AYDIN ADNAN MENDERES UNIVERSITY
ENGINEERING FACULTY
COMPUTER ENGINEERING DEPARTMENT

MOVIELENS RECOMMENDATION SYSTEM

Ilhan BAYRAMOGLU
Kani SUNGUR

Supervisor:

Dr. Öğr. Üyesi Hüseyin ABACI

2021

AYDIN ADNAN MENDERES UNIVERSITY
ENGINEERING FACULTY
COMPUTER ENGINEERING DEPARTMENT

MOVIELENS RECOMMENDATION SYSTEM

İlhan BAYRAMOGLU
Kani SUNGUR

Supervisor:
Asst. Prof. Dr. Hüseyin ABACI

2021

ABSTRACT

MOVIELENS RECOMMENDATION SYSTEM

Kani SUNGUR

Ilhan BAYRAMOGLU

B.Sc. Thesis, Computer Engineering Department

Supervisor: Asst. Prof. Dr. Hüseyin ABACI

{2021}, {50} pages

A movie recommendation system filters the data using different algorithms and recommends the most relevant movies to users. It first captures the past behavior of a user and based on that, recommends movies which the users might be likely to watch. If a completely new user visits the site, that site will not have any past history of that user. So how does the site go about recommending movies to the user in such a scenario? One possible solution could be to recommend the most watched and/or high rated movies. Another possible solution could be to recommend the movies which would need more diverse outcomes to process for database. Two main approaches are used for our recommender systems; First one is content-based filtering, where we try to profile the users interests using information collected, and recommend movies based on that profile. The other is collaborative filtering, where we try to group similar users together and use information about the group to make recommendations to the user.

Keywords: Matrix Factorization, Collaborative Filtering, ALS, SVD, RMSE

ÖZET

MOVIELENS RECOMENDATION SYSTEM

Ilhan BAYRAMOGLU

Kani SUNGUR

Lisans Bitirme Tezi, Bilgisayar Mühendisliği Bölümü

Tez Danışmanı: Dr. Öğr. Üyesi Hüseyin ABACI

{2021}, {50} sayfa

Bir film öneri sistemi, verileri farklı algoritmalar kullanarak filtreler ve kullanıcılara en alakalı filmleri önerir. Önce bir kullanıcının geçmiş davranışlarını yakalar ve buna dayanarak, kullanıcıların izlemesi muhtemel olan filmleri önerir. Tamamen yeni bir kullanıcı siteyi ziyaret ederse, o sitede o kullanıcının geçmiş geçmişi bulunmayacaktır. Peki site böyle bir senaryoda kullanıcıya film önerme konusunda nasıl bir yol izliyor? Olası bir çözüm, en çok izlenen ve/veya yüksek puan alan filmleri önermek olabilir. Bir başka olası çözüm, veritabanı için işlenmesi için daha çeşitli sonuçlara ihtiyaç duyan filmleri önermek olabilir. Öneri sistemlerimiz için iki ana yaklaşım kullanılmaktadır; Birincisi, toplanan bilgileri kullanarak kullanıcıların ilgi alanlarını profillemeye çalıştığımız ve bu profile dayalı olarak filmler önerdiğimiz içerik tabanlı filtrelemedir. Diğeri, benzer kullanıcıları bir araya getirmeye ve kullanıcıya önerilerde bulunmak için grup hakkındaki bilgileri kullanmaya çalıştığımız işbirlikçi filtrelemedir.

Anahtar Kelimeler: Matrix Factorization, Collaborative Filtering, ALS, SVD, RMSE

ACKNOWLEDGEMENT

It gives us immense pleasure to express our deepest sense of gratitude and sincere thanks to our respected guide Dr. Hüseyin ABACI(Assistant Professor), for their valuable guidance, encouragement and help for completing this work. Their useful suggestions for this whole work and co-operative behavior are sincerely acknowledged. We are also grateful to our teachers Asst. Prof. Dr. Hüseyin ABACI, Asst. Prof. Dr. Fatih SOYGAZİ for their constant support and guidance.

TABLE OF CONTENTS

1.11 Neural Collaborative Filtering(NCF)

LIST OF TABLES

LIST OF FIGURES

LIST OF DIAGRAMS

LIST OF ABBREVIATIONS

SVD	Singular Value Decomposition
NGCF	Neural Graph collaborative filtering
ALS	Alternating Least Squares
RMSE	Root Mean Squared Error
CPU	Central Process Unit

1. INTRODUCTION

A recommendation system is a type of information filtering system which attempts to predict the preferences of a user, and make suggests based on these preferences. There are a wide variety of applications for recommendation systems. These have become increasingly popular over the last few years and are now utilized in most online platforms that we use. The content of such platforms varies from movies, music, books and videos, to friends and stories on social media platforms, to products on e-commerce websites, to people on professional and dating websites, to search results returned on Google. Often, these systems are able to collect information about a users choices, and can use this information to improve their suggestions in the future. For example, Facebook can monitor your interaction with various stories on your feed in order to learn what types of stories appeal to you. Sometimes, the recommender systems can make improvements based on the activities of a large number of people. For example, if Amazon observes that a large number of customers who buy the latest Apple Macbook also buy a USB-C-to USB Adapter, they can recommend the Adapter to a new user who has just added a Macbook to his/her cart. Due to the advances in recommender systems, users constantly expect good recommendations. They have a low threshold for services that are not able to make appropriate suggestions. If a music streaming app is not able to predict and play music that the user likes, then the user will simply stop using it. This has led to a high emphasis by tech companies on improving their recommendation systems. However, the problem is more complex than it seems. Every user has different preferences and likes. In addition, even the taste of a single user can vary depending on a large number of factors, such as mood, season, or type of activity the user is doing. For example, the type of music one would like to hear while exercising differs greatly from the type of music he'd listen to when cooking dinner. Another issue that recommendation systems have to solve is the exploration vs exploitation problem. They must explore new domains to discover more about the user, while still making the most of what is already known about of the user. Three main approaches are being used for recommender systems these days. One is Demographic Filtering which means They offer generalized recommendations to every user, based on movie popularity and/or genre. The System recommends the same movies to users with similar demographic features. Since each user is different ,

this approach is considered to be too simple. The basic idea behind this system is that movies that are more popular and critically acclaimed will have a higher probability of being liked by the average audience. Second is content-based filtering, where we try to profile the users interests using information collected, and recommend items based on that profile. The other is collaborative filtering, where we try to group similar users together and use information about the group to make recommendations to the user.

1.2 Recommendations: What and Why?

1.2.1 What are Recommendations?

How does YouTube know what video you might want to watch next? How does the Google Play Store pick an app just for you? Magic? No, in both cases, an ML-based recommendation model determines how similar videos and apps are to other things you like and then serves up a recommendation. Two kinds of recommendations are commonly used:

- home page recommendations
- related item recommendations

1.2.1.1 Homepage Recommendations

Homepage recommendations are personalized to a user based on their known interests. Every user sees different recommendations.

If you go to the Google Play Apps homepage, you may see something like this:

1.2.1.2 Related Item Recommendations

As the name suggests, related items are recommendations similar to a particular item. In the Google Play apps example, users looking at a page for a math app may also see a panel of related apps, such as other math or science apps.

1.2.2 Why Recommendations?

A recommendation system helps users find compelling content in a large corpora. For example, the Google Play Store provides millions of apps, while YouTube provides billions of videos. More apps and videos are added every day. How can users find new compelling new content? Yes, one can use search to access content. However, a recommendation engine can display items that users might not have thought to search for on their own.

1.3Terminology

Before we dive in, there are a few terms that you should know:

1.3.1 Items (also known as documents)

The entities a system recommends. For the Google Play store, the items are apps to install. For YouTube, the items are videos.

1.3.2 Query (also known as context)

The information a system uses to make recommendations. Queries can be a combination of the following:

- user information
    - the id of the user
    - items that users previously interacted with
- additional context
    - time of day
    - the user's device

1.3.3 Embedding

A mapping from a discrete set (in this case, the set of queries, or the set of items to recommend) to a vector space called the embedding space. Many recommendation systems rely on learning an appropriate embedding representation of the queries and items.

1.4 Recommendation Systems Overview

One common architecture for recommendation systems consists of the following components:

- candidate generation
- scoring
- re-ranking

1.4.1 Candidate Generation

In this first stage, the system starts from a potentially huge corpus and generates a much smaller subset of candidates. For example, the candidate generator in YouTube reduces billions of videos down to hundreds or thousands. The model needs to evaluate queries quickly given the enormous size of the corpus. A given model may provide multiple candidate generators, each nominating a different subset of candidates.

1.4.2 Scoring

Next, another model scores and ranks the candidates in order to select the set of items (on the order of 10) to display to the user. Since this model evaluates a relatively small subset of items, the system can use a more precise model relying on additional queries.

1.4.3 Re-ranking

Finally, the system must take into account additional constraints for the final ranking. For example, the system removes items that the user explicitly disliked or boosts the score of fresher content. Re-ranking can also help ensure diversity, freshness, and fairness.

We will discuss each of these stages over the course of the class and give examples from different recommendation systems, such as YouTube.

1.5Candidate Generation Overview

1.5.1 Embedding Space

Both content-based and collaborative filtering map each item and each query (or context) to an embedding vector in a common embedding space $E=R^d$. Typically, the embedding space is low-dimensional (that is, d is much smaller than the size of the corpus), and captures some latent structure of the item or query set. Similar items, such as YouTube videos that are usually watched by the same user, end up close together in the embedding space. The notion of "closeness" is defined by a similarity measure.

1.5.2 Similarity Measures

A similarity measure is a function $s:E\times E\rightarrow R$ that takes a pair of embeddings and returns a scalar measuring their similarity. The embeddings can be used for candidate generation as follows: given a query embedding $q\in E$, the system looks for item embeddings $x\in E$ that are close to q, that is, embeddings with high similarity s(q,x).

To determine the degree of similarity, most recommendation systems rely on one or more of the following:

- cosine
- dot product
- Euclidean distance

1.5.3 Cosine

This is simply the cosine of the angle between the two vectors, $s(q,x)=\cos(q,x)$

1.5.3.1 Dot Product

The dot product of two vectors is $s(q,x)=\langle q,x\rangle=\sum_{i=1}^{d} q_i x_i$. It is also given by $s(q,x)=\|x\|\|q\|\cos(q,x)$ (the cosine of the angle multiplied by the product of norms). Thus, if the embeddings are normalized, then dot-product and cosine coincide.

1.5.3.2 Euclidean distance

This is the usual distance in Euclidean space, $s(q,x)=\|q-x\|=[\sum_{i=1}^{d}(q_i-x_i)^2]^{\frac{1}{2}}$. A smaller distance means higher similarity. Note that when the embeddings are

normalized, the squared Euclidean distance coincides with dot-product (and cosine) up to a constant, since in that case $12\|q-x\|2=1-\langle q,x\rangle$.

1.5.4 Comparing Similarity Measures

Consider the example in the figure to the right. The black vector illustrates the query embedding. The other three embedding vectors (Item A, Item B, Item C) represent candidate items. Depending on the similarity measure used, the ranking of the items can be different.

Using the image, try to determine the item ranking using all three of the similarity measures: cosine, dot product, and Euclidean distance.

1.5.5 Which Similarity Measure to Choose?

Compared to the cosine, the dot product similarity is sensitive to the norm of the embedding. That is, the larger the norm of an embedding, the higher the similarity (for items with an acute angle) and the more likely the item is to be recommended. This can affect recommendations as follows:

- Items that appear very frequently in the training set (for example, popular YouTube videos) tend to have embeddings with large norms. If capturing popularity information is desirable, then you should prefer dot product. However, if you're not careful, the popular items may end up dominating the recommendations. In practice, you can use other variants of similarity measures that put less emphasis on the norm of the item. For example, define $s(q,x)=\|q\|\alpha\|x\|\alpha\cos f_0(q,x)$ for some $\alpha\in(0,1)$.
- Items that appear very rarely may not be updated frequently during training. Consequently, if they are initialized with a large norm, the system may recommend rare items over more relevant items. To avoid this problem, be careful about embedding initialization, and use appropriate regularization. We will detail this problem in the first exercise.

1.6 Content-based Filtering

Content-based filtering uses item features to recommend other items similar to what the user likes, based on their previous actions or explicit feedback.

To demonstrate content-based filtering, let's hand-engineer some features for the Google Play store. The following figure shows a feature matrix where each row represents an app and each column represents a feature. Features could include categories (such as Education, Casual, Health), the publisher of the app, and many others. To simplify, assume this feature matrix is binary: a non-zero value means the app has that feature.

You also represent the user in the same feature space. Some of the user-related features could be explicitly provided by the user. For example, a user selects "Entertainment apps" in their profile. Other features can be implicit, based on the apps they have previously installed. For example, the user installed another app published by Science R Us.

The model should recommend items relevant to this user. To do so, you must first pick a similarity metric (for example, dot product). Then, you must set up the system to score each candidate item according to this similarity metric. Note that the recommendations are specific to this user, as the model did not use any information about other users.

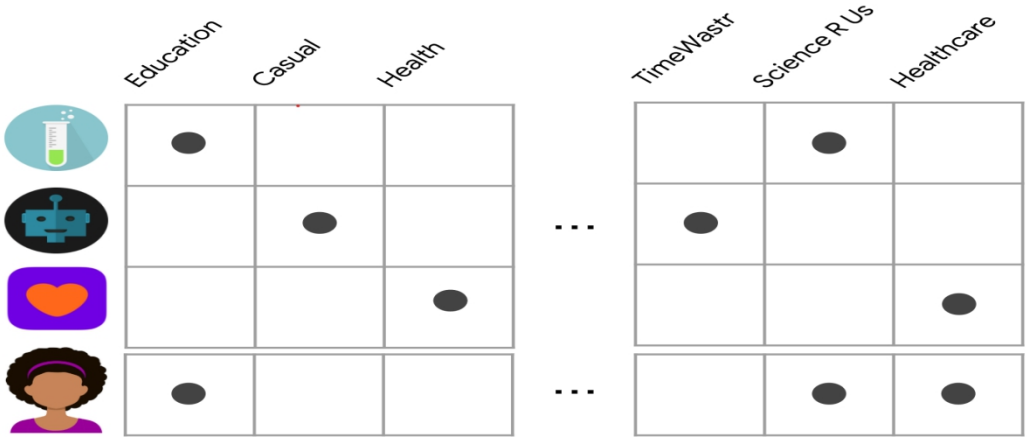


Figure 1. User dot product

1.6.1 Using Dot Product as a Similarity Measure

Consider the case where the user embedding x and the app embedding y are both binary vectors. Since $\langle x,y\rangle = \sum_{i=1}^{d} x_i y_i$, a feature appearing in both x and y contributes a 1 to the sum. In other words, $\langle x,y\rangle$ is the number of features that are active in both vectors simultaneously. A high dot product then indicates more common features, thus a higher similarity.

1.6.2 Content-based Filtering Advantages & Disadvantages

1.6.2.1 Advantages

- The model doesn't need any data about other users, since the recommendations are specific to this user. This makes it easier to scale to a large number of users.
- The model can capture the specific interests of a user, and can recommend niche items that very few other users are interested in.

1.6.2.2 Disadvantages

- Since the feature representation of the items are hand-engineered to some extent, this technique requires a lot of domain knowledge. Therefore, the model can only be as good as the hand-engineered features.
- The model can only make recommendations based on existing interests of the user. In other words, the model has limited ability to expand on the users' existing interests.

1.7 Collaborative Filtering

To address some of the limitations of content-based filtering, collaborative filtering uses *similarities between users and items simultaneously* to provide recommendations. This allows for serendipitous recommendations; that is, collaborative filtering models can recommend an item to user A based on the interests of a similar user B. Furthermore, the embeddings can be learned automatically, without relying on hand-engineering of features.

1.7.1 A Movie Recommendation Example

Consider a movie recommendation system in which the training data consists of a feedback matrix in which:

- Each row represents a user.
- Each column represents an item (a movie).

The feedback about movies falls into one of two categories:

- **Explicit -** users specify how much they liked a particular movie by providing a numerical rating.
- **Implicit -** if a user watches a movie, the system infers that the user is interested.

To simplify, we will assume that the feedback matrix is binary; that is, a value of 1 indicates interest in the movie.

When a user visits the homepage, the system should recommend movies based on both:

- similarity to movies the user has liked in the past
- movies that similar users liked

For the sake of illustration, let's hand-engineer some features for the movies described in the following table:

Movie	Rating	Description
The Dark Knight Rises	PG-13	Batman endeavors to save Gotham City from nuclear annihilation in this sequel to The Dark Knight, set in the DC Comics universe.
Harry Potter and the Sorcerer's Stone	PG	A orphaned boy discovers he is a wizard and enrolls in Hogwarts School of Witchcraft and Wizardry, where he wages his first battle against the evil Lord Voldemort.
Shrek	PG	A lovable ogre and his donkey sidekick set off on a mission to rescue Princess Fiona, who is emprisoned in her castle by a dragon.
The Triplets of Belleville	PG-13	When professional cycler Champion is kidnapped during the Tour de France, his grandmother and overweight dog journey overseas to rescue him, with the help of a trio of elderly jazz singers.
Memento	R	An amnesiac desperately seeks to solve his wife's murder by tattooing clues onto his body.

Table 1. Features for the movies

1.7.2 1D Embedding

Suppose we assign to each movie a scalar in [−1,1] that describes whether the movie is for children (negative values) or adults (positive values). Suppose we also assign a scalar to each user in [−1,1] that describes the user's interest in children's movies (closer to -1) or adult movies (closer to +1). The product of the movie embedding and the user embedding should be higher (closer to 1) for movies that we expect the user to like.

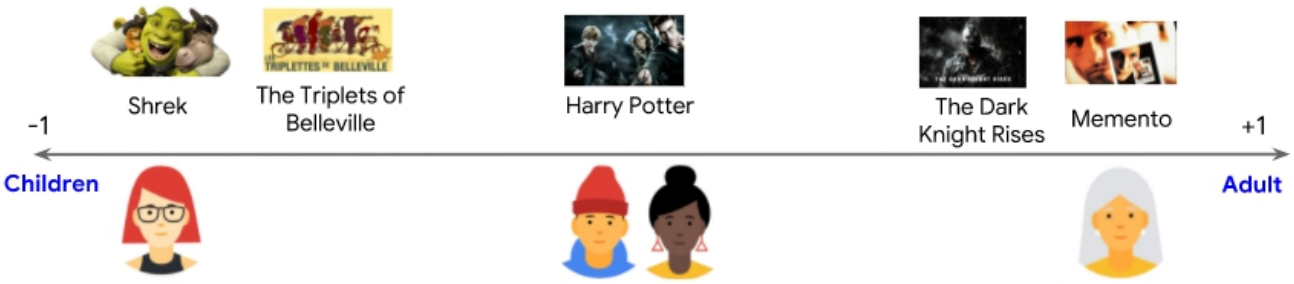


Diagram 1. A scalar for children or adults

In the diagram below, each checkmark identifies a movie that a particular user watched. The third and fourth users have preferences that are well explained by this feature—the third user prefers movies for children and the fourth user prefers movies for adults. However, the first and second users' preferences are not well explained by this single feature.

Figure 2. Movie that a particular user watched

1.7.3 2D Embedding

One feature was not enough to explain the preferences of all users. To overcome this problem, let's add a second feature: the degree to which each movie is a blockbuster or an arthouse movie. With a second feature, we can now represent each movie with the following two-dimensional embedding:

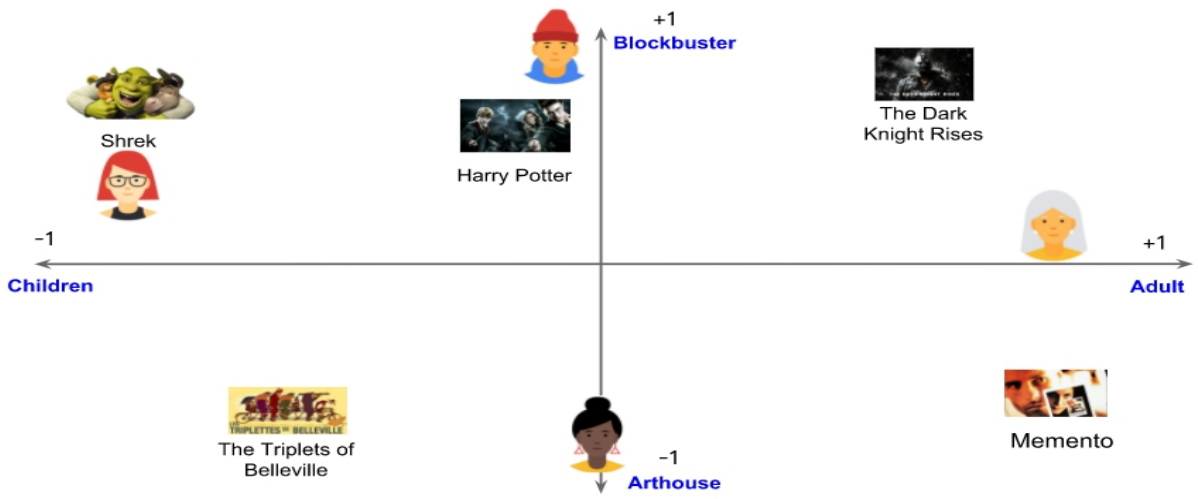


Diagram 2. Two-dimensional embedding

We again place our users in the same embedding space to best explain the feedback matrix: for each (user, item) pair, we would like the dot product of the user embedding and the item embedding to be close to 1 when the user watched the movie, and to 0 otherwise.

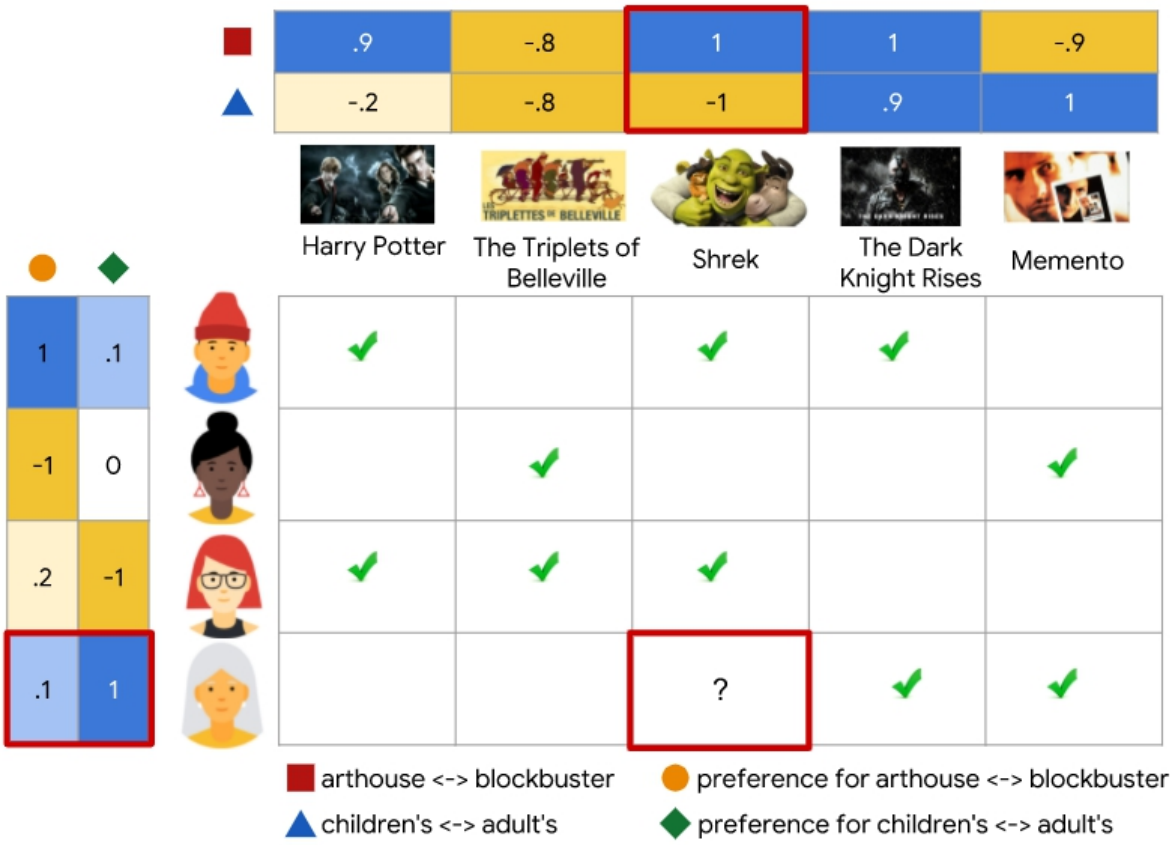


Figure 3. Feedback matrix

In this example, we hand-engineered the embeddings. In practice, the embeddings can be learned *automatically*, which is the power of collaborative filtering models. In the next two sections, we will discuss different models to learn these embeddings, and how to train them.

The collaborative nature of this approach is apparent when the model learns the embeddings. Suppose the embedding vectors for the movies are fixed. Then, the model can learn an embedding vector for the users to best explain their preferences. Consequently, embeddings of users with similar preferences will be close together. Similarly, if the embeddings for the users are fixed, then we can learn movie embeddings to best explain the feedback matrix. As a result, embeddings of movies liked by similar users will be close in the embedding space.

1.8 Matrix Factorization

In collaborative filtering, **matrix factorization** is the state-of-the-art solution for sparse data problem, although it has become widely known since *Netflix Prize Challenge*.

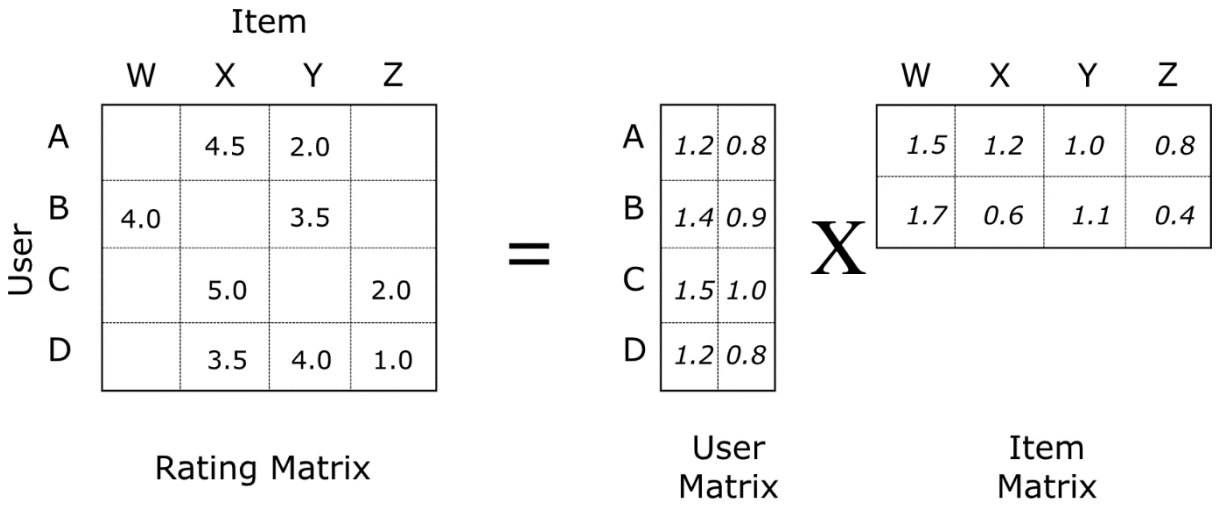


Figure 4. Item, User, Rating Matrix.

What is matrix factorization? Matrix factorization is simply a family of mathematical operations for matrices in linear algebra. To be specific, a matrix factorization is a factorization of a matrix into a product of matrices. In the case of collaborative filtering, **matrix factorization** algorithms work by **decomposing** the user-item interaction matrix into the product of two **lower dimensionality rectangular matrices**. One matrix can be seen as the user matrix where rows represent users and columns are latent factors. The other matrix is the item matrix where rows are latent factors and columns represent items.

How does matrix factorization solve our problems?

1. Model learns to factorize rating matrix into user and movie representations, which allows model to predict better personalized movie ratings for users

2. With matrix factorization, less-known movies can have rich latent representations as much as popular movies have, which improves recommender's ability to recommend less-known movies

In the sparse user-item interaction matrix, the predicted rating user u will give item i is computed as:

$$\tilde{r}_{ui} = \sum_{f=0}^{nfactors} H_{u,f} W_{f,i}$$

Rating of item i given by user u can be expressed as a dot product of the user latent vector and the item latent vector.

Notice in above formula, the number of **latent factors** can be tuned via cross-validation. **Latent factors** are the features in the lower dimension latent space projected from user-item interaction matrix. The idea behind matrix factorization is to use latent factors to represent user preferences or movie topics in a much lower dimension space. Matrix factorization is one of very effective **dimension reduction** techniques in machine learning.

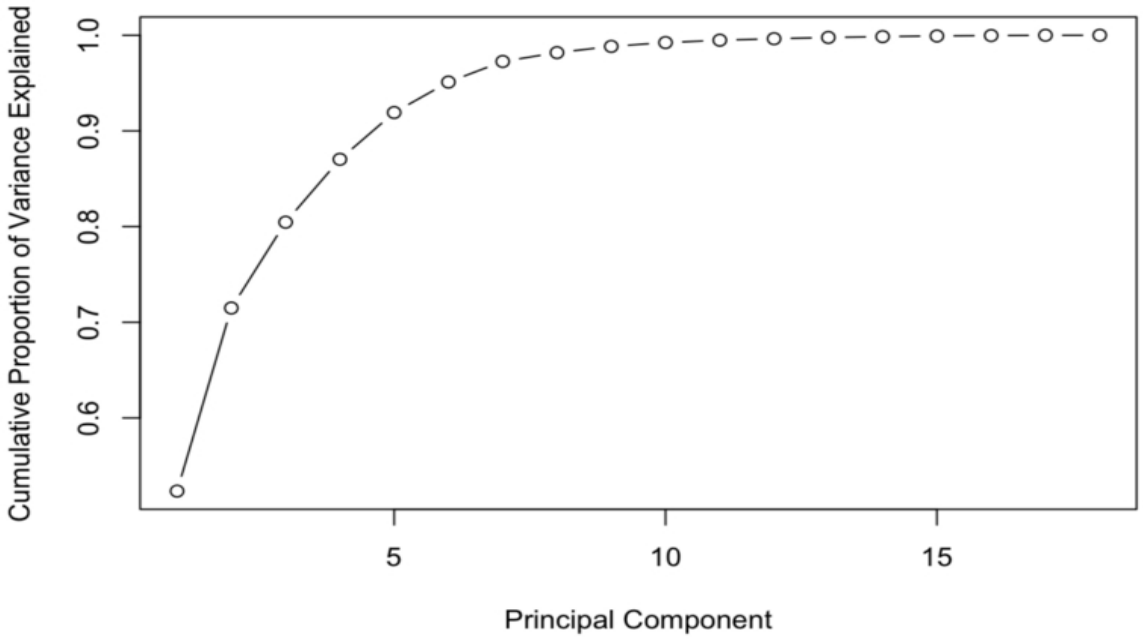


Diagram 3. Principal Component

Very much like the concept of **components** in **PCA**, the number of latent factors determines the amount of abstract information that we want to store in a lower dimension space. A matrix factorization with one latent factor is equivalent to a *most popular* or *top popular* recommender (e.g. recommends the items with the most interactions without any personalization). Increasing the number of latent factors will improve personalization, until the number of factors becomes too high, at which point the model starts to overfit. A common strategy to avoid overfitting is to add **regularization terms** to the objective function.

The objective of matrix factorization is to minimize the error between true rating and predicted rating:

$$\underset{H,W}{\arg\min} \left\| R - \tilde{R} \right\|_{\mathrm{F}} + \alpha \left\| H \right\| + \beta \left\| W \right\|$$

where H is user matrix, W is item matrix

Once we have an objective function, we just need a training routine (eg, gradient descent) to complete the implementation of a matrix factorization algorithm. This implementation is actually called **Funk SVD**. It is named after Simon Funk, who he shared his findings with the research community during Netflix prize challenge in 2006.

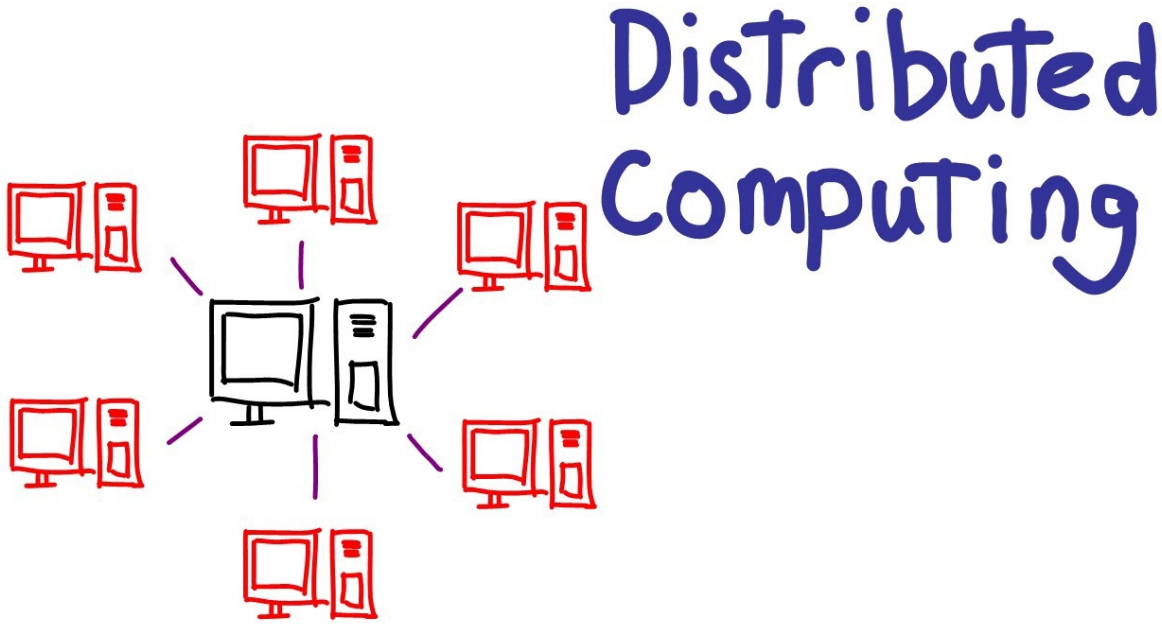


Figure 5. Distributed computing.

Although Funk SVD was very effective in matrix factorization with single machine during that time, it's not scalable as the amount of data grows today. With terabytes or even petabytes of data, it's impossible to load data with such size into a single machine. So we need a machine learning model (or framework) that can train on dataset spreading across from cluster of machines.

1.9 Alternating Least Square (ALS) with Spark ML

Alternating Least Square (ALS) is also a matrix factorization algorithm and it runs itself in a parallel fashion. ALS is implemented in Apache Spark ML and built for a larges-scale collaborative filtering problems. ALS is doing a pretty good job at solving scalability and sparseness of the Ratings data, and it's simple and scales well to very large datasets.

Some high-level ideas behind ALS are:

- Its objective function is slightly different than Funk SVD: ALS uses **L2** regularization while Funk uses L1 regularization
- Its training routine is different: ALS minimizes two loss functions alternatively; It first holds user matrix fixed and runs gradient descent with item matrix; then it holds item matrix fixed and runs gradient descent with user matrix
- Its scalability: ALS runs its gradient descent in **parallel** across multiple partitions of the underlying training data from a cluster of machines

Table 2. SGD Algorithm for MF

SGD Algorithm for MF

Input: training matrix V, the number of features K, regularization parameter λ, learning rate ϵ
Output: row related model matrix W and column related model matrix H

1: Initialize W, H to $UniformReal(0, \frac{1}{\sqrt{K}})$
2: **repeat**
3: **for random** $V_{ij} \in V$ **do**
4: $error = W_{i*}H_{*j} - V_{ij}$
5: $W_{i*} = W_{i*} - \epsilon(error \cdot H_{*j}^{\intercal} + \lambda W_{i*})$
6: $H_{*j} = H_{*j} - \epsilon(error \cdot W_{i*}^{\intercal} + \lambda H_{*j})$
7: **end for**
8: **until** convergence

If you are interested in learning more about ALS, you can find more details in this paper: *Large-scale Parallel Collaborative Filtering for the Netflix Prize*

Just like other machine learning algorithms, ALS has its own set of hyper-parameters. We probably want to tune its hyper-parameters via hold-out validation or cross-validation.

Most important hyper-params in Alternating Least Square (ALS):

- maxIter: the maximum number of iterations to run (defaults to 10)
- rank: the number of latent factors in the model (defaults to 10)
- regParam: the regularization parameter in ALS (defaults to 1.0)
- Hyper-parameter tuning is a highly recurring task in many machine learning projects. We can code it up in a function to speed up the tuning iterations.

1.9.1 Implementing ALS Recommender System

Now that we know we have a wonderful model for movie recommendation, the next question is: how do we take our wonderful model and productize it into a recommender system? Machine learning model productization is another big topic and I won't get into details about it. In this post, I will show how to build a MVP (minimum viable product) version for ALS recommender.

To productize a model, we need to build a work flow around the model. Typical ML work flow roughly starts with data preparation via pre-defined set of ETL jobs, offline/online model training, then ingesting trained models to web services for production. In our case, we are going to build a very minimum version of movie recommender that just does the job. Our work flow is following:

1. A new user inputs his/her favorite movies, then system create new user-movie interaction samples for the model
2. System retrains ALS model on data with the new inputs.
3. System creates movie data for inference (in my case, I sample all movies from the data)
4. System make rating predictions on all movies for that user.

5. System outputs top N movie recommendations for that user based on the ranking of movie rating predictions.

1.10 Singular Value Decomposition (SVD)

A well-known matrix factorization method is Singular value decomposition (SVD). Collaborative Filtering can be formulated by approximating a matrix X by using singular value decomposition. The winning team at the Netflix Prize competition used SVD matrix factorization models to produce product recommendations, for more information I recommend to read articles: Netflix Recommendations: Beyond the 5 stars and Netflix Prize and SVD. The general equation can be expressed as follows:

$$X = USV^T$$

Given *m x n* matrix *x*:

- *u* is an *(mxr)* orthogonal matrix
- *s* is an *(rxr)* diagonal matrix with non-negative real numbers on the diagonal
- *V^ T* is an *(rxn)* orthogonal matrix

Elements on the diagonal in *s* are known as *singular values of x.*

Matrix *x* can be factorized to *u*, *s* and *v.* The *u* matrix represents the feature vectors corresponding to the users in the hidden feature space and the V matrix represents the feature vectors corresponding to the items in the hidden feature space.

1.11 Neural Collaborative Filtering(NCF)

Neural Collaborative Filtering(NCF) replaces the user-item inner product with a neural architecture. By doing so NCF tried to achieve the following:

1- NCF tries to express and generalize MF under its framework.

2- NCF tries to learn User-item interactions through a multi-layer perceptron.

Despite the effectiveness of matrix factorization for collaborative filtering, it's performance is hindered by the simple choice of interaction function - inner product. Its performance can be improved by incorporating user-item bias terms into the interactiion function. This proves that the simple multiplication of latent features (inner product), may not be sufficient to capture the complex structure of user interaction data.

This calls for designing a better, dedicated interaction function for modeling the latent feature interaction between users and items. Neural Collaborative Filtering (NCF) aims to solve this by:-

1. Modeling user-item feature interaction through neural network architecture. It utilizes a Multi-Layer Perceptron(MLP) to learn user-item interactions. This is an upgrade over MF as MLP can (theoretically) learn any continuous function and has high level of nonlinearities(due to multiple layers) making it well endowed to learn user-item interaction function.

2. Generalizing and expressing MF as a special case of NCF. As MF is highly successful in the recommendation domain, doing this will give more credence to NCF.

1.11.1 Problem Statement

Given a set of users U = {u = 1, …, U}, a set of items I = {i = 1, …, I}, and a log of the users' past preferences of items O = (u, i, y), our goal is to recommend to each user u a ranked list of items that will maximize her/his satisfaction. y can be either 1(Case-1) or 0(Case-2).

Case 1: Observed entries:
It means the user u, have interacted with the item i, but does not mean that u like i.

Case 2: Unobserved entries:
It does not mean the user u dislike item i. Unobserved entries could be just missing data.

Intuitively speaking the recommendation algorithms estimates the scores of unobserved entries in Y, which are used for ranking the items.

Formally speaking they calculate

$$\hat{y}_{ui} = f(u, i|\Theta)$$

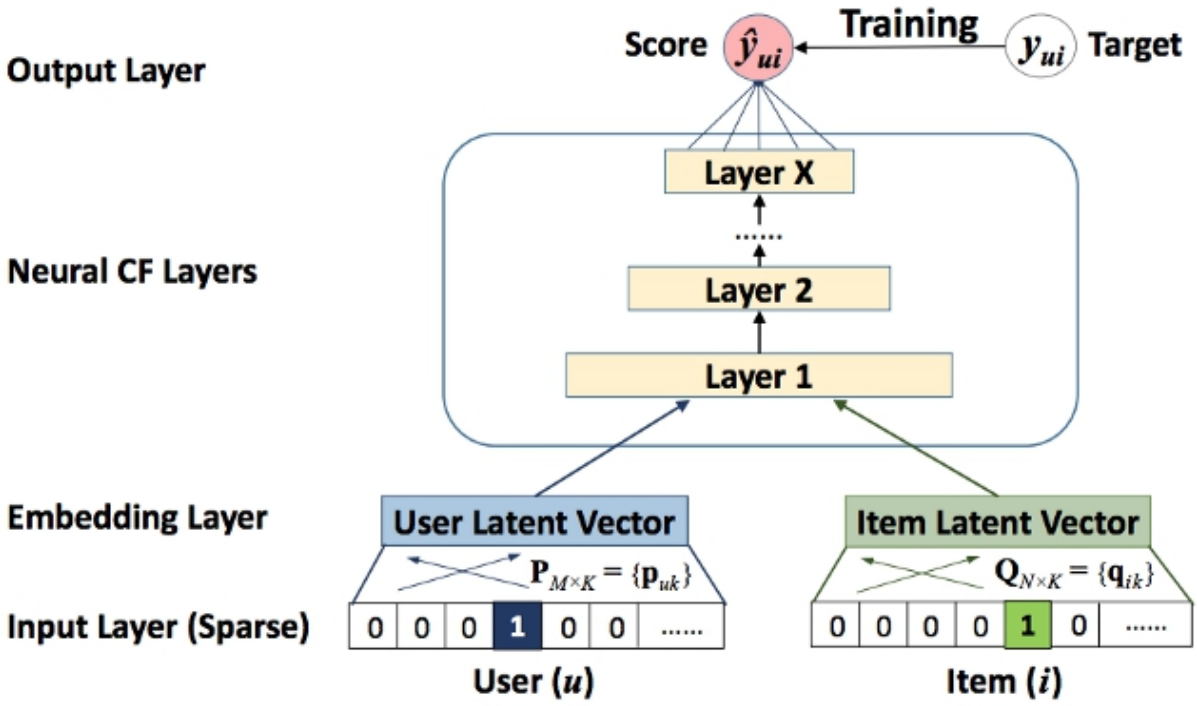
y(u,i): predicted score for interaction between user u and item i
theta: model parameters
f(Interaction Function): maps model parameters to the predicted score

In order to calculate theta, an objective function needs to be optimized. The 2 most popular loss functions for the recommendation system are a pointwise and pairwise loss.

1. Pointwise loss follows a regression framework by minimizing the squared loss between predicted score y_(u,i) and target value y(u,i). In order to account for negative feedback either all unobserved entries are considered as negative examples or some unobserved entries are sampled to be negative instances.
2. Pairwise loss aims to rank observed entries higher than the unobserved ones. It achieves this by maximizing the margin between observed entries and unobserved entries.

To summarise, pairwise loss maximizes the margin between observed and unobserved entries in contrast to pointwise loss which aims to minimize the loss between predicted and target score.

1.11.2 NCF Architecture



1. Input Layer binarise a sparse vector for a user and item identification where:
Item (i): 1 means the user u has interacted with Item(i)
User (u): To identify the user

2. Embedding layer is a fully connected layer that projects the sparse representation to a dense vector. The obtained user/item embeddings are the latent user/item vectors.

3. Neural CF layers use Multi-layered neural architecture to map the latent vectors to prediction scores.

4. The final output layer returns the predicted score by minimizing the pointwise loss/pairwise loss.

NCF modifies equation 1 in the following way:

$$\hat{y}_{ui} = f(\mathbf{P}^T \mathbf{v}_u^U, \mathbf{Q}^T \mathbf{v}_i^I | \mathbf{P}, \mathbf{Q}, \Theta_f),$$

P: Latent factor matrix for users (Size=M * K)

Q: Latent factor matrix for items (Size=N * K)

Theta(f): Model parameters

Since f is formulated as MLP it can be expanded as

$$f(\mathbf{P}^T \mathbf{v}_u^U, \mathbf{Q}^T \mathbf{v}_i^I) = \phi_{out}(\phi_X(...\phi_2(\phi_1(\mathbf{P}^T \mathbf{v}_u^U, \mathbf{Q}^T \mathbf{v}_i^I))...)), \quad (4)$$

Psi (out): mapping function for the output layer

Psi (x): mapping function for the x-th neural collaborative filtering layer

1.11.3 NCF's loss function

Pointwise squared loss equation is represented as

$$L_{sqr} = \sum_{(u,i) \in \mathcal{Y} \cup \mathcal{Y}^-} w_{ui} (y_{ui} - \hat{y}_{ui})^2,$$

y: observed interaction in Y

y negative: all/sample of unobserved interactions

w(u,i): the weight of training instance (hyperparameter)

The squared loss can be explained if we assume that the observations are from a Gaussian distribution which in our case is not true. Plus the prediction

score y_carat should return a score between [0,1] to represent the likelihood of the given user-item interaction. In short, we need a probabilistic approach for learning the pointwise NCF that pays special attention to the binary property of implicit data.

With the above settings, the likelihood function is defined as :

$$p(\mathcal{Y}, \mathcal{Y}^- | \mathbf{P}, \mathbf{Q}, \Theta_f) = \prod_{(u,i) \in \mathcal{Y}} \hat{y}_{ui} \prod_{(u,j) \in \mathcal{Y}^-} (1 - \hat{y}_{uj}).$$

Taking the negative log of the likelihood function

$$\begin{aligned} L &= - \sum_{(u,i) \in \mathcal{Y}} \log \hat{y}_{ui} - \sum_{(u,j) \in \mathcal{Y}^-} \log(1 - \hat{y}_{uj}) \\ &= - \sum_{(u,i) \in \mathcal{Y} \cup \mathcal{Y}^-} y_{ui} \log \hat{y}_{ui} + (1 - y_{ui}) \log(1 - \hat{y}_{ui}). \end{aligned}$$

which is nothing but the cross-entropy loss/log loss.

To account for negative instances y- is uniformly sampled from the unobserved interactions.

In the next segment, we can see how GMF(a component of NCF) generalizes the MF framework

1.11.4 Generalized Matrix Factorization (GMF)

The predicted output of the NCF can be expressed as

$$\hat{y}_{ui} = a_{out}(\mathbf{h}^T(\mathbf{p}_u \odot \mathbf{q}_i)),$$

a-out: activation function

h: edge weights of the output layer

We can play with a-out and h to create multiple variations of GMF.

	activation function	edge weights	variation
1	Identity function	1	Matrix factorisation
2	Identity function	learnable parameter	MF with varying importance of latent dimensions
3	non linear function	learnable parameter	Non-linear MF which is more expressive

As you can see from the above table that GMF with identity activation function and edge weights as 1 is indeed MF. The other 2 variations are expansions on the generic MF. The last variation of GMF with sigmoid as activation is used in NCF.

In the next segment, we will explain how NCF tries to model the user-item interaction using MLP

1.11.5 Multi-Layer Perceptron (MLP)

NCF is an example of multi modal deep learning as it contains data from 2 pathways namely user and item. The most intuitive way to combine them is by concatenation. But a simple vector concatenation does not account for user-item interactions and is insufficient to model the collaborative filtering effect. To address this NCF adds hidden layers on top of concatenated user-item vectors(MLP framework), to learn user-item interactions. This endows the model with a lot of

flexibility and non-linearity to learn the user-item interactions. This is an upgrade over MF that uses a fixed element-wise product on them. More precisely, the MLP alter Equation 1 as follows

$$\mathbf{z}_1 = \phi_1(\mathbf{p}_u, \mathbf{q}_i) = \begin{bmatrix} \mathbf{p}_u \\ \mathbf{q}_i \end{bmatrix},$$

$$\phi_2(\mathbf{z}_1) = a_2(\mathbf{W}_2^T \mathbf{z}_1 + \mathbf{b}_2),$$

$$\ldots\ldots$$

$$\phi_L(\mathbf{z}_{L-1}) = a_L(\mathbf{W}_L^T \mathbf{z}_{L-1} + \mathbf{b}_L),$$

$$\hat{y}_{ui} = \sigma(\mathbf{h}^T \phi_L(\mathbf{z}_{L-1})),$$

W(x): Weight matrix

b(x): bias vector

a(x): activation function for the x-th layer's perceptron

p: latent vector for the user

q: latent vector for an item

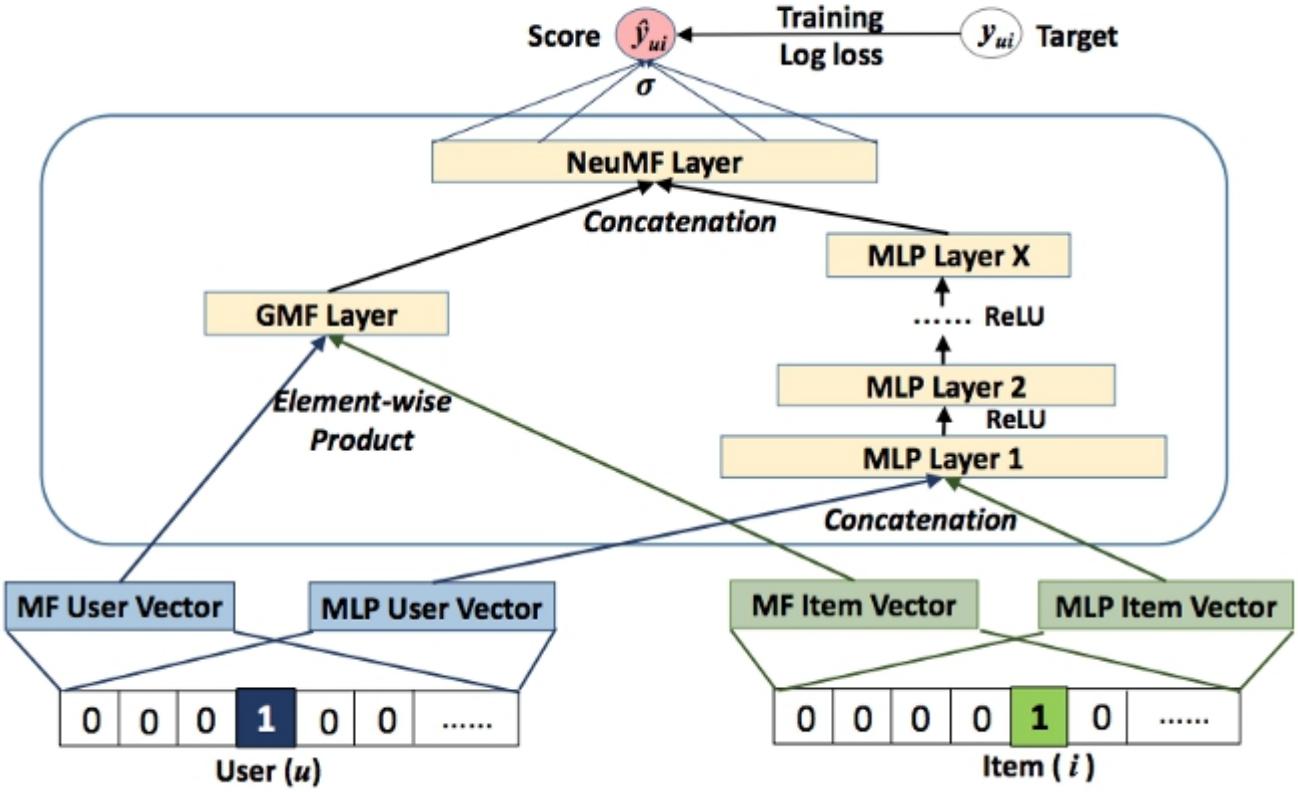
Due to multiple hidden layers, the model has sufficient complexity to learn user-item interactions as compared to the fixed element-wise product of their latent vectors(MF way).

1.11.6 NeuMF: A fusion of GMF and MLP

NCF has 2 components GMF and MLP with the following benefits

1. GMF that applies the linear kernel to model user-item interactions like vanilla MF
2. MLP that uses multiple neural layers to layer nonlinear interactions

NCF combines these models together to superimpose their desirable characteristics. NCF concatenates the output of GMF and MLP before feeding them into NeuMF layer.



Important points to notice

1. GMF/MLP have separate user and item embeddings. This is to make sure that both of them learn optimal embeddings independently.

2. GMF replicates the vanilla MF by element-wise product of the user-item vector.

3. MLP takes the concatenation of user-item latent vectors as input.

4. The outputs of GMF and MLP are concatenated in the final NeuMF(Neural Matrix Factorisation) layer.

The score function of equation 1 is modeled as

$$\phi^{GMF} = \mathbf{p}_u^G \odot \mathbf{q}_i^G,$$

$$\phi^{MLP} = a_L(\mathbf{W}_L^T(a_{L-1}(...a_2(\mathbf{W}_2^T \begin{bmatrix} \mathbf{p}_u^M \\ \mathbf{q}_i^M \end{bmatrix} + \mathbf{b}_2)...)) + \mathbf{b}_L),$$

$$\hat{y}_{ui} = \sigma(\mathbf{h}^T \begin{bmatrix} \phi^{GMF} \\ \phi^{MLP} \end{bmatrix}),$$

G: GMF

M: MLP

p: User embedding

q: Item embedding

This model combines the linearity of MF and non-linearity of DNNs for modeling user-item latent structures through the NeuMF (Neural Matrix Factorisation) layer.

Due to the non-convex objective function of NeuMF,gradient-based optimization methods can only find locally-optimal solutions. This could be solved by good weight initializations. To solve this NCF initializes GMF and MLP with pre-trained models. There are 2 ways to do this

1. Random Initialisation
    1. Train GMF+MLP with random initializations until convergence.
    2. Use model parameters of 1 to initialize NCF.
    3. The weights of the two models are concatenated for the output layer as

2. GMF + MLP from scratch

$$\mathbf{h} \leftarrow \begin{bmatrix} \alpha \mathbf{h}^{GMF} \\ (1-\alpha)\mathbf{h}^{MLP} \end{bmatrix}$$

h(GMF): h vector of the pre-trained GMF

h(MLP): h vector of the pre-trained MLP

alpha: Hyper-parameter determining the trade-off between the 2 pre-trained models

1. GMF + MLP from scratch

   1. Adaptive Moment Estimation (Adam) adapts the learning rate for each parameter by performing smaller updates for frequent and larger updates for infrequent parameters. The Adam method yields faster convergence for both models than the vanilla SGD and relieves the pain of tuning the learning rate.

   2. After feeding pre-trained parameters into NeuMF, we optimize it with the vanilla SGD, rather than Adam. Adam needs to save momentum information for updating parameters. As the initialization with pre-trained networks does not store momentum information.

1.12 What is Root Mean Square Error (RMSE)?

Root mean square error or root mean square deviation is one of the most commonly used measures for evaluating the quality of predictions. It shows how far predictions fall from measured true values using Euclidean distance.

To compute RMSE, calculate the residual (difference between prediction and truth) for each data point, compute the norm of residual for each data point, compute the mean of residuals and take the square root of that mean. RMSE is commonly used in supervised learning applications, as RMSE uses and needs true measurements at each predicted data point.

Root mean square error can be expressed as

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} \|y(i) - \hat{y}(i)\|^2}{N}},$$

where N is the number of data points, y(i) is the i-th measurement, and y (i) is its corresponding prediction.

Note: RMSE is NOT scale invariant and hence comparison of models using this measure is affected by the scale of the data. For this reason, RMSE is commonly used over standardized data.

1.11.2 Why is Root Mean Square Error (RMSE) Important?

In machine learning, it is extremely helpful to have a single number to judge a model's performance, whether it be during training, cross-validation, or monitoring after deployment. Root mean square error is one of the most widely used measures for this. It is a proper scoring rule that is intuitive to understand and compatible with some of the most common statistical assumptions.

Note: By squaring errors and calculating a mean, RMSE can be heavily affected by a few predictions which are much worse than the rest. If this is undesirable, using the absolute value of residuals and/or calculating median can give a better idea of how a model performs on most predictions, without extra influence from unusually poor predictions.

1.13 Problems we got,and their respective solutions:

13.1 Als

At early stages of ALS with big data(27m movielens data in our situation),we encountered one problem,which bothered us badly and took some time to solve:

- Errors caused by the small data set and big data set word differences.We were getting error:

```
TypeError: can't multiply sequence by non-int of type 'float'
```

- Here came different types of variables from integer. Solution was correcting it by deleting title and genres.

- Our biggest problem with ALS was running times,and we couldn't lower it without some sacrifice in accuracy at this setup of hardware and time limits. Our initial ranks were [10, 50, 100, 150] and our parametergrids were [.01, .05, .1, .15].

- Since we couldn't handle running times, we reduced both ranks and parametergrids.

```
In [15]: # Import the requisite items
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

# Add hyperparameters and their respective values to param_grid
param_grid = ParamGridBuilder() \
             .addGrid(als.rank, [10, 15, 20, 25]) \
             .addGrid(als.regParam, [.01, .015, .02, .025]) \
             .build()
             #              .addGrid(als.maxIter, [10, 15, 20, 25]) \

# Define evaluator as RMSE and print length of evaluator
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")
print ("Num models to be tested: ", len(param_grid))
executed in 19ms, finished 14:21:33 2021-06-08
Num models to be tested:  16

In [16]: # Build cross validation using CrossValidator
cv = CrossValidator(estimator=als, estimatorParamMaps=param_grid, evaluator=evaluator, numFolds=5)

# Confirm cv was built
print(cv)
executed in 10ms, finished 14:21:35 2021-06-08
CrossValidator_a4a2215dad56

In [17]: #Fit cross validator to the 'train' dataset
model = cv.fit(train)

#Extract best model from the cv model above
best_model = model.bestModel
executed in 3h 35m 14s, finished 17:56:52 2021-06-08
```

- Second problem was memory issue as it follows:

Py4JJavaError: An error occurred while calling o164.fit.

: org.apache.spark.SparkException: Job aborted due to stage failure: Task 2 in stage 19.0 failed 1 times, most recent failure: Lost task 2.0 in stage 19.0 (TID 894, DESKTOP-HC49LQF, executor driver): java.lang.OutOfMemoryError: ***Java heap space***

Root cause of problem was java's default memory allocation on a notebook proportional to system's available total physical memory. Which were 2048 megabytes. Since data we work with is very big,we were excessing that limit very early stages of cross validation training. We solved it with adjusting our allocated memory for spark session to 10240 megabytes.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .master('local[12]') \
    .config("spark.driver.memory", "10g") \
```

1.13.2 Svd

First of all, we didn't get any problems with SVD method while running,but we needed to use cross validation for rmse measurement.And it took some time to try out different k values(which is our approximation rank). Other than that,SVD method was smooth and straight.

Also there was a joint problem on both SVD and ALS methods,default workload distribution was only on physical cores of our CPU's. While this didn't bother Kani's try outs as he has non-hyperthreading CPU,İlhan needed to configure his sessions as seen above, because java and python were using just the half of processing power. Configured CPU usage can be seen below.

(Python didn't have any memory restrictions by default,it could easily exceed above 10240 megabytes.)

Figure 6. Memory used

1.13.3 Neural Graph Collaborative Filtering (NGCF)

We tried to implement NG method to our collaborative filtering. But it took too much time to give the results. Theorotically method was in need of 14 hour timeframe while we trying to make 50 epochs.

We tried to speed it up by lowering epoch values and increasing batch sizes. By the time we get acceptable running times,our accuracy was not a thing anymore:

```
history = model.fit(
    x=x_train,
    y=y_train,
    batch_size=2048,
    epochs=5,
    verbose=1,
    validation_data=(x_val, y_val),
)
```

executed in 1h 29m 6s, finished 02:28:04 2021-06-19

```
Epoch 1/5
9487/9487 [==============================] - 1012s 106ms/step - loss: 1.0416 - val_loss: 1.4347
Epoch 2/5
9487/9487 [==============================] - 1061s 112ms/step - loss: 1.8670 - val_loss: 2.1711
Epoch 3/5
9487/9487 [==============================] - 1095s 115ms/step - loss: 2.5954 - val_loss: 2.9616
Epoch 4/5
9487/9487 [==============================] - 1116s 118ms/step - loss: 3.3075 - val_loss: 3.9070
Epoch 5/5
9487/9487 [==============================] - 1061s 112ms/step - loss: 3.9018 - val_loss: 4.6001
```

Also the biggest drawback was lack of information on online websites and tutorials about usage of NG method.

Because of that problem,we couldn't find any efficient way to increase our recommender system's accuracy,so this method was out of commission very early.

Because of SVD's accuracy and speed,we tried to make something out of SVD method. Since we didn't have that much time to focus on NG method,we whink that we clearly didn't reach its potential,still needs a lot of attention and work hours.

1.14 Conclusions

NGCF

As we mentioned above data loss was so much that we had to leave NG method to try to fix SVD method's issues some more.

ALS(Rank=15,Iteration=10,Parameter=0.025):

```
# View the rmse
test_predictions = best_model.transform(test)
RMSE = evaluator.evaluate(test_predictions)
print(RMSE)
```

executed in 1m 40.0s, finished 18:05:55 2021-06-08

```
0.8168789588679283
```

SVD(k=100):

```
#The Reader object helps in parsing the file or dataframe containing ratings
ratings = df_ratings
reader = Reader()
#dataset creation
data = Dataset.load_from_df(ratings, reader)
#Define the SVD algorithm object
svd = SVD()
#Evaluate the performance in terms of RMSE
cross_validate(svd, data, measures=["RMSE"], cv = 3)
```

executed in 5m 27s, finished 20:36:55 2021-06-28

```
{'test_rmse': array([0.84311919, 0.84351096, 0.84268809]),
 'fit_time': (93.45157957077026, 94.6795699596405, 93.95454430580139),
 'test_time': (9.537140130996704, 9.151855707168579, 9.688456296920776)}
```

SVD(k=1000):

```
#The Reader object helps in parsing the file or dataframe containing ratings
ratings = df_ratings
reader = Reader()
#dataset creation
data = Dataset.load_from_df(ratings, reader)
#Define the SVD algorithm object
svd = SVD()
#Evaluate the performance in terms of RMSE
cross_validate(svd, data, measures=["RMSE"], cv = 3)
```

executed in 5m 26s, finished 20:56:21 2021-06-28

```
{'test_rmse': array([0.8430173 , 0.8429281 , 0.84185614]),
 'fit_time': (93.28102254867554, 93.53333163261414, 93.97608780860901),
 'test_time': (9.496597051620483, 9.092251062393188, 9.619818925857544)}
```

SVD(k=2500):

```
#The Reader object helps in parsing the file or dataframe containing ratings
ratings = df_ratings
reader = Reader()
#dataset creation
data = Dataset.load_from_df(ratings, reader)
#Define the SVD algorithm object
svd = SVD()
#Evaluate the performance in terms of RMSE
cross_validate(svd, data, measures=["RMSE"], cv = 3)
```

executed in 5m 27s, finished 11:02:56 2021-06-29

```
{'test_rmse': array([0.84313092, 0.84242579, 0.84220795]),
 'fit_time': (92.88638854026794, 93.74057364463806, 93.75009846687317),
 'test_time': (9.704850435256958, 9.292328834533691, 9.757637977600098)}
```

As seen on screenshots of our tryouts,from 3 examples of SVD,one with k=1000 was more accurate. Also accuracy of SVD method or ALS method doesn't change too much,so it's come down to preferences. Do we need faster method which is SVD or a little more accurate method which is ALS?

REFERENCES

medium.com/analytics-vidhya/recommendation-system-using-collaborative-filtering-cc310e641fde

github.com/nikitaa30/Recommender-Systems/

towardsdatascience.com/beginners-guide-to-creating-an-svd-recommender-system-1fd7326d1f65

github.com/saurabhmathur96/movie-recommendations/

grouplens.org/datasets/movielens/

spark.apache.org/downloads.html

www.python.org/downloads/release/python-390/

jupyter.org

developers.google.com/machine-learning/recommendation

towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-2-alternating-least-square-als-matrix-4a76c58714a1

arxiv.org/abs/1708.05031

github.com/microsoft/recommenders

github.com/hexiangnan/neural_collaborative_filtering