# Recommendation System – Two-Tower Model

Version: 1.0

File: rp\_two\_tower\_network.py

Author: Bhakti Ayarekar

Updated: 20 October 2025

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Problem Statement**

We want to build Recommendation System based on users and video(post) from keek’s platform.

Traditional recommendation is based on filter+sorting methods, we need to build personalised system. (Traditional method - Show me all videos matching conditions, sorted by entry\_date and country.)

**On What we are working: (Con’s of SQL)**

* Treats all users the same within a rule set
* Can’t *learn* preferences (e.g. "I like funny Hindi clips but not sports”)
* Relies on hard-coded conditions (fixed thresholds, no adaptation)
* Doesn’t improve with user behavior

**2. Dataset Used :**

1. From Production Postgress SQL used table - stream\_viewer, users, posts
2. In my project I used query on production dataset :

**For Interaction Table - On Streamer viewer**

SELECT

user\_id,

post\_id,

SUM(likes) AS likes,

SUM(view\_count) AS views,

SUM(CASE WHEN saved = 'Y' THEN 1 ELSE 0 END) AS saves

FROM

public.stream\_viewer

WHERE

post\_id IS NOT NULL

AND likes IS NOT NULL

AND likes <> 0

GROUP BY

user\_id,

post\_id

ORDER BY

user\_id, post\_id;

**For User Table - On User table**

SELECT DISTINCT

u.id AS user\_id,

u.country

FROM

users u

INNER JOIN (

SELECT DISTINCT user\_id

FROM public.stream\_viewer

WHERE

post\_id IS NOT NULL

AND likes IS NOT NULL

AND likes <> 0

) sv

ON u.id = sv.user\_id

WHERE

u.country IS NOT NULL

AND u.country <> ''

ORDER BY

u.id;

**For Post Table- On Post table**

SELECT

p.post\_id,

p.user\_id AS post\_owner\_id,

COALESCE(

NULLIF(

CASE

WHEN p.country IS NULL OR p.country = '' OR p.country = 'default'

THEN u.country

ELSE p.country

END,

''

),

'IN'

) AS effective\_country,

p.lang

FROM

public.post p

LEFT JOIN (

SELECT DISTINCT ON (u.id)

u.id AS user\_id,

u.country

FROM users u

INNER JOIN (

SELECT DISTINCT user\_id

FROM public.stream\_viewer

WHERE

post\_id IS NOT NULL

AND likes IS NOT NULL

AND likes <> 0

) sv

ON u.id = sv.user\_id

WHERE

u.country IS NOT NULL

AND u.country <> ''

) u

ON u.user\_id = p.user\_id

WHERE

p.post\_id IN (

SELECT DISTINCT post\_id

FROM public.stream\_viewer

WHERE

post\_id IS NOT NULL

AND likes IS NOT NULL

AND likes <> 0

)

ORDER BY

p.post\_id;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Then, After Query

**Data Tables look like this :**

**Interaction Table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **user\_id** | **post\_id** | **likes** | **Views** | **Saves** |
| 1001 | 1401 | 1 | 1 | 0 |

**User Table:**

|  |  |  |
| --- | --- | --- |
| **user\_id** | **country** | **Supported\_languages** |
| 1001 | US | [‘en’,’es’,’hi’] |

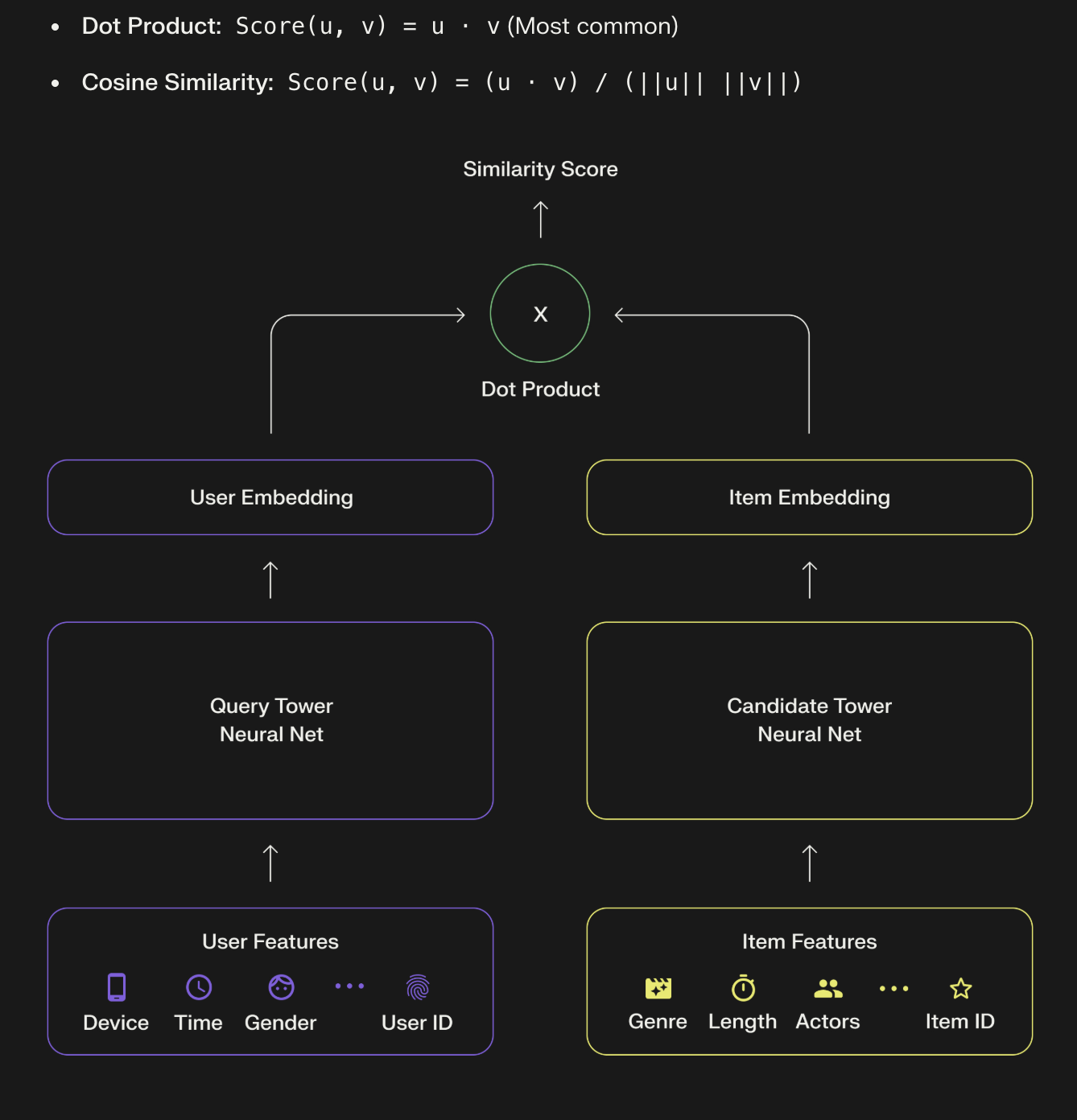
**Post Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **post\_id** | **post\_owner\_id** | **effective\_country** | **Lang** |
| 1001 | US | IN | En |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm Used: Two-Tower Network**

The **Two-Tower** (dual encoder) model is a neural network architecture commonly used in recommendation systems and information retrieval. It gets its name from having two separate neural network “towers”: one for the **user** (often called the *query tower*) and one for the **item** (the *candidate tower*). Each tower learns to **encode** its inputs (user features or item features) into a dense **embedding vector** in a shared feature space. The core idea is that if a user and an item are a good match, their embedding vectors will be close or aligned in this space, which we measure by a **similarity score** (usually a dot product).



**Why Two-Tower Model?**

It enables scalable and flexible recommendations by learning separate embeddings for users and items, allowing fast retrieval via nearest-neighbor search—ideal for large catalogs.

Unlike traditional matrix factorization, it incorporates rich user/item features using neural networks, making it effective for personalized candidate generation in real-time systems.

**System Architecture :**

User Data → User Tower ┐

├→ Shared Embedding Space → Similarity Scoring → Ranked Results

Item Data → Item Tower ┘

**Components :**

|  |  |
| --- | --- |
| **Component** | **Description** |
| **User Tower** | Learns representations from user features (e.g., country, language). |
| **Item Tower** | Learns representations from item features (e.g., country, language, tags). |
| **Embedding Space** | Shared vector space for user and item embeddings. |
| **Similarity Function** | Computes similarity (e.g., dot product or cosine) between user and item vectors. |

**Two - Tower Network Algorithm Working :**

* **User tower**

Embeddings: `user\_id\_emb(32)`, `user\_country\_emb(8)`, `user\_lang\_emb(8)`

Concatenate → `user\_features`

MLP: `Dense(32, relu)` → `user\_vector = Dense(32, linear)`

* **Item tower**
* Embeddings: `item\_id\_emb(32)`, `item\_country\_emb(8)`, `item\_lang\_emb(8)`
* Concatenate → `item\_features`
* MLP: `Dense(32, relu)` → `item\_vector = Dense(32, linear)`
* **Similarity:**  `Dot(axes=1)` between `user\_vector` and `item\_vector`
* **Output activation:**  `sigmoid` → probability of interaction

**Loss & Optimisation -**

* Loss: binary\_crossentropy
* Optimizer: adam
* Metrics tracked: accuracy

**Hyperparameters -**

* Embedding dims: 32 (IDs), 8 (country/lang)
* Hidden units per tower: 32
* Batch size: 256
* Epochs: 5
* Validation split: 0.1

**Tech Stack -**

* Language: Python 3
* Libraries: TensorFlow/Keras, Pandas, NumPy, (standard Python `random`, `ast`)
* Training device: CPU/GPU (Keras-compatible)

**Output of Model -**

|  |  |  |  |
| --- | --- | --- | --- |
| **user\_id** | **recommended\_post\_id** | **score** | **rank** |
| 10114 | 2655 | 0.624968 | 1 |
| 10114 | 2701 | 0.585291 | 2 |
| 10114 | 713 | 0.578722 | 3 |
| 10000000942 | 2655 | 0.741727 | 1 |
| 10000000942 | 2701 | 0.703937 | 2 |
| 10000000942 | 713 | 0.697708 | 3 |
| 10000001189 | 2655 | 0.721780 | 1 |
| 10000001189 | 2701 | 0.683907 | 2 |

**Accuracy of Model -**

Final training accuracy: 90.01557230949402 %

Final validation accuracy: 6.69826865196228 %

**Further Improvements -**

Need to train on also negative sampling to improve validation accuracy. So, we can do train on likes = 0.

**End of Document**