National School of Applied Sciences of Al Hoceima

2WatchM: A Movie Platform with Deep Recommendation via Retrieval and Ranking Using TensorFlow Recommenders

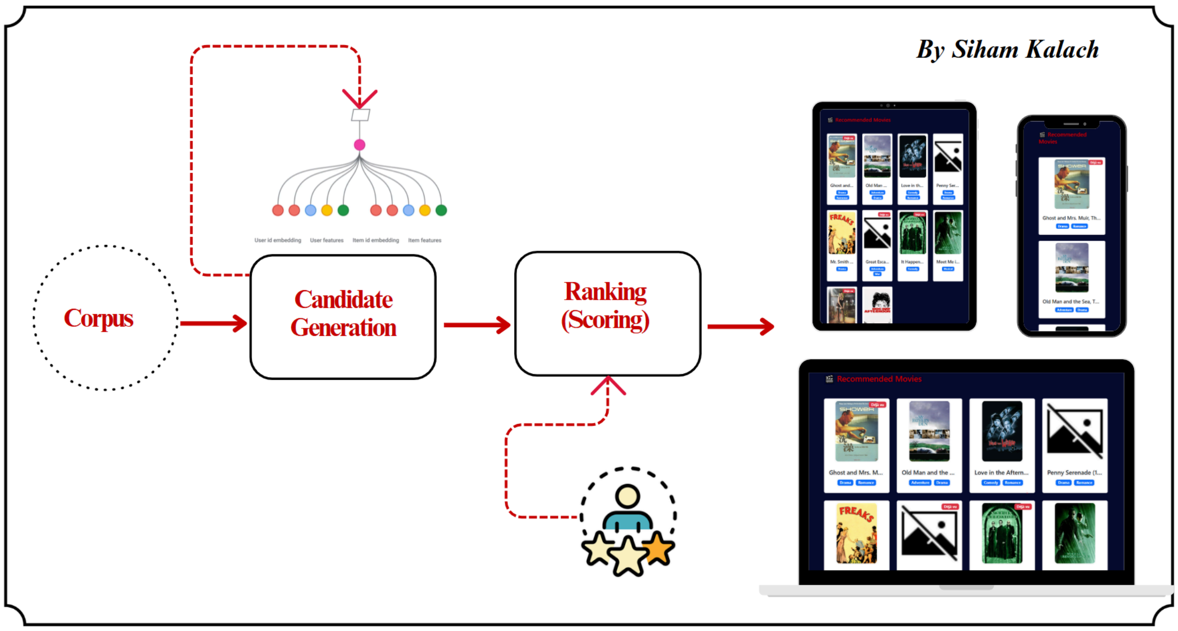


Figure 1: Two-Stage Recommendation Pipeline of 2WatchM

Prepared by: **Siham Kalach**

Academic Year: 2024–2025

1. ABSTRACT

In today’s era of overwhelming digital media, providing users with timely, relevant content is essential. This paper introduces **2WatchM**, an end-to-end movie recommendation platform that delivers personalized suggestions using a deep learning pipeline implemented with TensorFlow Recommenders. The system integrates a scalable full-stack architecture: a React frontend and a Django backend, combined with a custom-trained recommendation model.

At its core, the recommendation engine follows a two-stage architecture. The **retrieval stage** employs a Two-Tower model where user and movie IDs are projected into a shared embedding space. Candidate generation leverages both **BruteForce** and **ScaNN** indexing layers to enable scalable nearest-neighbor search, enhancing performance in dynamic environments. The **ranking stage** refines the top candidates by combining user and movie embeddings through a deep neural network trained to predict ratings, optimized using Mean Squared Error and evaluated via Root Mean Squared Error.

2WatchM also includes a data pipeline for extracting and transforming user interactions, genres, and ratings into structured datasets, which are stored and accessed through Django ORM. Recommendations are served in real-time using the **TFSMLayer**, allowing the model to be deployed seamlessly in production. A key feature is the “Already Seen” indicator, which tags previously rated movies to enhance user experience and recommendation relevance.

This work showcases the integration of modern recommendation architectures into full-stack web platforms and lays the groundwork for future enhancements, including real-time deployment using **Apache Kafka** for scalable and dynamic recommendation delivery.

2. INTRODUCTION

As the volume of digital media content continues to grow at an unprecedented pace, helping users navigate and discover relevant material has become a central challenge for online platforms. Recommendation systems have emerged as essential tools for personalizing user experiences, enabling intelligent content delivery tailored to individual preferences and interaction history. These systems are now deeply embedded in domains such as video streaming, online retail, and social platforms, where the quality of recommendations can significantly influence user engagement and platform retention.

Building recommendation systems that are not only accurate but also deployable within real-world, production-grade web environments requires addressing several engineering and modeling challenges. These include designing architectures capable of learning meaningful user-item relationships, scaling retrieval across large catalogs, and integrating machine learning workflows into web-based interfaces. While deep learning methods have shown great promise in modeling complex user-item interactions, their deployment in full-stack platforms remains a non-trivial task.

This paper introduces **2WatchM**, a web-based movie recommendation platform that demonstrates the integration of a deep learning recommendation engine within a modern, scalable web application. At the heart of the system lies a two-stage recommendation architecture built using **TensorFlow Recommenders (TFRS)**. The **retrieval model** employs a Two-Tower network structure that learns user and movie embeddings in a shared representation space, enabling candidate selection based on similarity. This stage incorporates both **BruteForce** and **ScaNN** indexing mechanisms to enable efficient large-scale retrieval. The **ranking model** further processes the retrieved candidates using deep neural layers to estimate user preferences in the form of predicted ratings.

To support the end-to-end deployment, the platform features a **Django backend** and a **React frontend**, with a custom data ingestion pipeline that processes user interactions, genres, and ratings into structured formats consumable by the models and interface. Recommendations are served in real-time using **TFSMLayer**, allowing the trained TensorFlow model to be directly queried from within the application. A user-centric feature is also introduced: movies previously rated by the user are marked with an **"Already Seen"** label to enhance transparency and personalization.

By bridging deep learning recommendation models with real-world deployment considerations, this work illustrates a practical path from model training to user-facing application. It also sets the stage for further developments, including **real-time streaming infrastructure** using tools like **Apache Kafka**, to enable dynamic, event-driven recommendation flows.

The remainder of this paper is structured as follows:

- ❖ **Section 2** describes the system architecture, including the retrieval and ranking models.
- ❖ **Section 3** details the integration of the trained models within the platform.
- ❖ **Section 4** presents evaluation results and interface features.
- ❖ **Section 5** concludes with insights and outlines directions for future work.

Keywords —

Deep Learning; Recommendation Systems; TensorFlow Recommenders; Two-Tower Architecture; Retrieval and Ranking Models; ScaNN; BruteForce Indexing; Movie Recommendation; TFSMLayer; Full-Stack Deployment; Django; React; User Embeddings; MovieLens Dataset; Web Application Integration

3. LITERATURE REVIEW

Traditional recommendation systems have historically relied on collaborative filtering (Koren et al., 2009) and content-based filtering (Lops et al., 2011). Collaborative filtering leverages user-item interaction data but faces challenges such as scalability and the cold-start problem, particularly when dealing with new users or items. In contrast, content-based filtering utilizes item metadata to generate recommendations but lacks the ability to model complex user-item interactions. While both methods have proven effective in specific contexts, they often struggle to scale and adapt to the vast and dynamic datasets characteristic of modern digital platforms.

In recent years, deep learning-based approaches have emerged as powerful alternatives capable of overcoming the limitations of traditional techniques. The Neural Collaborative Filtering model (He et al., 2017), for instance, employs deep neural networks to learn nonlinear interactions between users and items. Another influential architecture is the Two-Tower model introduced by Covington et al. (2016) at YouTube, which encodes users and items into a shared latent space independently, facilitating efficient retrieval of relevant items from extensive catalogs.

To further address the issue of scalability, indexing methods have become critical. While brute-force search is straightforward, it becomes computationally intensive as the dataset grows. Advanced techniques like ScaNN (Guo et al., 2020) offer significant improvements by enabling approximate nearest neighbor searches, drastically reducing inference time while preserving high-quality recommendations. ScaNN has thus become a standard component in real-time, large-scale recommendation systems.

Despite these advancements in algorithm design, there is a notable lack of comprehensive studies detailing the integration of deep learning recommendation models into fully functional web environments. Some research, such as Zhao et al. (2019), discusses partial deployments, but complete end-to-end implementations that encompass data extraction, model training, and real-time user interface personalization remain scarce.

Identified limitations in the literature:

- Difficulty simultaneously addressing scalability, recommendation quality, and user experience in a complete system.
- Lack of comprehensive studies covering both backend (database, API) and frontend (display, interaction) integration.

4. METHOLOY

4.1. Notation :

To facilitate clarity and precision, we define the symbols and terms used throughout this section:

Symbol	Definition

u	Scalar identifier for a user
m	Scalar identifier for a movie
eu ∈ Rd	Embedding vector representing user uuu in a latent space of dimension d
em ∈ Rd	Embedding vector representing movie mmm in the same latent space
r^um	Predicted rating of movie mmm by user uuu
rum	Ground-truth rating of movie mmm by user uuu (from dataset)
L	Loss function used to train the model

4.2.Theoretical Framework

Before detailing our method, we contextualize it within the established theoretical paradigm of **deep learning-based recommendation systems**, which typically employ a two-stage architecture:

❖ **Representation Learning :** The cornerstone of modern recommendation systems is the transformation of users and items into **embedding vectors** within a shared latent space. These embeddings encode semantic and behavioral characteristics, enabling similarity comparisons.

❖ **Dot Product Similarity :** In the retrieval phase, the relevance between a user u and a candidate item m is estimated using the **dot product eu · em** This computation is both efficient and effective for large-scale candidate generation.

❖ **Learning-to-Rank :** In the ranking phase, the system refines predictions by modeling nonlinear relationships between eu and em using a deep neural network. This allows for more expressive scoring functions than simple similarity measures.

4.3 System Overview :

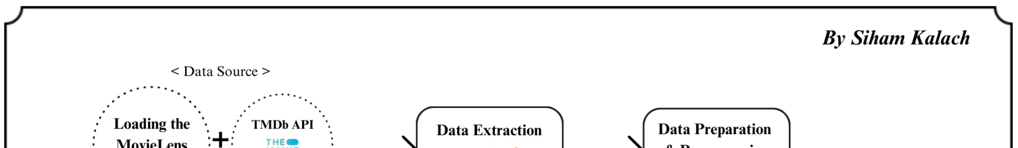


Figure 2: Pipeline Architecture for 2WatchM Movie Recommendation Platform

The proposed system follows a modular and hierarchical methodology based on a two-stage recommendation pipeline: candidate generation (retrieval) and ranking (scoring). This architecture is inspired by large-scale industry recommender systems such as those used by YouTube and Google Play.

Our platform, *2WatchM*, utilizes the MovieLens dataset, enriched with metadata obtained through the TMDb API, to deliver personalized movie recommendations for each authenticated user. The system is built as a full-stack application, comprising a Django backend with a PostgreSQL database, a React-based frontend, and TensorFlow-powered deep learning recommendation models.

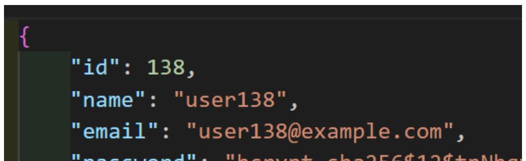
4.4 Processing Pipeline

4.4.1 Data Extraction

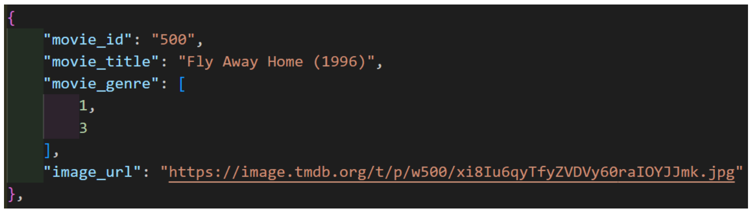
We leverage the **MovieLens 100K** dataset as our primary source of user-movie interactions, including ratings and demographic attributes. To enhance this dataset visually and semantically, we integrate metadata from The Movie Database (TMDb) API. For each movie, the TMDb API is queried to retrieve poster images, ensuring that every recommended movie can be visually displayed in the user interface.

Scripts used:

- **get_movie_poster(movie_id) :** function retrieves image URLs from TMDb.
- **movies_data.json** contains titles, genres, and image links.
- **users_data.json**, **ratings_data.json**, and **movie_genre.json** were generated from TensorFlow Datasets.



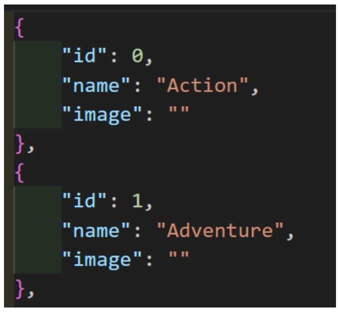
```
{
    "id": 138,
    "name": "user138",
    "email": "user138@example.com",
    "password": "bcrypt_sha256$12$tpNbg
```



```
{
    "movie_id": "500",
    "movie_title": "Fly Away Home (1996)",
    "movie_genre": [
        1,
        3
    ],
    "image_url": "https://image.tmdb.org/t/p/w500/xi8Iu6qyTfyZVDVy60raIOYJJmk.jpg"
},
```

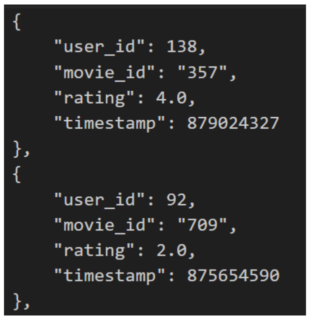
Figure 4: Example structure of **movies_data.json**

*Figure 3: Example structure of **users_data.json***



```
{
    "id": 0,
    "name": "Action",
    "image": ""
},
{
    "id": 1,
    "name": "Adventure",
    "image": ""
},
```

Figure 5: Example structure of **movie_genre.json**



```
{
    "user_id": 138,
    "movie_id": "357",
    "rating": 4.0,
    "timestamp": 879024327
},
{
    "user_id": 92,
    "movie_id": "709",
    "rating": 2.0,
    "timestamp": 875654590
},
```

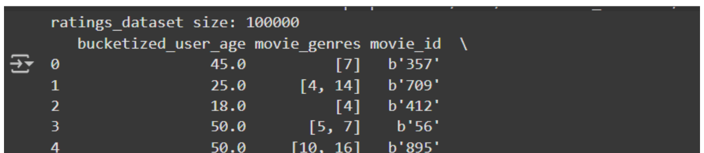
Note : *Once the data is extracted and enriched, it is uploaded to the **PostgreSQL database**. The **Django models** used for storing this data are designed based on the exact same structure and logic as these JSON files, ensuring consistency between the data layer and the backend application logic.*

Figure 6: Example structure of **ratings_data.json**

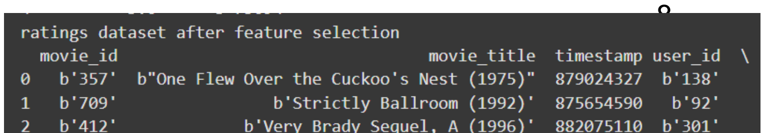
4.4.2 Data Preparation and Preprocessing

4.4.2.a Dataset Acquisition and Cleaning

The dataset was loaded using TensorFlow Datasets (TFDS), which provides structured access to the MovieLens corpus. Only the relevant fields—namely user identifiers, movie identifiers, movie titles, ratings, and timestamps—were retained. Records with missing or malformed entries were removed to ensure data integrity, and extraneous fields not relevant to our models were discarded.



```
ratings_dataset size: 100000
   bucketized_user_age movie_genres movie_id  \
0                 45.0          [7]   b'357'
1                 25.0      [4, 14]   b'709'
2                 18.0          [4]   b'412'
3                 50.0       [5, 7]    b'56'
4                 50.0     [10, 16]   b'895'
```



```
ratings dataset after feature selection
  movie_id                             movie_title  timestamp user_id  \
0   b'357'  b"One Flew Over the Cuckoo's Nest (1975)"  879024327  b'138'
1   b'709'                b'Strictly Ballroom (1992)'  875654590   b'92'
2   b'412'              b'Very Brady Sequel, A (1996)'  882075110  b'301'
```

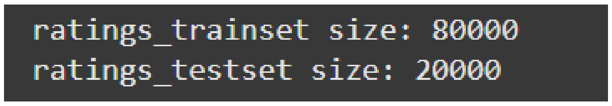
Figure 9: Dataset sample after feature selection filtering to relevant fields

Figure 8: Sample of raw ratings data from MovieLens dataset

Note : *Other potentially useful features available in the original dataset—such as movie genres, user demographics, or additional metadata—were not included in this initial version in order to maintain model simplicity and focus on core attributes. However, these features hold promise for enhancing model accuracy and personalization, and will be considered in future iterations of the system.*

4.4.2.b Train/Test Splitting

The dataset was randomly shuffled and partitioned into training and testing subsets using an 80/20 split. A fixed random seed was used during shuffling to ensure reproducibility across experiments. This manual splitting process was necessary because the TFDS version of MovieLens 100k does not provide predefined splits.



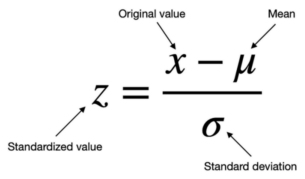
```
ratings_trainset size: 80000
ratings_testset size: 20000
```

Figure 10: Training & Test Dataset Size

4.4.2.c Normalization of Continuous Features

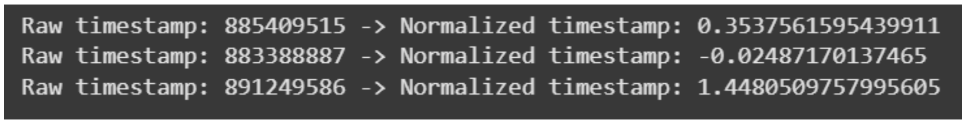
Among the selected features, the **timestamp** represents a **continuous numerical** variable indicating the moment at which each user provided a rating. Since raw timestamps (typically expressed as Unix epoch seconds) span a wide and uneven range, they were normalized prior to model training

To normalize the timestamp, we applied **Z-score normalization**, a standard statistical method that transforms a feature to have zero mean and unit variance , to ensure that different features contribute equally to the analysis or model training .



$$z = \frac{x - \mu}{\sigma}$$

Figure 11: Z-score Normalization Formula



```
Raw timestamp: 885409515 -> Normalized timestamp: 0.3537561595439911
Raw timestamp: 883388887 -> Normalized timestamp: -0.02487170137465
Raw timestamp: 891249586 -> Normalized timestamp: 1.4480509757995605
```

Figure 12: Example of Timestamp Normalization Using Z-score in the Training Set

4.4.2.d Encoding of Categorical Features

Categorical features such as **user_id** and **movie_id** are non-numeric variables that represent discrete values. Since neural networks operate on continuous inputs, these features must be transformed into a numerical format. We adopt a two-stage transformation to convert categorical string values into dense vector embeddings, which can be optimized during training to learn user preferences and item characteristics.

1. String-to-Integer Mapping via Vocabulary Encoding

The first step is to convert raw string values into unique integer indices using a StringLookup layer. This mapping can be defined as a function:

$$f_{\text{lookup}} : \mathcal{C} \rightarrow \mathbb{Z}_{\geq 0}$$

Where:

- C is the set of all categorical values (e.g., user or movie IDs),
- flookup(c) =i assigns an integer index i to a category c.

For instance, if **"898"** is a **movie_id**, then: flookup("898")=i_898

=> This indexing function allows efficient and consistent encoding of categorical inputs, with reserved indices for unknown or out-of-vocabulary (OOV) entries.

2. Embedding Lookup: Integer-to-Vector Transformation

Once categorical values are mapped to integers, these indices are used to retrieve trainable dense vector representations through an embedding layer:

$$f_{\text{embed}} : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}^d$$

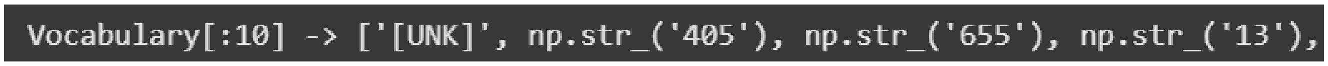
Where:

- d is the embedding dimension (set to 32 in our experiments),
- fembed(i)=ei ∈Rd is the embedding vector for index i.

This can be thought of as a matrix lookup operation. Let E∈Rn×d be the embedding matrix, where n is the vocabulary size. Then, the embedding for a category c is:

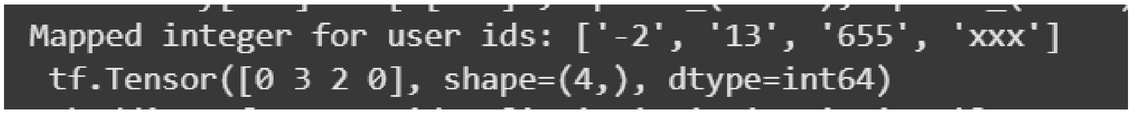
$$\mathbf{e}_c = \mathbf{E}[f_{\text{lookup}}(c)]$$

=>These embeddings are initialized randomly and learned jointly with the rest of the model via backpropagation. During training, the embeddings of users and items that co-occur (e.g., a user watching a movie) are adjusted to become more similar, often measured via inner product or cosine similarity.



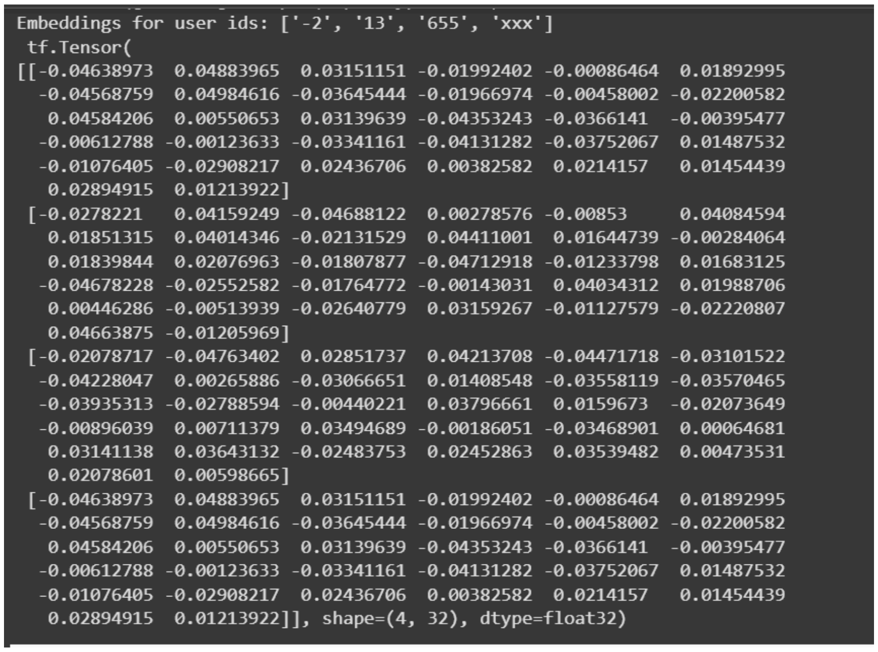
```
Vocabulary[:10] -> ['[UNK]', np.str_('405'), np.str_('655'), np.str_('13'),
```

Figure 13.a: Vocabulary Mapping Output for User IDs



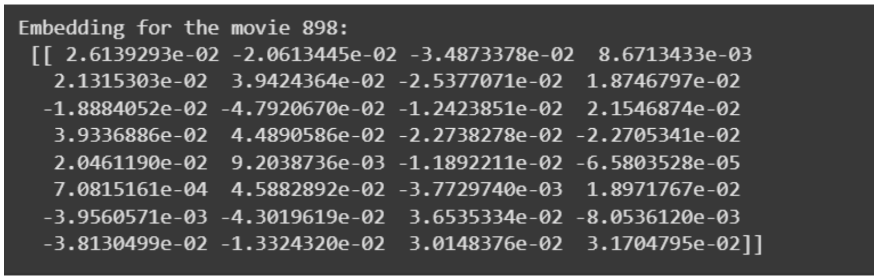
```
Mapped integer for user ids: ['-2', '13', '655', 'xxx']
 tf.Tensor([0 3 2 0], shape=(4,), dtype=int64)
```

Figure 13.b: Integer Encoding of Sample User IDs Using StringLookup



```
Embeddings for user ids: ['-2', '13', '655', 'xxx']
 tf.Tensor(
[[-0.04638973  0.04883965  0.03151151 -0.01992402 -0.00086464  0.01892995
  -0.04568759  0.04984616 -0.03645444 -0.01966974 -0.00458002 -0.02200582
   0.04584206  0.00550653  0.03139639 -0.04353243 -0.0366141  -0.00395477
  -0.00612788 -0.00123633 -0.03341161 -0.04131282 -0.03752067  0.01487532
  -0.01076405 -0.02908217  0.02436706  0.00382582  0.0214157   0.01454439
   0.02894915  0.01213922]
 [-0.0278221   0.04159249 -0.04688122  0.00278576 -0.00853     0.04084594
   0.01851315  0.04014346 -0.02131529  0.04411001  0.01644739 -0.00284064
   0.01839844  0.02076963 -0.01807877 -0.04712918 -0.01233798  0.01683125
  -0.04678228 -0.02552582 -0.01764772 -0.00143031  0.04034312  0.01988706
   0.00446286 -0.00513939 -0.02640779  0.03159267 -0.01127579 -0.02220807
   0.04663875 -0.01205969]
 [-0.02078717 -0.04763402  0.02851737  0.04213708 -0.04471718 -0.03101522
  -0.04228047  0.00265886 -0.03066651  0.01408548 -0.03558119 -0.03570465
  -0.03935313 -0.02788594 -0.00440221  0.03796661  0.0159673  -0.02073649
  -0.00896039  0.00711379  0.03494689 -0.00186051 -0.03468901  0.00064681
   0.03141138  0.03643132 -0.02483753  0.02452863  0.03539482  0.00473531
   0.02078601  0.00598665]
 [-0.04638973  0.04883965  0.03151151 -0.01992402 -0.00086464  0.01892995
  -0.04568759  0.04984616 -0.03645444 -0.01966974 -0.00458002 -0.02200582
   0.04584206  0.00550653  0.03139639 -0.04353243 -0.0366141  -0.00395477
  -0.00612788 -0.00123633 -0.03341161 -0.04131282 -0.03752067  0.01487532
  -0.01076405 -0.02908217  0.02436706  0.00382582  0.0214157   0.01454439
   0.02894915  0.01213922]], shape=(4, 32), dtype=float32)
```

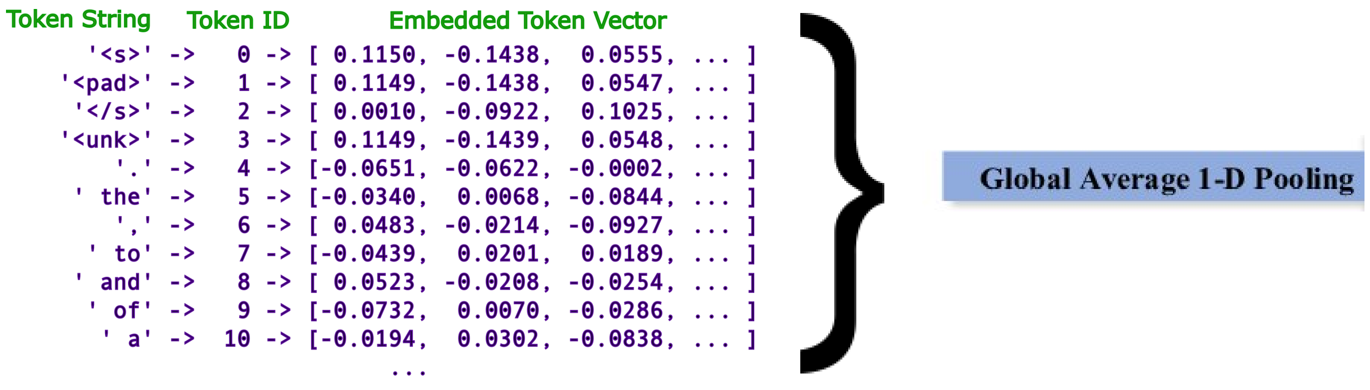
Figure 13.c: Embedding Vectors for Sample User IDs



```
Embedding for the movie 898:
 [[ 2.6139293e-02 -2.0613445e-02 -3.4873378e-02  8.6713433e-03
   2.1315303e-02  3.9424364e-02 -2.5377071e-02  1.8746797e-02
  -1.8884052e-02 -4.7920670e-02 -1.2423851e-02  2.1546874e-02
   3.9336886e-02  4.4890586e-02 -2.2738278e-02 -2.2705341e-02
   2.0461190e-02  9.2038736e-03 -1.1892211e-02 -6.5803528e-05
   7.0815161e-04  4.5882892e-02 -3.7729740e-03  1.8971767e-02
  -3.9560571e-03 -4.3019619e-02  3.6535334e-02 -8.0536120e-03
  -3.8130499e-02 -1.3324320e-02  3.0148376e-02  3.1704795e-02]]
```

Figure 13.d: Embedding Vector for Movie ID '898'

4.4.2.e Vectorization of Text Features



```
Token String   Token ID          Embedded Token Vector
      '<s>' ->    0 -> [ 0.1150, -0.1438,  0.0555, ... ]
    '<pad>' ->    1 -> [ 0.1149, -0.1438,  0.0547, ... ]
     '</s>' ->    2 -> [ 0.0010, -0.0922,  0.1025, ... ]
    '<unk>' ->    3 -> [ 0.1149, -0.1439,  0.0548, ... ]
        '.' ->    4 -> [-0.0651, -0.0622, -0.0002, ... ]
     ' the' ->    5 -> [-0.0340,  0.0068, -0.0844, ... ]
        ',' ->    6 -> [ 0.0483, -0.0214, -0.0927, ... ]
      ' to' ->    7 -> [-0.0439,  0.0201,  0.0189, ... ]
     ' and' ->    8 -> [ 0.0523, -0.0208, -0.0254, ... ]
      ' of' ->    9 -> [-0.0732,  0.0070, -0.0286, ... ]
       ' a' ->   10 -> [-0.0194,  0.0302, -0.0838, ... ]
                                ...
```

Figure 14.a: Tokenization Process

While the MovieLens 100k dataset does not contain full textual metadata such as plot summaries or user reviews, it does provide **movie titles**, which we treat as a lightweight form of textual content. To leverage this information, we incorporated a **text vectorization and embedding pipeline** that converts movie titles into dense, fixed-size vector representations. These title embeddings serve as **auxiliary item features**, enriching the model's understanding of items — particularly in cold-start scenarios where rating history is limited.

Step 1: Tokenization and Vectorization

Raw title strings are first transformed using the **TextVectorization** layer from Keras. This process involves:

- **Lowercasing and standardizing** the input text

- **Splitting** the text into individual tokens (usually word-based)
- **Mapping** each token to an integer index using a learned vocabulary

Let **T=[w1,w2,…,wk]**represent the list of **k** words in a movie title. The vectorization layer applies a vocabulary mapping **f_vec : wj↦tj ∈ Z ≥1**, resulting in a token sequence:

$$\mathbf{t} = [t_1, t_2, \dots, t_k] \in \mathbb{Z}^k$$

=>This integer sequence corresponds to positions in a vocabulary table created during an adaptation phase, where the layer scans all training data to build a frequency-based vocabulary.

Step 2: Embedding the Token Sequences

The sequence of integers t is passed through an **Embedding** layer, which maps each token **tj** to a dense vector **ej ∈ Rd** where d is the embedding dimension (e.g., 32). This creates a 2D tensor of shape k×d

$$\mathbf{E} = \begin{bmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \\ \vdots \\ \mathbf{e}_k \end{bmatrix} \in \mathbb{R}^{k \times d}$$

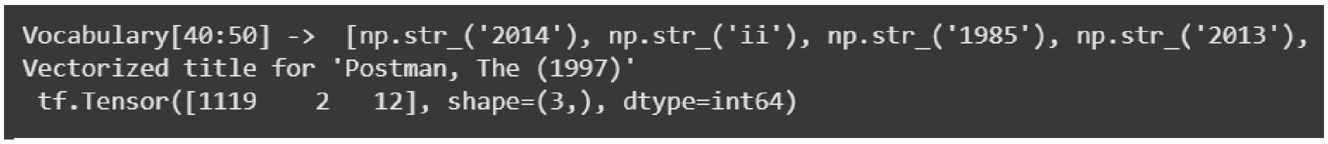
The embedding layer is initialized randomly and updated via gradient descent during training, allowing the model to learn semantic relationships between words.

Step 3: Aggregating Word Embeddings

Since movie titles vary in length, the embedded sequences must be reduced to **fixed-length vectors**. We apply a **Global Average Pooling** operation across the token dimension:

$$\mathbf{v}_{\text{title}} = \frac{1}{k} \sum_{j=1}^{k} \mathbf{e}_j \in \mathbb{R}^d$$

This results in a single vector per title, capturing the **average semantic meaning** of the words it contains. While more advanced models like RNNs, Transformers, or attention mechanisms could be used here, average pooling provides a **lightweight and effective** baseline for this task.



```
Vocabulary[40:50] ->  [np.str_('2014'), np.str_('ii'), np.str_('1985'), np.str_('2013'),
Vectorized title for 'Postman, The (1997)'
 tf.Tensor([1119    2   12], shape=(3,), dtype=int64)
```

Figure 14.b: Exemple of extracted vocabulary tokens & Tokenized integer sequence corresponding to the raw title "Postman, The (1997)"

4.5 Recommendation Modeling

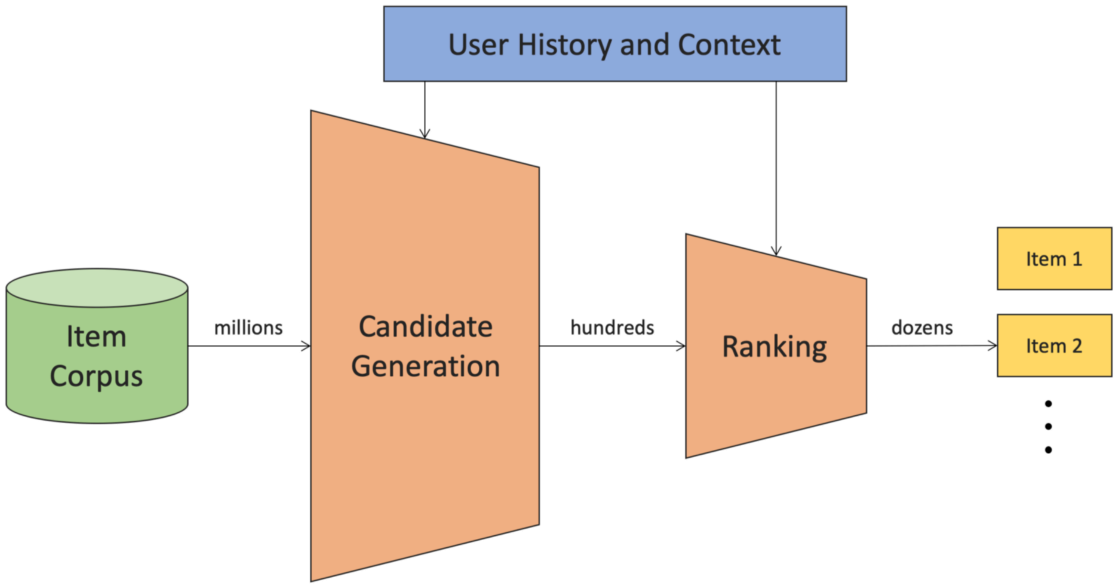


Figure 15: Two-Stage Recommendation Pipeline | Retrieval - Ranking

The core of our recommendation system is a two-stage architecture combining a **Retrieval Model** and a **Ranking Model**. This design balances scalability and accuracy: the retrieval stage quickly narrows down the vast movie catalog to a smaller set of relevant candidates, while the ranking stage refines personalized recommendations by accurately predicting user preferences.

4.5.1 Retrieval Model

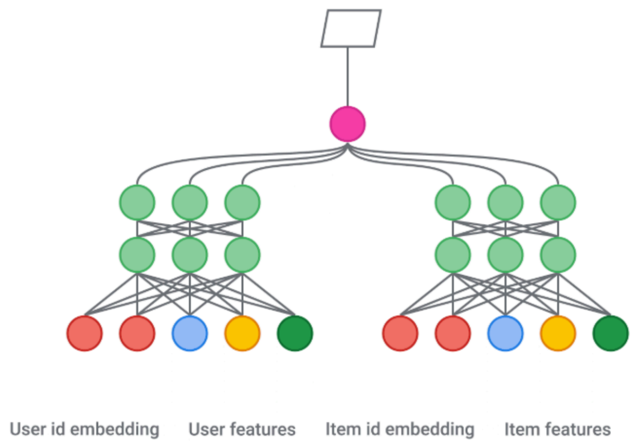


Figure 16: Two-Tower Neural Network Architecture

The retrieval model serves as the initial filtering step that selects a subset of potentially relevant movies from the entire corpus. This is formulated as a representation learning task where users and movies are embedded into a common latent space, enabling efficient similarity-based retrieval.

1.Data Preparation:

For retrieval, we only require **user** and **movie identifiers**, omitting explicit ratings. The training and test datasets are mapped to contain only user_id and movie_id, reflecting implicit feedback.

2.Candidate Corpus:

The complete set of movies used for retrieval is extracted from the MovieLens 100k dataset and consists of all available movies, regardless of whether they have been rated by users. Each movie is represented by its unique movie_id, which is used to construct embeddings for the candidate tower.

The goal of the **retrieval** model is **to surface potentially relevant movies, not just those previously rated**. Limiting candidates to rated items would prevent the system from recommending new content. By embedding the entire movie corpus, the system can retrieve **unseen movies** that are semantically close to a user's preferences, enabling genuine discovery and recommendation.

3.Model Architecture:

We use a **Two-Tower Neural Network** architecture implemented with TensorFlow Recommenders (TFRS):

- **The Query Tower** encodes users into embeddings based on their identifiers.
- **The Candidate Tower** encodes movies into embeddings.

Both towers are trained simultaneously to maximize the similarity between embeddings of users and the movies they have interacted with.

4.Loss and Metrics:

a.Retrieval Loss: Softmax with In-Batch Negatives

To train the retrieval model, we use a **categorical cross-entropy loss** over a softmax distribution, constructed using **in-batch negative sampling**. The goal is to maximize the similarity between the embeddings of users and their relevant (positive) items, while minimizing it with respect to irrelevant (negative) items.

Let:

- ui ∈ Rd be the embedding vector of user i,
- vj ∈ Rd be the embedding vector of movie (item) j,
- S(ui,vj)=⟨ui,vj⟩ denote the similarity score (dot product) between user and item.

For a batch of N (user, positive item) pairs {(ui,vi+)}[i=1 N], each training example treats the other N−1 items in the batch as **negative samples**. The softmax probability of a correct match for user iii is defined as:

$$P(i) = \frac{\exp(\langle u_i, v_i^+ \rangle)}{\sum_{j=1}^{N} \exp(\langle u_i, v_j \rangle)}$$

The retrieval loss is then the negative log-likelihood over the batch:

$$-\frac{1}{N}\sum_{i=1}^{N} \log P(i) = -\frac{1}{N}\sum_{i=1}^{N} \log\left(\frac{\exp(\langle u_i, v_i^+ \rangle)}{\sum_{j=1}^{N} \exp(\langle u_i, v_j \rangle)}\right)$$

b.Evaluation Metrics: Factorized Top-K Accuracy

To evaluate the model's effectiveness in recommending relevant movies, we use **Factorized Top-K Accuracy**, a set of metrics measuring how often the true positive item appears within the top KKK recommendations for a user.

For each user u, we compute the similarity scores against a set of candidate items C, rank them, and evaluate whether the true item v+ is within the top-K. Let ranku(v+) be the rank of the true item in the sorted list of candidates:

$$\text{Top-K Accuracy} = \frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} \mathbf{1}[\text{rank}_u(v^+) \leq K]$$

- 1[ranku(v+)≤K]:equals **1 if the condition inside is true**, and **0 otherwise**.
- So for each user uuu, we check: *"Is the true (positive) item in the top K retrieved items?"*.
- If yes, the indicator is 1, meaning a correct prediction.
- If not, the indicator is 0, meaning a miss.

5.Training:

The retrieval model is trained for **5 epochs** using the **Adagrad** optimizer with a **learning rate of 0.1**. Adagrad (Duchi et al., 2011) is a first-order gradient-based optimization algorithm that adapts the learning rate for each parameter individually based on the historical gradients. This is particularly effective in sparse settings like recommendation systems, where different embedding parameters (e.g., for rare users or movies) may require different levels of updates.

Formally, Adagrad updates a parameter θi at time step t according to:

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta}{\sqrt{G_i^{(t)} + \epsilon}} \cdot g_i^{(t)}$$

where:

- η is the global learning rate (set to 0.1),
- gi(t) is the gradient of the loss with respect to θi\theta_iθi at time ttt,
- Gi(t) is the accumulated sum of squared gradients,
- ϵ epsilonϵ is a small constant added for numerical stability.

To improve training **efficiency**, both the training and validation datasets are **batched and cached** using TensorFlow's data pipeline. Batching enables the model to process multiple examples simultaneously on the GPU, significantly speeding up training. Caching pre-processes and stores the data in memory or disk so that it does not need to be recomputed on every epoch, further reducing overhead.

6.Indexing for Efficient Retrieval:

After training the retrieval model, two **approximate nearest neighbor (ANN)** search indices are constructed to enable scalable recommendation serving:

1. **BruteForce Index**
2. **ScaNN Index (Scalable Nearest Neighbors)**

Each index is responsible for **mapping a user embedding vector**—obtained from the trained query model—to its top-k most relevant **movie embeddings** from the candidate corpus.

a. BruteForce Index

The **BruteForce index** performs an **exact nearest neighbor search** over the entire set of movie embeddings. Given a user embedding u ∈ Rd and a set of N movie embeddings {v1,…,vN}, it computes the similarity score for all i=1…N:

$$\mathbf{score(u,vi)=\langle u,vi\rangle}$$

The top-k candidates are selected by ranking all movies by their dot product with the user vector. While this approach is **computationally expensive** at scale (O(N)), it provides **exact results** and is thus primarily used for **testing and debugging purposes**, where accuracy and reproducibility are paramount.

b.ScaNN Index

The **ScaNN (Scalable Nearest Neighbors)** index, developed by Google Research, implements a highly optimized ANN retrieval technique that approximates nearest neighbors in **sublinear time**. It partitions the embedding space using **tree-based partitioning**, **asymmetric hashing**, and **quantization-aware reordering**, significantly reducing the number of distance computations required at inference time.

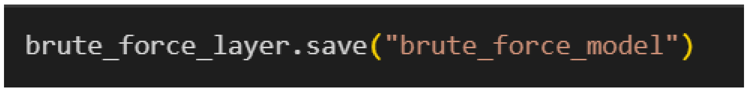
While it trades off some accuracy for speed, it scales much better to large corpora (millions of items). Formally, it still performs:

$$\mathbf{score(u,vi)\approx\langle u,vi\rangle} \text{ , for top k items}$$

but only computes dot products for a **filtered subset** of candidates, using a learned or precomputed partitioning scheme.

7.Model Saving and Loading:

The BruteForce index is saved and later loaded in the backend system for live recommendation serving.



```
brute_force_layer.save("brute_force_model")
```

4.5.2 Ranking Model

The ranking model refines the candidates retrieved by the retrieval model, predicting precise user preference scores to generate a personalized ranked list.

1. **Input Features:**
   The ranking model takes the concatenated embeddings of the user and each candidate movie as input.
2. **Model Architecture:**
   A feedforward neural network with three dense layers is employed:
   1. The first hidden layer has 256 units with **ReLU** activation.
   2. The second hidden layer has 64 units with **ReLU** activation.
   3. The output layer predicts a single scalar rating value.

3. **Loss and Metrics:**
   The model is trained to minimize **Mean Squared Error (MSE)** between predicted and actual user ratings. **Root Mean Squared Error (RMSE)** is used as an evaluation metric.

$$MSE = \frac{1}{n}\Sigma\underbrace{\left(y - \widehat{y}\right)^2}_{\text{The square of the difference between actual and predicted}}$$

4. **Training:**
   Similar to the retrieval model, the ranking model is trained for 5 epochs using **Adagrad optimizer** and

uses batched and cached datasets.

5. **Inference Pipeline:**
   For each user:
   1. The retrieval model retrieves the top-N candidate movies using either BruteForce or ScaNN indexes.
   2. The ranking model scores these candidates by predicting user-specific ratings.
   3. The candidates are sorted by predicted rating to produce the final top-K personalized recommendations.

4.6 Deployment and System Integration

To make the system accessible to users in real time, we deploy the trained models within a full-stack web application. The system is structured as follows :

4.6.1 Backend (Django REST Framework)

The backend of the system is implemented using the Django REST Framework (DRF), which enables modular and scalable development of web APIs. This component is responsible for data ingestion, user management, recommendation serving, and system interaction with the frontend.

1. **User Management and Authentication**

   To capture personalized user features, we define a custom user model (CustomUser) extending Django's AbstractUser. This model includes additional demographic attributes such as age, gender, occupation, and zip code. User registration and login are handled via JWT (JSON Web Token) authentication, providing secure and stateless session management.

2. **Data Ingestion and Management**

   The system supports dynamic loading of the MovieLens dataset through dedicated endpoints that import genres, movies, users, and ratings into a PostgreSQL database. The main data models are as follows:

   - Genre: Represents movie categories with optional image thumbnails.

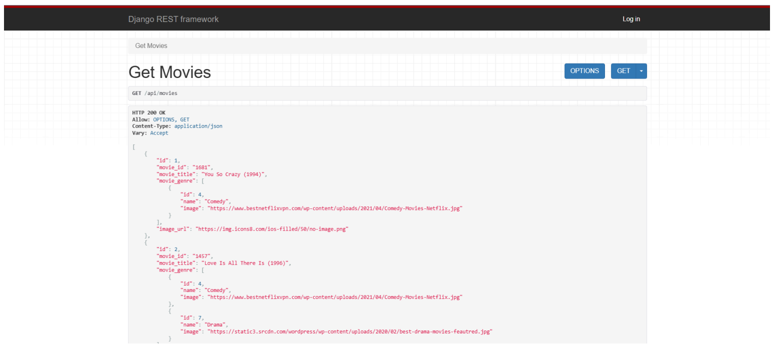
- Movie: Contains movie metadata and is linked to multiple genres via a many-to-many relationship.
- Rating: Stores user-movie interactions with a floating-point score and timestamp.

These entities are serialized using DRF's **ModelSerializer** classes and exposed through RESTful endpoints.

3. Public Data Access

Unauthenticated users can retrieve general movie-related information via public API endpoints. These include:

- A list of all movies and genres.
- Movies filtered by genre.
- Detailed information about a specific movie.



```
GET /api/movies
```

```
HTTP 200 OK
Allow: OPTIONS, GET
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "movie_id": "1681",
        "movie_title": "You So Crazy (1994)",
        "movie_genre": [
            {
                "id": 4,
                "name": "Comedy",
                "image": "https://www.bestnetflixvpn.com/wp-content/uploads/2021/04/Comedy-Movies-Netflix.jpg"
            }
        ],
        "image_url": "https://img.icons8.com/ios-filled/50/no-image.png"
    },
    {
        "id": 2,
        "movie_id": "1457",
        "movie_title": "Love Is All There Is (1996)",
        "movie_genre": [
            {
                "id": 4,
                "name": "Comedy",
                "image": "https://www.bestnetflixvpn.com/wp-content/uploads/2021/04/Comedy-Movies-Netflix.jpg"
            },
            {
                "id": 7,
                "name": "Drama",
                "image": "https://static3.srcdn.com/wordpress/wp-content/uploads/2020/02/best-drama-movies-feautred.jpg"
            }
```

*Figure 17:JSON Response from the **/api/movies** Endpoint of the Django Backend*

These endpoints provide structured JSON responses suitable for frontend rendering and client-side filtering.

4. Personalized Recommendation Endpoint

The core of the recommendation logic is encapsulated in a secure endpoint:

a. **GET /api/recommendations/ :** This endpoint is accessible only to authenticated users. It invokes a TensorFlow-based recommender model using the **recommend_movies_for_user(user_id)** function, which returns a list of relevant movie IDs for the requesting user. The backend then queries the database to retrieve movie metadata for these IDs and serializes the response.

To support frontend tagging of previously seen movies, the backend also extracts all movie IDs previously rated by the user. The complete response includes:

- The user ID,
- A list of recommended movie objects,
- A list of already rated movie IDs.

b. **GET /api/user-rated-movies/:** This authenticated endpoint returns all movies rated by the user, ordered by rating in descending order. It supports profile display and enables the user to revisit previously engaged content , this can help us to check the performance of our recommendation.

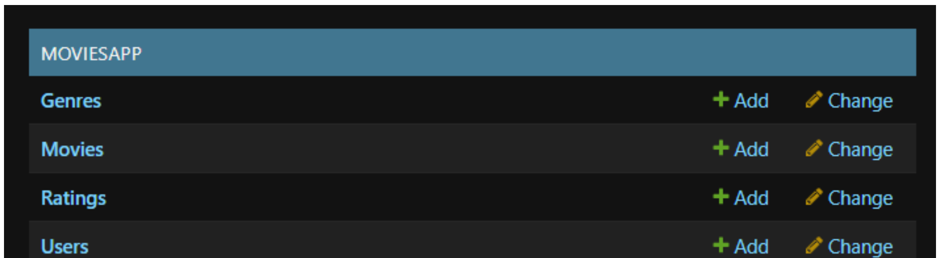
4.6.2 Database

The system's persistent data storage is managed through a relational database implemented with PostgreSQL, chosen for its robustness, scalability, and advanced relational capabilities. The database schema is designed to efficiently support the core functionalities of the recommendation system, including user management, movie metadata storage, and rating tracking.

Data Models

The primary entities and their relationships are modeled as follows:

- **User Model (CustomUser)**:
  Extending the standard Django AbstractUser, this model incorporates additional demographic fields such as age, gender, occupation, and zip code to provide enriched user profiles for personalized recommendations.
- **Genre Model**:
  Represents movie categories or genres (e.g., Action, Comedy, Drama). Each genre record includes a name and an optional image URL for display purposes.
- **Movie Model**:
  Stores metadata for each movie, including a unique identifier (movie_id), title, and an optional image URL. A many-to-many relationship links movies to multiple genres, enabling flexible categorization.
- **Rating Model**:
  Captures user-item interactions, consisting of a foreign key relationship to both the user and the movie, a floating-point rating value, and a timestamp indicating when the rating was recorded.



MOVIESAPP		
Genres	+ Add	Change
Movies	+ Add	Change
Ratings	+ Add	Change
Users	+ Add	Change

Figure 18:Representation of the Movie Model in the Django Admin Interface

4.6.3 Frontend (React.js)

Figure 19:React Logo

The frontend of the recommendation system is built using React, enabling a modular and responsive user interface. Key features include a dynamic navigation bar with interactive search functionality and seamless routing across pages such as Home, Genres, Movies, About, and Contact.

The homepage integrates components for curated movie collections, personalized recommendations, and top-rated movies. Users can browse movies by genre through a visually appealing card layout, with each genre linking to filtered movie lists.

Personalized recommendations are securely fetched from the backend and visually indicate movies the user has already rated via an “Already Seen” badge, enhancing user experience. State management is handled through React hooks, with API calls secured using authentication tokens.

The frontend also supports a user dashboard for managing profiles and viewing history, designed to be responsive and accessible across devices.

5. RESULTS

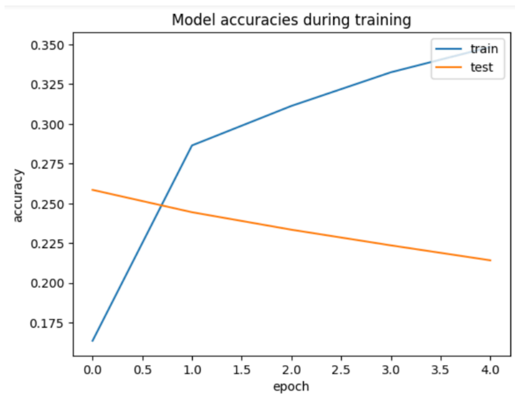


Figure 20:Top-K Categorical Accuracy of the Retrieval Model over Epochs for Training and Validation Sets

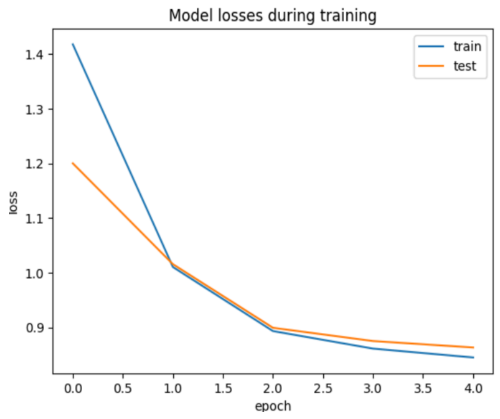


Figure 21:Training and Validation Loss (MSE) of the Ranking Model Across Epochs

❖ Interpretation and Analysis

a.Retrieval Model Accuracy Analysis

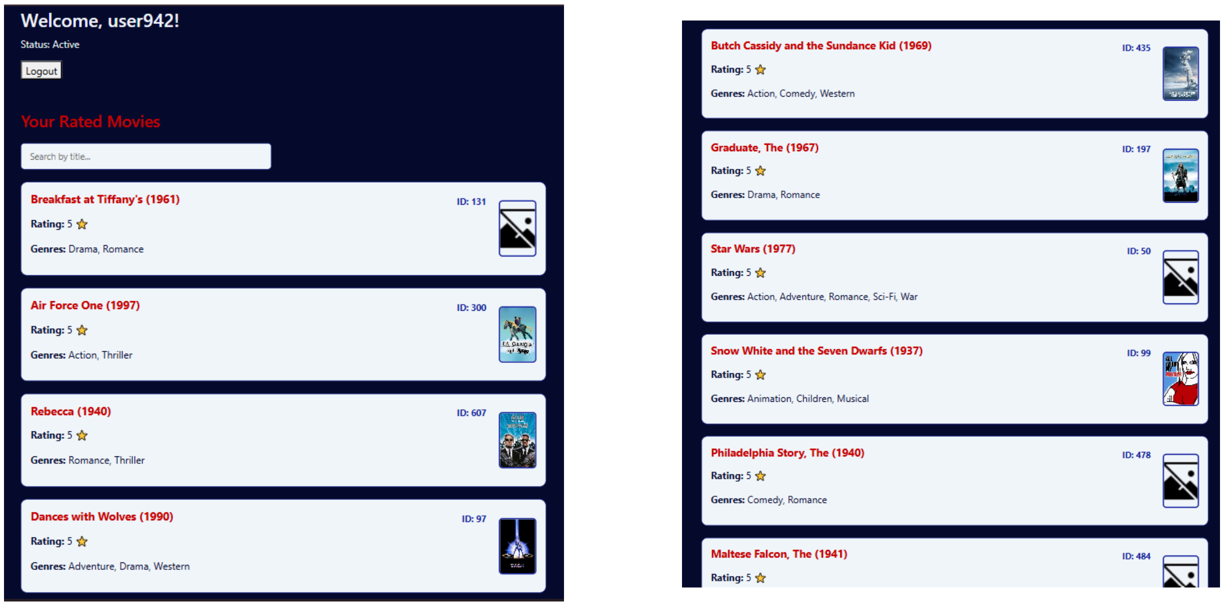
As shown in **Figure 20**, the Top-K categorical accuracy steadily improves on the training dataset across epochs. However, the validation accuracy plateaus early and, in some cases, declines slightly as training progresses. This pattern strongly indicates that the model is **overfitting** to the training data.

Overfitting occurs when the model learns to memorize training examples instead of generalizing patterns that would apply to new, unseen data. This is a common issue in deep learning models, especially when the model has a large number of parameters and insufficient regularization. In this case, the retrieval model fits the training data very well, but fails to achieve similar performance on validation data, which is a more meaningful indicator of real-world performance.

To mitigate overfitting, one could consider:

- Adding **regularization** (e.g., L2 regularization on embeddings).
- Incorporating **rich features** such as user demographics or movie metadata (genres, release year).

Despite the modest validation performance, the model performs well in practical usage. For example, for **User 942**, the recommended movies share similar genres and high ratings with previously rated items, demonstrating that the system is able to **capture user preferences** and deliver semantically relevant recommendations in the live application.



Welcome, user942!

Status: Active

Your Rated Movies

Breakfast at Tiffany's (1961) ID: 131
Rating: 5 ⭐
Genres: Drama, Romance

Air Force One (1997) ID: 300
Rating: 5 ⭐
Genres: Action, Thriller

Rebecca (1940) ID: 607
Rating: 5 ⭐
Genres: Romance, Thriller

Dances with Wolves (1990) ID: 97
Rating: 5 ⭐
Genres: Adventure, Drama, Western

Butch Cassidy and the Sundance Kid (1969) ID: 435
Rating: 5 ⭐
Genres: Action, Comedy, Western

Graduate, The (1967) ID: 197
Rating: 5 ⭐
Genres: Drama, Romance

Star Wars (1977) ID: 50
Rating: 5 ⭐
Genres: Action, Adventure, Romance, Sci-Fi, War

Snow White and the Seven Dwarfs (1937) ID: 99
Rating: 5 ⭐
Genres: Animation, Children, Musical

Philadelphia Story, The (1940) ID: 478
Rating: 5 ⭐
Genres: Comedy, Romance

Maltese Falcon, The (1941) ID: 484
Rating: 5 ⭐

Figure 21:History of Rated Movies Of user942

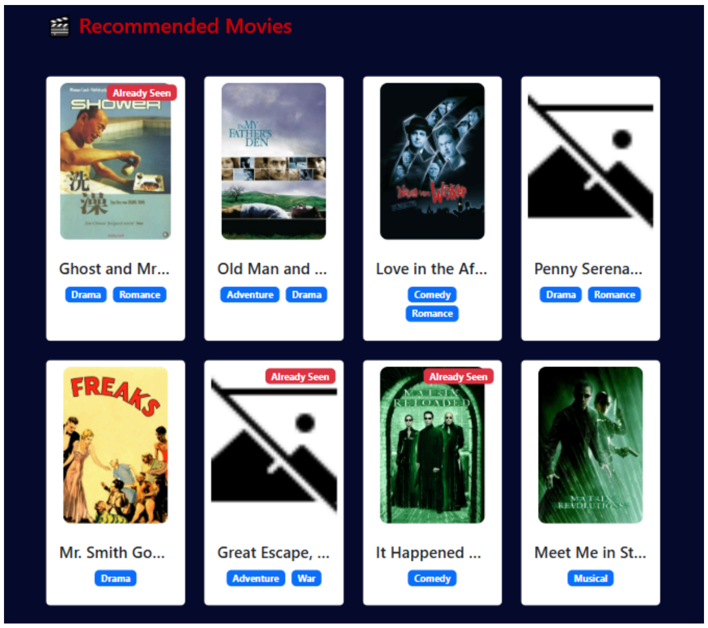


Figure 21:Recommended Movies For user942

Analysis of this user's historical ratings reveals a clear preference for genres such as drama, romance, and adventure.

Upon reviewing the recommendations provided by the system, we observe that the suggested movies consistently align with these genres. For instance, the recommendation list includes titles belonging to **romantic dramas** and **adventure-themed films**, which closely resemble the types of movies this user has previously rated highly.

This behavior illustrates the model's ability to **capture and generalize user preferences** effectively

b.Retrieval Model Accuracy Analysis

From the first to the last epoch, both losses decrease significantly:

- Training loss drops from 10.19 to 0.85,
- Validation loss drops from 1.20 to 0.86,

This sharp decrease indicates that the model rapidly learns to approximate user ratings more accurately after just a few epochs. Unlike the retrieval model, the ranking model does not exhibit strong signs of overfitting—the validation loss follows the training loss closely, showing good generalization to unseen data.

6. CONCLUSION

This research presents a hybrid recommendation system built upon the MovieLens 100k dataset, combining a deep learning retrieval model and a ranking model to provide personalized movie suggestions. The retrieval component effectively narrows down the candidate movies, while the ranking model refines the predictions by estimating user ratings. Although evaluation metrics indicate some overfitting and modest generalization performance, the system demonstrates practical effectiveness by recommending movies aligned with users' historical preferences and favored genres.

The use of approximate nearest neighbor search techniques, such as ScaNN, ensures scalability and responsiveness in real-time recommendation scenarios. Meanwhile, the integration of user and movie embeddings, alongside carefully designed neural network architectures, allows the model to capture complex interactions between users and items.

Future work could focus on enhancing generalization through additional regularization methods, incorporating richer side information (e.g., user demographics or movie metadata), and experimenting with more advanced ranking architectures. Additionally, integrating Apache Kafka or similar streaming platforms could enable real-time recommendation updates and user feedback processing, further improving responsiveness and personalization in dynamic environments.

Ultimately, this project illustrates the potential and challenges of deep learning approaches in recommendation systems, underscoring the importance of both quantitative evaluation and qualitative assessment in measuring success.

7. REFERENCES

- NVIDIA Developer. (2020, April 28). *How to Build a Winning Recommendation System, Part 2: Deep Learning for Recommender Systems*. https://developer.nvidia.com/blog/how-to-build-a-winning-recommendation-system-part-2-deep-learning-for-recommender-systems/
- Kohler, V. (2018, August 27). *Collaborative Filtering using Deep Neural Networks in TensorFlow*. Medium. https://medium.com/@victorkohler/collaborative-filtering-using-deep-neural-networks-in-tensorflow-96e5d41a39a1
- EPFL Machine Learning for Computer Vision Lab. (2021). *CIL Project 1 - Collaborative Filtering*. https://colab.research.google.com/github/dalab/lecture_cil_public/blob/master/exercises/2021/Project_1.ipynb
- Sun, W., & Shi, Y. (2023). *Unified Recommender System via Sequential and Contrastive Learning*. arXiv preprint. https://arxiv.org/pdf/2312.06145v1
- Shrivastava, D. (2022). *Movie Recommendation System Using Content-Based Filtering and Collaborative Filtering Techniques*. *International Journal of Research and Analytical Reviews (IJRAR), 9*(2), 865–869. https://ijrar.org/papers/IJRAR22B1151.pdf
- NVIDIA Developer. (2020, May 12). *Building Recommender Systems Faster Using Jupyter Notebooks from NGC*. https://developer.nvidia.com/blog/building-recommender-systems-faster-using-jupyter-notebooks-from-ngc/
- The Movie Database (TMDb). (n.d.). *API Settings*. https://www.themoviedb.org/settings/api
- Yan, E. (2020, October 13). *Real-Time Recommendations*. https://eugeneyan.com/writing/real-time-recommendations/
- TensorFlow. (n.d.). *TensorFlow Recommenders*. https://www.tensorflow.org/recommenders

- TensorFlow Recommenders GitHub. (n.d.). *Issue #712: Discussions and Implementation Notes*. https://github.com/tensorflow/recommenders/issues/712

- Covington, P., Adams, J., & Sargin, E. (2016). *Deep Neural Networks for YouTube Recommendations*. *Proceedings of the 10th ACM Conference on Recommender Systems*, 191–198. https://dl.acm.org/doi/pdf/10.1145/2959100.2959190
- GroupLens Research. (n.d.). *MovieLens Dataset*. https://grouplens.org/datasets/movielens/
- Apache Software Foundation. (n.d.). *Apache Kafka*. https://kafka.apache.org/