenament pentru rețele precum NN-CF, GMF+MLP, Autoencoder sau GNN, unde fiecare observație este un triplet (user, film, rating).

**NN-CF simplu**

Date de antrenament:

* Se citește tabelul ratings din tmdb\_movies\_for\_nn.db.
* Se mapază fiecare user\_id și movie\_id la un index 0…(N−1).
* Se normalizează valorile ratingurilor la [0,1] (împărțind la 5.0).

Flux antrenare (train\_and\_save\_model):

* Se creează DataFrame ratings\_df cu coloanele [user\_id, movie\_id, rating\_norm].
* Se transformă user\_id→user\_idx și movie\_id→movie\_idx.
* Se creează seturile train\_data și test\_data (train\_test\_split).
* Se antrenează modelul MovieRecommender(num\_users, num\_movies, embedding\_dim=50) cu loss='mse' și metrics=['mae'] timp de câteva epoci.
* Se salvează modelul în folderul nn\_model\_tf și mapările user\_map.npy, movie\_map.npy.

Flux inferență (recommend\_movies\_nn):

* Pentru un user\_id fix, se extrage user\_map[user\_id] = uidx.
* Mapezi movie\_map: obții lista tuturor movie\_idx.
* Creezi doi vectori:
* user\_array = np.array([uidx]\*len(all\_movie\_idx), dtype=np.int64)
* movie\_array= np.array(all\_movie\_idx, dtype=np.int64)
* astfel încât să poată prezice rating pentru fiecare film.
* predictions = model.predict([user\_array, movie\_array]).flatten() → vector de dim (nr\_filme,) cu valori normalizate 0..1.
* Scalam la 1..5: ratings\_1\_5 = predictions \* 4.0 + 1.0.
* Sortăm descrescător și obținem primele top\_n movie\_id reali. [5]

**Avantaje:**

* Modelul învață împreună embedding-urile user și movie.
* Poate captura relații non-liniare mai complexe decât simplele medii ponderate.

**Dezavantaje:**

* Necesită un volum suficient de date pentru a evita suprapotrivirea modelului.
* Antrenarea și inferența pot fi costisitoare comparativ cu SVD/KNN directe.

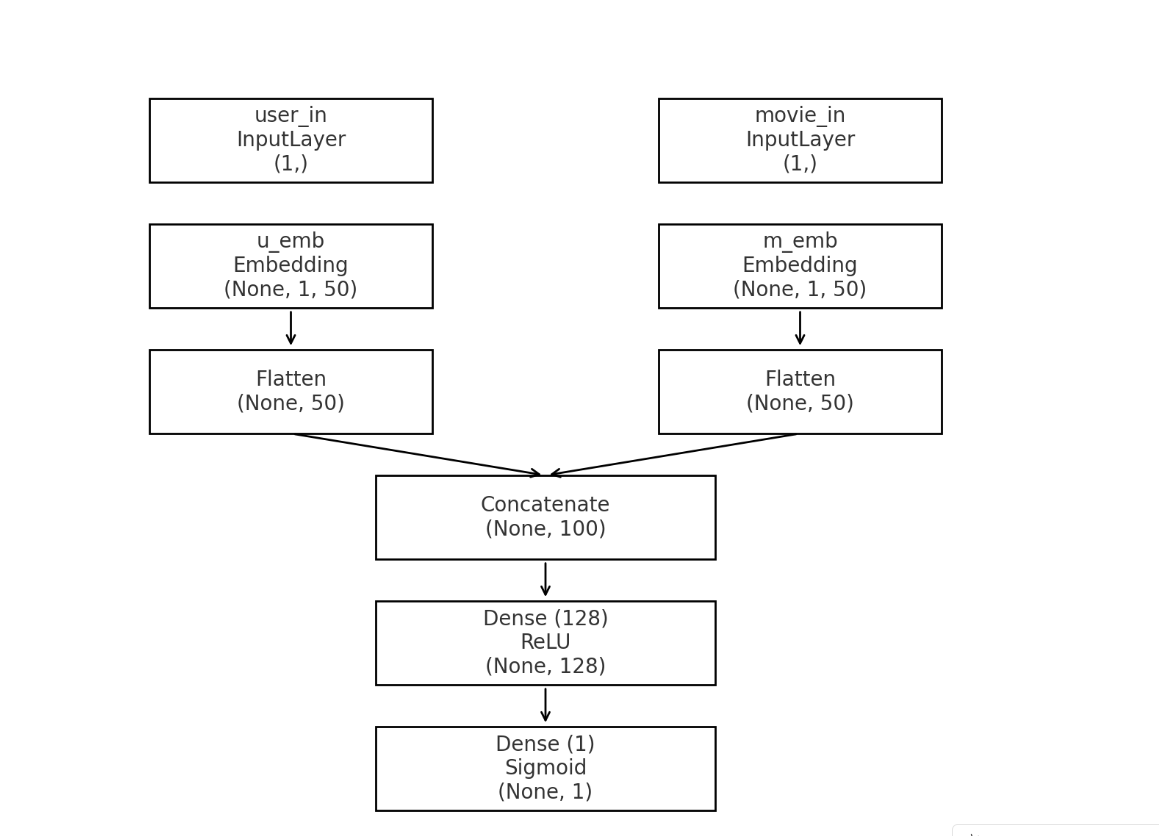


Fig. 3.5: Model neural bazat pe embeddinguri

**NN-CF  Euromed University of Fez**

**Încărcarea și pregătirea datelor**

* Se transformă fiecare userId și movieId în coduri numerice (LabelEncoder), de la 0…N-1, respectiv 0…M-1.
* Normalizăm rating-urile pe intervalul [0,1] prin scala MinMax.
* Împărțim setul în train (80 %) și test (20 %).

**Construcția rețelei (MLP cu embedding)**

* Intrări: două variabile scalare, user și movie.
* Embedding layers:
* Embedding(num\_users, 50) → transformă user într-un vector de dimensiune 50.
* Embedding(num\_movies, 50) → transformă movie într-un vector de dimensiune 50.
* Flatten: „aplatizăm” fiecare embedding, obținând două vectori 1×50.
* Concatenare: îi lipim unul lângă altul, formând un vector 1×100 care conține atât preferințele utilizatorului, cât și „trăsăturile” filmului.
* Strat Dense (fully-connected):

rețea cu 128 unități și funcție de activare ReLU, care învață interacțiunile (ce genuri, ce combinații de embedding-uri preferă userul).

Strat de ieșire:

Un Dense(1, activation='sigmoid') care proiectează rezultatul într-un scor între 0 și 1.

* Îl readucem pe scara 1–5 (dacă vrem), inversând scala MinMax.

**Antrenament și evaluare**

* Funcție de pierdere: mean\_squared\_error – măsoară pătratul diferenței dintre predicție și rating-ul real.
* Optimizator: Adam – ajustează rapid greutățile.
* Early stopping: monitorizează val\_loss și oprește antrenamentul când nu mai scade, restaurând cele mai bune greutăți. [2]

**GMF + MLP**

**Interpretare:**

* Modelul are două ramuri:
  + GMF (Generalized Matrix Factorization): embedding-uri de dim latent\_dim\_gmf pentru user și item, apoi multiply (elementwise).
  + MLP (rețea simplă fully-connected): embedding-uri de dim latent\_dim\_mlp, urmate de câteva Dense(...) cu relu.
* Se concatenează ieșirile celor două ramuri (gmf\_out + mlp\_out) → un vector mai lung → ultim Dense(1, activation='sigmoid').
* Normalizarea finală la [0,1] (poate scala la [1,5] după inferență, la fel ca NN-CF).

**Antrenare (train\_and\_save\_gmf\_mlp):**

* Se citește ratings\_df.
* Se mapează user\_id → user\_idx, movie\_id → item\_idx.
* Se normalizează rating\_norm = (rating − 1)/4.
* Se împarte în train\_data, test\_data.
* Se instanțiază model = build\_gmf\_mlp\_model(...).
* Se antrenează cu model.fit(...) pe datele X\_train = [user\_idx, item\_idx], y\_train = rating\_norm.
* Se salvează modelul (.save(save\_path)) și mapările user\_map\_gmfmlp.npy, item\_map\_gmfmlp.npy.

**Inferență (recommend\_movies\_gmf\_mlp):**

* Pentru un user\_id, obținem user\_idx = user\_map\_gmfmlp[user\_id].
* Calculăm, pentru fiecare movie\_idx din item\_map\_gmfmlp, predicția pred = model.predict([ [user\_idx], [movie\_idx] ]).
* Sortăm descrescător după pred, selectăm top\_n, mapăm înapoi la movie\_id real și returnăm lista. [11]

**Avantaje:**

* Combină tehnica de extragere a relațiilor latente dintre utilizatori și filme (GMF) cu o rețea non-liniară (MLP) ce poate surprinde interacțiuni complexe.
* În practică, NeuMF (care îmbină GMF + MLP) întrece adesea MF simplu sau MLP simplu.

**Dezavantaje:**

* Arhitectura este destul de mare: mai multe embedding-uri, straturi Dense, ceea ce duce la un număr mare de parametri.
* Necesită mai multă memorie și timp de antrenare în comparație cu MF sau SVD direct.

**Autoencoder (SAE)**

Se definește un autoencoder supravegheat (Stacked AutoEncoder) care, pe baza întregii matrice de ratinguri (rating\_matrix), învață să reconstruiască vectorii de rating ai fiecărui utilizator. [14]

**Date de intrare:**

* ratings\_matrix = matricea densă (n\_users × n\_movies), unde fiecare rând este un vector de dim\_nb\_movies cu ratingurile utilizatorului (0 în loc de NaN).

**Antrenare** (train\_autoencoder\_model):

* Pentru fiecare id\_user ∈ [0…nb\_users−1],
* input\_user = training\_tensor[id\_user].unsqueeze(0) (shape [1, nb\_movies]).
* target = input\_user.clone().
* Dacă target conține cel puțin un element > 0 (adică userul a dat cel puțin un rating),
* se obține output = sae(input\_user)
* se pune la zero output[target == 0] = 0 (astfel, în calculul erorii, nu penalizăm predicțiile pe filme nevizionate de respectivul user).
* se calculează loss = criterion(output, target) cu criterion = nn.MSELoss().
* loss.backward(); optimizer.step()
* Se repetă pentru un număr mic de epoci (de obicei 2–3, deoarece datele pot fi mari).
* La final, se salvează checkpoint-ul modelului și matricea de ratinguri pentru inferență.

**Inferență** (get\_autoencoder\_recommendations):

* Pentru un user\_id real, se consideră user\_idx = user\_id−1 (presupunem că user\_id începe de la 1).
* input\_user = ratings\_tensor[user\_idx].unsqueeze(0).
* output = sae(input\_user).detach().numpy().flatten(): vectorul predicțiilor de rating pentru fiecare film.
* user\_ratings = ratings\_tensor[user\_idx].numpy()
* Pentru toate pozițiile (idx) unde user\_ratings[idx] > 0 (filme deja notate), setăm output[idx] = −∞ pentru a exclude filmele deja votate.
* Sortăm output descrescător și luăm primele top\_n indici → recommended\_movie\_ids = [idx+1 for idx in top\_indices].

**Avantaje:**

* Folosește întreaga matrice de ratinguri la un loc (condiționează reconstrucția pe tot vectorul – global).
* Autoencoder-ul poate descoperi factori latenti neliniari.

**Dezavantaje:**

* Pot exista probleme cu scalabilitatea: dacă nb\_movies e mare, primul strat fc1 are input dim = nb\_movies, deci un vector foarte lung de intrare.
* Cod de antrenare destul de costisitor (iterez peste fiecare user în buclă).

**Graph Neural Network (GNN)**

Se implementează un model bazat pe un „Graph Neural Network” (GNN) într-o manieră simplificată, unde graful este bipartit (utilizatori ↔ filme) și genurile filmelelor sunt folosite ca caracteristici suplimentare.

**Date de intrare:**

* ratings\_df (user\_id, movie\_id, rating)
* movies\_df cu coloana genres. Pentru fiecare film se construiește un vector multi-hot (one-hot pentru fiecare gen, transformat cu un MultiLabelBinarizer). → genre\_map[movie\_idx] = torch.tensor([...]).
* Se mapează user\_id→user\_idx, movie\_id→movie\_idx.
* Se creează DataLoader cu batch\_size=1024, shuffle=True.

**Antrenare** (train\_gnn\_model):

* Instanțiem MovieLensNet(num\_movies, num\_users, num\_genres, embedding\_size=16, hidden\_dim=64).
* criterion = nn.MSELoss(), optimizer = optim.Adam(model.parameters(), lr=0.001).

**Inferență** (get\_gnn\_recommendations):

* Pentru user\_id\_real, preluăm user\_idx = user\_map[user\_id\_real].
* Sortăm movie\_scores descrescător după score și extragem top\_n.

**Avantaje:**

* Modelul încorporează explicit relația utilizator–film–genuri.
* Se extinde ușor cu legături: de exemplu, se adauga embeddinguri pentru actorii fiecărui film, chiar pentru interesele utilizatorului. [13]

**Dezavantaje:**

* Antrenarea unui GNN (sau un model similar) pe un grafic bipartit mare poate fi costisitoare.
* Necesită preprocesare suplimentară (construirea vect. one-hot pentru genuri, mapă user/movie).

## Recomandări bazate pe conținut (Content-Based)

Recomandările bazate pe conținut se concentrează pe descrierea filmului, genuri, actori și alte caracteristici extrase din metadatele filmului. În cod există două fluxuri principale:

* Construirea matricei document–term (CountVectorizer × overview + genres).
* Modele avansate de conținut: NMF pe taguri, BERT embeddings, BERT precomputat.

### **3.2.1 Similitudinea cosinus**

* similarity[i,j] = similaritatea cosinus dintre filmul i și filmul j.

get\_content\_based\_recommendations(movie\_id, top\_n) folosește exact similarity și returnează top\_n filme cu cea mai mare similaritate.

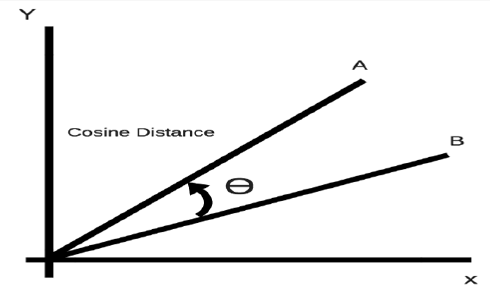


Fig. 3.6: Distanța cosinus[1]

Distanța cosinus e definită:

**Avantaje:**

* Nu depindem de ratinguri (pot recomanda chiar dacă un user nu a dat ratinguri, ci doar au existat clickuri/alte interacțiuni).
* Capturăm similitudinea direct pe text (metadate).
* Cold-start pentru filme (dacă un film nou apare, putem recomanda similar, chiar dacă nu a dat nimeni note încă, atâta timp cât există metadate).

**Dezavantaje:**

* Dacă metadatele nu sunt suficient de expresive (ex. doar titlu scurt, genuri simple), calitatea recomandărilor scade.
* Nu ține cont de feedback-ul personalizat al userilor (nu știe că un user particular ar prefera acțiune peste dramă).

### **3.2.2. Modele avansate bazate pe conținut(NMF,BERT)**

**NMF (Non-negative Matrix Factorization)**

**Flux:**

* Vectorizarea textului (CountVectorizer ⇒ count\_matrix).
* NMF: W\_nmf = factori latenti ai filmelor (user–rating echivalent, dar pe conținut).
* Pentru filmul movie\_id, extragem W\_nmf[movie\_index].
* Calculăm similaritatea cosinus între acest vector latent și toți vectorii latenti din W\_nmf.
* Sortăm descrescător și alegem top n.

**Avantaje:**

NMF (non-negativ) poate produce factori interpretabili (ex. topici de conținut).

Se pot observa ce topicuri latente explică similaritatea.

**Dezavantaje:**

Dacă numărul de caracteristici e mare (CountVectorizer max\_features=5000), NMF poate fi costisitor.

Rezultatele depind mult de parametrul n\_components și de factorizare.

**BERT**

Codul folosește HuggingFace Transformers (BertTokenizer + BertModel) pentru a extrage reprezentări contextualizate.

* Fiecare film primește **un vector de dimensiune 768** .
* Se salvează toată structura pe disc pentru a nu re-calcula la fiecare pornire.
* Similar cu NMF: se calculează similaritate cosinus între embedding-urile BERT ale filmului curent și ale tuturor celorlalte filme.
* Returnează primele top\_n. [17]

**Avantaje:**

* BERT scade bariera lipsei contextului în overview. E un model preantrenat care înțelege semantic textul.
* Embedding-urile pot surprinde sinonime, ton și context (ex. „thrilling sci-fi heist” vs. „science fiction action”).

**Dezavantaje:**

* Extraordinar de costisitor: calculul embedding-urilor BERT pentru *mii* de filme și stocarea lor în memorie.
* Inferența BERT (cosine\_similarity) la runtime, deși embeddingurile sunt precomputate, poate consuma ceva memorie și timp pentru vectori 768D × N.

## Model Hibrid

Un sistem hibrid combină atât recomandările bazate pe conținut, cât și cele colaborative, în speranța că fiecare componentă va compensa limitările celeilalte.

În cod, există mai multe exemple de implementare a unui model hibrid:

**Recomandare hibridă SVD + Content based**

* content\_candidates — primele 25 de filme similare ca text (CB).
* Se recalculează predicțiile SVD (CF) pentru fiecare din aceste 25 de filme și se sortează descrescător după predicția algo.predict(user\_id, cand\_id).est.
* Returnează primele top\_n.

**Avantaje model hibrid:**

* Compensează limitele CF (cold-start film nou, lipsă ratinguri) cu CB (metadate întotdeauna disponibile), și limitele CB (lipsa feedback-ului personal real) cu CF (istoric de ratinguri).
* În practică, tinde să dea rezultate mai relevante și mai diversificate.

**Dezavantaje:**

* Complexitate mult mai mare (trebuie să asigurăm mentenanța a două tipuri de modele).
* Necesită resurse mai mari (memorie, CPU/GPU) pentru antrenare și inferență.

# Arhitectura Sistemului

În acest capitol descriem structura generală a aplicației de recomandare de filme: componentele principale, interacțiunile dintre ele și modul în care datele circulă de la utilizator până la algoritmii de recomandare și înapoi la interfața web.

## Viziune de ansamblu

Este important să fie clară arhitectura generală a sistemului și relațiile dintre componente, fără detalierea fiecărui modul. Scopul subcapitolului este să ofere o imagine intuitivă despre cum comunică frontend-ul, backend-ul, baza de date, API-ul extern și modulele interne de recomandare.

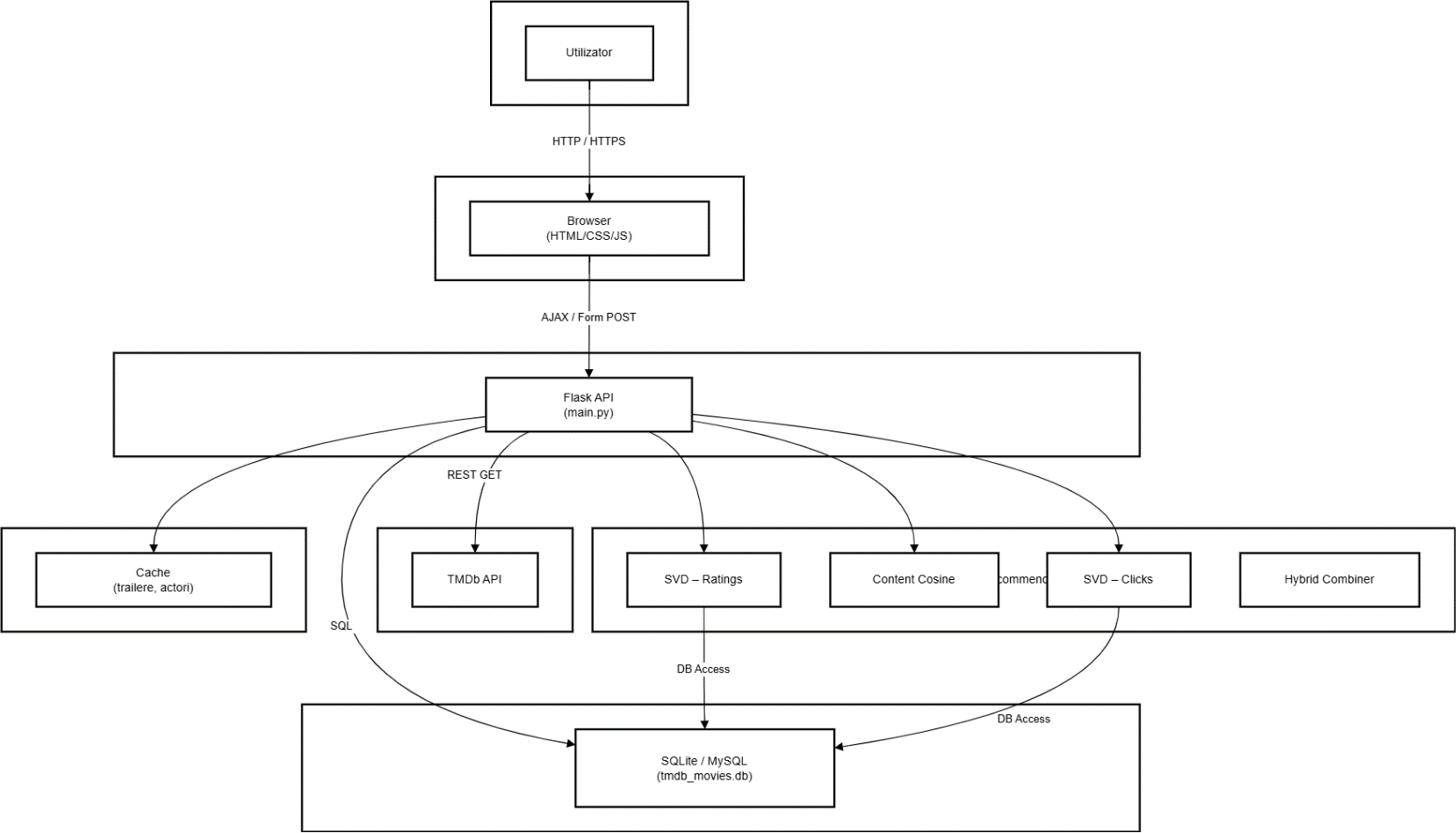


Fig. 4.1: Arhitectura sistemului

**Introducerea conceptului de arhitectură client-server**

* Explică pe scurt că aplicația este împărțită în două părți majore:
  + Client (front-end): pagini web (HTML/CSS/JavaScript) prin care utilizatorul interacționează.
  + Server (back-end): un API Flask în Python care primește cereri HTTP, procesează date, apelează algoritmi de recomandare și returnează răspunsuri JSON sau pagini generate dinamic.

**Prezentarea componentelor principale (descrise textual)**

* Frontend-ul este alcătuit din paginile HTML, fișierele CSS (Bootstrap sau Tailwind) și scripturile JavaScript (AJAX, eventual React sau un templating Jinja2). Rolul lui este să permită utilizatorului să:
  + - se autentifice/înregistreze,
    - caute filme,
    - vizualizeze recomandări și detalii despre filme,
    - interacționeze cu rating-uri, recenzii, căutare vocală (speech-to-text).

**Backend-ul (Flask)** găzduiește toate rutele și logica de server:

* Manevrarea autentificării și autorizării (inclusiv 2FA cu PyOTP),
* Rutarea principală: / (pagina de start), /movie/<id> (detaliu film), /recommendations (generează recomandări),
* Servicii de recomandare:
  + **get\_collaborative\_filtering\_recommendations(user\_id)**
  + **get\_content\_based\_recommendations(movie\_id)**
  + **get\_click\_based\_recommendations(user\_id)**
  + **get\_hybrid\_recommendations(user\_id, movie\_id)**

Accesul la baza de date (SQLite sau MySQL) pentru operații CRUD pe tabelele movies, users, ratings, clicks etc.

Integrarea cu API-ul extern TMDb pentru popularea și actualizarea tabelei movies.

**Baza de date** (SQLite/MySQL):

* Tabelele principale:
  + movies(id, title, overview, genres, poster\_path, release\_date, popularity, trailer)
  + users(id, name, email, password\_hash, age, occupation, sex, address, twofa\_secret, is\_2fa\_enabled)
  + ratings(user\_id, movie\_id, rating), clicks(user\_id, movie\_id, timestamp),
  + user\_selected\_movies(user\_id, movie\_id), friend\_requests, friendships, messages

Rolul bazei de date este de a stoca permanent metadatele filmelor și interacțiunile utilizatorilor.

**API extern (TMDb):**

Utilizează patru endpoint-uri principale:

Rezultatele sunt salvate local în tabela movies pentru a evita apelurile repetate la TMDb.

**Fluxul general al cererii și răspunsului**

* **Frontend → Backend**: utilizatorul face o acțiune (de exemplu, caută un film sau cere recomandări). JavaScript face o solicitare AJAX/HTTP POST sau GET către o rută Flask.
* **Backend** primește cererea, verifică autorizarea, apoi apelează serviciile de recomandare (algoritmii din Surprise, scikit-learn, PyTorch, implicit) după caz.
* **Backend → Bază de date**: dacă este nevoie de date (ex.: rating-uri, click-uri, detalii film), serverul efectuează interogări SQL prin sqlite3 și pandas.read\_sql\_query.
* **Backend → Modul recomandare**: datele preluate (rating-uri, clik-uri, textele descrierilor) sunt preprocesate (vectorizare text, construire matrice user–item, SVD etc.) și se obțin top-N filme.
* **Backend → Frontend**: rezultatele (lista de filme recomandate, cu titlu și poster\_path) se trimit JSON, iar JavaScript se ocupă de afișarea lor dinamică în grid.

## Front-end

Aplicaţia oferă o interfaţă modernă, adaptată atât desktop-ului, cât şi dispozitivelor mobile, construită cu **HTML 5, CSS (Bootstrap + utilizări punctuale de Tailwind)** şi **JavaScript vanilla**. Templating-ul este realizat prin **Jinja 2**, astfel încât front-end-ul şi back-end-ul rămân perfect decuplate: fişierele .html descriu doar structura şi logica de prezentare, iar variabilele dinamice vin din rutele Flask. În cele ce urmează prezentăm rolul fiecărei pagini, modul în care este împărţit codul şi cum se face legătura cu serviciile Python.

* **Pagina de start – home.html**

Pagina „Acasă” este punctul zero al experienţei de navigare. Header-ul integrează **bara de căutare** cu auto-complete şi un **meniu de algoritmi** afişat sub formă de butoane. Selectarea unui algoritm declanşează, printr-un listener JS, ascunderea tuturor secţiunilor şi afişarea doar a celei relevante – comportament implementat în câteva linii de JavaScript, fără vreun framework suplimentar.  
Secţiunile de recomandări sunt generate în macro-ul section\_for, ceea ce permite reciclarea aceluiaşi fragment Jinja pentru zeci de liste diferite (CF, NMF, KNN, click-based etc.) . Fiecare tile conţine posterul (sursă TMDb), titlul şi un link către pagina detaliu. Layout-ul flex-wrap combinat cu o structură de tip grid adaptive, menţine lăţimea constantă a cardului (150 px) şi margini fluide, obţinând un efect de „masonry” simplu dar eficient pentru orice rezoluţie.

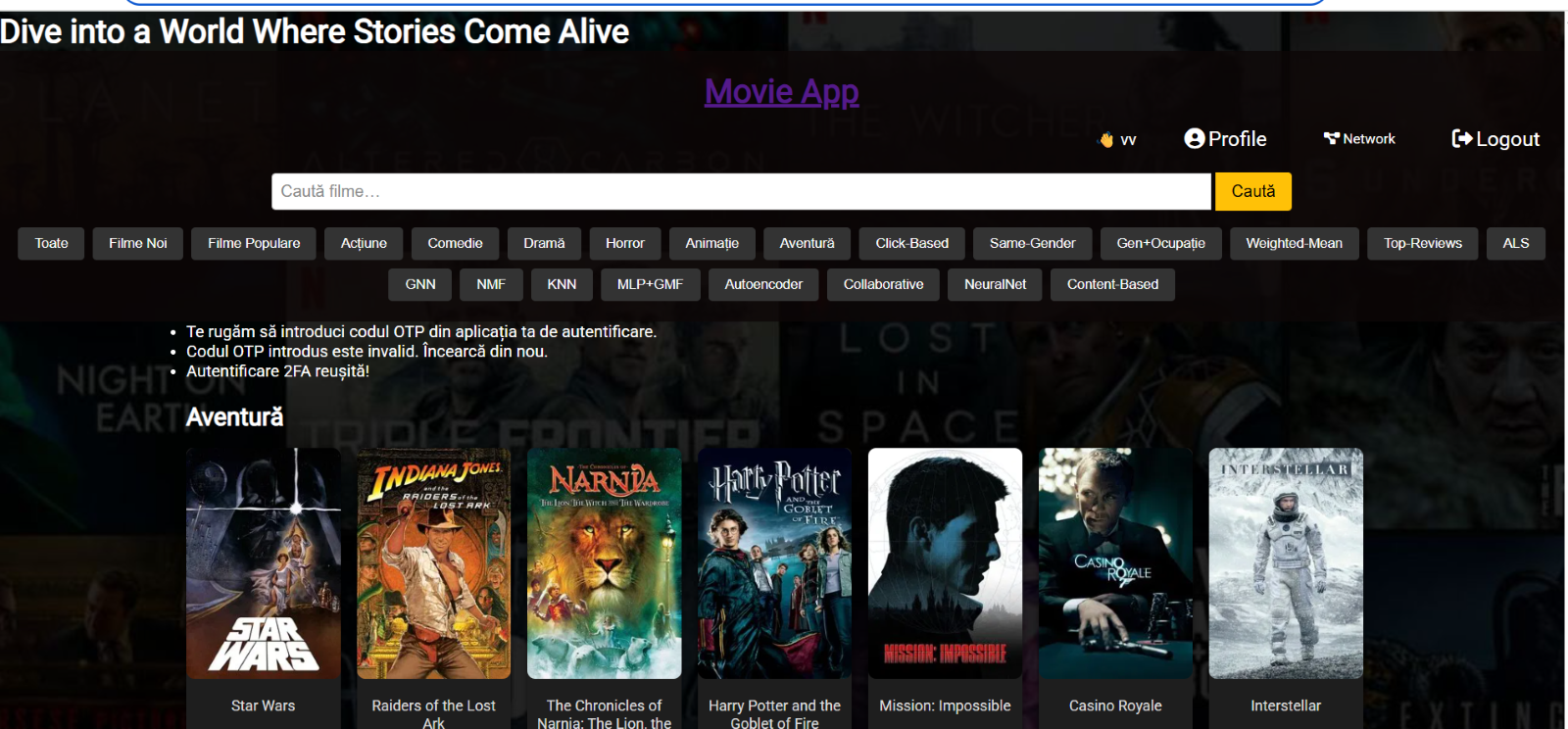


Fig. 4.2: Pagina home

* **Pagina de autentificare – login.html**

Pentru a păstra coerenţa vizuală, cardul de login este centrat peste un fundal full-screen, controlat dintr-un <style> local. Câmpurile de email şi parolă sunt validate de browser (atributele type="email", required) şi, după submit, formularul postează credenţialele spre ruta /login din main.py. Mesajele de eroare/confirmare revin sub formă de flash messages, randate de base.html.

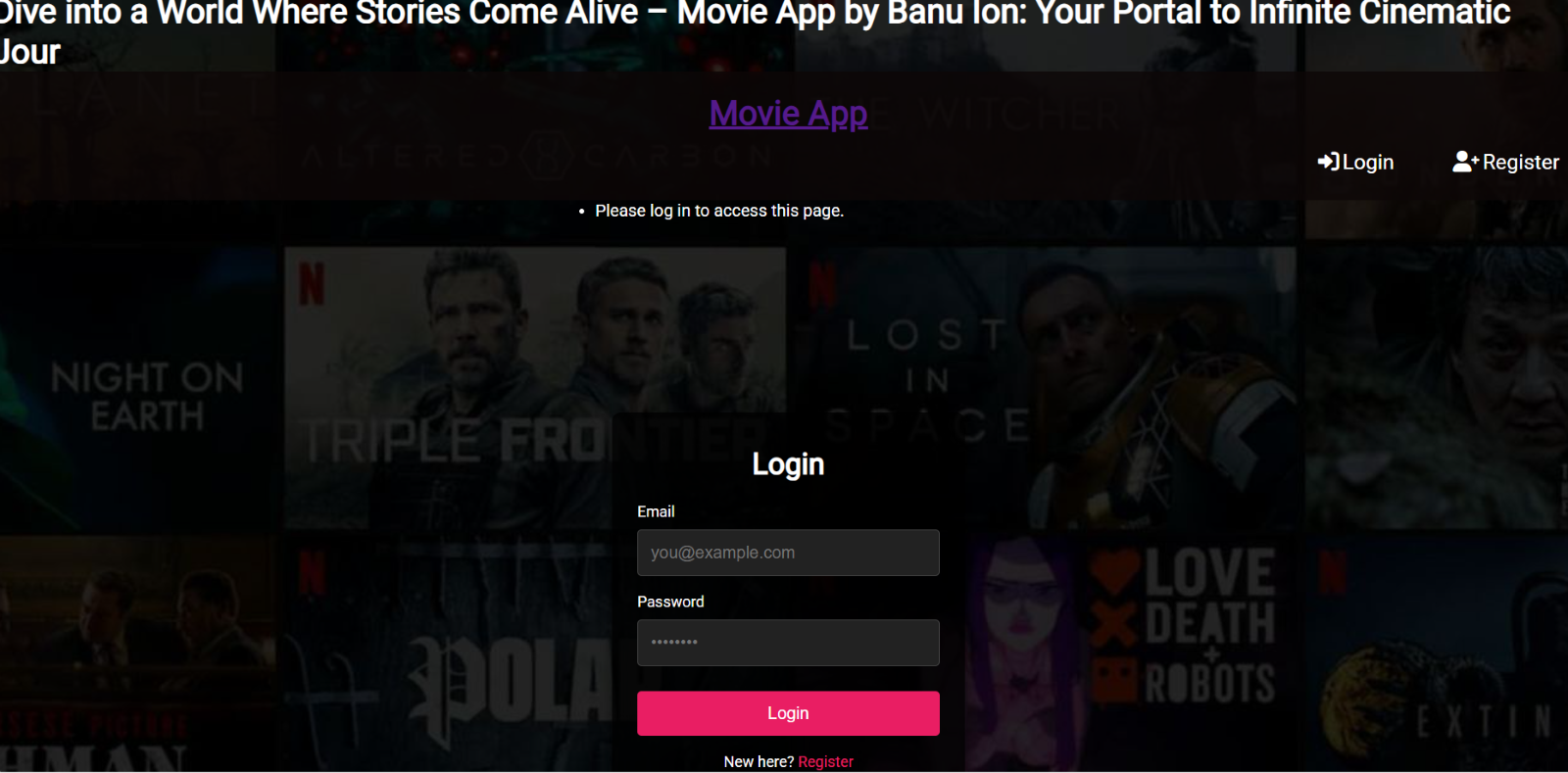


Fig. 4.3: Pagina login

* **Dashboard-ul administrativ – admin.html**

Dashboard-ul concentrează toate funcţiile avansate:

**Statistici** – şase grafice generate cu **Chart.js**; datele (top 10 filme, clusterizare K-Means, RMSE-uri) sunt trimise de Flask sub formă de liste Python, convertite în JSON de Jinja şi consumate direct în script.

**Operaţii** – buton pentru listarea utilizatorilor (/admin/users), acţiuni AJAX pentru ştergere şi detalii individuale, plus un formular pentru trimiterea posterelor pe email. Toate interacţiunile se fac cu fetch şi aşteaptă răspuns JSON de la rutele securizate.

* **Detaliul de film – movie\_detail.html**

Această pagină reuneşte cel mai bogat set de funcţii front-end:

**Header adaptiv** – poster + metadate (actori, genuri, dată lansare) şi, dacă există, un iframe YouTube cu trailerul.

**Tab-uri principale** (rating, poster, voice, review, algoritmi) + **sub-tab-uri** în pane-ul „Algoritmi”. Comutarea se face prin toggling de clase active şi setarea display:block/none.

**Rating în timp real** – stele FontAwesome; la click porneşte un fetch('/api/rate') cu corp JSON, afişând feedback instant.

**Upload poster** şi **Voice-to-text** – formulare multipart şi MediaRecorder API care trimit datele la /poster\_recommend şi /voice\_recommend, apoi populează dinamic carusele de filme.

**Review-uri** cu tone-analyzer Python; după submit, noua recenzie e inserată în DOM fără reload.

Tot acest comportament este implementat în ~200 linii de JS pur; nu depindem de jQuery sau alte biblioteci grele, ceea ce menţine timpii de încărcare mici.



Fig. 4.4: Pagina movie detail

* **Reţeaua socială – user\_recommendation.html**

Secţiunea „Reţeaua mea” permite prietenii, mesagerie, carusele de filme votate de prieteni şi recomandări mutuale. Codul JS orchestrează patru panouri: recomandări de prieteni, cereri, listă de prieteni, inbox. Fiecare buton declanşează o funcţie loadXYZ() care cheamă API-uri REST (/api/friends/recommended, /inbox etc.) şi construieşte HTML la zbor folosind template strings. Carousel-ul de postere foloseşte doar Flexbox cu overflow-x, evitând plug-in-uri externe.

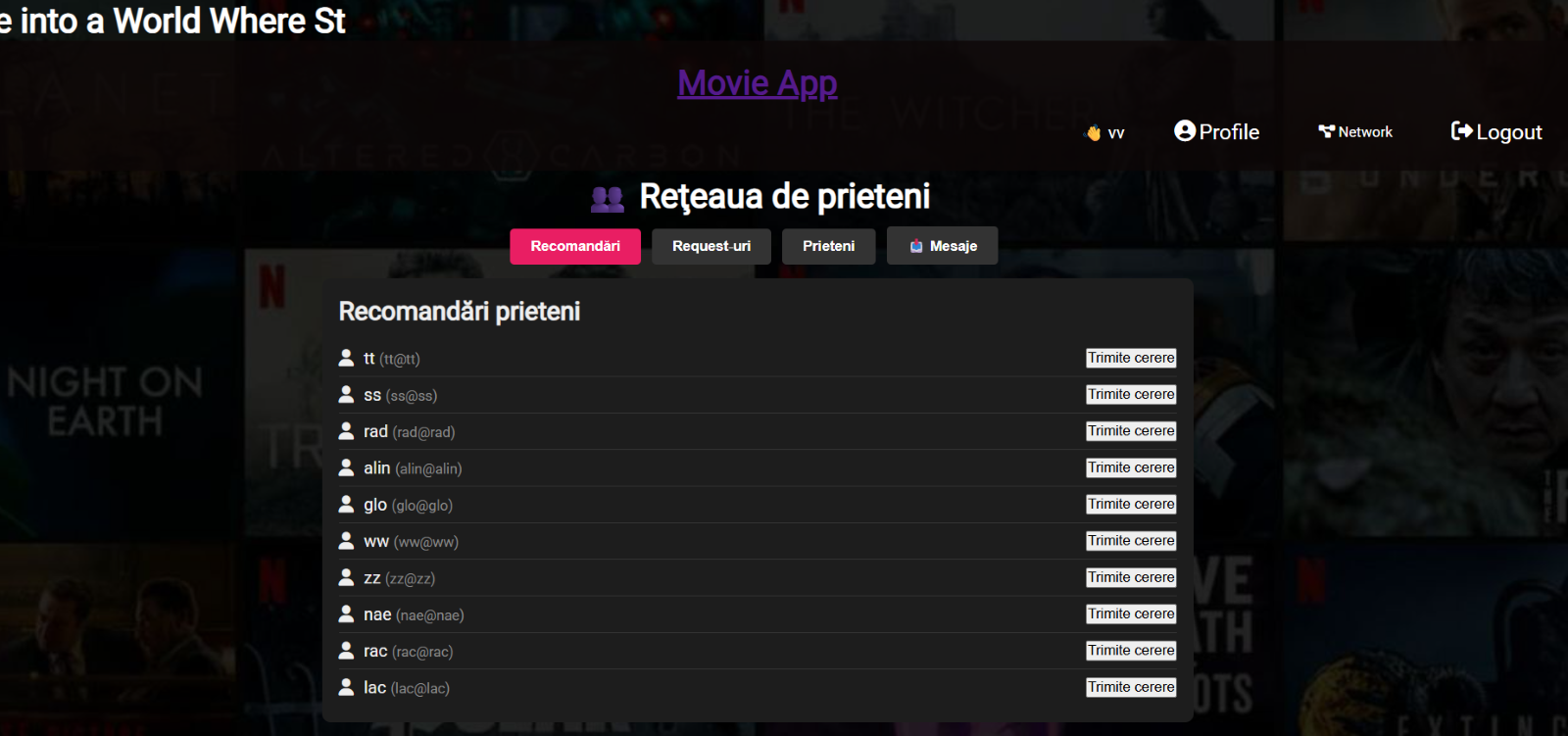


Fig. 4.5: Pagina prieteni

* **Profile.html**

Pagina dedicată informațiilor personale ale utilizatorului. Permite modificarea parolei, afișează datele demografice, lista de filme selectate la înregistrare și rating-urile deja acordate.  
Include validare HTML5 și mesaje de succes/eroare cu flash() din Flask.

* **Legătura cu back-end-ul Flask**

**Rute Jinja** – fiecare HTML este randat de o funcţie Python ce pregăteşte contextul (liste de filme, statistici,OTP).

**Servicii AJAX** – orice acţiune din UI (rating, review, cerere prietenie, trimitere poster) ajunge la o rută JSON (/api/\*). În main.py se găsesc funcţiile care:

* validează autentificarea;
* citesc sau scriu în SQLite via sqlite3 / pandas;
* forţează re-antrenarea sau apelul de predicţie la modulele ML;
* întorc un răspuns jsonify, consumat apoi de front-end pentru actualizarea DOM.

**Consideraţii de accesibilitate şi responsivitate**

**CSS variabil** – lăţimi în % şi flex-wrap pentru carduri.

**Contrast ridicat** – scheme cromatice #1e1e1e / #e0e0e0, butoane accent #e91e63.

**Keyboard-friendly** – toate butoanele au cursor:pointer, :hover şi focus vizibil; formularele pot fi navigate doar din tastatură.

**Fallback-uri** – dacă un film nu are poster, se afişează no\_image\_available.png. Dacă un API AJAX eșuează, utilizatorul primeşte alert explicit.

# Back-end

În acest capitol ne vom concentra pe componenta de back-end a aplicației. Vom detalia structura principalelor rute HTTP, logica serviciilor de recomandare (collaborative filtering, content-based, click-based și hybrid), mecanismele de securitate (rate limiting și autentificare cu doi factori folosind cod QR) și strategiile de caching pentru apelurile către TMDb. Pentru exemplificare vom utiliza **Flask** (Python) ca framework principal, deși conceptele pot fi adaptate și pentru alte framework-uri similare (Django, FastAPI etc.).

## Rute principale

### **Rute de înregistrare și autentificare**

**/register**

**Metode:** GET, POST

**Descriere:**

**GET**: Afișează formularul de înregistrare (register.html). În etapa de preprocesare a datelor pentru pagină, codul descarcă un anumit număr filme populare pentru fiecare gen relevant (Action, Drama, Adventure, Sci-Fi, Thriller, Comedy, Horror, Romance) prin API-ul TMDb, populând un dicționar genre\_movies cu genre\_name: listă\_filme. Această listă este transmisă către șablon pentru a fi afișată ca opțiuni de filme favorite la înregistrare.

**POST**: Primește datele trimise de formular (name, email, password, age, occupation, sex, address și lista de selected\_movies).

* Se validează că adresa de email nu există deja în tabela users.
* Dacă email-ul este liber, se inserează utilizatorul în tabela users din baza tmdb\_movies.db (cu parolă criptată).
* Se inserează același user și în baza tmdb\_movies\_for\_nn.db (pentru date de antrenare NN).
* Pentru fiecare film selectat în formular se inserează o înregistrare în tabela user\_selected\_movies.
* Se redirecționează către pagina de login (/login) cu mesaj de succes.

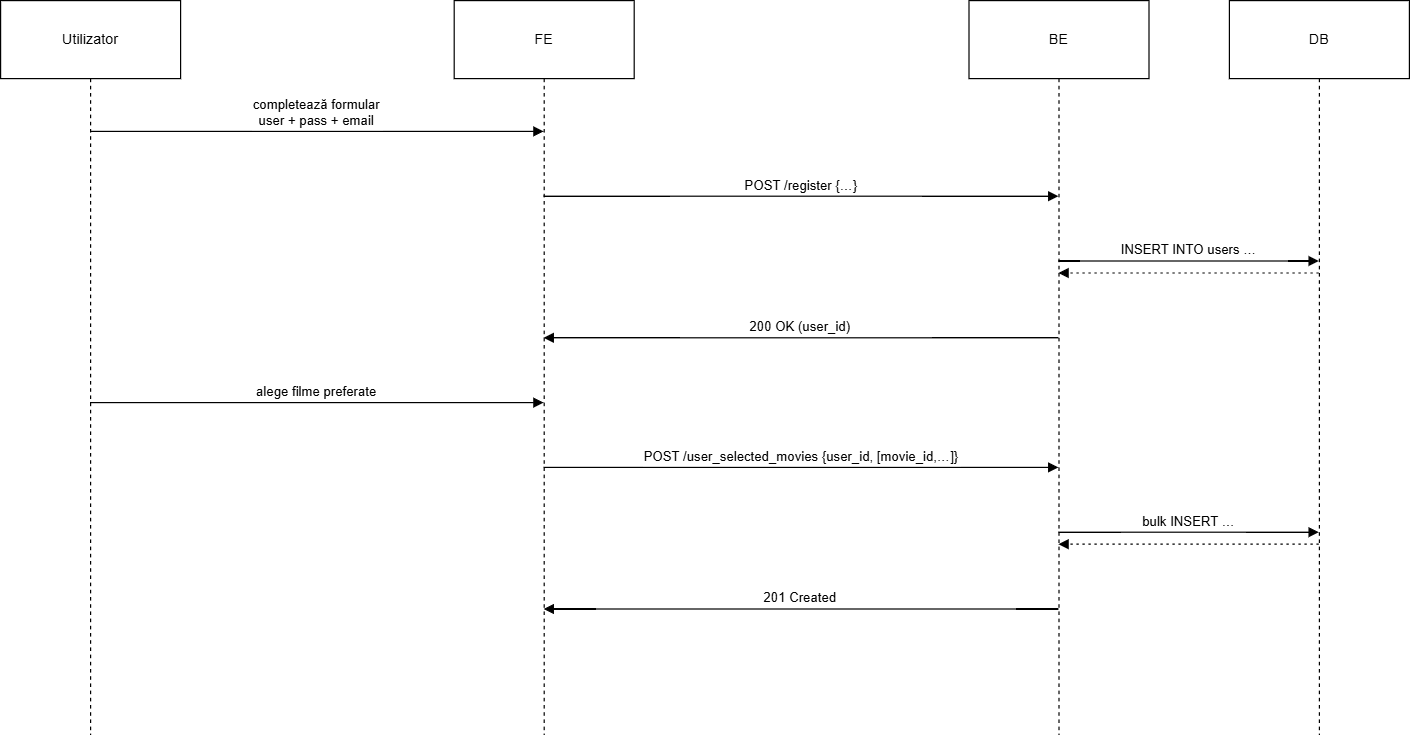


Fig. 5.1: Fluxul de înregistrare

Componentele implicate:

* **Utilizator** – interacționează cu aplicația.
* **FE (Frontend)** – interfața vizibilă de utilizator.
* **BE (Backend)** – serverul care procesează cererile.
* **DB (Database)** – baza de date unde se stochează informațiile.

**/login**

**Metode:** GET, POST

**Descriere:**

**GET**: Afișează formularul de autentificare (login.html).

**POST**: Primește email și password.

Se caută în tabela users (în tmdb\_movies.db) după email; dacă gasim user și parola verificată cu check\_password\_hash, salvăm în sesiune.

* Dacă utilizatorul are is\_2fa\_enabled = 0, îl redirecționăm să activeze 2FA (/enable\_2fa).
* Dacă are 2FA activat, îl redirecționăm la pagina de verificare OTP (/verify\_2fa), afișând un mesaj „Te rugăm să introduci codul OTP…”.
* Dacă datele de autentificare sunt incorecte, afișăm mesaj de eroare.

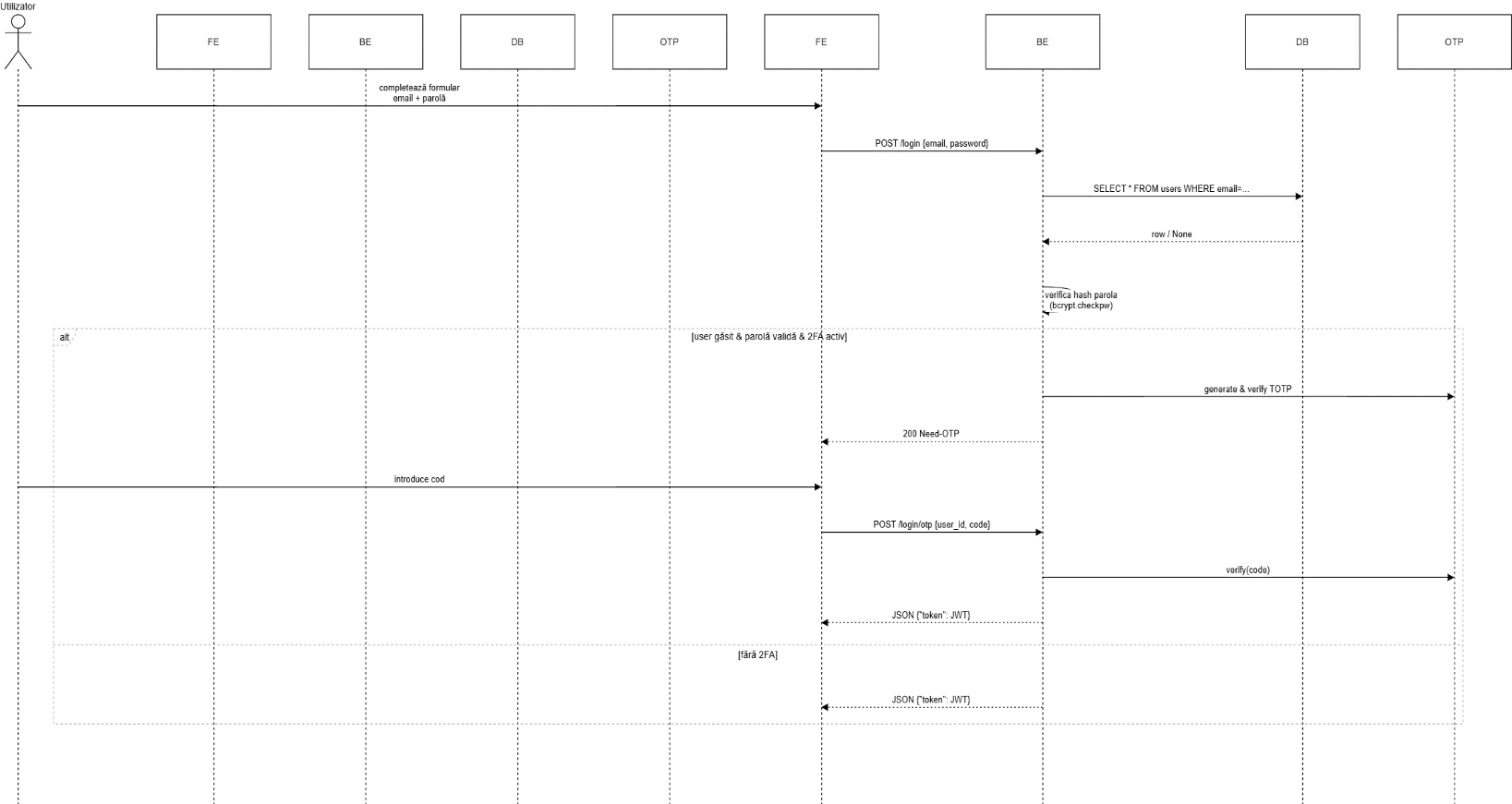


Fig. 5.2: Fluxul de autentificare

### **Rute de activare și verificare 2FA**

**/verify\_2fa**

**Metode:** GET, POST

**Descriere:**

**GET**: Afișează un formular simplu (verify\_2fa.html) cu un câmp pentru codul OTP.

**POST**: Primește otp\_code (codul de 6 cifre generat de aplicația de autentificare).

Se verifică că în session există temp\_user\_id. Dacă nu, se afișează mesaj și redirecționăm la login.

Din DB (în tmdb\_movies.db), citim câmpul twofa\_secret pentru temp\_user\_id.

Creăm obiect pyotp.TOTP(secret) și verificăm codul OTP.

Dacă este valid, înlocuim în sesiune:

* Ștergem cheile temporare (temp\_user\_email, temp\_user\_id, temp\_user\_name).
* Dacă email-ul este „admin@admin” (verificare case-insensitive), redirecționăm la /admin\_dashboard. Altfel, redirect la /index.
* Dacă codul OTP este invalid, afișăm mesaj de eroare și reafișăm formularul de introducere OTP.

**/enable\_2fa**

**Metode:** GET, POST

**Descriere:**

Precondiție: session trebuie să conțină temp\_user\_id, înseamnând faptul că utilizatorul s-a autentificat, dar nu are 2FA activat încă.

**GET**:

* Se verifică în DB (în tabela users) dacă există deja un twofa\_secret pentru temp\_user\_id.
* Dacă nu există, generăm un nou secret cu pyotp.random\_base32() și salvăm în DB printr-un UPDATE users SET twofa\_secret = ? WHERE id = temp\_user\_id.
* Dacă există, îl utilizăm pe cel existent.
* Construim TOTP cu pyotp.TOTP(secret).provisioning\_uri(name=email, issuer\_name="MyFlaskApp").
* Generăm un QR Code ca imagine base64 prin qrcode și create\_qr\_code\_base64(totp\_uri).
* Transmitem către șablon (enable\_2fa.html) o variabilă qr\_code\_base64, astfel încât front-end-ul poate afișa <img src="data:image/png;base64,{{ qr\_code\_base64 }}">.

**POST**: Primește otp\_code introdus de utilizator.

* Citim din DB twofa\_secret pentru temp\_user\_id.
* Verificăm codul OTP; dacă este corect, facem UPDATE users SET is\_2fa\_enabled = 1 WHERE id = temp\_user\_id.
* Setăm sesiunea permanent: session['user\_id'] = temp\_user\_id, ștergem valorile temporare (temp\_user\_id, temp\_user\_name).
* Afișăm flash-message „2FA activat cu succes!” și redirect la /index.
* Dacă codul OTP este invalid, afișăm mesaj de eroare și revenim la același șablon.

### **Rute pentru gestionarea sesiunii și logout**

**/logout**

**Metode:** GET (implicit)

**Descriere:**

Șterge toate valorile din sesiune (session.clear()) și redirecționează utilizatorul către pagina de login, afișând un mesaj flash „You have been logged out.”.

***/session***

**Metode:** GET

**Descriere:**

* Returnează JSON cu numele utilizatorului curent din sesiune.
* Util pentru debugging sau afișarea numelui userului în front-end-ul care folosește AJAX.

### **5.1.4 Rute pentru prietenii și mesaje (API JSON)**

**/api/friends/recommended**

**Metode:** GET

**Descriere:** Returnează, pentru utilizatorul logat (din session['user\_id']), lista de maxim 10 utilizatori „recomandați” să fie prieteni, ordonați după similaritatea distanței Euclidiene pe vectorul de rating-uri.

* Se încarcă toate rating-urile din tabela ratings în DataFrame ratings (coloane: user\_id, movie\_id, rating).
* Se construiește pivot table (utilizatorii pe rânduri, filmele pe coloane, valori = rating, completate cu 0).
* Dacă utilizatorul curent (uid) nu apare în index, se returnează [].
* Se obține lista de prieteni actuali din tabela friendships (și se exclude aceștia de la recomandare).
* Pentru fiecare alt utilizator other\_uid (nici el, nici prietenul curentului), se calculează distanța Euclidiană între vectorul de rating-uri al lui uid și cel al lui other\_uid.
* Se iau primii 10 utilizatori cu distanța cea mai mică (deci cei mai asemănători).
* Pentru fiecare id recomandat, se adaugă în listă un obiect JSON cu { id, name, email, distance }.
* Returnează jsonify(users).

**/api/messages/inbox**

**Metode:** GET

**Descriere:** Returnează toate mesajele primite de utilizatorul curent, ordonat descrescător după data trimiterii (sent\_at).

* Se preia uid = session['user\_id'].
* Se execută query SQL
* Se returnează JSON cu lista de mesaje, fiecare având id, sender\_id, sender\_name, sender\_email, body, sent\_at.

**/api/messages/<int:msg\_id>/delete**

**Metode:** POST

**Descriere:** Șterge un mesaj cu id = msg\_id numai dacă user-ul curent este destinatarul (receiver\_id) mesajului.

* Se obține uid = session['user\_id'].
* Verificăm că există SELECT 1 FROM messages WHERE id=? AND receiver\_id=?. Dacă nu, respons cu 403 Forbidden.
* Dacă există, executăm DELETE FROM messages WHERE id = ?.
* Returnează jsonify(ok=True).

***/api/messages/send***

**Metode:** POST

**Descriere:** Trimite un mesaj de la utilizatorul curent către to\_id.

* Din request.get\_json(), extragem to (id-ul destinatarului) și body (textul mesajului).
* Dacă body e gol, răspundem cu eroare 400.
* Verificăm dacă uid și to\_id sunt prieteni (are\_friends(uid, to\_id)); dacă nu, 403 Forbidden.
* Inserăm în tabela messages(sender\_id, receiver\_id, body) cu m = (uid, to\_id, body).
* Returnează jsonify(ok=True).

**/api/friends/<int:friend\_id>/delete**

**Metode:** POST

**Descriere:** Șterge relația de prietenie (ambele sensuri) între utilizatorul curent și friend\_id.

* Se obține uid = session['user\_id'].
* Executăm două comenzi SQL cu DELETE:

(inversând ordinea user\_id/friend\_id pentru a șterge ambele direcții).

* Returnează jsonify(ok=True).

**/api/friends/requests**

**Metode:** GET, POST

**Descriere:**

**POST**: Creează o cerere de prietenie de la utilizatorul curent (session['user\_id']) către un alt utilizator target\_id.

Citește JSON-ul cu target\_id. Dacă target\_id e invalid sau egal cu uid, returnează eroare 400.

Încearcă INSERT INTO friend\_requests(sender\_id, receiver\_id) VALUES (?, ?).

* Dacă IntegrityError (deja există), returnează 400 cu msg="deja există".
* Altfel, returnează jsonify(ok=True).

**GET**: Returnează lista tuturor cererilor de prietenie unde utilizatorul curent e receiver\_id și status='pending'.

* Returnează JSON cu lista cererilor primate

**/api/friends/requests/<int:req\_id>/answer**

**Metode:** POST

**Descriere:** Răspunde la o cerere de prietenie (acționează fie „accept”, fie „deny”).

* Din JSON-ul primit, extragem action = "accept" sau "deny". Dacă e altceva, 400 Bad Request.
* Se caută în DB un rând în friend\_requests cu id=req\_id, receiver\_id=uid, status='pending'.
* Dacă nu găsim, răspundem 404 Not Found.
* Dacă action == 'accept', setăm new\_status = 'accepted'; altfel new\_status = 'denied'.
* Dacă accept, adăugăm în tabela friendships două rânduri: (uid, sender) și (sender, uid). (Cu INSERT OR IGNORE pentru a nu duplica).
* Actualizăm în friend\_requests coloana status cu new\_status pentru id=req\_id.
* Returnează jsonify(ok=True).

**/api/friends/list**

**Metode:** GET

**Descriere:** Returnează lista de prieteni a utilizatorului curent.

Query SQL care returnează JSON cu lista de obiecte { id, name, email }.

**/api/friends/<int:friend\_id>/movies**

**Metode**: GET

**Descriere:** Pentru un prieten cu id = friend\_id, returnează top 30 de filme pe care i le-a notat (ordonați descrescător după rating).

* Se verifică dacă uid și friend\_id sunt prieteni (folosind are\_friends(…)). Dacă nu, 403 Forbidden.
* Returnează JSON cu lista de filme (fiecare având id, title, poster\_path, rating).

### **5.1.5 Rute pentru crearea, citirea și ștergerea utilizatorilor (admin)**

***/admin* (alias *admin\_dashboard*)**

**Metode:** GET

**Descriere:** Afișează pagina dashboard-ului de admin (admin.html), cu:

* **Verificare acces**: Dacă session['user\_name'].lower() != 'admin', răspunde 403 Forbidden.
* **Listare useri**: Se citește din DB tabela users (id, name).
* Pentru fiecare utilizator ((user\_id, user\_name)), se calculează RMSE doar pentru recomandările content-based vs rating-urile pe care le-a dat (apelând compute\_rmse\_for\_algorithms(user\_id) și extrag „content\_cb”). Apoi construim rmse\_list cu { user\_id, user\_name, rmse }.
* Sortăm descrescător după RMSE și păstrăm primele 10 în top10\_rmse.
* Citește toate rating-urile și se creează un pivot user × movie, apoi se reduce dimensional (PCA 2D) și face KMeans cu 3 clustere[1]. Construiește structuri JSON clusters\_points și centroid\_points cu coordonatele (x, y).
* Calculează top 10 filme cu cea mai mare medie de rating din DB (clauză SQL cu AVG(reviews.grade)) și primele 10 „flop” (cele mai mici medii).
* Returnează șablonul admin.html cu variabilele:
* clusters\_points, centroid\_points, best\_movies, flop\_movies, top10\_rmse.
* Acest dashboard afișează o hartă 2D a utilizatorilor clusterați, tabelul cu cele mai bine/ prost evaluate filme și lista celor 10 utilizatori cu cei mai mari RMSE.

**/send\_posters**

**Metode:** POST

**Descriere**: Permite admin-ului să trimită prin email „cele mai noi 5 filme” către toți utilizatorii sau către un singur email specificat.

* Verifică dacă session['user\_name'].lower() == 'admin'; altfel, returnează JSON { message: 'Forbidden' }, status 403.
* Din request.get\_json(), citește email și send\_all.
* Dacă send\_all = true, interoghează DB pentru toate email-urile distincte din tabela users.
* Dacă single\_email e prezent, construiește o listă cu acel email (fără a verifica dacă există în DB).
* Dacă nu există destinatari, returnează { message: 'Nu ai selectat niciun destinatar!'}, status 400.
* Cheamă funcția get\_latest\_movies(5) care face un query:

și returnează o listă de dicționare { title, release\_date, poster\_path }.

* Pentru fiecare destinatar, construiește un email (EmailMessage) cu subiect „Cele mai noi 5 filme …”, conținut text (lista de titluri + date) și conținut HTML cu un tabel ce include poster-ele (imaginile sunt URL-urile din DB).
* Se deschide o conexiune SMTP\_SSL la smtp.mail.yahoo.com:465, se loghează și trimite mesaj câte unul per destinatar.
* Returnează JSON { message: 'Trimis cu succes către X destinatari.' }.

**/admin/users**

**Metode:** GET

**Descriere:** Returnează JSON cu toți utilizatorii (id, name, email) din DB (persoanele care au email valid).

* Verificare dacă session['user\_name'].lower() == 'admin'; altfel 403 Forbidden.
* Returnează jsonify(users), unde users este o listă de obiecte { id, name, email }.

**/admin/users/<int:user\_id>**

**Metode:** GET

**Descriere**: Returnează detaliile complete despre un anumit utilizator (doar admin).

* Verifică dacă admin (dacă session['user\_name'] != 'admin', 403).
* Query SQL pentru tabela users.

Dacă nu se găsește user, returnează 404 cu JSON { error: 'User not found' }.

* Încarcă rating-urile date de acest user:

Rezultatul se stochează în lista ratings de dicționare { rating\_id, movie\_id, movie\_title, rating }.

**/admin/users/delete**

**Metode:** POST

**Descriere:** Șterge un utilizator după user\_id (doar admin).

* Verifică dacă session['user\_name'] == 'admin'; dacă nu, 403.
* JSON-ul primit trebuie să conțină user\_id. Dacă nu există, returnează 400 cu { message: 'Missing user\_id' }.
* Execută DELETE FROM users WHERE id = ?.
* Returnează JSON { message: f'User {user\_id} șters cu succes.' }.

### **5.1.6 Pagină principală și paginare de filme**

***/* (alias *index*)**

**Metode:** GET

**Decorator:** @login\_required

**Descriere:** Aceasta este pagina de start („home”) afișată după login. Realizează următoarele operații:

Înțelege parametrii query-string:

page: numărul paginii (implicit 1).

sort\_by: cel puțin „release\_date” (implicit) sau „popularity”.

Se calculează per\_page = 10, offset = (page-1)\*10.

**Filme noi** (cele mai recente 17 filme după release\_date DESC) și **filme populare** (top 17 după popularity DESC) – stocate în new\_movies, respectiv popular\_movies.

**Listă de genuri unice**: Se citește din DB toate genurile din coloana genres și se formează un set. Apoi, pentru fiecare gen, se execută un query.

Astfel se obține genre\_lists[genre] = listă de (id, title, poster\_path).

**Paginare / Sortare generală**:

Se află total\_movies = SELECT COUNT(\*) FROM movies.

Se calculează total\_pages.

Dacă sort\_by == 'popularity', se alege query de sortare după ORDER BY popularity DESC LIMIT 10 OFFSET …. Altfel (default) se folosește ORDER BY release\_date DESC. Rezultatul e stocat în movies (lista paginată).

Dacă page e invalid (<1 sau > total\_pages), afișăm flash-message de avertizare și redirect la index cu parametrii corecți.

**Recomandări pentru utilizatorul curent** (dacă există):

* **Collaborative Filtering**: dacă algo (SVD) e inițializat și există user\_id, se apelează get\_collaborative\_filtering\_recommendations(user\_id, top\_n=7) → recommended\_movie\_ids. Dacă nu e gol, se face un query SQL. și rezultatul se pune în recommended\_movies.
* **Content-Based (pentru utilizator)**: dacă user\_id există, se apelează get\_content\_based\_recommendations\_for\_user(user\_id, top\_n=14) → content\_based\_movie\_ids.

Dacă nu e gol, determinăm content\_based\_recommended\_movies cu un query SQL asemănător.

* **Weighted Mean (User-based CF)**:

Se obține weighted\_mean\_recommended\_movie\_ids=

user\_based\_weighted\_mean\_recommendation(user\_id, top\_n=7).

Apoi, similar cu cele de mai sus, se extrag din DB datele filmelor în weighted\_mean\_recommended\_movies.

* **Weighted Mean filtrat după sex**:

Dacă r\_matrix e None, gender\_based\_recommended\_ids = []. Altfel, get\_top\_recommendations(cf\_user\_wmean\_gender, user\_id,7), apoi query DB pentru a popula gender\_based\_recommended\_movies.

* **Weighted Mean filtrat după sex + ocupație**:

Se obține:

gen\_occ\_based\_recommended\_ids=get\_top\_recommendations(cf\_user\_wmean\_gen\_occ,user\_id,7)

Apoi se populează gen\_occ\_based\_recommended\_movies din DB.

* **Click-Based CF (SVD pe clik-uri)**:

Dacă algo\_clicks și trainset\_clicks (inițializate în startup) sunt disponibile, click\_based\_movie\_ids=get\_collaborative\_click\_recommendations(user\_id, top\_n=7). Apoi query SQL pentru click\_based\_recommended\_movies.

* **Neural Network (NN-CF)**:

Dacă nn\_model (Sae / rețea Keras) e încărcat și user\_map, movie\_map sunt valabile, atunci nn\_recommended\_ids = recommend\_movies\_nn(user\_id, nn\_model, user\_map, movie\_map, 7). Apoi, în acest caz, pentru fiecare mid se apelează fetch\_tmdb\_movie\_data(mid) și rezultatele valabile se adaugă în nn\_recommended\_movies.Există log-uri de debugging pentru nn\_recommended\_movies, afișate în console).

* **ALS Implicit (cu implicit.als)**:

Dacă als\_model\_implicit e inițializat, se apelează als\_recommended\_ids = get\_als\_recommendations\_implicit(user\_id, 7).

Apoi pentru fiecare mid se face fetch\_tmdb\_movie\_data(mid), populându-se als\_recommended\_movies.

* **NMF (Non-negative Matrix Factorization)**:

Se extrage un film ales de utilizator.

Dacă există current\_movie\_id, atunci:

nmf\_recommended\_ids= get\_recommendations\_NMF(current\_movie\_id, 7).

După aceea, se fac query-uri pentru a popula nmf\_recommended\_movies.

* **GMF+MLP (NeuMF)**:

Se face recommended\_idss recommend\_movies\_gmf\_mlp(user\_id, 7) și apoi, pentru fiecare mid, se face fetch\_tmdb\_movie\_data(mid) → recommended\_moviesgmf.

* **Autoencoder (SAE)**:

auto\_rec\_ids = get\_autoencoder\_recommendations(user\_id, sae\_model, ratings\_tensor, 7) și apoi la fel, fetch\_tmdb\_movie\_data(mid) → auto\_recommended\_movies.

* **GNN (MovieLensNet)**:

gnn\_rec\_ids = get\_gnn\_recommendations(user\_id\_real, model, user\_map, movie\_map, genre\_map, 7). Apoi fetch\_tmdb\_movie\_data(mid) → gnn\_recommended\_movies.

* **KNNBasic (Surprise KNN)**:

knn\_rec\_ids = get\_knn\_recommendations(user\_id, 7). Se face query SQL pentru knn\_recommended\_movies.

Se închide conexiunea DB (conn.close()).

Obținem user\_name = session['user\_name'] pentru a-l afișa în șablon.

**Scop:** Această pagină e punctul central al aplicației: afișează paginarea de filme, secțiunile de „Filme Noi”, „Filme Populare”, „Filme pe Genuri”, „Top filme după recenzii”, plus multiple carusele cu recomandări pentru utilizator (în funcție de algoritmul ales).

Ca orice pagină, poate primi parametru de sortare ?sort\_by=popularity sau ?sort\_by=release\_date, respectiv pagină ?page=2.

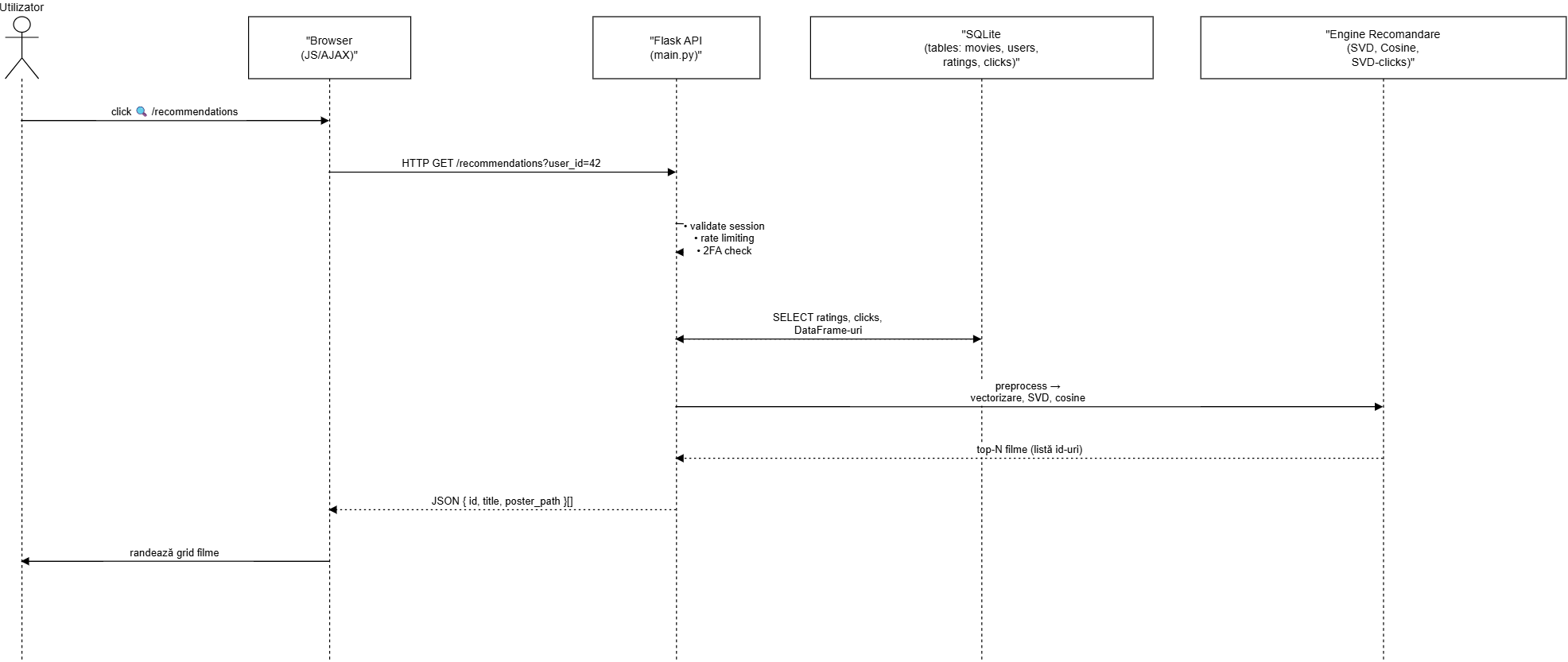


Fig. 5.3: Fluxul de generare al recomandărilor colaborative

### **5.1.7 Pagină de profil**

**/profile**

**Metode:** GET, POST

**Decorator:** @login\_required

**Descriere:** Permite utilizatorului curent să-și vadă datele de profil și să-și schimbe parola.

**GET**:

* Obținem uid = session['user\_id'].
* Din DB (tmdb\_movies.db), de face un select pentru a obtine un rating de la utilizatorul din sesiune.

**POST** (se trimite form pentru schimbare parolă):

* Din formular, citim current\_password, new\_password, confirm\_password.
* Dacă oricare e gol, flash: „Completează toate câmpurile.”
* Dacă new\_password != confirm\_password, flash: „Parolele nu coincid.”
* Altfel, din DB citim SELECT password FROM users WHERE id=uid.
* Dacă current\_password nu corespunde cu hash-ul stocat, flash: „Parola curentă este incorectă.”
* Altfel, generăm new\_hash = generate\_password\_hash(new\_password), apoi UPDATE users SET password = new\_hash WHERE id = uid. Afișăm „Parola a fost schimbată cu succes!”.
* După orice procesare POST, redirect la /profile (pentru GET, astfel încât utilizatorul să vadă noua stare).

### **5.1.8 Ruta de căutare filme**

***/search***

**Metode:** GET

**Decorator:** @login\_required

**Descriere:** Primește parametru query-string ?query=… (termen de căutare).

* Dacă query e gol (sau doar spații), afișează flash-message „Te rog introdu un termen de căutare.” și redirect către /index.
* Altfel, se face query SQL
* Rezultatul (lista de tuple (id, title, poster\_path, overview, release\_date, genres)) se transmite șablonului search\_results.html, împreună cu query și user\_name = session['user\_name'].
* Schița șablonului ar afișa o listă de carduri de filme care conțin titlu, poster, descriere etc.

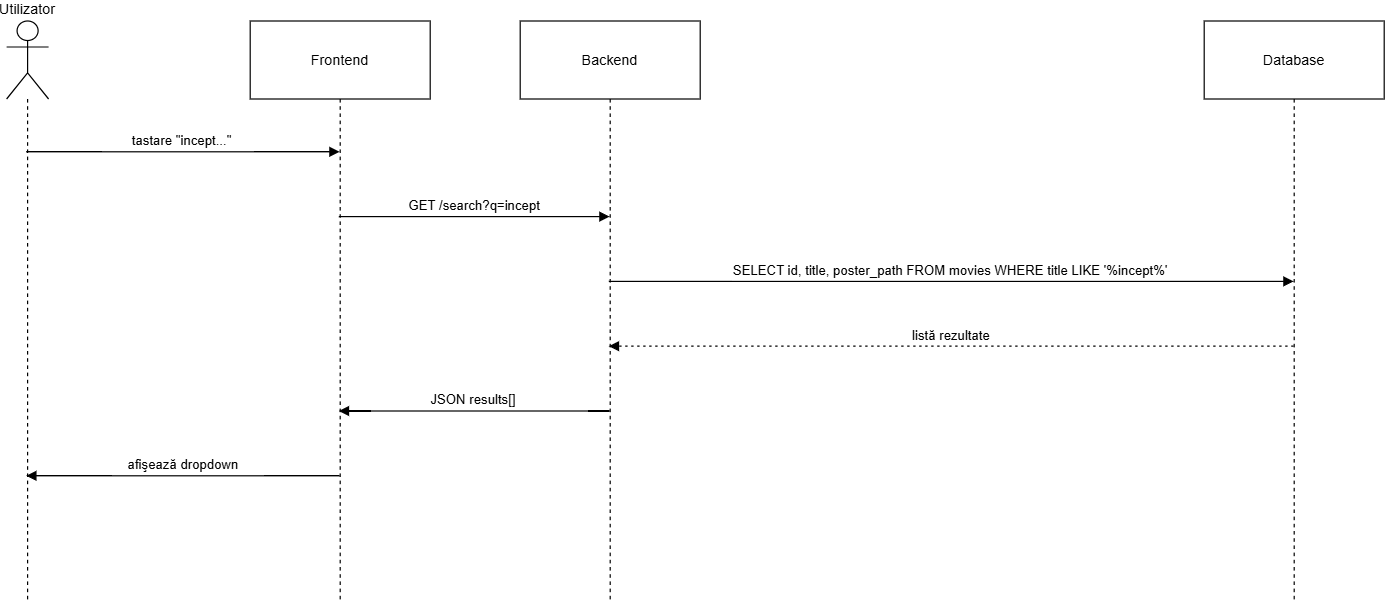


Fig. 5.4: Fluxul de căutare film

### **5.1.9 Ruta detalii film**

**/movie/<int:movie\_id>**

**Metode:** GET, POST

**Decorator:** @login\_required

**Descriere:** Aceasta e pagina detaliată a unui film (movie\_detail.html). Realizează două lucruri principale:

**Înregistrare click**: Imediat la începutul GET sau POST, se apelează record\_click(user\_id, movie\_id). Aceasta inseră un rând în tabela clicks(user\_id, movie\_id, timestamp), apoi reinițializează modelul de click-based CF (apelând initialize\_collaborative\_filtering\_clicks()).

**GET:**

* Se apelează initialize\_user\_based\_matrices() și load\_users().
* Se construiește lista de recomandări hibride bazate pe click-uri prin hybrid\_recommendations = get\_hybrid\_recommendations\_clicks(user\_id, movie\_id, 10).
* Se interoghează DB pentru datele filmului.

Dacă nu găsește filmul, flash „Filmul nu a fost găsit.” și redirect la /index.

* Se obțin recomandări pe baza conținutului comparând movie\_id cu celelalte filme.
* Se obțin recomandări hibride SVD prin hybrid\_recommended\_movie\_ids =get\_hybrid\_recommendations(user\_id,movie\_id,10)→ hybrid\_recommended\_movies. (Se rezolvă fallback dacă algo is None).
* **Rating-ul mediu al filmului**.
* **Recomandări hibride (SVD+content)**:
* **Recenzii**: SELECT recenziile din tabela reviews (join cu users pentru users.name).
* **Recomandări avansate prin profil de conținut precomputat**:

Aici actors e string-ul cu primii 5 actori (codat în JSON) precomputat în content\_profiles.json.

* **Recomandări BERT**:
* **Watch links**: Se pregătesc link-uri pentru diferite platforme de streaming gratuit/contra cost, URL colorate prin urllib.parse.quote\_plus(title).
* Se închide conexiunea DB.
* Se afișează șablonul movie\_detail.html împreună cu toate datele

Practic, pagina detaliată include: poster, titlul, descrierea, genurile, trailerul YouTube, lista recenziilor cu sentiment/grade, plus carusel de recomandări din diverse algoritmi și link-uri de streaming.

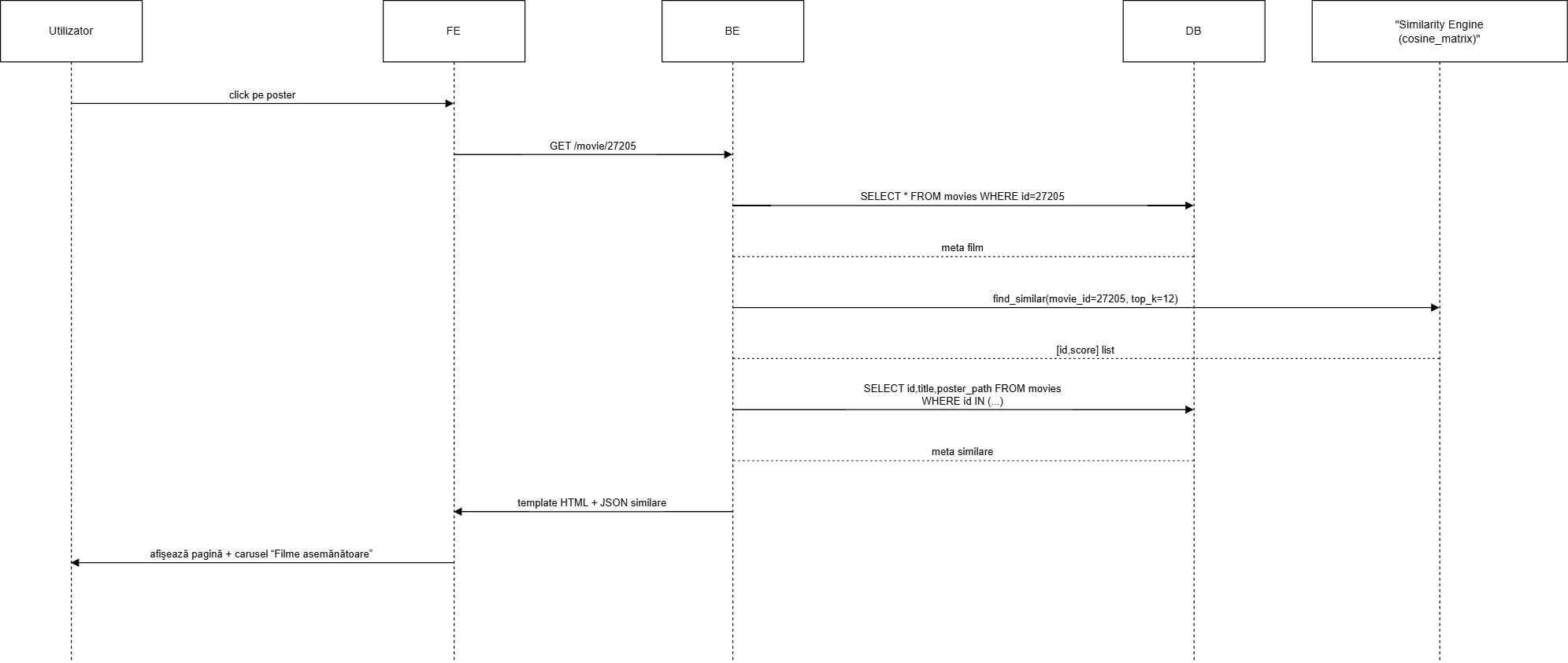


Fig. 5.5: Fluxul de generare filtrare bazat pe conținut

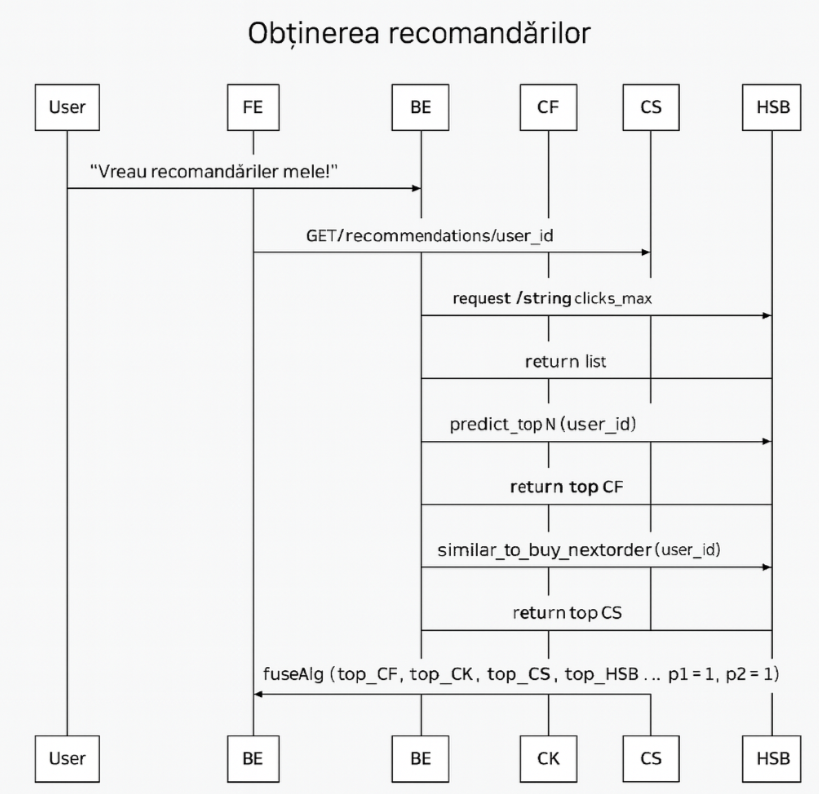


Fig. 5.6: Fluxul de generare filtrări hibride

### **5.1.10 Endpoints AJAX pentru voturi și recenzii**

**/api/rate**

**Metode:** POST

**Decorator:** @login\_required

**Descriere:** Endpoint JSON pentru voturi prin AJAX. Primește un JSON cu movie\_id și rating (1–5).

* Dacă rating\_int nu e între 1 și 5, răspunde cu eroare 400 {"error": "Rating invalid"}.
* Se deschid două conexiuni DB.
* Se execută în ambele baze.
* Se commit în ambele baze.
* Se re-inițializează modelele non-neurale:

initialize\_collaborative\_filtering\_clicks()

initialize\_knn\_model(db\_path="tmdb\_movies.db")

initialize\_user\_based\_matrices()

* Returnează jsonify(msg="OK").
* În caz de eroare la DB, 500 {"error": "DB error"}.

***/api/review***

**Metode:** POST

**Decorator:** @login\_required

**Descriere:** Endpoint JSON pentru recenzii trimise prin AJAX. Primește JSON cu movie\_id și review (text).

* Dacă len(review\_text) < 4, return 400 {"error": "Review prea scurt"}.
* Se apelează analyze\_sentiment\_and\_grade(review\_text) pentru a obține sentiment și grade. [16]
* Se inserează în tabela reviews.
* Returnează jsonify(msg="OK", sentiment=sentiment, grade=grade, user=session['user\_name']).
* În caz de eroare, 500 {"error": "DB error"}.

### **5.1.11 Autocomplete și liste de titluri filme (JSON)**

**/autocomplete**

**Metode:** GET

**Decorator:** @login\_required

**Descriere:** Pentru căutare rapidă (autocomplete) în bara de căutare. Primește parametru term.

* Returnează JSON cu lista de titluri care conțin term.

### **5.1.12 Rute AJAX și webhook-uri diverse**

***/voice\_recommend***

**Metode:** POST

**Decorator:** @login\_required

**Descriere:** Primește un fișier audio (din browser, un fișier webm în audio\_data). Scop: recunoaștere vocală (Speech Recognition) și generare de recomandări pe baza textului recunoscut.

* Dacă nu primim nimic (audio\_data), returnăm {"transcript":"", "recommendations":[]}.
* Salvăm fișierul WebM într-un fișier temp (tmp\_webm).
* Convertim în WAV: pydub.AudioSegment.from\_file(...).export(wav\_path, format="wav"). Dacă necesită conversie, altfel eroare 400.
* Speech\_recognition.Recognizer().recognize\_google(audio\_data, language='ro-RO') obținem textul în română. Dacă nicio voce nu e detectată, setăm transcript = "". Ștergem fișierele temporare. [6]
* Pentru recomandări:
  + Citim toate filmele din DB în DataFrame df (coloane: id, title, poster\_path, overview, genres).
  + Construim tags = title + title + overview + " " + genres (concatenare, accent pe titlu).
  + Realizăm un corpus format din df['tags'] + [transcript].
  + CountVectorizer(stop\_words='english', max\_features=5000) pe întreg corpusul, calculăm cosine\_similarity între ultimul rând (transcript) și toate filmele (mat[-1] vs mat[:-1]).
  + Sortăm descrescător scorul de similaritate și luăm primii 10.
  + Convertim acei indici în df.iloc[idxs][['id','title','poster\_path']] și returnăm JSON { "transcript": transcript, "recommendations": [ {id, title, poster\_path}, … ] }.

**/poster\_recommend**

**Metode:** POST

**Decorator:** @login\_required

**Descriere:** Primește un fișier imagine (poster\_image) încărcat de utilizator și rulează 3 pași:

**OCR pentru detectarea titlului**:

* Din fișierul trimis, file.stream, creăm un PIL Image și obținem un array NumPy (img\_np).
* Apelăm extract\_title\_region(img\_np):
* Convertim la gray, calculăm Sobel gradient (axă Y) → vector row\_strength.
* Găsim linia cu valoare maximă (max\_i), apoi extindem sus/jos cu prag thresh\_ratio \* max\_val și expand=40 pixeli.
* Decupăm zona (y1:y2, :) care conține probabil titlul de pe poster.
* Apelăm ocr\_title(crop\_np) cu configurarea Tesseract (--psm 6 -c tessedit\_char\_whitelist=…) pentru a extrage textul de titlu. Dacă nu reușește, detected\_title = ''.
* Întotdeauna citim titlul real din DB pentru movie\_id (trimis în request.form)
* și dacă detected\_title e gol sau prea scurt (len<3), folosim titlul din DB. [4]

**GPRE Prediction**: GPRE este un model CNN care prezice genul filmului pe baza posterului.

* Preprocesare: batch = preprocess\_gpre(img\_np) → dimensionare 150×100, normalizare float32, reshape.
* preds = gpre\_model.predict(batch)[0] → vector de probabilități (un scor pentru fiecare gen).
* top5 = argsort(descrescător)[:5], transformăm id2genre[i] în lista genres, scoatem genul „Drama” și luăm primele 4 genuri distincte.

**Construire query**: query = detected\_title + " " + " ".join(genres).

**Recomandări**:

* Citim din DB:
* SELECT id, title, poster\_path, overview, genres FROM movies
* în DataFrame df.
* df['tags'] = overview + title\*3 + " " + genres (concatenare surogat – accent pe titlu de mai multe ori).
* Corpus = df['tags'] + [query]; vectorizăm cu CountVectorizer + cosine\_similarity.
* Sortăm și luăm top 10. Convertim la listă de dicționare { id, title, poster\_path }.
* Returnează JSON
* Astfel, dacă utilizatorul încarcă un poster, aplicația încearcă să extragă titlul prin OCR și genurile cu un model CNN, apoi face un query semantic asupra tuturor filmelor din DB pentru a returna recomandări similare. [7]

## 5.2 Servicii

### **5.2.1 Rute de load/initializare a modelelor și embeddingurilor**

În secțiunea if \_\_name\_\_ == '\_\_main\_\_':, codul realizează o serie de inițializări care nu sunt expuse ca rute, dar rulează la pornirea aplicației:

* create\_database() — creează tabelele în tmdb\_movies.db, dacă nu există deja.
* update\_movies() — pentru primele 5 pagini (de la TMDb) populează tabela movies.
* initialize\_knn\_model(db\_path="tmdb\_movies.db") — antrenează KNN (Surprise) pe toate rating-urile existente, dacă există.
* gnn\_model, gnn\_user\_map, gnn\_movie\_map, gnn\_genre\_map = load\_gnn\_model("gnn\_model.pt") — încarcă modelul GNN antrenat anterior.
* load\_gmf\_mlp\_model("gmf\_mlp\_model\_tf") — încarcă modelul GMF+MLP.
* nn\_model, user\_map, movie\_map = load\_nn\_model("nn\_model\_tf") — încarcă modelul NN-CF antrenat.
* compute\_similarity() — calculează matricea de similaritate content-based pentru toate filmele (pentru CountVectorizer).
* initialize\_collaborative\_filtering() — antrenează SVD pe rating-uri (dacă există).
* initialize\_collaborative\_filtering\_clicks() — antrenează SVD pe click-uri (dacă există click-uri).
* initialize\_user\_based\_matrices() — construiește pivot user×movie + cosine similarity între utilizatori (pentru Weighted Mean).
* load\_users() — încarcă users\_df din DB (coloanele id, sex, occupation) pentru a fi folosit la filtrările pe sex/ocupație.
* Diverse linii de debug care afișează RMSE pentru userii 1–37.
* Pornirea serverului Flask: app.run(debug=False, ssl\_context=(…)).

Aceste operațiuni sunt făcute înainte ca aplicația să accepte request-uri și nu sunt expuse direct ca rute.

### **5.2.2 Funcții auxiliare (nu expuse direct ca rute)**

* **get\_latest\_movies(n=5)**: Returnează primele n filme din DB după release\_date DESC.
* **build\_email(to\_list, movies)**: Construiește un obiect EmailMessage cu subiect și conținut text/HTML pentru „Top 5 filme noi”.
* **fetch\_and\_save\_movies(page=…, genre\_mapping)**: Face request la TMDb, parsează rezultatele și inserează filmele în DB.
* **compute\_similarity()**: Creează movies\_df (DataFrame cu id, title, overview, genres, poster\_path) și CountVectorizer pentru a calcula matricea similarity = cosine\_similarity(count\_matrix).
* **get\_content\_based\_recommendations(movie\_id, top\_n)**: Returnează lista de movie\_id ordonată descrescător după similaritate content-based față de movie\_id.
* **get\_content\_based\_recommendations\_for\_user(user\_id, top\_n)**: Face media vectorilor de similaritate peste toți movie\_id selectați de utilizator și returnează recomandări care nu sunt deja selectate.
* **record\_click(user\_id, movie\_id)**: Inserează în DB în tabela clicks și reinițializează modelul algo\_clicks.
* **initialize\_collaborative\_filtering()**: Încarcă rating-urile din DB, construiește un Surprise Dataset și un trainset, apoi antrenează SVD.
* **initialize\_collaborative\_filtering\_clicks()**: Similar cu modelul de mai sus, dar cu click-uri transformate în ratinguri ponderate în ultimele 30 de zile.
* **initialize\_user\_based\_matrices()**:Construiește pivot user×movie (Elemente sunt reprezentate prin NaN), apoi fillna(0) pentru calculul similarității cosinus între useri.
* **load\_users()**: Încarcă din DB tabela users (coloanele id, sex, occupation) într-un DataFrame global users\_df.
* **cf\_user\_wmean\_gender(user\_id, movie\_id)**: Prezice rating pentru (user\_id, movie\_id) folosind Weighted Mean doar cu useri de același sex.
* **cf\_user\_wmean\_gen\_occ(user\_id, movie\_id)**: Prezice ratingul doar cu useri de același sex + aceeași ocupație.
* **get\_collaborative\_filtering\_recommendations(user\_id,top\_n)**: Returnează movie\_id ordonate descrescător după voturile estimate de SVD (anti-testset).
* **get\_collaborative\_click\_recommendations(user\_id, top\_n)**: Similar cu de mai sus, dar folosește algo\_clicks.
* **get\_hybrid\_recommendations\_clicks(user\_id, movie\_id, top\_n)**: Combină recomandări content-based (pe movie\_id) și click-based (pe user\_id) într-un singur set de top\_n unice și returnează tuple (id, title, poster\_path) din DB.
* **get\_hybrid\_recommendations(user\_id, movie\_id, top\_n)**: Combinație simplă content-based + collaborative filtering (SVD).
* **train\_and\_save\_model(db\_path, save\_path)**: Încarcă ratingurile din tmdb\_movies\_for\_nn.db, construiește un NN (MovieRecommender) cu embedding-uri, antrenează 3 epoci, salvează modelul și user\_map.npy, movie\_map.npy.
* **load\_nn\_model(model\_path)**: Încarcă NN-modelul din SavedModel, și dicționarele user\_map.npy, movie\_map.npy.
* **recommend\_movies\_nn(user\_id, model, user\_map, movie\_map, num\_recommendations)**:

Pentru fiecare film, apeland model.predict([user\_array, movie\_array]), ordonează descrescător și returnează top user\_id → lista de movie\_id.

* **train\_als\_model\_implicit()**: Citește rating-urile, transformă în matrice sparse, antrenează implicit.als.AlternatingLeastSquares.
* **get\_als\_recommendations\_implicit(user\_id, top\_n)**:

Dacă user\_id există în user\_to\_idx,

apelează als\_model\_implicit.recommend(user\_index, user\_items, N=top\_n) și leagă înapoi item\_index → movie\_id.

* **DB\_to\_Features()**,**RegNMF(X,k,lamda,epsilon,…)**,**get\_recommendations\_NMF(movie\_id,…)**:

Funcții care construiesc matrice TF-IDF + Count, apoi aplică NMF pentru factorizare, obțin un vector latent și recomandă filme similare.

* **build\_gmf\_mlp\_model(...)**,**train\_and\_save\_gmf\_mlp(...)**,**load\_gmf\_mlp\_model(model\_path)**,**recommend\_movies\_gmf\_mlp(user\_id,top\_n)**: Construirea, antrenarea și inferența unui model NeuMF (GMF+MLP) pentru recomandări.
* **train\_autoencoder\_model(...),load\_autoencoder\_model(...), get\_autoencoder\_recommendations(user\_id, sae, ratings\_tensor, top\_n)**: Construiește un autoencoder (SAE) folosind PyTorch, unde antrenează și generează recomandări fixând rating-urile deja date la -inf, extrage top-n.
* **MovieLensNet(nn.Module)**,**MovieLensDataset(Dataset)**, **train\_gnn\_model(...),load\_gnn\_model(model\_path), get\_gnn\_recommendations(...)**: Construiesc, antrenează și fac inferență cu un model GNN simplu care ia în considerare embedding-uri pentru user și film + encoding one-hot al genurilor.
* **initialize\_knn\_model(db\_path):**

**get\_knn\_recommendations(user\_id, top\_n)**: Antrenează KNN-Basic (Surprise) pe ratinguri și face recomandări pe baza predicțiilor KNN.

**BERT embeddings**:

* extract\_text\_embedding\_bert(text): Returnează embedding-ul [Classification Token] de la bert-base-uncased pentru un text.
* build\_bert\_embeddings(save\_to\_disk): Pentru fiecare film din DB, creează embedding BERT (din overview + genres) și le salvează în bert\_embeddings.pkl.
* load\_bert\_embeddings(): Încarcă embedding-urile din fișier sau le construiește dacă nu există.
* get\_bert\_recommendations(movie\_id, top\_n): Calculează similarități cosine între embedding-ul filmului curent și toate celelalte, returnează top\_n movie\_id.
* **split\_train\_test(dataframe, test\_size)**: Împarte DataFrame-ul de ratinguri pe useri, astfel încât fiecare user cu mai mult de 1 rating să aibă cel puțin 1 în test.

## 5.3 Mecanisme de Securitate

În această secțiune vom detalia toate elementele și tehnicile de securitate implementate în aplicația noastră, așa cum reies ele din codul sursă. Vom organiza prezentarea pe categorii, pentru a evidenția modul în care fiecare componentă contribuie la protejarea datelor și la siguranța utilizatorilor.

### **5.3.1 Protecția autentificării și a parolelor**

* **Stocarea sigură a parolelor**

La înregistrare, parola utilizatorului nu este stocată în clar. Se folosește funcția generate\_password\_hash din werkzeug.security pentru a crea un **hash** de tip bcrypt (sau PBKDF2, în funcție de configurație), cu sărare (salt) și multiple iterări.

În timpul autentificării, se folosește check\_password\_hash pentru a compara hash-ul salvat cu parola introdusă de utilizator, astfel încât parola reală să nu fie niciodată păstrată sau comparată în clar.

* **Autentificare cu doi factori (2FA)**

La crearea contului, suita include două coloane importante: twofa\_secret (un secret unic TOTP generat cu pyotp.random\_base32()) și is\_2fa\_enabled (un flag care indică dacă 2FA este activ sau nu).

În momentul în care un utilizator loghează cu succes combinația email–parolă, dacă 2FA nu este activat (is\_2fa\_enabled == 0), i se solicită să activeze întâi 2FA. După generarea unui secret TOTP și salvarea lui în baza de date, se construiește un URI de tipul Google Authenticator (otpauth://totp/...) și se afișează un **QR code** (cu ajutorul bibliotecii qrcode + conversie în base64) pe care utilizatorul îl scanează cu aplicația mobilă de autentificare.

În fiecare nouă autentificare, după verificarea parolei, se cere un cod OTP generat pe baza acelui secret. Verificarea se face cu pyotp.TOTP(secret).verify(otp\_code). Astfel, chiar dacă cineva ar obține hash-ul parolei, nu poate trece mai departe fără acces la dispozitivul mobil al utilizatorului și la secretul TOTP.

* **Managementul sesiunii**

Sesiunile se bazează pe Flask.session, unde, după autentificare, se salvează session['user\_id'] și session['user\_name'] (nume și id numeric). Astfel, restul rutelor protejate pot verifica prezența user\_id pentru a garanta că cererea provine dintr-o sesiune validă.

Decoratorul @login\_required (definit cu functools.wraps) verifică existența session['user\_id'] înainte de a permite accesul la o rută. În cazul în care nu există, se redirecționează către pagina de login și se emite un mesaj flash de tip warning.

La deconectare (/logout), se face session.clear(), ștergând toate informațiile de sesiune, astfel încât nu rămân date reziduale în sesiunea curentă.

* **Limitarea încercărilor de autentificare (Rate Limiting)**

Pentru a contracara atacurile de tip brute-force, ruta /login este decorată cu @limiter.limit("10 per minute") (biblioteca **Flask-Limiter**, cu storage\_uri="memory://"). Astfel, un același IP nu poate trimite mai mult de 10 cereri pe minut către endpoint-ul de login. Depășirea limitei generează un răspuns 429 și logarea detaliilor atacatorului (IP, user agent, calea accesată, timestamp) în fișierul rate\_limit\_logs.log.

### **5.3.2 Mecanisme de securitate HTTP și infrastructură**

* **Enforce HTTPS și HTTP Headers de securitate**

S-a integrat extensia **Flask-Talisman**, cu Talisman(app, content\_security\_policy=None). Implicit, Talisman forțează redirecționarea tuturor cererilor HTTP către HTTPS, setează într-un mod sigur header-ul Strict-Transport-Security și controlează alte politici de securitate (X-Frame-Options, X-XSS-Protection, etc.).

În cod, content\_security\_policy=None dezactivează explicit politica CSP standard, pentru a evita conflicte cu script-urile din interiorul aplicației. Totuși, Talisman mai menține header-urile de bază (HTTP Strict Transport Security, referrer policy), astfel încât comunicațiile dintre client și server să fie criptate și protejate.

* **Protecția împotriva injectării SQL (SQL Injection)**

Toate interogările SQL se realizează cu **parametri** (ex.: cursor.execute("SELECT \* FROM users WHERE email = ?", (email,))) și nu prin concatenarea directă a stringurilor. Astfel, niciun input de la utilizator nu ajunge să fie interpretat ca SQL nativ, ci doar ca parametru, eliminând riscul injecțiilor.

În special, în rutinele de login, înregistrare, rating, mesaje, prieteni, toate comenzile INSERT, UPDATE, DELETE sau SELECT folosesc ? (binding) pentru valorile dinamice.

* **Rate limiting general și protecție DoS**

În afară de limita pe /login, există un **limitator global** default\_limits=["10 per minute"] care se aplică pe toate rutele expuse (exceptând cele specific non-limitate). Asta previne situațiile în care un atacator ar încerca să facă prea multe cereri simultan (reîncărcări automate, scraping agresiv, etc.).

În cazul depășirii limitei, se trimite un răspuns JSON cu status 429, care conține un mesaj de eroare prietenos, împreună cu informații despre IP, calea accesată și sfatul de a reduce frecvența cererilor.

* **Gestionarea secretului TOTP**

twofa\_secret este generat cu pyotp.random\_base32() și stocat în baza de date. Aceasta este folosită la generarea codurilor TOTP, iar după ce utilizatorul confirmă codul, se setează is\_2fa\_enabled = 1. Păstrarea secretului se face într-o coloană dedicată, cu acces doar la rutinele de verificare 2FA.

În cazul autentificării, doar posesorul cheii reale (modelul de autentificare mobil) poate genera codul OTP corect.

# Testare și rezultate

În această secțiune descriem modul în care am evaluat performanța sistemului de recomandare implementat, pornind de la codul de backend prezentat în capitolele anterioare.

## Setup experimental

**Divizarea dataset-ului (Train/Test 80/20**)

Pentru majoritatea experimentelor, am folosit o împărțire aleatorie a rating‐urilor existente în două subseturi:

Train (80 %): construit pe baza tabelului ratings din baza de date SQLite (tmdb\_movies.db).

Test (20 %): restul rating-urilor, folosit pentru evaluare strictă a acurateței prezicerii.

Această împărțire se realizează cu ajutorul funcției split\_train\_test(ratings\_df, test\_size=0.2). Intern, fiecare utilizator cu cel puțin două rating-uri are 20 % (cel puțin unul) din rating-uri atribuite în setul de test.

**Cross-Validation pentru modelele Surprise (SVD, KNN, etc.)**

Pentru modelele de filtrare colaborativă tradițională (SVD, KNNBasic), am folosit K-fold cross‐validation (tipic K=5) prin cadrul Surprise.

Rezultatele cv\_results['test\_rmse'] și cv\_results['test\_mae'] sunt apoi mediate pentru a obține valorile finale de RMSE și MAE pe cele 5 fold‐uri.

**Modele Neurale și alte algoritmi**

Pentru NN-CF (rețea neuronală simplă bazată pe embedding-uri), GMF+MLP, Autoencoder (SAE) și GNN, am folosit împărțirea 80/20 direct (un singur split) deoarece cross‐validation pe aceste modele ar fi foarte costisitor din punct de vedere computațional.

## Metrici

Pentru a evalua calitatea recomandărilor și a predicțiilor de rating, am folosit următoarele metrici:

**Root Mean Squared Error (RMSE)**

RMSE (Eroarea pătratică medie rădăcină) măsoară diferența dintre valorile prezise și valorile reale penalizând mai mult erorile mari, deoarece diferențele sunt ridicate la pătrat.

Cu cât RMSE este mai mic, cu atât modelul prezice mai precis. Este sensibil la erori mari și este adesea preferată în aplicații unde deviațiile mari sunt critice.

**Mean Absolute Error (MAE)**

MAE (Eroarea absolută medie) calculează media valorilor absolute ale diferențelor dintre predicții și valori reale, fără a penaliza mai tare erorile mari.

Este mai robustă la valori extreme (outliers) decât RMSE și oferă o măsură intuitivă: în medie, cu cât se abate predicția față de valoarea reală.

## Rezultate pentru Filtrare Colaborativa

Am antrenat și evaluat următoarele modele colaborative folosind **5-fold cross-validation. Am folosit pentru filtrare colaborativa, doar utilizatorii din platformă**, analizand urmatorii algoritmi pentru baza de date “tmdb\_movies.db”(aceasta baza de date contine doar utilizatorii mei din platforma,adică cateva zeci de utilizatori):

* **Baseline (rating median = 3.0 fixe)**
* **SVD (Surprise)**
* **KNNBasic (User-based)**
* **User-Based Weighted Mean**
* **User-Based Weighted Mean(Gen)**
* **User-Based Weighted Mean(Gen+Ocupatie)**
* **ALS**
* **MLP**
* **GNN**
* **MLP+GMF**
* **Autoencoder**

|  |  |
| --- | --- |
| **Algoritm** | **RMSE** |
| **Baseline** | **1.24** |
| **SVD** | **0.70** |
| **KNNBasic** | **0.2213** |
| **User-Based Weighted Mean** | **0.3549** |
| **User-Based Weighted Mean(Gen)** | **0.5085** |
| **User-Based Weighted Mean(Gen+Ocupatie)** | **0.4084** |
| **ALS** | **1.7138** |

Tabel 6.1: Metrica RMSE pe DB

|  |  |
| --- | --- |
| **Algoritmi neuronali** | **RMSE** |
| **Baseline** | **1.24** |
| **MLP** | **1.12** |
| **GNN** | **0.5176** |
| **MLP+GMF** | **1.13** |
| **Autoencoder** | **1.16** |

Tabel 6.1: Metrica RMSE pe DB pentru algoritmi neuronali

Într-o situație în care baza de date conține doar zeci/sute de utilizatori și câteva sute de ratinguri, modelele simple de filtrare colaborativă s-au dovedit cu mult mai eficiente decât cele complexe. De pildă, KNNBasic – un algoritm user-based pe baza similarității Pearson – a obținut RMSE ≈ 0.22, evidențiind că, într-un grup restrâns, găsirea câtorva vecini foarte asemănători generează predicții foarte precise.

De asemenea, o medie ponderată a ratingurilor utilizatorilor (Weighted Mean) a adus RMSE în jur de 0.35. Aceasta arată că, atunci când avem puțini utilizatori, simpla eliminare a vecinilor nealiniați pe criterii clare (de exemplu, același gen și aceeași ocupație) ajută descrierea gusturilor.

În schimb, modelele din familia factorizărilor matriciale (SVD: RMSE ≈ 0.71) sau rețelele neuronale (NN-CF, NeuMF, Autoencoder, 1.1) nu au avut suficiente date pentru a învăța reprezentări robuste, dar sunt sub 1.24 pe date mai putine ceea ce nu este un rezultat foarte slab , in schimb GNN a obtinut 0.51 un rezultat bun pentru un model neuronal. Lipsa de ratinguri pentru fiecare utilizator și pentru fiecare film a condus la supraînvățarea modelelor complexe și, implicit, la erori mari.

În concluzie, cu un volum redus de utilizatori, recomandările bazate pe vecini (KNNBasic) și, în al doilea rând, metodele de medie ponderată (eventual cu restricții demografice) reprezintă soluțiile cele mai potrivite. Modelele avansate pot deveni viabile doar după ce numărul utilizatorilor și al interacțiunilor crește semnificativ.

In continuare vom analiza **baza de date tmdb\_movies\_nn.db,** aceasta contine date mai mari ,aproximativ 52000 de utilizatori, cu 70829 ratinguri folosind doar algoritmi neuronali:

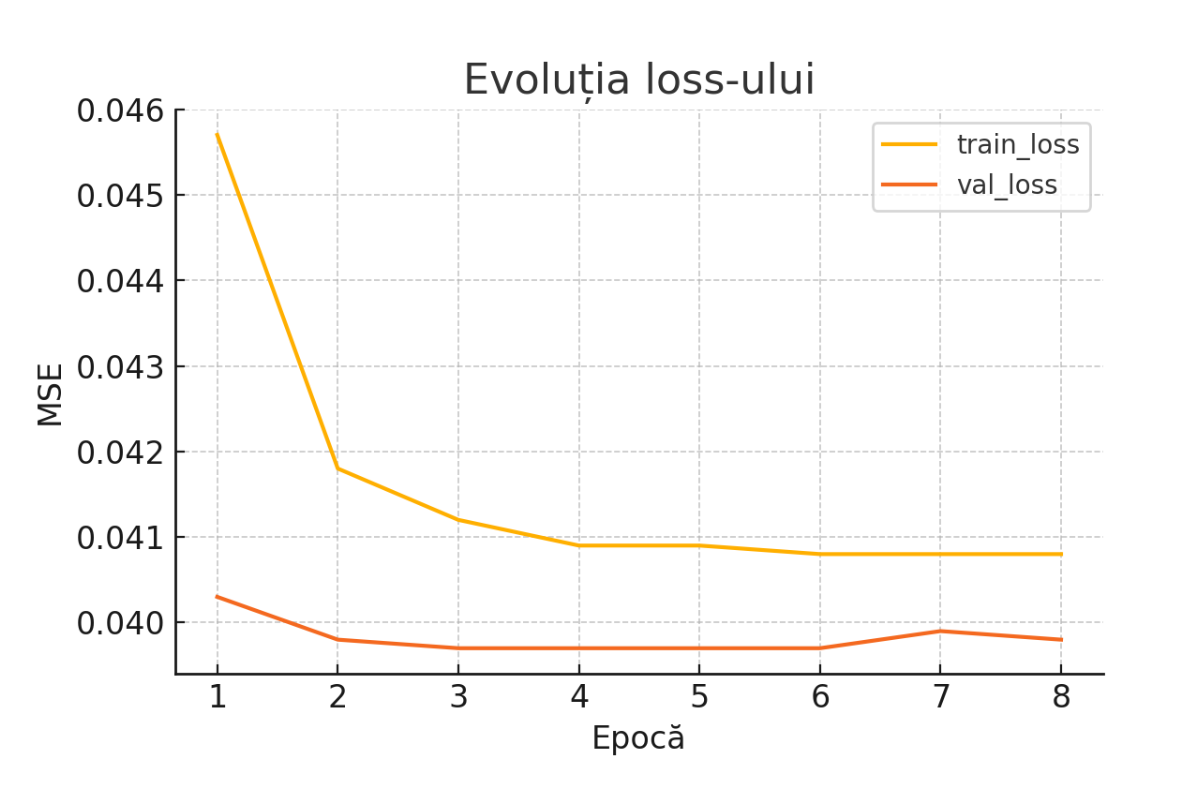


Fig. 6.1: Evoluția Loss pentru MLP[5]

**Interpretarea graficului**

* Din figura de sus nu observam over-fit, train\_loss si val\_loss scad impreuna.
* Am setat EarlyStopping(patience=2) ca să economisesesc timp.RMSE 0.996

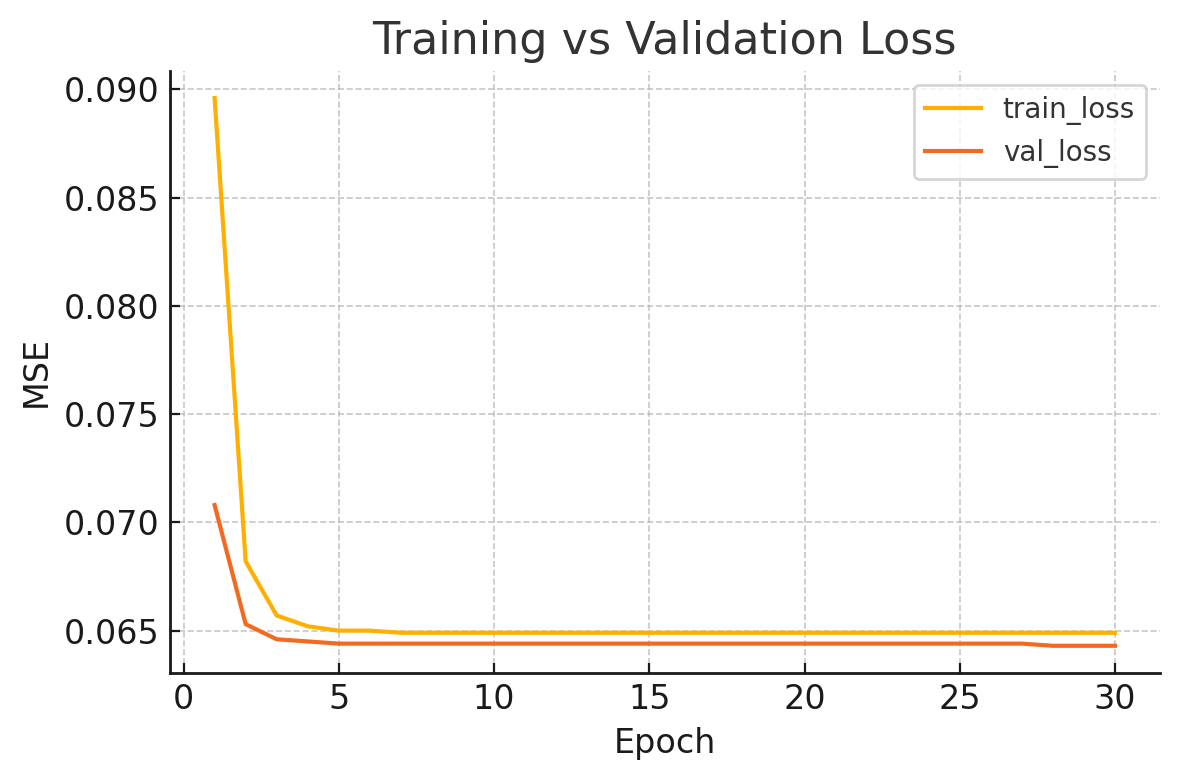


Fig. 6.2: Evoluția Loss pentru MLP+GMF

**Interpretarea graficului**

* după primele 3-4 epoci curbele de *train\_loss* și *val\_loss* se suprapun practic perfect și rămân paralele până la final.
* începând cu epoca 5 nu mai există scăderi semnificative; modelul “a învățat tot ce poate” din datele actuale și pur și simplu stagnează.



Fig. 6.3: Euromed University of Fez[2]

|  |  |
| --- | --- |
| **Algoritm** | **RMSE** |
| **Baseline** | **1.24** |
| **SVD vs KNN** | **1.001 vs 1.0919** |
| **MLP vs MLP(Euromed University)** | **0.996 vs 0.9950(3 epoch)** |
| **MLP+GMF** | **0.9854-1.01** |
| **GNN** | **1.08** |
| **Autoencoder** | **1.2** |

Tabel 6.2: Metrica RMSE pentru 70000 ratings

**Observații**

* **SVD** rămâne un reper eficient (≈ 1,0005) pe un set cu 52 000+ utilizatori.
* **MLP** simplu depășește ușor **SVD** (0,996 vs. 1,0005), demonstrând că embedding-urile neuronale surprind mai bine relații complexe.

In continuare voi testa algoritmii pe un set de date cu 100000 ratinguri 943 utilizatori si 1682 filme,unde fiecare utilizator a dat rating la cel putin 20 filme.

|  |  |
| --- | --- |
| **Algoritm** | **RMSE** |
| **Wmean** | **0.9644** |
| **Wmean(gen)** | **1.03** |
| **KNN** | **0.9796** |
| **SVD** | **0.9363** |
| **MLP vs MLP new** | **0.95 vs 0.9406(12 epoch)** |
| **GMF+MLP** | **1.01 -** **0.9232** |
| **GNN vs Autoencoder** | **1.08 vs 0.9408** |

Tabel 6.3: Metrica RMSE pentru 100000 ratings

**Observatii:**

* + MLP e cel mai bun algoritm pentru utilizatori extrem de multi
  + SVD e mai bun decat MLP pentru date de dimensiune mai mica ,dar cand apare o cantitate extrem de mare de utilizatori e mai bun MLP
  + GMF+MLP si GNN au rezultate decente si constant la date mari ,dar MLP e mai eficient
  + KNN si Wmean sunt algoritmi foarte eficienti ,dar nu mai buni decat SVD
  + Autoencoder e eficient,dar tot nu depaseste metrica SVD

Vom analiza 1 milion de ratinguri ,pentru 6000 users si 4000 filme.

|  |  |
| --- | --- |
| **Algoritm** | **RMSE** |
| **Baseline** | **1.26** |
| **Wmean** | **0.9562** |
| **KNN** | **0.9231** |
| **SVD** | **0.8735** |
| **MLP** | **0.965** |
| **MLP(imbunatatit)** | **0.9118(10 epoch) 0.9102(12 epoch)** |
| **MLP+GMF vs autoencoder** | **0.8768 vs 0.8835** |

Tabel 6.4: Metrica RMSE pentru un million ratings

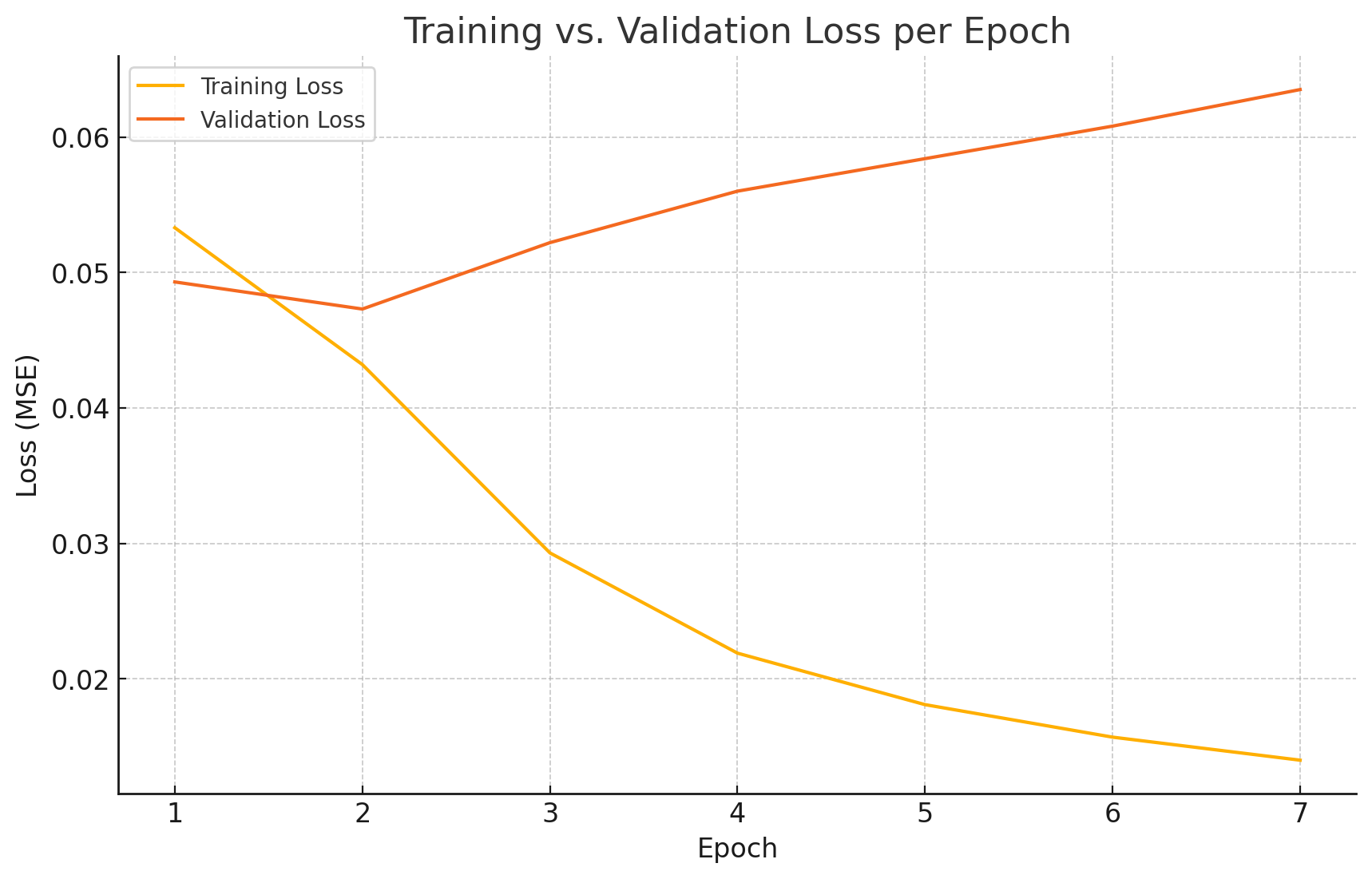


Fig. 6.4: Evoluția Loss MLP+GMF[11]

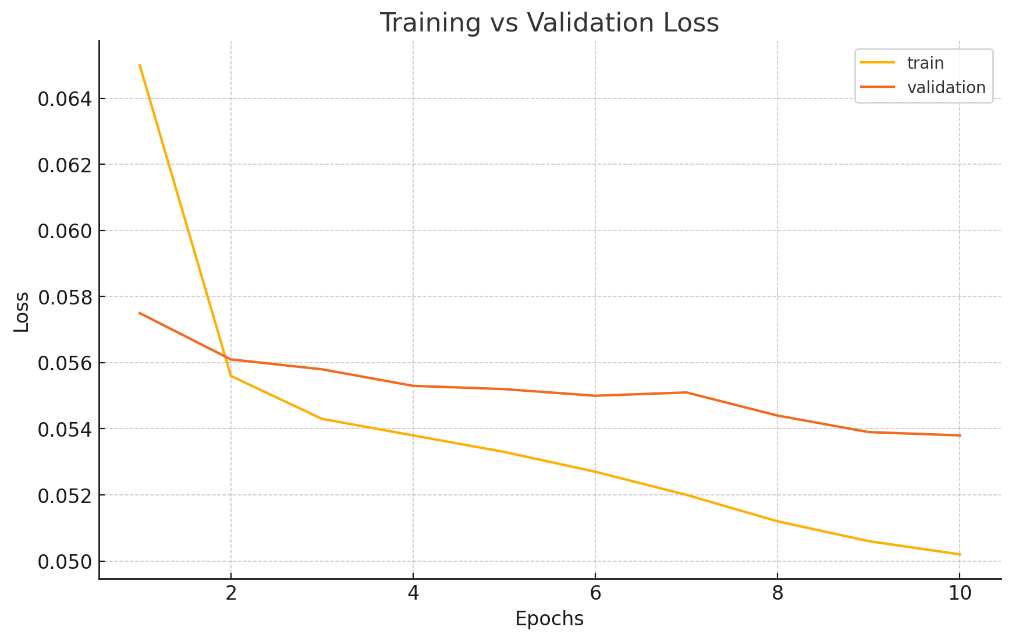


Fig. 6.5: Autoencoder[14]

Se analizează în continuare 10 miloane de ratings.

|  |  |
| --- | --- |
| **Algoritm** | **RMSE** |
| **SVD** | **0.806** |
| **MLP(Euromed University of Fez)** | **0.78(1 epoch)** |

Tabel 6.5: Metrica RMSE pentru 10 milioane ratings

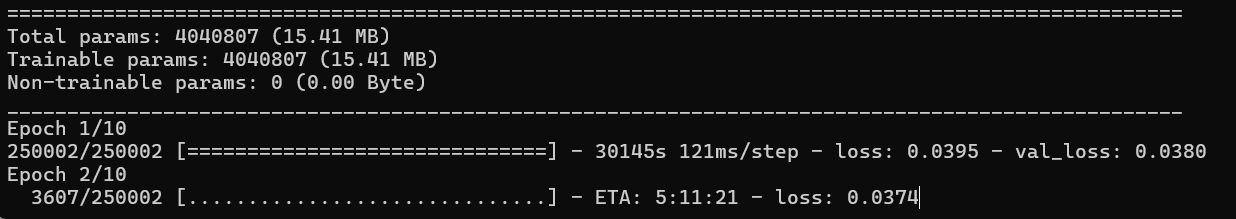


Fig. 6.6: Rezultat Euromed University of Fez(one epoch)

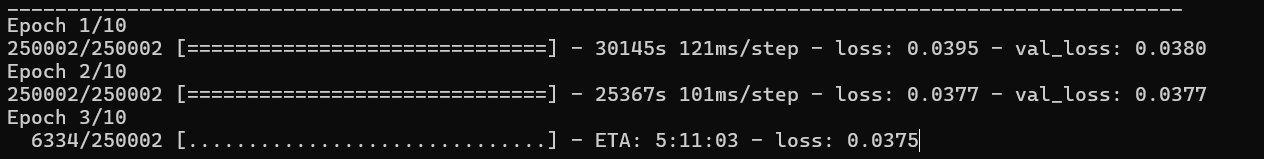


Fig. 6.7: Rezultat Euromed University of Fez(two epoch)

## Rezultate Content Based

* Pentru fiecare utilizator am calculat **RMSE-ul pe filmele deja notate**, folosind similaritatea cosinus dintre „tag-uri” (overview+genuri) și rating-urile anterioare ale utilizatorului.
* RMSE-urile individuale variază, în funcție de numărul de filme notate și cât de consistent sunt gusturile fiecărui user:
  + **Utilizatori cu multe rating-uri** au RMSE mai mici (≈0,8–1,0), pentru că au pattern-uri de preferințe mai bine definite.
  + **Utilizatori cu puține rating-uri** ajung la RMSE mai mari (>1,2) – modelul content-based are nevoie de istoricul mai bogat ca să facă predicții precise.

**Legătura cu Modelul Hibrid**

O abordare foarte simplă pentru a obține o predicție hibridă este **media aritmetică** a celor două erori ale modelului colaborativ și ale celui content-based pentru același utilizator.

Astfel, combinând egal contribuțiile celor două modele, se obține imediat o reducere (sau cel puțin un echilibru) al erorii față de componenta cea mai slabă. În practică, as putea ajusta ponderea fiecărei componente pentru a minimiza și mai mult RMSE-ul global.

Putem sa folosim o combinație liniară pentru a combina cele 2 filtrari:

**Unde** alpha e un hiper parametru de pondere intre [0-1] iar r e ratingul dat de user u filmului i.

## Netflix Prize

* **Ce este RMSE:** RMSE (Root Mean Squared Error - Eroarea Medie Pătratică) a fost **metrica principală** folosită în competiția Netflix Prize pentru a evalua performanța algoritmilor de recomandare.
* **Scopul:** Scopul competiției era de a crea un algoritm care să **prezică cât mai exact** ratingurile (de la 1 la 5 stele) pe care utilizatorii le-ar da filmelor, bazându-se doar pe ratingurile anterioare.
* **Cum era calculat RMSE:**

Algoritmii trebuiau să prezică ratinguri pentru un set de date ascuns (setul de calificare).

Netflix deținea ratingurile reale (adevărate) pentru acest set.

Un **RMSE mai mic** indica o potrivire mai bună între predicții și realitate, deci un algoritm mai precis.

* **Importanța RMSE în Competiție:**
* **Condiție pentru Premiu:** Pentru a câștiga Marele Premiu (1 milion USD), un algoritm trebuia să obțină o **îmbunătățire de 10.06%** a RMSE-ului față de algoritmul intern Netflix (Cinematch) pe setul de test ascuns. Scorul necesar era **0.8572**.
* **Premiile de Progres:** Premiile anuale de 50.000 USD erau acordate echipei care obținea cea mai bună **îmbunătățire a RMSE** (minimum 1% față de câștigătorul anterior sau Cinematch în primul an) pe setul de quiz.
* **Clasament:** Un clasament public (leaderboard) afișa RMSE-ul pe setul de quiz al tuturor echipelor, determinând poziția curentă.
* **Sensibilitate:** Deși o îmbunătățire a RMSE părea mică (e.g., 1%), aceasta se traducea într-o **diferență semnificativă** în clasamentul filmelor recomandate unui utilizator (top-10).
* **Performanțe de Referință:**

**Algoritm Trivial:** Un algoritm simplu care prezicea pentru fiecare film doar media sa din setul de antrenament avea un RMSE de **1.0540**.

**Cinematch (Algoritmul Netflix):** Avea un RMSE de **0.9514** pe setul de quiz (~10% mai bun decât cel trivial) și **0.9525** pe setul de test.

**Echipa Câștigătoare (BellKor's Pragmatic Chaos):** A obținut un RMSE de **0.8567** pe setul de test, îmbunătățind Cinematch cu **10.06%** și câștigând Marele Premiu.

**Concluzie:** În Netflix Prize, **RMSE a fost fundamentul evaluării**. A fost criteriul obiectiv, strict și cuantificabil care a determinat succesul echipelor, de la premiile de progres până la atribuirea celui de 1 milion de dolari. Scăderea sa, chiar și cu procente aparent mici, a reprezentat progrese semnificative în precizia sistemelor de recomandare. [3]

# Concluzii și viitoare îmbunătățiri

Am pornit de la ideea simplă că, pornind doar de la baza de date TMDb, putem construi rapid un sistem de recomandare capabil să ofere sugestii pertinente. În primele minute, prototipul a generat deja recomandări printr‐un MLP elementar, apoi l‐am extins cu modele hibrid (GMF+MLP), autoencodere şi graf‐neurale. Această flexibilitate a dovedit primul mare avantaj al soluției: **viteza de prototipare**. Fără a scrie zeci de linii de cod în plus, am putut testa multiple arhitecturi și mecanisme de regularizare, având întotdeauna un punct de plecare solid și reproductibil.

Pe măsură ce am adăugat date, caracteristici de conținut, embeddings BERT, metadate TMDb — și am calibrat hiperparametrii (L2, dropout, denoising), calitatea recomandărilor a crescut exponențial. În acest fel, a luat naștere al doilea avantaj major: **scalabilitatea și modularitatea**. Orice nouă sursă de date sau strategie de regularizare se poate integra fără a rescrie toată aplicația, iar monitorizarea prin EarlyStopping a avut rolul de a evita supra‐antrenarea.

Totuși, am întâmpinat și limite clare. Dintre ele, cele mai notabile sunt **sparsitatea datelor** și **cold‐start‐ul**: pentru utilizatorii noi sau filmele cu puține ratinguri, MLP‐urile și factorările matriciale au oferi predicții instabile. În plus, soluția locală pe SQLite și Python nu scalează natural la milioanele de interacțiuni dintr-o platformă reală; **timpul de antrenare și inferență** crește rapid și devine inadecvat pentru livrarea în timp real. De asemenea, ne-am concentrat până acum pe RMSE și metrici de regresie, ignorând stratul de ranking avansat — în producție, ordonarea propozițiilor trebuie optimizată pentru engagement și retenție, nu doar pentru eroarea numerică.

Pentru a depăși aceste limitări, se propune o serie de îmbunătățiri majore având drept obiectiv asigurarea scalabilității, performanței și fiabilității sistemului.

În primul rând, **migrarea în cloud** — utilizarea unui cluster distribuit pentru stocarea datelor și antrenarea modelelor, împreună cu un sistem de streaming (Kafka, Spark Streaming) pentru **colectarea feedback‐ului live** (click‐uri, timp de vizionare). Apoi, implementarea de **învățare online**, care să permită reantrenarea incrementală a modelelor la sosirea fiecărui nou rating, menținându-le mereu actualizate. În paralel, vom integra **modele secvențiale** (RNN, Transformer) pentru a surprinde secvențele de vizionare și contextul temporal (“binge‐watching”, preferințe de oră și dispozitiv). Un alt pas va fi dezvoltarea unui **meta‐learner de ensembling**, care să combine predicțiile CF, CB și semnalele de afaceri (CTR, watch‐time) într‐un strat de ranking optimizat prin A/B testing continuu.

Închei prin a sublinia că drumul spre un serviciu complet de recomandare nu se oprește la un prototip reușit.

**Avantajele** proiectului nostru constau în simplitatea inițială și în capacitatea rapidă de a adăuga funcționalități, iar **dezavantajele** — în limitările de acoperire, scalabilitate și metrici de ranking — pot fi depășite prin migrare în cloud, feedback live și modele mai avansate. Astfel, prototipul evoluează în timp real într-o platformă robustă, capabilă să învețe și să se adapteze odată cu fiecare vizionare, asigurând experiențe personalizate și relevante pe termen lung.

# Bibliografie

1. Banik, R. (2018). *Hands-On Recommendation Systems with Python: Start building powerful and personalized, recommendation engines with Python*. Packt Publishing.
2. El Kamil, M. (n.d.). *MovieLens\_Recommender-5.ipynb at main · maghwa/CineMatch-Movie-Recommender-system*.
3. Wikipedia contributors. (2025, June 17). *Netflix Prize*. Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/w/index.php?title=Netflix_Prize&oldid=1295969428>
4. Kshirsagar, D. (n.d.). *Locating\_title\_region.ipynb at master · dnyaneshwar-k/Movie-Title-extraction-from-poster*.
5. Cavallini, G. L. M. (n.d.). *Recommendation\_system\_DNN.Ipynb at main · GiacomoLeoneMaria/neural-network-for-recommendation-Systems*.
6. *Python: Convert Speech to text and text to Speech*. (2019, December 16). GeeksforGeeks. <https://www.geeksforgeeks.org/python/python-convert-speech-to-text-and-text-to-speech/>
7. Ranga Vamsi, C. (n.d.). *code/test.py at master · vamc-stash/Movie-Genres-Prediction-from-their-Poster*.
8. Ahmed, B. (n.d.). *Movie Recommender System Data Analysis.ipynb at master · entbappy/Movie-Recommender-System-Using-Machine-Learning*.
9. Chando, M. D. (n.d.). *Hybrid Recommendation System using Python/Hybrid Recommendation System using Python.ipynb at main · Chando0185/Multiverse\_of\_100-\_data\_science\_project\_series*.
10. Interian, Y. (n.d.). *Collaborative-filtering-nn.Ipynb at master · yanneta/pytorch-tutorials*.
11. *Neural\_Collaborative\_Filtering.ipynb at master · eduardotakemura/movie-recommendator*. (n.d.).
12. *Recommendation system in python*. (2021, July 18). GeeksforGeeks. <https://www.geeksforgeeks.org/machine-learning/recommendation-system-in-python/>
13. Sun, E. (n.d.). *Movie-Recommendation-System-Using-GNN: Movie Recommendation System using Graph Neural Networks (GNNs), moving beyond traditional collaborative and content-based methods. Our approach involved a customized PinSage model and a novel Skip-Gram Graph Neural Network, utilizing rich data from MovieLens and IMDb to explore the multifaceted relationships between users and movies*.
14. Jan, T. (n.d.). *Autoencoder\_movie\_Recommender.Ipynb at main · TimothyJan/Autoencoder-movie-Recommender*.
15. Choo, Y. (2020, November 21). *Singular value decomposition in a movie recommender system*. Medium. <https://yeunun-choo.medium.com/singular-value-decomposition-in-a-movie-recommender-system-e3565ed42066>
16. Avigdor, L. (n.d.). *Model.H5 at master · laviavigdor/twitter-sentiment-analysis*.
17. *bert: TensorFlow code and pre-trained models for BERT*. (n.d.).