5 mins Recommender systems: Matrix Factorization Collaborative Filtering

5 min read · May 10, 2024

Omar Tafsi Follow

Open in app ↗ Sign up Sign in

Medium Search

Introduction:

Collaborative Filtering (CF) is a popular technique of recommendation systems that leverages the collective wisdom of users to provide personalised recommendations. Among the various CF methods, Matrix Factorization (MF) has emerged as a powerful approach for modelling user-item interactions. In this article, we dig into the workings of Matrix Factorization for collaborative filtering, its implementation in Python, and its key strengths and weaknesses.

Note that we're focusing on explicit rating schemas (user rates item on a scale of 1 to 5) instead of the implicit (1s if the user interacted with the item and 0 elsewhere) in this series of articles as it is the more intuitive use-case.

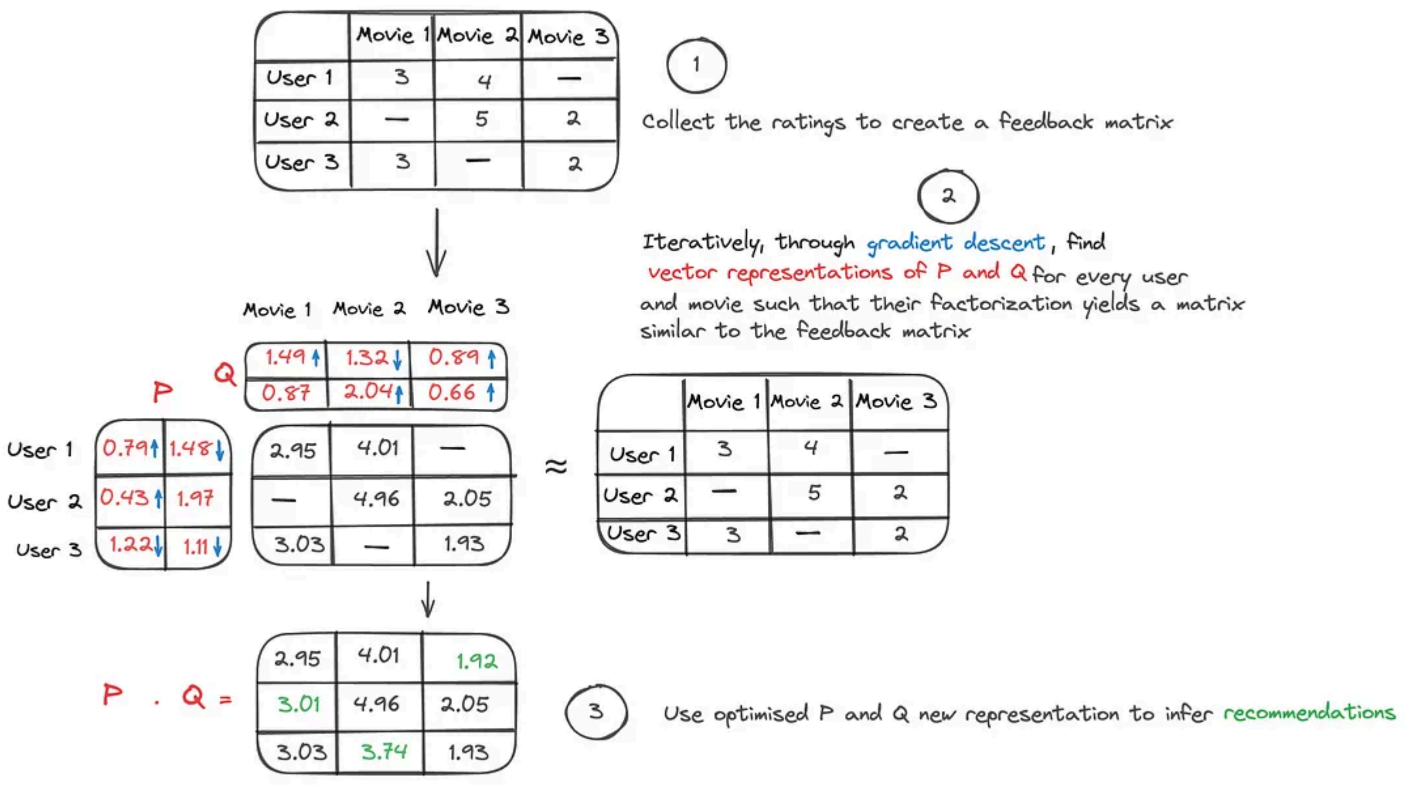
Understanding Matrix Factorization:

At its core, Matrix Factorization aims to decompose the user-item interaction matrix into lower-dimensional matrices, capturing latent features of users and items. By representing users and items in a lower-dimensional space, Matrix Factorization can effectively capture complex patterns and preferences, enabling accurate recommendation generation. The method comprizes of the following steps:

1. **Data Representation**: Represent user-item interactions as the feedback matrix.
2. **Initialization**: Initialize user (P) and item (Q) matrices.
3. **Optimization**: Use gradient descent to update P and Q to minimize the error between the output of the P . Q and the feedback matrix.
4. **Evaluation**: Assess model performance using metrics like MSE or RMSE through predicting on a small subset of data.
5. **Inference**: Use optimized P and Q to predict interactions for new data.

To make it more intuitive, many researchers and machine learning practitioners like to think of these latent vectors dimensions as explicit item or user characteristics ex: if we represent movies with 3-d vectors, in dimension 1 we would have a float that could encode the movie category, dimension 2 is country and dimension 3 is a float that encodes the decade of the movie.

Here's an example where we choose to represent users and items as a 2-d vectors to illustrate these steps, for the sake of simplicity we will skip the evaluation step:


Movie 1
Movie 2
Movie 3
User 1
User 2
User 3
1
Collect the ratings to create a feedback matrix
2
Iteratively, through gradient descent, find vector representations of P and Q for every user and movie such that their factorization yields a matrix similar to the feedback matrix
P
Q
≈
P . Q =
3
Use optimised P and Q new representation to infer recommendations

Python Implementation:

In this implementation, to perform our gradient descent optimisation, we use the following squared error formula containing both an error and a regularisation component:

$$e_{ij}^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2 + \frac{\beta}{2} \sum_{k=1}^{K} (||P||^2 + ||Q||^2)$$

It yields an Item/User vector element updates with the following formulas:

$$p'_{ik} = p_{ik} + \alpha \frac{\partial}{\partial p_{ik}} e_{ij}^2 = p_{ik} + \alpha(2e_{ij}q_{kj} - \beta p_{ik})$$
$$q'_{kj} = q_{kj} + \alpha \frac{\partial}{\partial q_{kj}} e_{ij}^2 = q_{kj} + \alpha(2e_{ij}p_{ik} - \beta q_{kj})$$

Here’s the Python implementation:

```
import numpy as np

class MatrixFactorization:
    def __init__(self, feedback_matrix, nb_features, steps=5000, lr=0.0002, reg
        #
        self.feedback_matrix = feedback_matrix  # rating matrix
        self.nb_features = nb_features  # latent features
        self.steps = steps  # iterations
        self.lr = lr  # learning rate
        self.reg = reg  # regularization parameter
        self.P = None  # User features matrix
        self.Q = None  # Item features matrix

    def fit(self):
        num_users, num_items = self.feedback_matrix.shape
        self.P = np.random.rand(num_users, self.nb_features)
        self.Q = np.random.rand(num_items, self.nb_features).T

        for step in range(self.steps):
            for i in range(num_users):
                for j in range(num_items):
                    if self.feedback_matrix[i][j] > 0:
                        # calculating the loss to be used in the vector update
                        eij = self.feedback_matrix[i][j] - np.dot(self.P[i, :],

                        for k in range(self.nb_features):
                            # updating every element of the user/item vectors w
                            self.P[i][k] = self.P[i][k] + self.lr * (2 * eij *
                            self.Q[k][j] = self.Q[k][j] + self.lr * (2 * eij *

            eR = np.dot(self.P, self.Q)

            e = 0
            # calculating error term with newly updated vectors
            for i in range(num_users):
                for j in range(num_items):
                    if self.feedback_matrix[i][j] > 0:
                        e += pow(self.feedback_matrix[i][j] - np.dot(self.P[i,
                        for k in range(self.nb_features):
                            e += (self.reg / 2) * (pow(self.P[i][k], 2) + pow(s

            # stop training if error below 0.001
            if e < 0.001:
                break

    def get_matrices(self):
        return self.P, self.Q.T

feedback_matrix = [
        [3,4,0],

        [0,5,2],

        [3,0,2],

      ]
matrix_factorization = MatrixFactorization(np.array(feedback_matrix),2)
matrix_factorization.fit()
print(matrix_factorization.get_matrices())
```

Strengths of Matrix Factorization:

1. **Scalability:** Although this property depends on specific implementation details and characteristics of the dataset and the number of epochs it takes to achieve the targeted performance, in general, Matrix Factorization techniques are computationally efficient and can handle large-scale datasets as oppose to methods like Memory based CF for instance. The decomposition of the user-item interaction matrix into lower-dimensional matrices reduces computational complexity.

2. **Interpretability:** MF provides interpretable latent factors representing users and items. These latent factors capture meaningful user preferences and item characteristics, facilitating the generation of transparent and explainable recommendations.

3. **Handling Sparsity:** Matrix Factorization effectively handles sparse user-item interaction data by inferring missing values based on learned latent factors. This capability is crucial in recommendation systems where user-item interactions are often sparse, ensuring robust recommendation generation. In the implemented above we can see for instance how we discard the 0 values (interaction that didn't happen yet) from the training which can have a huge impact on the time complexity.

As part of this series of articles, **we're planning to add a dedicated article post where we compare how all the RS methods we have introduced by testing each one against a large dataset** to measure several training and serving metrics (MAP@K, time to train, time to infer, ease of integrating new users/items, ease of parallelization. etc)

Weaknesses of Matrix Factorization:

1. **Cold Start Problem:** Similar to other CF methods, MF suffers from the cold start problem, where it struggles to provide accurate recommendations for new users or items with limited interaction data. In such cases, MF may fails to capture sufficient information to generate meaningful recommendations.

2. **Lack of Contextual Information:** Matrix Factorization relies solely on user-item interaction data and does not consider contextual information such as user demographics or item attributes. This limitation may lead to less personalised recommendations, especially in domains where contextual information is crucial for understanding user preferences.

3. **Overfitting:** MF models are prone to overfitting, especially when dealing with noisy or sparse datasets. Without proper regularization techniques, MF may learn spurious patterns from the training data, resulting in poor generalisation performance on unseen data.

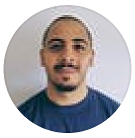
Conclusion:

Matrix Factorization remains a powerful and widely used technique for collaborative filtering in recommendation systems. Its ability to capture latent features of users and items, coupled with its scalability and interpretability, makes it a valuable tool for generating personalised recommendations. However, addressing its limitations, such as the cold start problem and lack of contextual information, is crucial for enhancing its performance in real-world applications. By understanding the strengths and weaknesses of Matrix Factorization, developers and data scientists can design more effective recommendation systems tailored to the needs of users and businesses.

References:

Matrix factorization : http://www.quuxlabs.com/blog/2010/09/matrix-factorization-a-simple-tutorial-and-implementation-in-python/

Recommendation System Collaborative Filtering Matrix Factorization



Follow

Written by Omar Tafsi

8 followers · 1 following

Responses (1)

Write a response

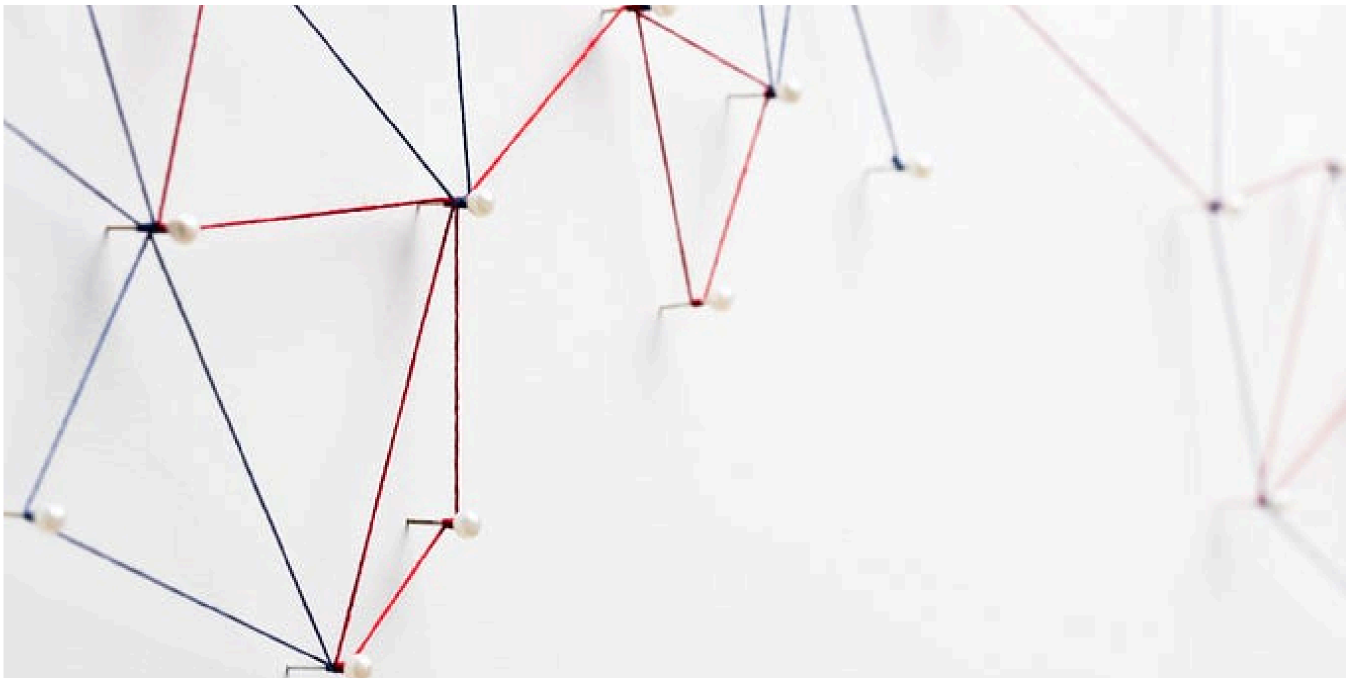
What are your thoughts?

mohanakumaar Karhikeyan
May 8

Great Article.

Reply

More from Omar Tafsi



Omar Tafsi

5 mins Recommender Systems: Neural Graph Collaborative Filtering

In the ever-evolving field of recommender systems, neural graph collaborative filtering represents a great addition, merging graph theory...

Oct 8, 2024 10

Omar Tafsi

5 mins Recommender systems: Neural Collaborative Filtering

Introduction:

Jun 3, 2024 4

Omar Tafsi

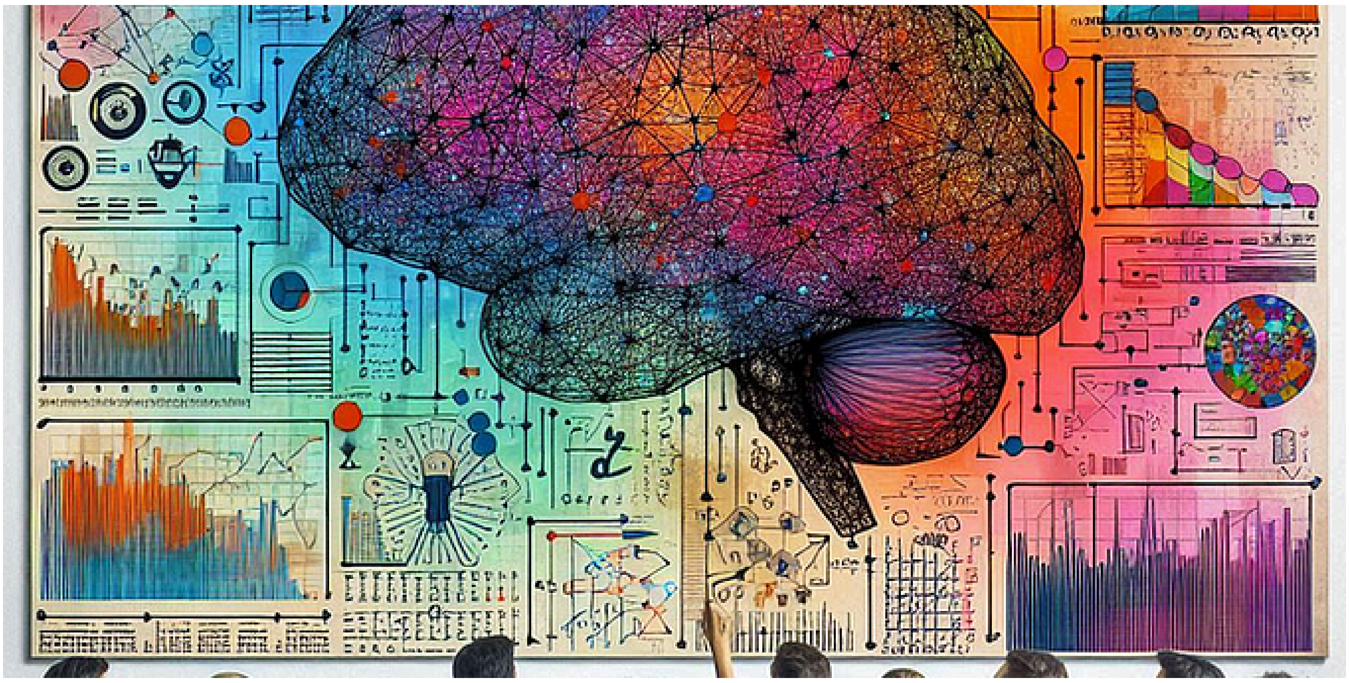
5 mins Recommender Systems: Memory Based Collaborative Filtering

Why do we use Recommender Systems?

Mar 23, 2024 8

See all from Omar Tafsi

Recommended from Medium

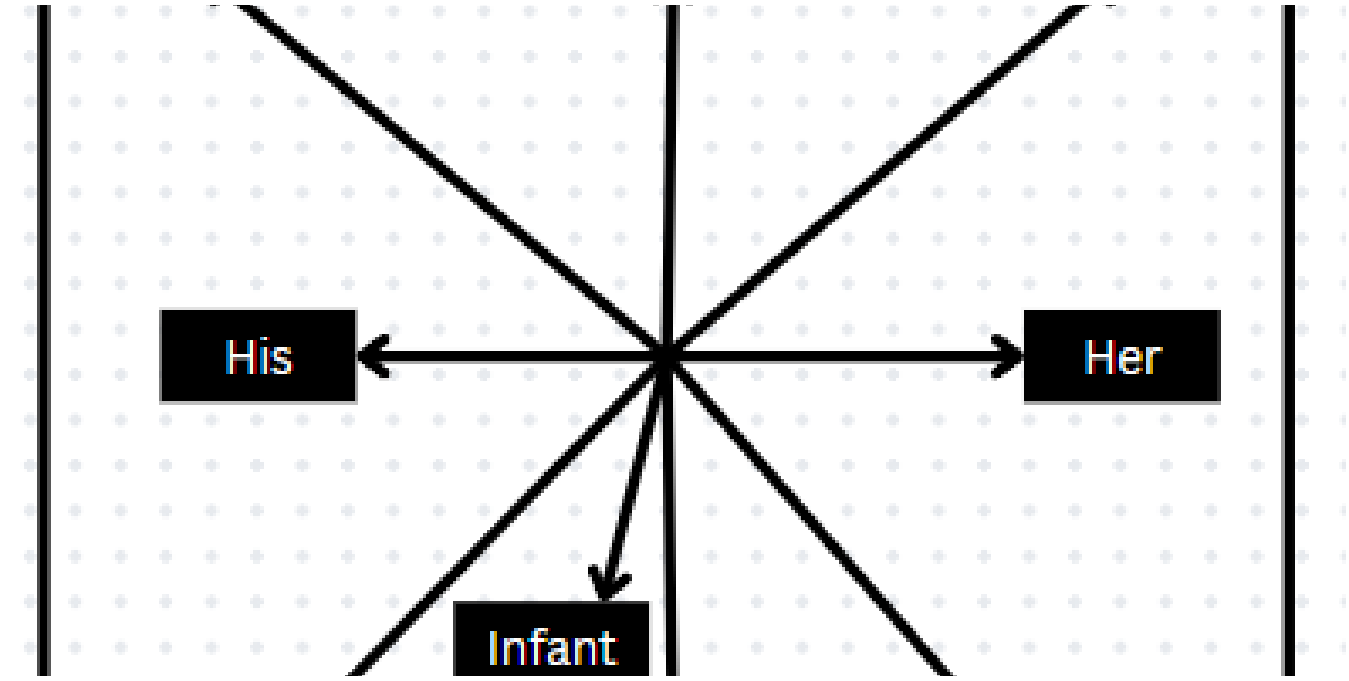


Rohan Rao

Machine Learning with Python: Building a Product Ranking System

Learn how to use python and Machine Learning to rank products effectively

Feb 12 16 1

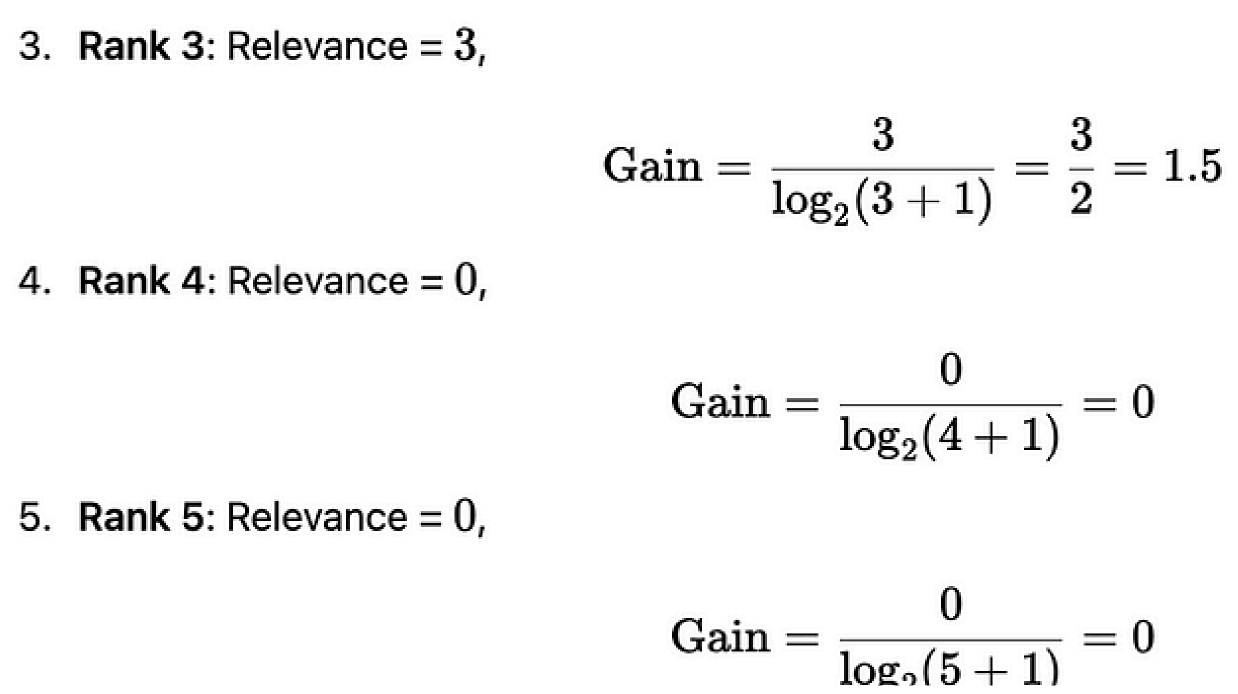


Imran Khan

Understanding Vector Embeddings, Semantic Search and Its Implementation

A vector embedding converts data—such as text, images, or audio—into a numerical representation (a high-dimensional vector, e.g., a...

May 26



3. **Rank 3**: Relevance = 3,

$$\text{Gain} = \frac{3}{\log_2(3+1)} = \frac{3}{2} = 1.5$$

4. **Rank 4**: Relevance = 0,

$$\text{Gain} = \frac{0}{\log_2(4+1)} = 0$$

5. **Rank 5**: Relevance = 0,

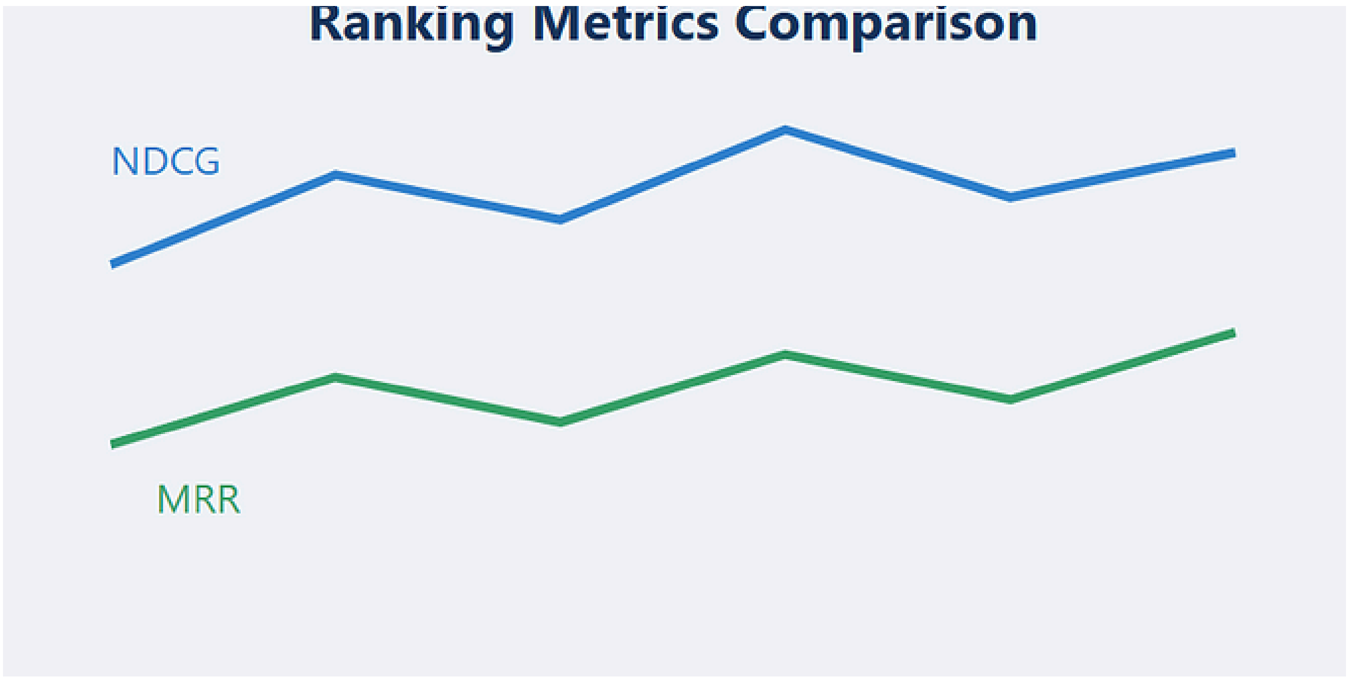
$$\text{Gain} = \frac{0}{\log_2(5+1)} = 0$$

Yan Xu

Overview: Learning to Rank

Overview

Jan 1 3

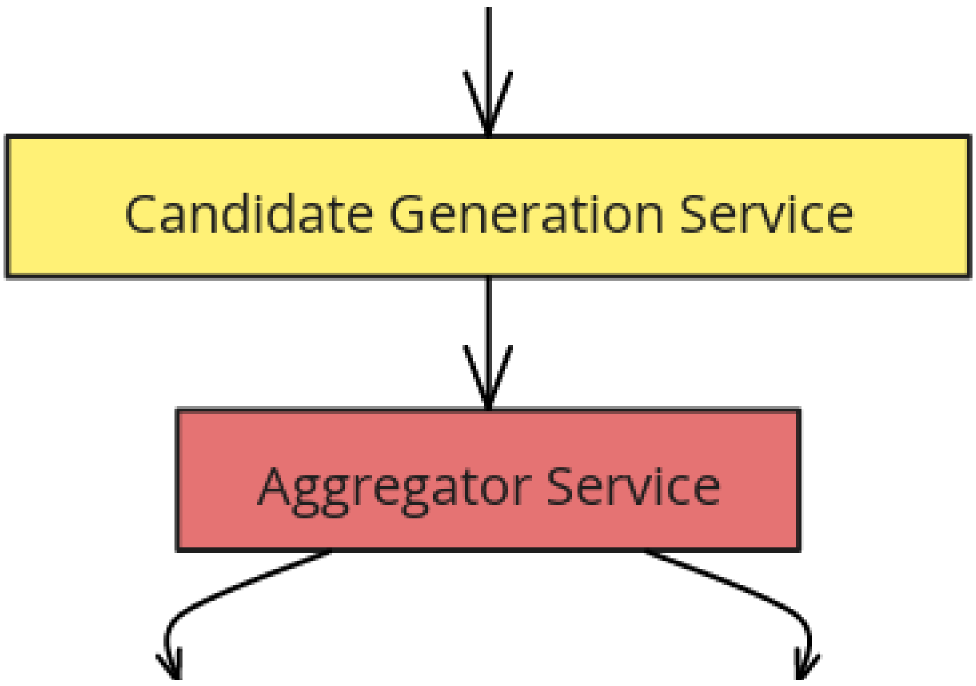


In Stackademic by BavalpreetSinghh

NDCG vs. MRR: Ranking Metrics for Information Retrieval in RAG's

Introduction

Dec 5, 2024 325

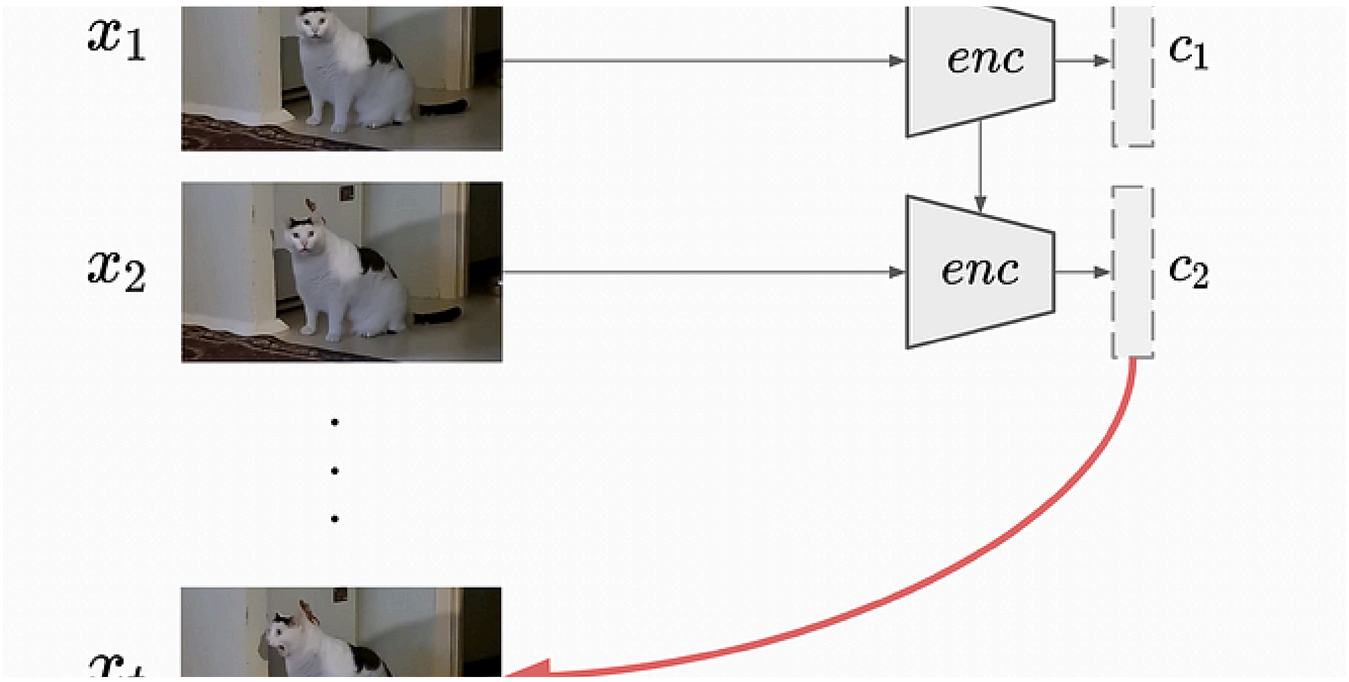


In NextGenAI by Prem Vishnoi(cloudvala)

Ad Click Prediction ML System Design

Introduction:

Dec 4, 2024 56 1



Allen Liang

Contrastive Learning for Beginners: InfoNCE Loss Explained in Details

InfoNCE, or Information Noise-Contrastive Estimation, is a cornerstone in contrastive learning. Proposed by van den Oord et al. in the 2018

May 3 1

See more recommendations