Problem Statement

Let's get acquainted with the task of building a recommendation system.

We'll cover the following

Introduction

Recommendation systems are used by most of the platforms we use daily.

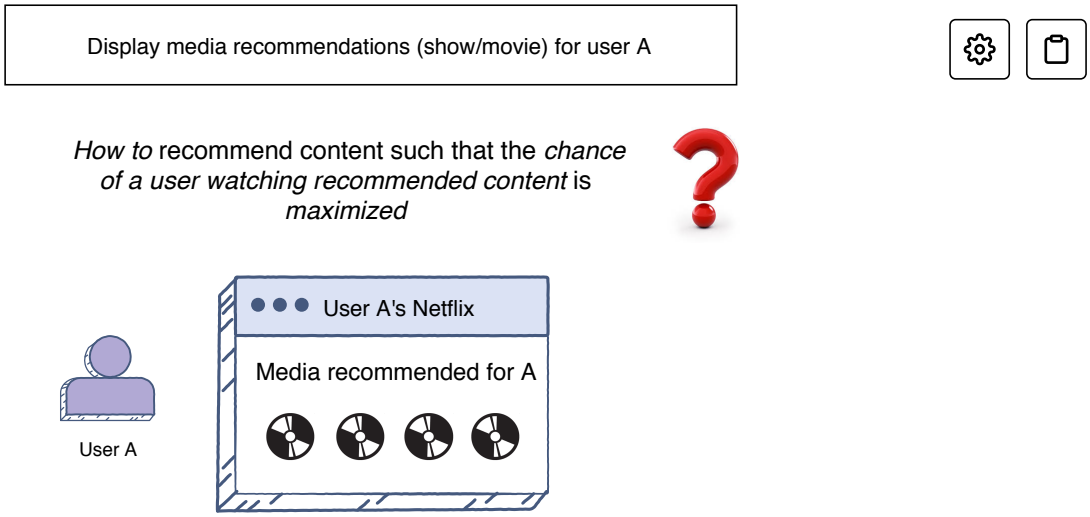
For example:

1. The Amazon homepage recommends personalized products that we might be interested in.
2. The Pinterest feed is full of pins that we might like based on trends and our historical browsing.
3. Netflix shows movie recommendations based on our taste, trending movies, etc.

In this chapter, we will go over the Netflix movie recommendation problem, but a similar technique can be applied to nearly all other recommendation systems.

Problem statement

The interviewer has asked you to display media (movie/show) recommendations for a Netflix user. Your task is to make recommendations in such a manner that the chance of the user watching them is maximized.



Visualizing the problem

The prime factor that led to Netflix's success was its recommendation system. The algorithm does a great job of bringing the right kind of content to the right users. Unlike Netflix's recommendation system, a simple recommendation system would have simply recommended the trending movies/shows with little regard for the particular user's preferences. At most, it would look at the viewer's past watches and recommend movies/shows of the same genre.

Another key aspect of Netflix's approach is that they have found ways to recommend content that seemed different from the user's regular choices. However, Netflix's recommendations are not based on wild guesses, but they are based on *other users' watch histories*, these users share some common patterns with the concerned user. This way, customers got to discover new content that they wouldn't have found otherwise.

> 📝 80% of the shows watched on Netflix are driven by its recommendations, as opposed to someone searching for a particular show and watching it.

The task at hand is to create such a recommendation system that keeps the viewers hooked and introduces them to varied content that expands their horizons.

Scope of the problem

Now that you know the problem at hand, let's define the scope of the problem:

1. The total number of subscribers on the platform as of 2019 is 163.5 million.
2. There are 53 million international daily active users.

Hence, you have to build a system for a large number of users who require good recommendations on a daily basis.

One common way to set up the recommendation system in the machine learning world is to pose it as a classification problem with the aim to predict the probability of user engagement with the content. So, the problem statement would be:

"Given a user and context (time, location, and season), predict the probability of engagement for each movie and order movies using that score."

Problem formulation

We established that you will predict the probability of engagement for each movie/show and then order/rank movies based on that score. Moreover, since our main focus is on getting the users to watch most of the recommendations, the recommendation system would be based on implicit feedback (having binary values: the user has watched movie/show or not watched).

Let's see why we have opted for ranking movies using "implicit feedback" as our probability predictor instead of using "explicit feedback" to predict movie ratings.

Click to expand: rating prediction problem

Types of user feedback

Generally, there are two types of feedback coming from an end-user for a given recommendation.

1. **Explicit feedback:**

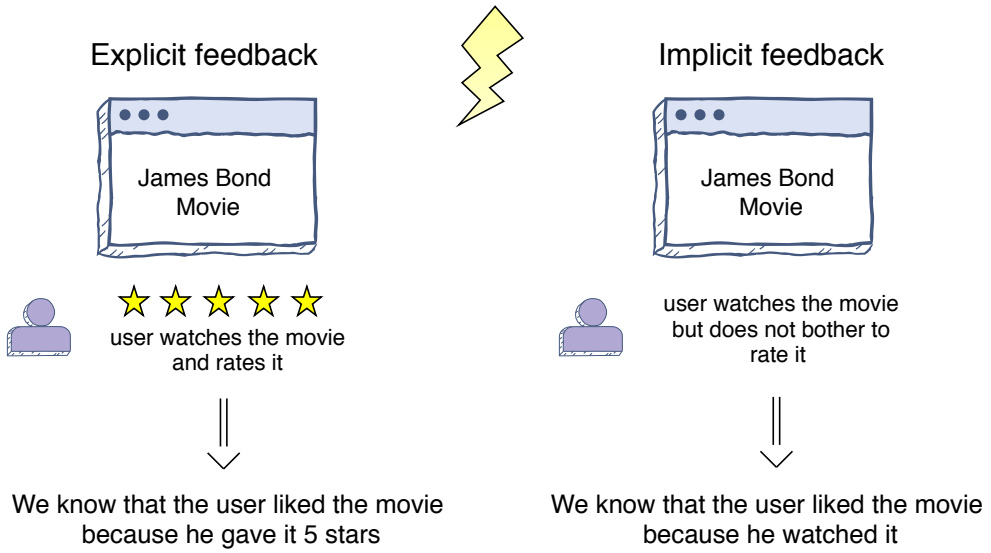
   A user provides an explicit assessment of a recommendation. In our case, it would be a star rating, e.g., a user rates the movie four out of five stars.

Here, the recommendation problem will be viewed as a rating prediction problem.

2. **Implicit feedback:**

   Implicit feedback is extracted from a user's interaction with the recommended media. Most often, it is binary in nature. For instance, a user watched a movie (1), or they did not watch the movie (0).

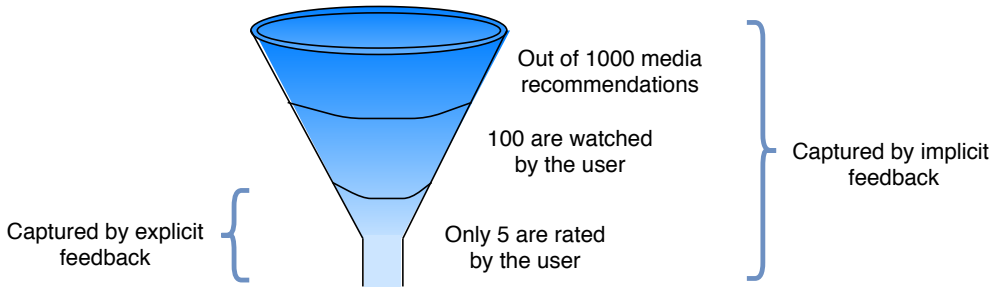
Here, the recommendation problem will be viewed as a ranking problem.



Explicit feedback is provided by user, whereas implicit feedback is assumed by user action

One key advantage of utilizing implicit feedback is that it allows collecting a large amount of training data. This allows us to better personalize recommendations by getting to know our users more.

However, this is not the case with explicit feedback. People seldom rate the movies after watching them, as depicted by the funnel below.



The difference in data available to a system built on explicit feedback versus a system built on implicit feedback.

Explicit feedback faces the *missing not at random* (MNAR) problem. Users will generally rate those media recommendations that they liked. This means $\frac{4}{5}$ or $\frac{5}{5}$ star ratings are more common than $\frac{1}{5}$, $\frac{2}{5}$ or $\frac{3}{5}$. Therefore, we won't get much information on the kind of movies that a user does not like. Also, movies with fewer ratings would have less impact on the recommendation process.

Back
Online Experimentation
Next
Metrics
Completed

Metrics

Let's look at the online and offline metrics used to judge the performance of the recommendation system.

We'll cover the following

In this lesson, you will look at different metrics that you can use to gauge the performance of the movie/show recommendation system.

Types of metrics

Like any other optimization problem, there are two types of metrics to measure the success of a movie/show recommendation system:

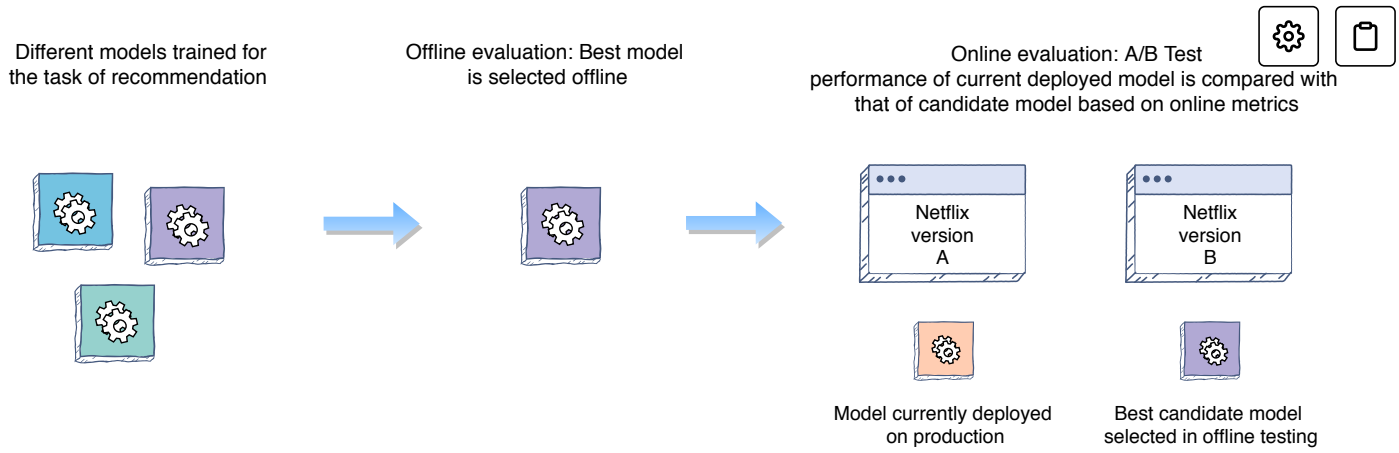
1. **Online metrics**

   Online metrics are used to see the system's performance through online evaluations on live data during an A/B test.

2. **Offline metrics**

   Offline metrics are used in offline evaluations, which simulate the model's performance in the production environment.

We might train multiple models and tune and test them *offline* with the *held-out* test data (historical interaction of users with recommended media). If its performance gain is worth the engineering effort to bring it into a production environment, the best performing model will then be selected for an *online* A/B test on live data.



📝 If a model performs well in an offline test but not in the online test, we need to think about where we went wrong. For instance, we need to consider whether our data was biased or whether we split the data appropriately for train and test.

Have a look at the lesson (https://www.educative.io/collection/page/10370001/6237869033127936/5766564994351104) about online experimentation.

Driving online metrics in the right direction is the ultimate goal of the recommendation system.

Online metrics

The following are some options for online metrics that we have for the system. Let's go over each of them and discuss which one makes the most sense to be used as the key online success indicator.

Engagement rate

The success of the recommendation system is directly proportional to the number of recommendations that the user engages with. So, the engagement rate ($\frac{sessions\ with\ clicks}{total\ number\ of\ sessions}$) can help us measure it. However, the user might click on a recommended movie but does not find it interesting enough to complete watching it. Therefore, only measuring the engagement rate with the recommendations provides an incomplete picture.

Videos watched

To take into account the unsuccessful clicks on the movie/show recommendations, we can also consider the average number of videos that the user has watched. We should only count videos that the user has spent at least a significant time watching (e.g., more than two minutes).

However, this metric can be problematic when it comes to the user starting to watch movie/series recommendations but not finding them interesting enough to finish them.

Series generally have several seasons and episodes, so watching one episode and then not continuing is also an indication of the user not finding the content interesting. So, just measuring the average number of videos watched might miss out on overall user satisfaction with the recommended content.

Session watch time

Session watch time measures the overall time a user spends watching content based on recommendations in a session. The key measurement aspect here is that the user is able to find a meaningful recommendation in a session such that they spend significant time watching it.

To illustrate intuitively on why session watch time is a better metric than engagement rate and videos watched, let's consider an example of two users, A and B. User A engages with five recommendations, spends ten minutes watching three of them and then ends the session. One the other end, user B engages with two recommendations, spends five minutes on first and then ninety minutes on the second recommendation. Although user A engaged with more content, user B's session is clearly more successful as they found something interesting to watch.

Therefore, measuring session watch time, which is indicative of the session success, is a good metric to track online for the movie recommendation system.

Offline metrics

The purpose of building an offline measurement set is to be able to evaluate our new models quickly. Offline metrics should be able to tell us whether new models will improve the quality of the recommendations or not.

Can we build an ideal set of documents that will allow us to measure recommendation set quality? One way of doing this could be to look at the movies/series that the user has completely watched and see if your recommendation system gets it right using historical data.

Once we have the set of movies/series that we can confidently say should be on the user's recommendation list, we can use the following offline metrics to measure the quality of your recommendation system.

mAP @ N

One such metric is the Mean Average Precision(mAP @ N).

> 📝 N = length of the recommendation list

Let's go over how this metric is computed so you can build intuition on why it's good to measure the offline quality.

Precision measures the ratio between the relevant recommendations and total recommendations in the movie recommendation list. It will be calculated as follows:

$$P = \frac{number\ of\ relevant\ recommendations}{total\ number\ of\ recommendations}$$

We can observe that precision alone does not reward the early placement of relevant items on the list. However, if we calculate the precision of the subset of recommendations up until each position, **k** (k = 1 to N), on the list and take their weighted average, we will achieve our goal. Let's see how.

Assume the following:

1. The system recommended **N** = 5 movies.
2. The user watched three movies from this recommendation list and ignored the other two.
3. Among all the possible movies that the system could have recommended (available on the Netflix platform), only **m** = 10 are actually relevant to the user (historical data).

Position	Movie Recommendation	Did user watch recommend-ed movie
1	Interstellar	True positive
2	Inception	True positive
3	Avengers	False positive
4	Harry Potter	True positive
5	The Imitation Game	False positive

In the following diagram, we calculate the precision of recommendation subsets up to each position, k, from 1 to 5.

Position	Movie Recommendation	Did user watch recommended movie
1	Interstellar	True positive
2	Inception	True positive
3	Avengers	False positive
4	Harry Potter	True positive
5	The Imitation Game	False positive

k=1

P(k=1) = 1/1 → p @k = proportion of all examples above that rank which are from the positive class

Calculating precision up to cutoff k = 1

1 of 5

The movie recommendation list and precisions at each cutoff “k” (1 to 5) can be represented as follows.

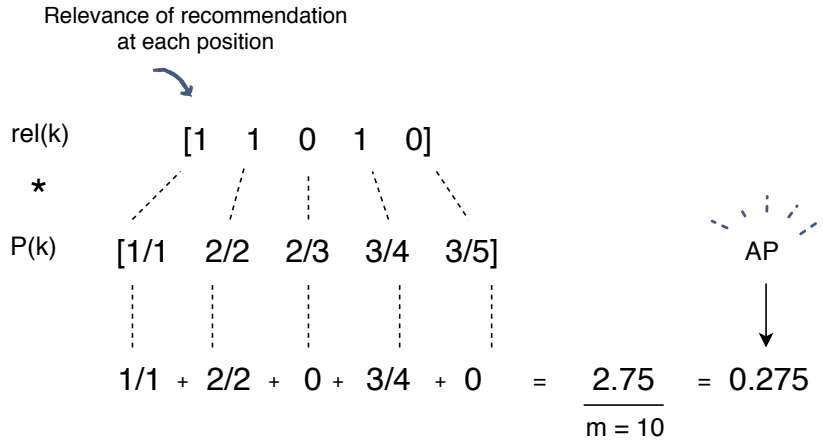
Position	Movie Recommendation	Did user see this movie
1	Interstellar	True positive
2	Inception	True Positive
3	Avengers	False Positive
4	Harry Potter	True Positive
5	The Imitation Game	False Positive

Now to calculate the average precision (AP), we have the following formula:

$$\text{AP@N} = \frac{1}{m}\sum_{k=1}^{N}(P(k) \text{ if } k^{th} \text{ item was relevant}) = \frac{1}{m}\sum_{k=1}^{N} P(k) \cdot rel(k),$$

In the above formula, rel(k) tells whether that k^{th} item is relevant or not.

Applying the formula, we have:



Average precision of the recommendation list given above

Here, we see that P(k) only contributes to AP if the recommendation at position k is relevant. Also, observe the “placement legalization” by AP by the following scores of three different recommendation lists:

User interaction with recommendation	Precision @ k	AP @ 3
[1 0 0]	[1/1 1/2 1/3]	(1/10)*(1/1) = 0.1
[0 1 0]	[0/1 1/2 1/3]	(1/10)*(1/2) = 0.05
[0 0 1]	[0/1 0/2 1/3]	(1/10)*(1/3) = 0.03

Note that a true positive *(1)*, down the recommendation list, leads to low a mAP compared to the one that is high up in the list. This is important because we want the best recommendations to be at the start of the recommendation set.

Lastly, the “mean” in mAP means that we will calculate the AP with respect to each user’s ratings and take their mean. So, mAP computes the metric for a large set of users to see how the system performs overall on a large set.

mAR @ N

Another metric that rewards the previously mentioned points is called Mean Average Recall (mAR @ N). It works similar to mAP @ N. The difference lies in the use of recall instead of precision.

Recall for your recommendation list is the ratio between the number of relevant recommendations in the list and the number of all possible relevant items(shows/movies). It is calculated as:

$$r = \frac{number\ of\ relevant\ recommendations}{number\ of\ all\ possible\ relevant\ items}$$

We will use the same recommendation list as used in the mAP @ K example, where N = 5 and m = 10. Let's calculate the recall of recommendation subsets up to each position, k.

k=1

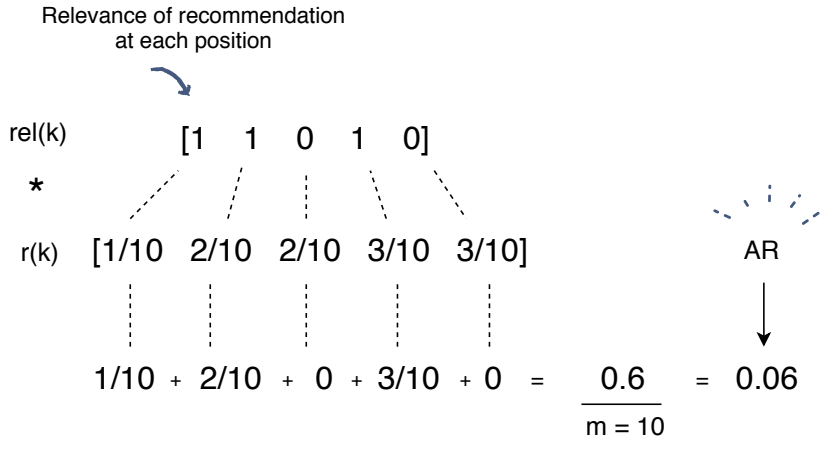
Position	Movie Recommendation	Did user watch recommended movie
1	Interstellar	True positive
2	Inception	True positive
3	Avengers	False positive
4	Harry Potter	True positive
5	The Imitation Game	False positive

r(k=1) = 1/10 → r @k = proportion of all positive examples ranked above a given rank k.

Calculating recall up to cutoff k = 1

1 of 5

The average recall (AR) will then be calculated as follows:



Lastly, the “mean” in mAR means that we will calculate AR with respect to each user’s ratings and then take their mean.

So, mAR at a high-level, measures how many of the top recommendations (based on historical data) we are able to get in the recommendation set.

F1 score

> 📝 Consider that we have two models, one is giving a better mAP @ N score and the other one was giving a better mAR @ N score. How should you decide which model has better overall performance? If you want to give equal importance to precision and recall, you need to look for a score that conveys the balance between precision and recall.

mAP @ N focuses on how relevant the top recommendations are, whereas mAR @ N shows how well the recommender recalls all the items with positive feedback, especially in its top recommendations. You want to consider both of these metrics for the recommender. Hence, you arrive at the final metric “*F1 score*”.

F1 score = $2 * \frac{mAR*mAP}{mAP+mAR}$

So, the F1 score based on mAP and mAR will be a fairly good offline way to measure the quality of your models. Remember that we selected our recommendation set size to be five, but it can be differ based on the recommendation viewport or the number of recommendations that users on the platform generally engage with.

Offline metric for optimizing ratings

We established above that we optimize the system for implicit feedback data. However, what if the interviewer says that you have to optimize the recommendation system for getting the ratings (explicit feedback) right. Here, it makes sense to use root mean squared error (RMSE) to minimize the error in rating prediction.

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

$\hat{y}_i$ is the recommendation system’s predicted rating for the movie, and y_i is the ground truth rating actually given by the user. The difference between these two values is the error. The average of this error is taken across N movies.

Architectural Components

Let's take a look at the architectural components of the recommendation system.

We'll cover the following

- Candidate generation
- Ranker
- Training data generation

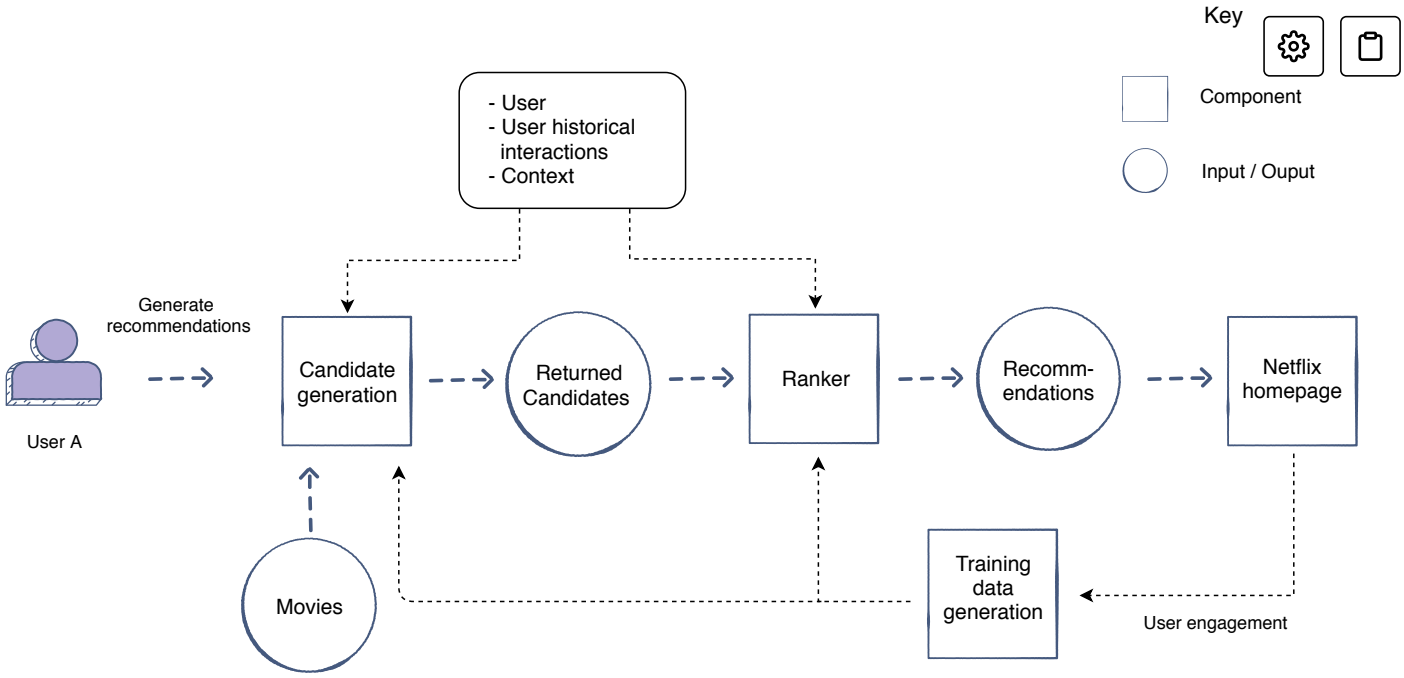
It makes sense to consider generating the best recommendation from a large corpus of movies, as a multi-stage ranking problem. Let’s see why.

We have a huge number of movies to choose from. Also, we require complex models to make great, personalized recommendations. However, if we try to run a complex model on the whole corpus, it would be inefficient in terms of execution time and computing resources usage.

Therefore, we split the recommendation task into two stages.

- Stage 1: Candidate generation
- Stage 2: Ranking of generated candidates

Stage 1 uses a simpler mechanism to sift through the entire corpus for possible recommendations. Stage 2 uses complex strategies only on the candidates given by stage 1 to come up with personalized recommendations.



Architectural diagram of the recommendation system

Candidate generation

Candidate generation is the first step in coming up with recommendations for the user. This component uses several techniques to find out the best candidate movies/shows for a user, given the user's historical interactions with the media and context.

> 📝 This component focuses on **higher recall**, meaning it focuses on gathering movies that might interest the user from all perspectives, e.g., media that is relevant based on historical user interests, trending locally, etc.

Ranker

The ranker component will score the candidate movies/shows generated by the candidate data generation component according to how interesting they might be for the user.

> 📝 This component focuses on **higher precision**, i.e., it will focus on the ranking of the top k recommendations.

It will ensemble different scores given to a media by multiple candidate generation sources whose scores are not directly comparable. Moreover, it will also use a lot of other dense and sparse features to ensure highly relevant and personalized results.

Training data generation

The user's engagement with the recommendations on their Netflix homepage will help to generate training data for both, the ranker component and the candidate generation component.

> 📝 Find out more about training data generation for the ranker component and the candidate generation component in the upcoming lessons.

Back

Metrics

Next

Feature Engineering

Completed

69% completed, meet the criteria and claim your course certificate!

Report an Issue

Ask a Question (https://discuss.educative.io/tag/architectural-components__recommendation-system__grokking-the-machine-learning-interview)

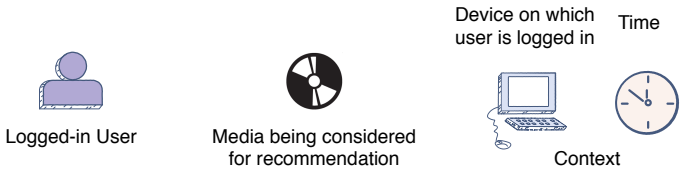
Feature Engineering

Let's engineer features for the candidate generation and ranking model.

We'll cover the following

To start the feature engineering process, we will first identify the main **actors** in the movie/show recommendation process:

1. Logged-in user
2. Movie/show
3. Context (e.g., season, time, etc.)



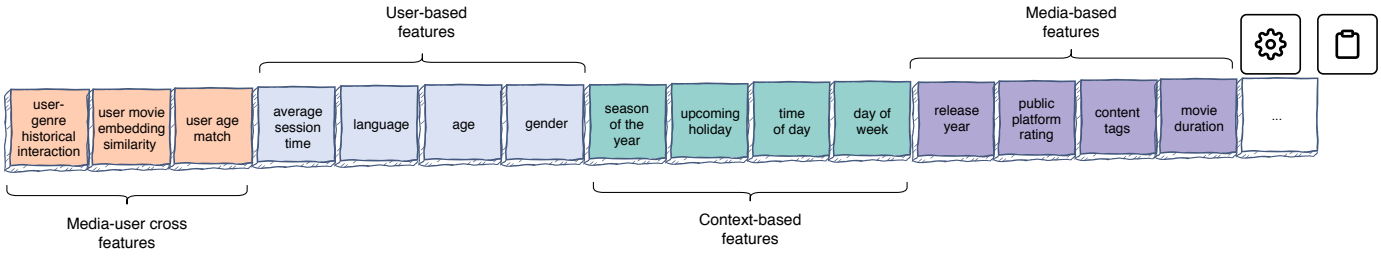
Main actors in media recommendation

Features

Now it's time to generate features based on these actors. The features would fall into the following categories:

1. User-based features
2. Context-based features
3. Media-based features
4. Media-user cross features

A subset of the features is shown below.



Features in the training data row

User-based features

Let's look at various aspects of the user that can serve as useful features for the recommendation model.

- **age**

  This feature will allow the model to learn the kind of content that is appropriate for different age groups and recommend media accordingly.

- **gender**

  The model will learn about gender-based preferences and recommend media accordingly.

- **language**

  This feature will record the language of the user. It may be used by the model to see if a movie is in the same language that the user speaks.

- **country**

  This feature will record the country of the user. Users from different geographical regions have different content preferences. This feature can help the model learn geographic preferences and tune recommendations accordingly.

- **average_session_time**

  This feature (user's average session time) can tell whether the user likes to watch lengthy or short movies/shows.

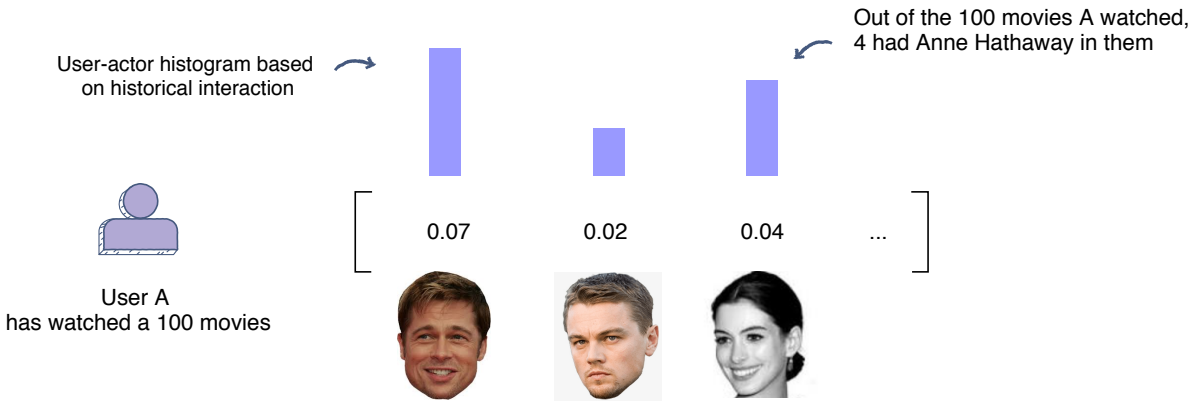
- **last_genre_watched**

  The genre of the last movie that a user has watched may serve as a hint for what they might like to watch next. For example, the model may discover a pattern that a user likes to watch thrillers or romantic movies.

The following are some user-based features (derived from historical interaction patterns) that have a *sparse representation*. The model can use these features to figure out user preferences.

- **user_actor_histogram**

  This feature would be a vector based on the histogram that shows the historical interaction between the active user and all actors in the media on Netflix. It will record the percentage of media that the user watched with each actor cast in it.



User-actor histogram vector as a feature for the model

- **user_genre_histogram**

  This feature would be a vector based on the histogram that shows historical interaction between the active user and all the genres present on Netflix. It will record the percentage of media that the user watched belonging to each genre.

- **user_language_histogram**

  This feature would be a vector based on the histogram that shows historical interaction between the active user and all the languages in the media on Netflix. It will record the percentage of media in each language that the user watched.

Context-based features

Making context-aware recommendations can improve the user's experience. The following are some features that aim to capture the contextual information.

- **season_of_the_year**

  User preferences may be patterned according to the four seasons of the year. This feature will record the season during which a person watched the media. For instance, let's say a person watched a movie tagged "summertime" (by the Netflix tagger) during the *summer season*. Therefore, the model can learn that people prefer "summertime" movies during the summer season.

- **upcoming_holiday**

This feature will record the upcoming holiday. People tend to watch holiday-themed content as the different holidays approach. For instance, Netflix tweeted that fifty-three people had watched the movie “A Christmas Prince” daily for eighteen days before the Christmas holiday. Holidays will be region-specific as well.

- **days_to_upcoming_holiday**

  It is useful to see how many days before a holiday the users started watching holiday-themed content. The model can infer how many days before a particular holiday users should be recommended holiday-themed media.

- **time_of_day**

  A user might watch different content based on the time of the day as well.

- **day_of_week**

  User watch patterns also tend to vary along the week. For example, it has been observed that users prefer watching shows throughout the week and enjoy movies on the weekend.

- **device**

  It can be beneficial to observe the device on which the person is viewing content. A potential observation could be that users tend to watch content for shorter periods on their mobile when they are busy. They usually chose to watch on their TV when they have more free time. So, they watch media for a longer period consecutively on their TV. Hence, we can recommend shows with short episodes when a user logs in from their mobile device and longer movies when they log in from their TV.

Media-based features

We can create a lot of useful features from the media’s metadata.

- **public-platform-rating**

  This feature would tell the public’s opinion, such as IMDb/rotten tomatoes rating, on a movie. A movie may launch on Netflix well after its release. Therefore, these ratings can predict how the users will receive the movie after it becomes available on Netflix.

- **revenue**

  We can also add the revenue generated by a movie before it came to Netflix. This feature also helps the model to figure out the movie’s popularity.

- **time_passed_since_release_date**

  The feature will tell how much time has elapsed since the movie’s release date.

- **time_on_platform**

  It is also beneficial to record how long a media has been present on Netflix.

- **media_watch_history**

  Media's watch history (number of times the media was watched) can indicate its popularity. Some users might like to stay on top of trends and focus on only watching popular movies. They can be recommended popular media. Others might like less discovered indie movies more. They can be recommended less watched movies that had good implicit feedback (the user watched the whole movie and did not leave it midway). The model can learn these patterns with the help of this feature.

We can look at the media's watch history for different time intervals as well. For instance, we can have the following features:

- media_watch_history_last_12_hrs
- media_watch_history_last_24_hrs

> 📝 The media-based features listed above can collectively tell the model that a particular media is a blockbuster, and many people would be interested in watching it. For example, if a movie generates a large revenue, has a good IMDb rating, came to the platform 24 hours ago, and a lot of people have watched it, then it is definitely a blockbuster.

- **genre**

  This feature records the primary genre of content, e.g., comedy, action, documentaries, classics, drama, animated, and so on.

- **movie_duration**

  This feature tells the movie duration. The model may use it in combination with other features to learn that a user may prefer shorter movies due to their busy lifestyle or vice versa.

- **content_set_time_period**

  This feature describes the time period in which the movie/show was set in. For example, it may show that the user prefers shows that are set in the '90s.

- **content_tags**

Netflix has hired people to watch movies and shows to create extremely detailed, descriptive, and specific tags for the movies/shows that capture the nuances in the content. For instance, media can be tagged as a “Visually-striking nostalgic movie”. These tags greatly help the model understand the taste of different users and find the similarity between the user’s taste and the movies.

- **show_season_number**

  If the media is a show with multiple seasons, this feature can tell the model whether a user likes shows with fewer seasons or more.

- **country_of_origin**

  This feature holds the country in which the content was produced.

- **release_country**

  This feature holds the country where the content was released.

- **release_year**

  This feature shows the year of theatrical release, original broadcast date or DVD release date.

- **release_type**

  This feature shows whether the content had a theatrical, broadcast, DVD, or streaming release.

- **maturity_rating**

  This feature contains the maturity rating of the media with respect to the territory (geographical region). The model may use it along with a user’s age to recommend appropriate movies.

Media-user cross features

In order to learn the users’ preferences, representing their historical interactions with media as features is very important. For instance, if a user watches a lot of Christopher Nolan movies, that would give us a lot of information about what kind of movies the user likes. Some of these interaction-based features are as follows:

User-genre historical interaction features

These features represent the percentage of movies that the user watched with the same genre as the movie under consideration. This percentage is calculated for different time intervals to cater to the dynamic nature of user preferences.

- **user_genre_historical_interaction_3months**

  The percentage of movies that the user watched with the same genre as the movie under consideration in the last 3 months. For example, if the user watched 6 comedy movies out of the 12 he/she watched in the last 3 months, then the feature value will be:

  $\frac{6}{12}$ = 0.5 or 50%

  This feature shows a more recent trend in the user's preference for genres as compared to the following feature.

- **user_genre_historical_interaction_1year**

  This is the same feature as above but calculated for the time interval of one year. It shows a more long term trend in the relationship between the user and genre.

- **user_and_movie_embedding_similarity**

  Netflix has hired people to watch movies and shows to create incredibly detailed, descriptive, and specific tags for the movies/shows that capture the nuances in the content. For instance, media can be tagged as "Visually-striking nostalgic movie".

  You can have a user embedding based on the tags of movies that the user has interacted with and a media embedding based on its tags. The dot product similarity between these two embeddings can also serve as a feature.

- **user_actor**

  This feature tells the percentage of media that the user has watched, which has the same cast (actors) as that of the media under consideration for recommendation.

- **user_director**

  This feature tells the percentage of movies that the user has watched with the same director as the movie under consideration.

- **user_language_match**

  This feature matches the user's language and the media's language.

- **user_age_match**

  You will keep a record of the age bracket that has mostly viewed a certain media. This feature will see if the user watching a particular movie/show falls into the same age bracket. For instance, movie A is mostly (80% of the times) watched by people who are 40+. Now, while considering movie A for a recommendation, this feature will see if the user is 40+ or not.

Some ***sparse features*** are described below. Each of them can show popular trends in their respective domains and also the preferences of individual users. We will go over how these sparse features are used in the ranking chapter. You can also go over the embedding chapter (https://www.educative.io/collection/page/10370001/6237869033127936/6130870193750016) about how to generate vector representation of this sparse data to use them in machine learning models.

- **movie_id**

  Popular movie IDs are be repeated frequently.

- **title_of_media**

  This feature holds the title of the movie or the TVV series.

- **synopsis**

  This feature holds the synopsis or summary of the content.

- **original_title**

  This feature holds the original title of the movie in its original language. The media may be released for a different country with a different title keeping in view the preference of the nationals. For example, Japanese/Korean movies/shows are released for English speaking countries with English titles as well.

- **distributor**

  A particular distributor may be selecting very good quality content, and hence users might prefer content from that distributor.

- **creator**

  This feature contains the creator/s of the content.

- **original_language**

  This feature holds the original spoken language of the content. If multiple, you can record the choose the majority language.

- **director**

  This feature holds the director/s of the content. This feature can indicate directors who are widely popular, such as Steven Spielberg, and it can also showcase the individual preference of users.

- **first_release_year**

This feature holds the year in which content had its first release anywhere (this is different from production year).

- **music_composer**

  The music in a show or a film's score can greatly enhance the storytelling aspect. Users may fancy the work of a particular composer and may be more drawn to their work.

- **actors**

  This feature includes the cast of the movie/show.

Click to expand: Sparse vs dense representation of features

Back
Architectural Components

Next
Candidate Generation

Completed

Candidate Generation

The purpose of candidate generation is to select the top k (let's say one-thousand) movies that you would want to consider showing as recommendations to the end-user. Therefore, the task is to select these movies from a corpus of more than a million available movies.

We'll cover the following

In this lesson, we will be looking at a few techniques to generate media candidates that will match user interests based on the user's historical interaction with the system.

Candidate generation techniques

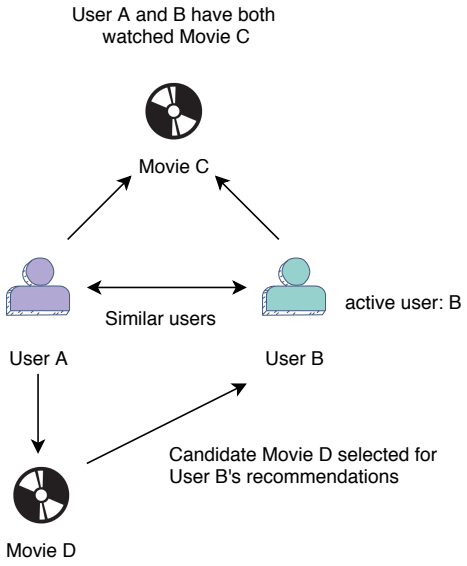
The candidate generation techniques are as follows:

1. Collaborative filtering
2. Content-based filtering
3. Embedding-based similarity

Each method has its own strengths for selecting good candidates, and we will combine all of them together to generate a complete list before passing it on to the ranked (this will be explained in the ranking lesson).

Collaborative filtering

In this technique, you find users similar to the *active user* based on the intersection of their historical watches. You, then, collaborate with similar users to generate candidate media for the active user, as shown below.



Collaborative filtering

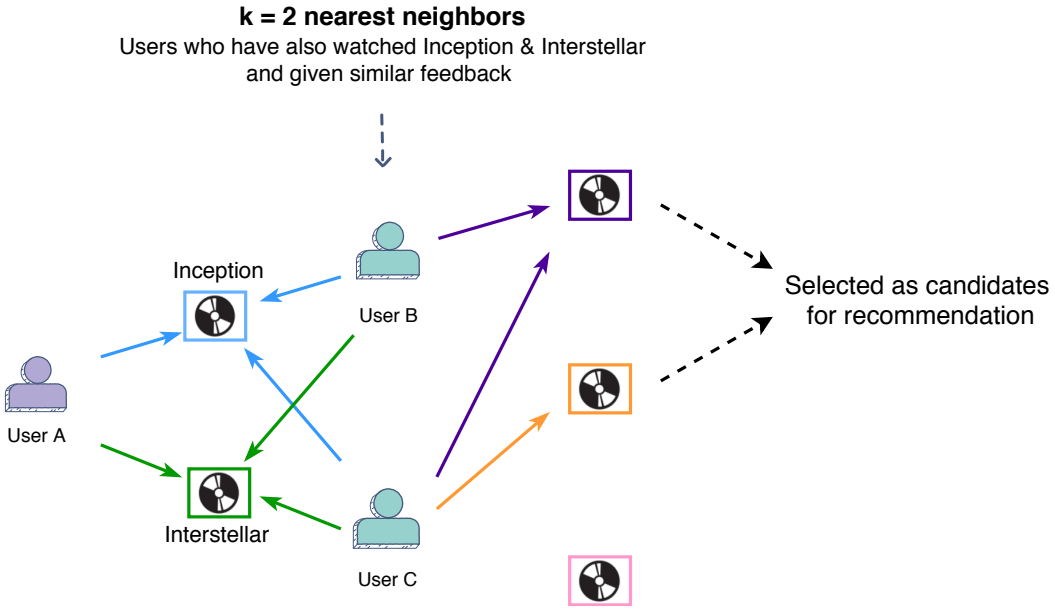
Click to expand: Intuition behind collaborative filtering

There are two methods to perform collaborative filtering:

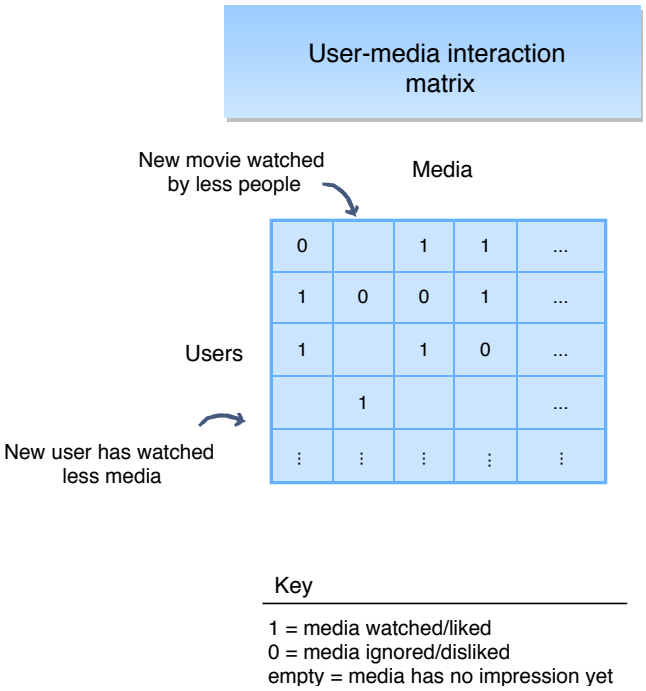
1. Nearest neighborhood
2. Matrix factorization

Method 1: Nearest neighborhood

User A is similar to user B and user C as they have watched the movies Inception and Interstellar. So, you can say that user A's nearest neighbours are user B and user C. You will look at other movies liked by users B and C as candidates for user A's recommendations.



Let's see how this concept is realized. You have a (n x m) matrix of user $u_{i(i=1\ to\ n)}$ and movie m_j $(j=1\ to\ m)$. Each matrix element represents the feedback that the user *i* has given to a movie *j*. An empty cell means that user *i* has not watched movie *j*.



To generate recommendations for user *i*, you need to predict their feedback for all the movies they haven't watched. You will collaborate with users similar to user *i* for this process. Their ratings for a movie, not seen by user i, would give us a good idea of how user *i* would like it.

So, you will compute the similarity (e.g. cosine similarity) of other users with user *i* and then select the top k similar users/nearest neighbours (KNN(u_i)). Then, user *i's* feedback for an unseen *movie j* (f_{ij}) can be predicted by taking the weighted average of feedback that the top k similar users gave to movie *j*. Here the feedback by the nearest neighbour is weighted by their similarity with user *i*.

$$f_{ij} = \frac{\sum_{v \in KNN(u_i)} Similarity(u_i, u_v) f_{vj}}{k}$$

The unseen movies with good predicted feedback will be chosen as candidates for user *i's* recommendations.

Collaborative filtering by identifying similar media

It is evident that this process will be computationally expensive with the increase in numbers of users and movies. The sparsity of this matrix also poses a problem when a movie has not been rated by any user or a new user has not watched many movies.

Method 2: Matrix factorization

As explained above, you need to represent the *user-media interaction matrix* in a way that is scalable and handles sparsity. Here, matrix factorization helps us by factoring this matrix into two lower dimensional matrices:

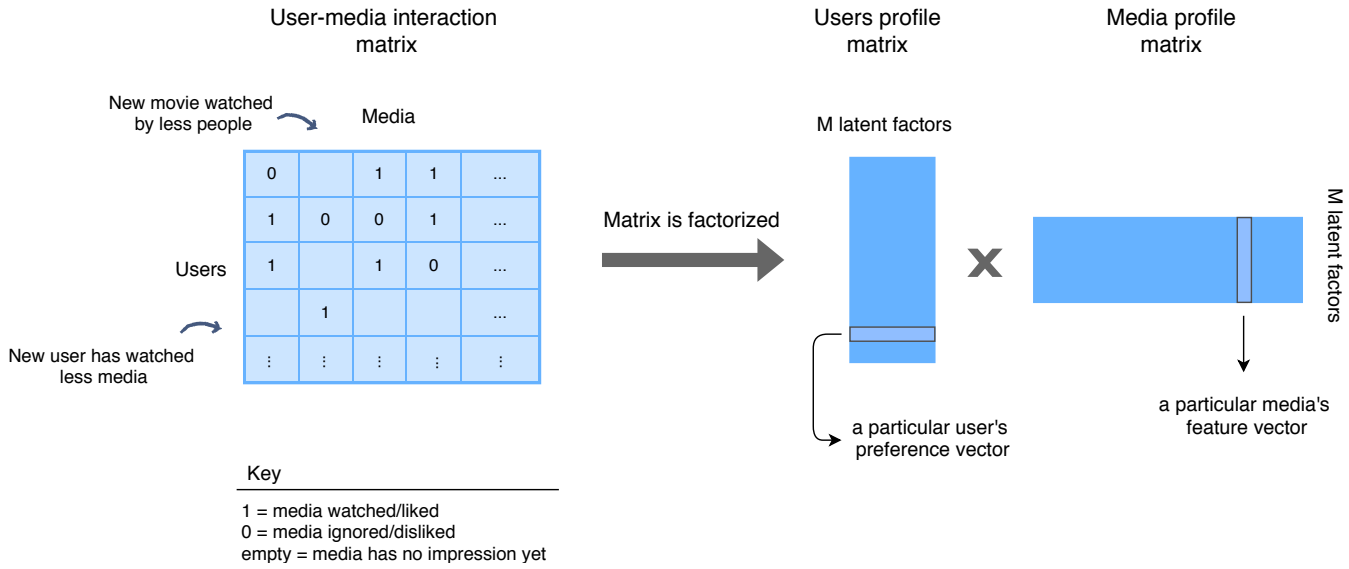
1. User profile matrix (n x M)

   Each user in the user profile matrix is represented by a row, which is a *latent vector of M dimensions*.

2. Media profile matrix (M x m)

   Each movie in the movie profile matrix is represented by a column, which is a latent vector of M dimensions.

The dimension M is the number of latent factors we're using to estimate the user-media feedback matrix. *M* is much smaller than the actual number of users and number of media.

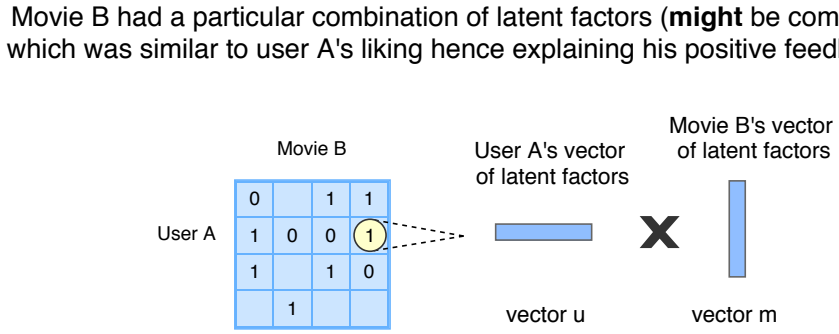


Factorization of user-media interaction matrix into two smaller factors (user profile matrix and media profile matrix)

The representation of users and media in the form of a *vector of latent factors* aims to explain the reason behind a user's particular feedback for a media. Latent vectors can also be thought of as features of a movie or a user. For example, in the diagram below, two of the latent factors might loosely represent the amount of comedy and quirkiness.

> 📝 Note that we say the word “might” and “loosely” for latent factors because we don’t know exactly what each dimension means, it is “hidden” from us.

User A’s vector has their preferences: 1 for the “comedy” and 0 for “quirkiness”. Movie B’s vector contains the presence/absence of the two factors in the movie, e.g., 1 and 0. The *dot product of these two vectors* will tell how much movie B aligns with the preferences of the user A; hence, the feedback.



User A's feedback for movie B is captured in the form of alignment between A's preferences and B's characteristics

The first step is to create the user profile and movie profile matrices. Then, you can generate good candidates for movie recommendation by predicting user feedback for unseen movies. This prediction can be made simply by computing the dot product of the user vector with the movie vector.

Now, let’s go over the process of how to learn the latent factor matrices for users and media.

You will initialize the user and movie vectors randomly. For each known/historical user-movie feedback value f_{ij}, you will predict the movie feedback by taking the dot product of the corresponding user profile vector u_i and movie profile vector m_j. The difference between the actual (f_{ij}) and the predicted feedback ($u_i.m_j$) will be the error (e_{ij}).

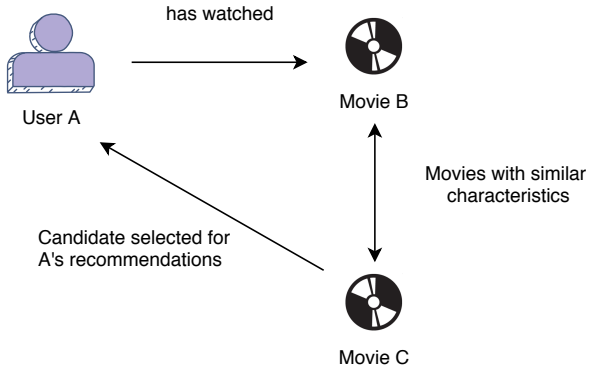
$$e_{ij} = f_{ij} - u_i.m_j$$

You will use stochastic gradient descent to update the user and movie latent vectors, based on the error value. As you continue to optimize the user and movie latent vectors, you will get a semantic representation of the users and movies, allowing us to find new recommendations that are closer in that space.

📝 The user profile vector will be made based on the movies that the user has given feedback on. Similarly, the media/movie profile vector will be made based on its user feedbacks. By utilizing these user and movie profile vectors, you will generate candidates for a given user based on their predicted feedback for unseen movies.

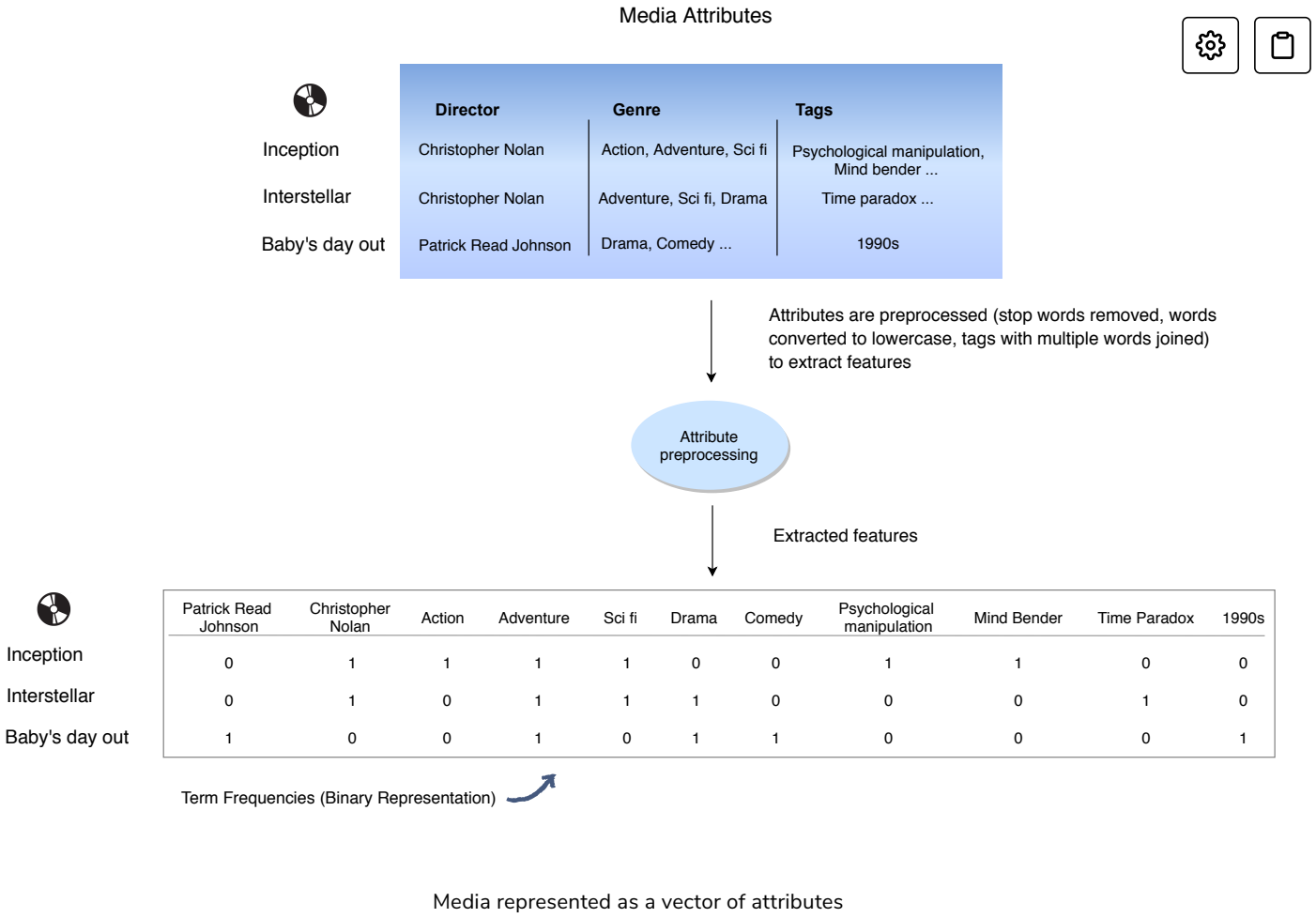
Content-based filtering

Content-based filtering allows us to make recommendations to users based on the characteristics or attributes of the media they have already interacted with.



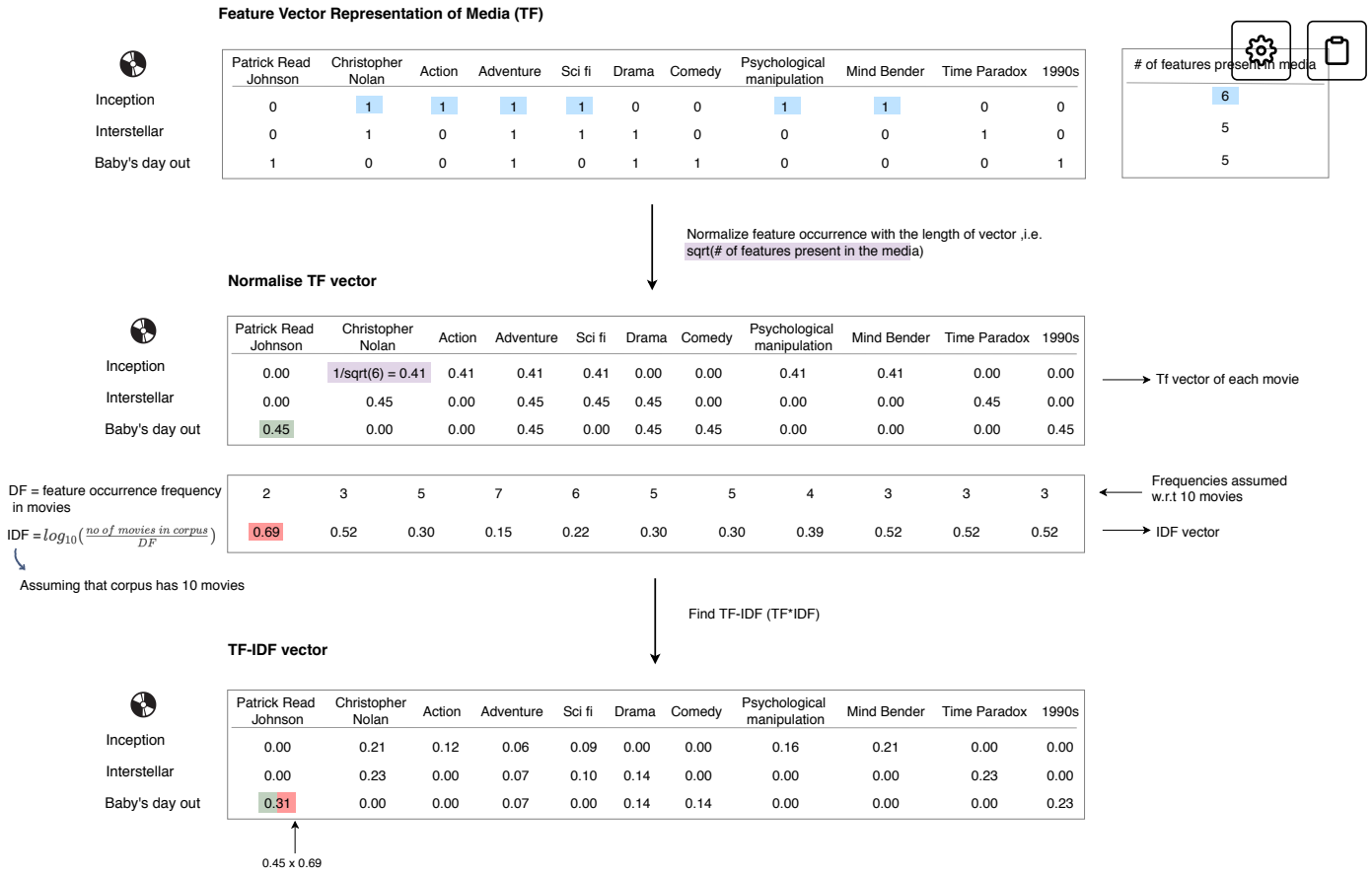
Collaborative filtering

As such the recommendations tend to be relevant to users' interest. The characteristics come from metadata (e.g., genre, movie cast, synopsis, director, etc.) information and manually assigned media-descriptive-tags (e.g., visually striking, nostalgic, magical creatures, character development, winter season, quirky indie rom-com set in Oregon, etc.) by Netflix taggers. The media is represented as a vector of its attributes. The following explanation takes a subset of the attributes for ease of understanding.



Media represented as a vector of attributes

Initially, you have the media's attributes in raw form. They need to be preprocessed accordingly to extract features. For instance, you need to remove stop words and convert attribute values to lowercase to avoid duplication. You also have to join the director's first name and last name to identify unique people since there maybe two directors with the first name Christopher. Similar preprocessing is required for the tags. After this, you are able to represent the movies as a vector of attributes with elements depicting the term frequency (TF) (binary representation of attribute's presence (1) or absence (0)).



Click to enlarge: making TF-IDF vector representations of media

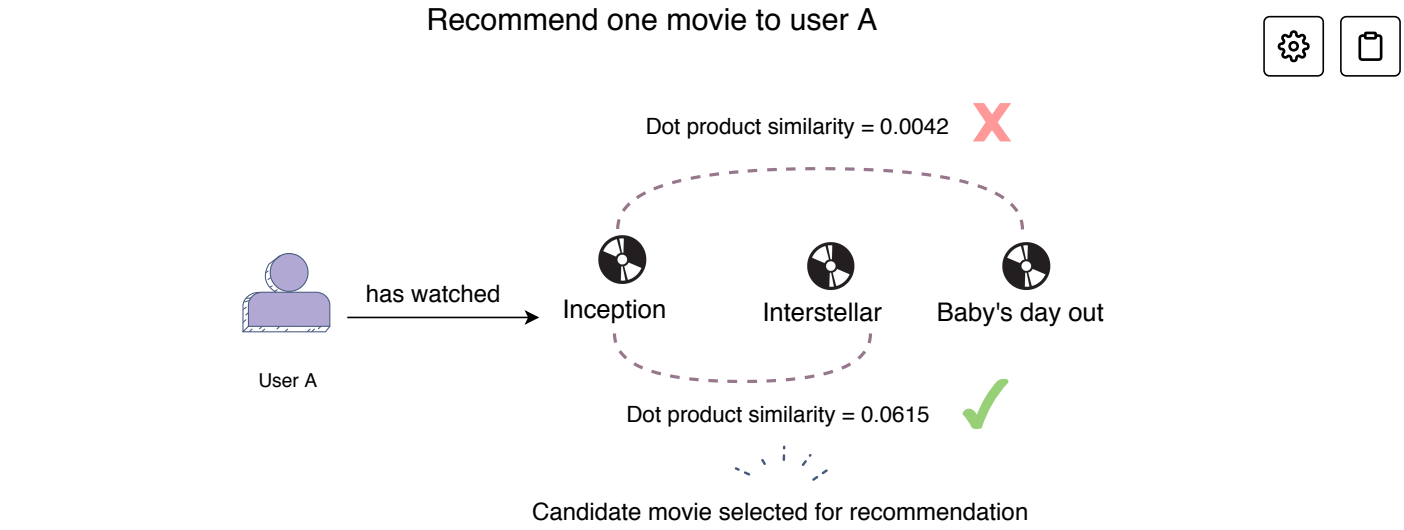
Now, you have the movies represented as vectors containing the TF (term/attribute frequency) of all the attributes. This is followed by normalizing the term frequencies by the length of the vectors. Finally, you multiply the movies' normalized TF vectors and the IDF vector element-wise to make TF-IDF vectors for the movies.

Click to expand: TF-IDF

Given the TF-IDF representation of each movie/show, you have two options for recommending media to the user:

1. **Similarity with historical interactions**

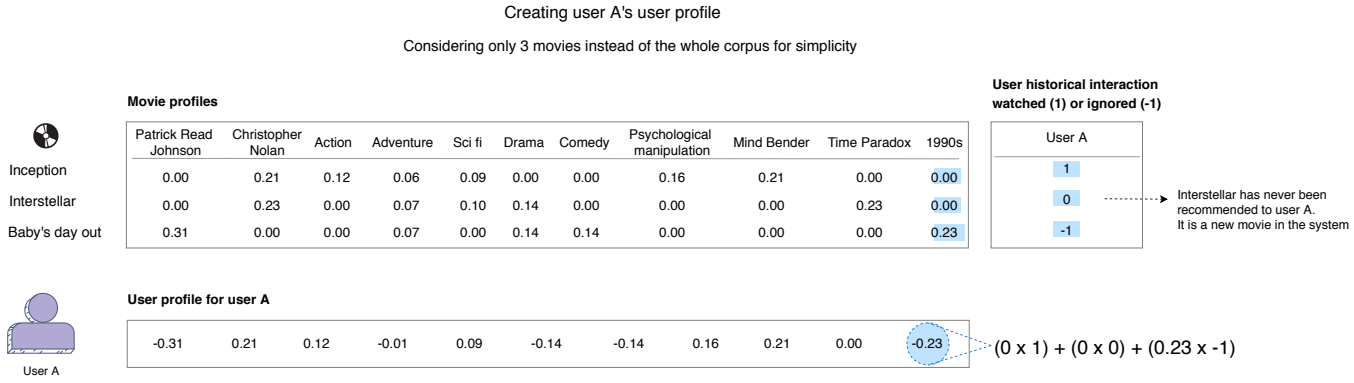
   You can recommend movies to the user similar to those they have interacted (seen) with in the past. This can be achieved by computing the dot product of movies.



Interstellar is selected as a candidate for recommendation to user A based on their historical watch (Inception)

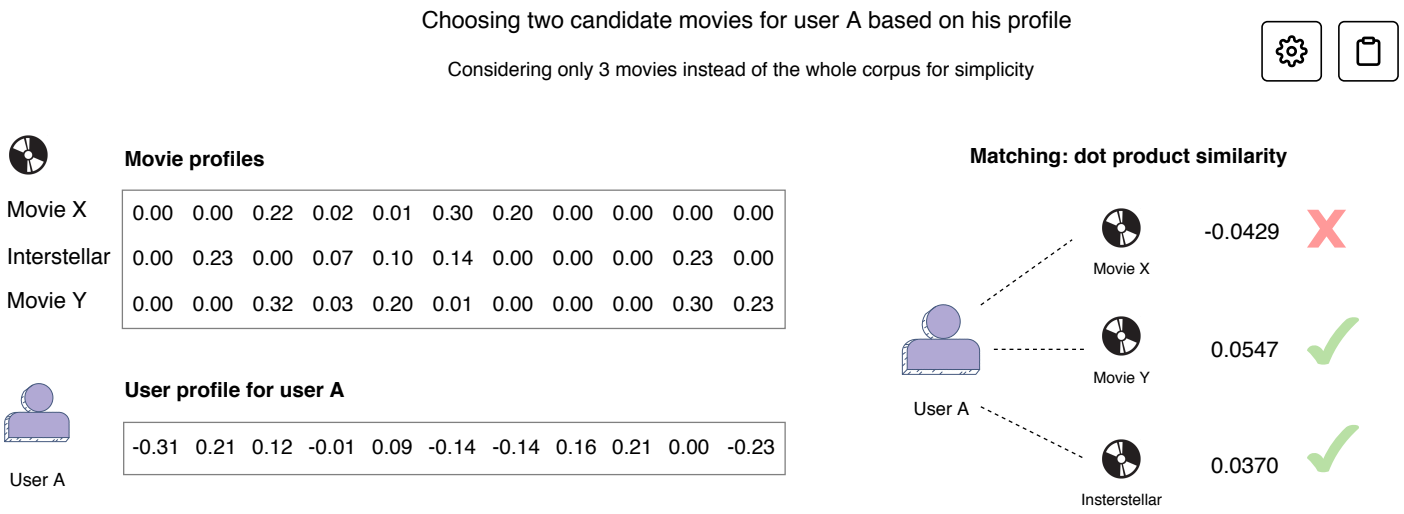
2. **Similarity between media and user profiles**

The media's TF-IDF vectors can be viewed as their *profile*. Based on user preferences, which can be seen from its historical interactions with media, you can build *user profiles* as well as shown below.



Creating user A's user profile

Now, instead of using past watches to recommend new ones, you can just compute the similarity (dot product) between the user's profile and media profiles of unseen movies to generate relevant candidates for user A's recommendations.



Choosing two candidate movies for user A based on their profile

Generate embedding using neural networks/deep learning

Given the historical feedback (u,m), i.e., user u's feedback for movie m, you can use the power of deep learning to generate latent vectors/embeddings to represent both movies and users. Once you have generated the vectors, you will utilize KNN (k nearest neighbours) to find the movies that you would want to recommend to the user. This method is similar to generating latent vectors, that you saw in matrix factorization but is much more powerful because using NNs allows us to use all the sparse and dense features in the data to generate user and movie embedding.

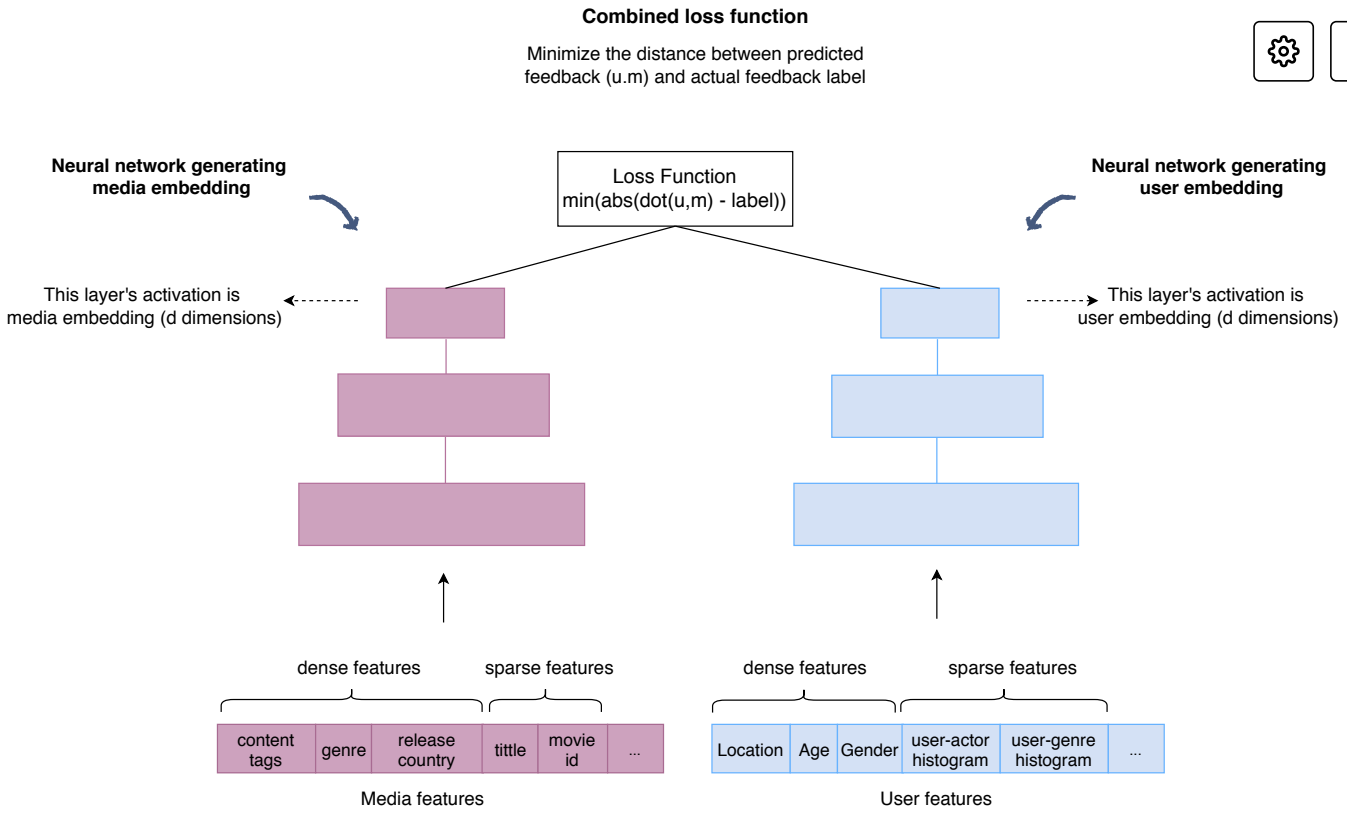
Embedding generation

You set up the network as two towers with one tower feeding in *media only* sparse and dense features and the other tower feeding in *user-only* sparse and dense features. The activation of the first tower's last layer will form the media's vector embedding *(m)*. Similarly, the activation of the second tower's last layer will form the user's vector embedding *(u)*. The combined optimization function at the top aims to minimize the distance between the dot product of *u* and *m* (predicted feedback) and the actual feedback label.

$$min(|dot(u,m) - label|) \text{ where } u,m \in R^d, d = dimension\ of\ u\ and\ v's\ embedding$$

Let's look at the intuition behind this cost function. The actual feedback label will be positive when the media aligns with user preferences and negative when it does not. To make the predicted feedback follow the same pattern, the network learns the user and media embeddings in such a way that their distance will be minimized if the user would like this media and maximized if the user would not like the media.

> 📝 The vector representation of the user, which you would get as a result, will be nearest to the kind of movies that the user would like/watch.



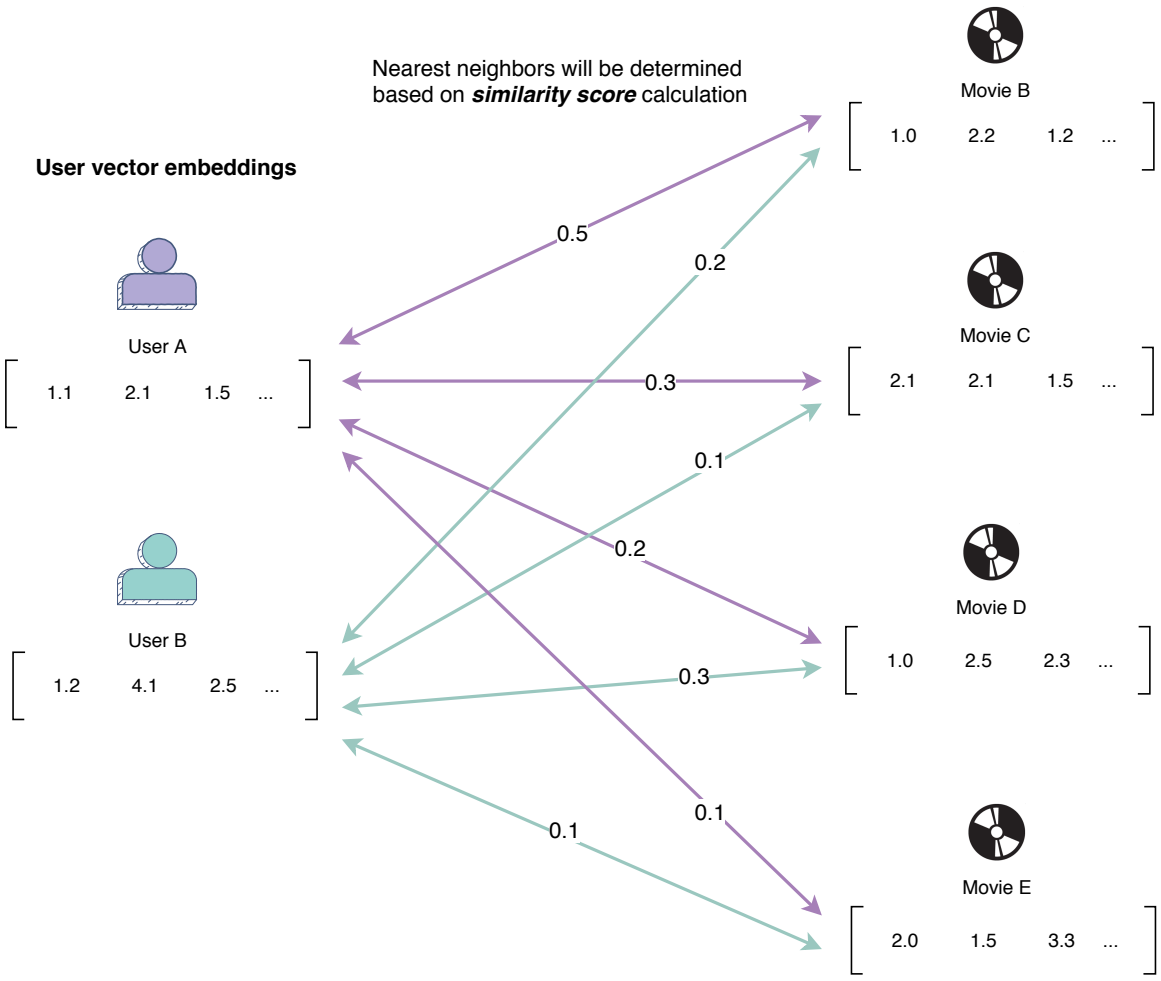
Generating user and movie embedding through neural networks with a combined loss function

📝 The feature *user-actor histogram* shows the percentage of media watched by the user with particular actors (cast) in it. For instance, let's say 2% of the content viewed by the user had Brad Pitt in it, and 5% had Leonardo DiCaprio in it.

Click to expand: Hyper parameters

Candidate selection (KNN)

Let's assume that you have to generate two candidates for each user, as shown in the diagram below. After the user and media vector embeddings have been generated, you will apply KNN and select candidates for each user. It can be observed in the diagram below, that user A's nearest neighbors are movie B and movie C, based on high *vector similarity*. Whereas, for user B, movie D and movie B are the nearest neighbors.



Finding k=2 nearest neighbor (movies) for users based on their vector similarity (dot product)

Techniques' strengths and weaknesses

Let's look at some strengths and weaknesses of the approaches for candidate generation discussed above.

Collaborative filtering can suggest candidates based solely on the historical interaction of the users. Unlike content-based filtering, it *does not require domain knowledge to create user and media profiles*. It may also be able to capture data aspects that are often elusive and difficult to profile using content-based filtering. However, collaborative filtering suffers from the *cold start problem*. It is difficult to find users similar to a new user in the system because they have less historical interaction. Also, new media can't be recommended immediately as no users have given feedback on it.

The **neural network technique** also suffers from the *cold start problem*. The embedding vectors of media and users are updated in the training process of the neural networks. However, if a movie is new or if a user is new, both would have fewer instances of feedback received and feedback

given, respectively. By extension, this means there is a lack of sufficient training examples to update their embedding vectors accordingly. Hence, the cold start problem.

Content-based filtering is superior in such scenarios. It does require some initial input from the user regarding their preferences to start generating candidates, though. This input is obtained as a part of the onboarding process, where a new user is asked to share their preferences. *Once we have the initial input, it can create and then match the user's profile with media profiles*. Moreover, new medias' profiles can be built immediately as their description is provided manually.

Back

Feature Engineering

Next

Training Data Generation

Completed

69% completed, meet the criteria and claim your course certificate!

Report an Issue

Ask a Question (https://discuss.educative.io/tag/candidate-generation__recommendation-system__grokking-the-machine-learning-interview)

Training Data Generation

Let's generate training data for the recommendation task with respect to implicit user feedback.

We'll cover the following

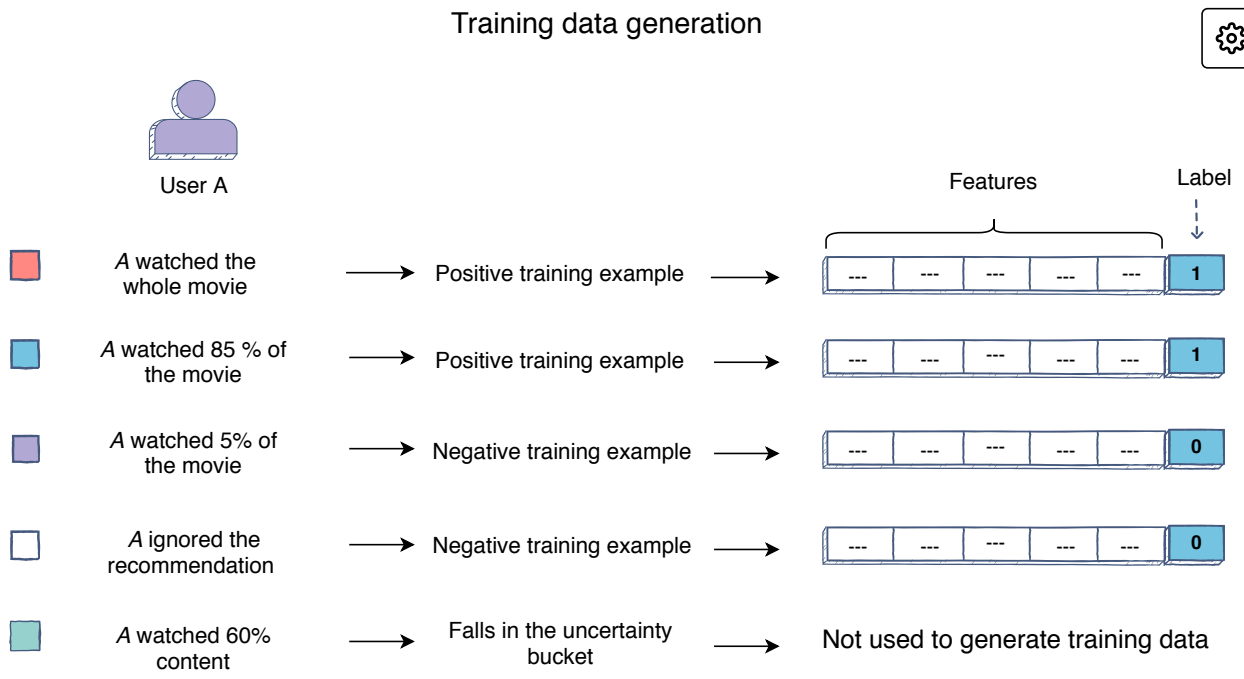
As mentioned previously, you will build your model on implicit feedback from the user. You will look at how a user interacts with media recommendations to generate positive and negative training examples.

Generating training examples

One way of interpreting *user actions* as positive and negative training examples is based on the duration for which the user watched a particular show/movie. You take positive examples as ones where the user ended up watching most of a recommended movie/show, i.e., watched 80% or more. You take negative examples, again where we are confident that the user ignored a movie/show, i.e., watched 10% or less.

If the percentage of a movie/show watched by the user falls between 10% and 80%, you will put it in the uncertainty bucket. This percentage is not clearly indicative of a user's like or dislike, so you ignore such examples. For instance, let's say a user watched 55% of a movie. This could be considered a positive example considering that they liked it enough to watch it midway. However, it could be that a lot of people had recommended it to them, so they wanted to see what all the hype was about by at least watching it halfway through. However, they, ultimately, decided that it was not according to their liking.

Hence, to avoid these kinds of misinterpretations, you label examples as positive and negative only when you are certain about it to a higher degree.

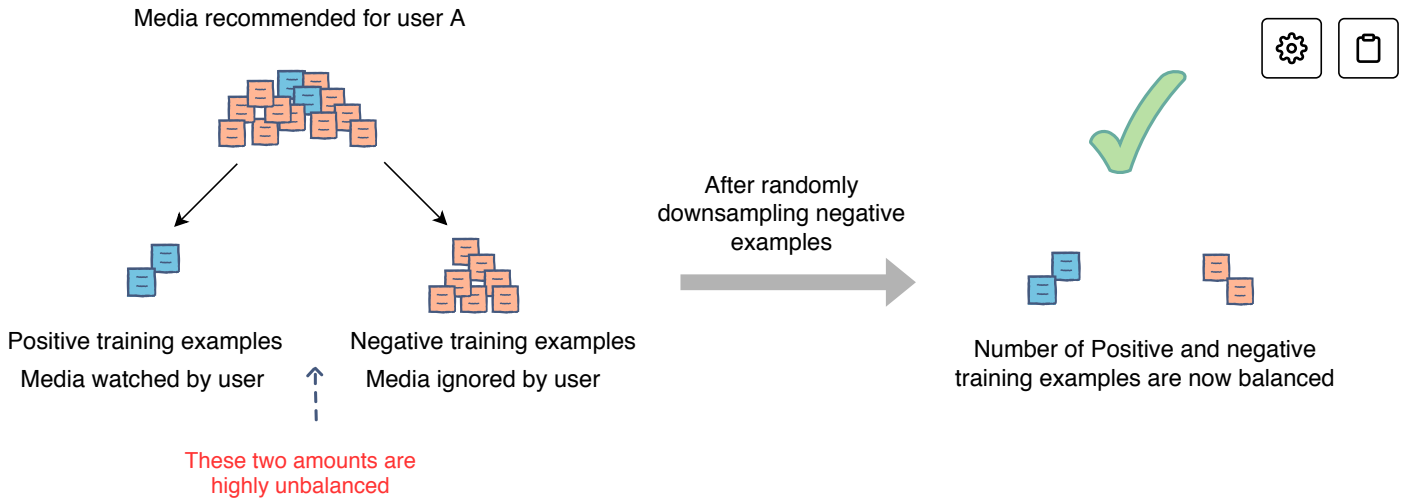


Positive and negative training examples being generated from user actions

Balancing positive and negative training examples

Each time a user logs in, Netflix provides *a lot of recommendations*. The user cannot watch all of them. Yes, people do binge-watch on Netflix, but still, this does not improve the positive to negative training examples ratio significantly. Therefore, you have a lot more negative training examples than positive ones. To balance the ratio of positive and negative training samples, you can **randomly downsample** the negative examples.

> 📝 We balance the negative and positive examples to prevent classifier from favouring the outcome that has more examples.



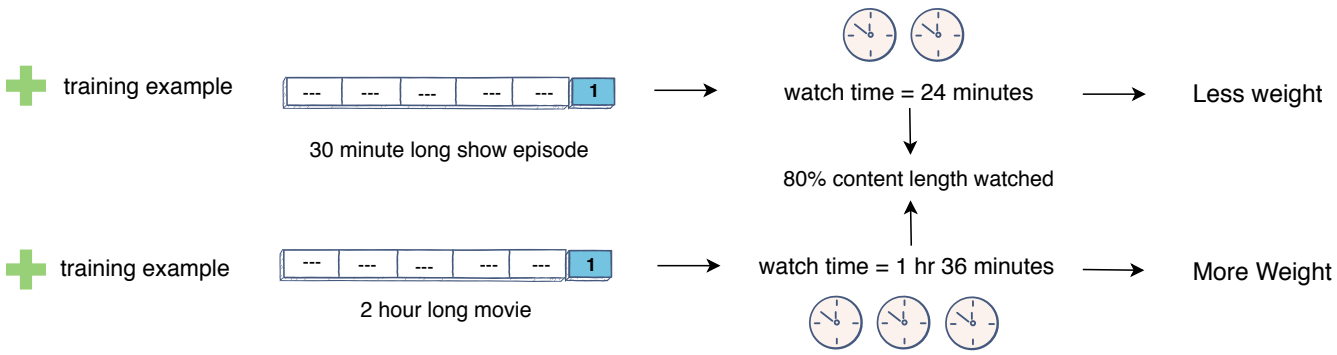
Balancing positive and negative training examples

Weighting training examples

Based on our training data discussion so far, all of the training examples are weighted equally, i.e., all have a weight of 1. According to Netflix's business objectives, the main goal could be to increase the time a user spends on the platform.

One way to incentivize your model to focus more on examples that have a higher contribution to the session watch time is to weight examples based on their contribution to session time. Here, you are assuming that your prediction model's optimization function utilizes weight per example in its objective.

Reweighing training data based on watch time



In the diagram above, you have two positive training examples. The first is a thirty-minute long show episode while the second is a two-hour long movie. The user watches 80% of both media. For the show, this equals twenty-four minutes, but for the movie, it means one hour and thirty-six minutes. You would assign more weight to the second example than the first one so that your model learns which kinds of media increases the session watch time.

📝 One caveat of utilizing these weights is that the model might only recommend content with a longer watch time. So, it's important to choose weights such that we are not solely focused on watch time. We should find the right balance between user satisfaction and watch time, based on our online A/B experiments.

Train test split

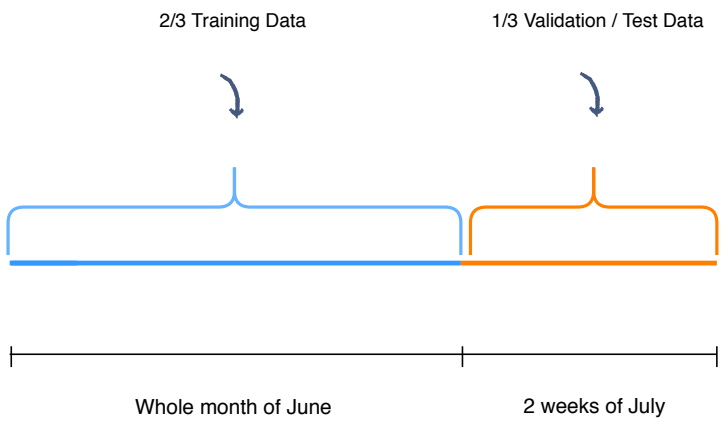
Now, it's time to split your training data for training and then testing your models. While doing so, you need to be mindful of the fact that the user's interaction patterns may differ throughout the week. Hence, you will use the interaction with recommendations throughout a week to capture all of the patterns during model training.

You can randomly select $\frac{2}{3}^{rd}$, or 66.6%, of the training data rows generated and utilize them for training purposes. The rest of the $\frac{1}{3}^{rd}$, or 33.3%, can be used for validation and model testing.

However, this random splitting defeats the purpose of training the model on a whole week's data. Also, the data has a time dimension, i.e., you know the interaction on previous recommendations, and you want to predict the interaction with future recommendations. Hence, you will train the model on data from one time interval and validate it on the data from its succeeding time interval. This will give you a more accurate picture of how your model will perform in a real scenario.

📝 You are building models with the intent to forecast the future.

In the following illustration, you are training the model using data generated from the month of June and using data generated in the first and second week of July for validation and testing purposes.



Splitting data for training, validation, and testing

Even though Netflix has millions of subscribers, on average they may watch a maximum of 3 movies/1 show per week. Therefore, to generate a sufficient amount of data, you need to increase the time span over which you gather the data. Hence, we are using an entire month's data to train your model.

Back

Candidate Generation

Next

Ranking

Completed

69% completed, meet the criteria and claim your course certificate!

Report an Issue

Ask a Question (https://discuss.educative.io/tag/training-data-generation__recommendation-system__grokking-the-machine-learning-interview)

Ranking

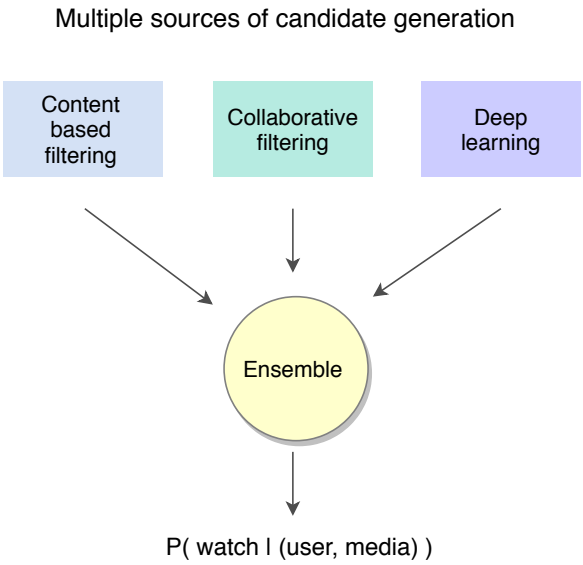
Let's look at modeling options for the recommendation system.

We'll cover the following

- Approach 1: Logistic regression or random forest
- Approach 2: Deep NN with sparse and dense features
- Network structure
- Re-ranking

The ranking model takes the top candidates from multiple sources of candidate generation that we have discussed (https://www.educative.io/collection/page/10370001/6237869033127936/5808795478392832). Then, an ensemble of all of these candidates is created, and the candidates are ranked with respect to the chance of the user watching that video content.

Here, your goal is to rank the content based on the probability of a user watching a media given a user and a candidate media, i.e., P(watch | (User, Media)).



Ensemble of candidate generation models

There are a few ways in which you can try to predict the probability of *watch*. It would make sense to first try a simplistic approach to see how far you can go and then apply complex modelling approaches to further optimize the system.

First, we will discuss some approaches using logistic regression or tree ensemble methods and then a deep learning model with dense and sparse features.

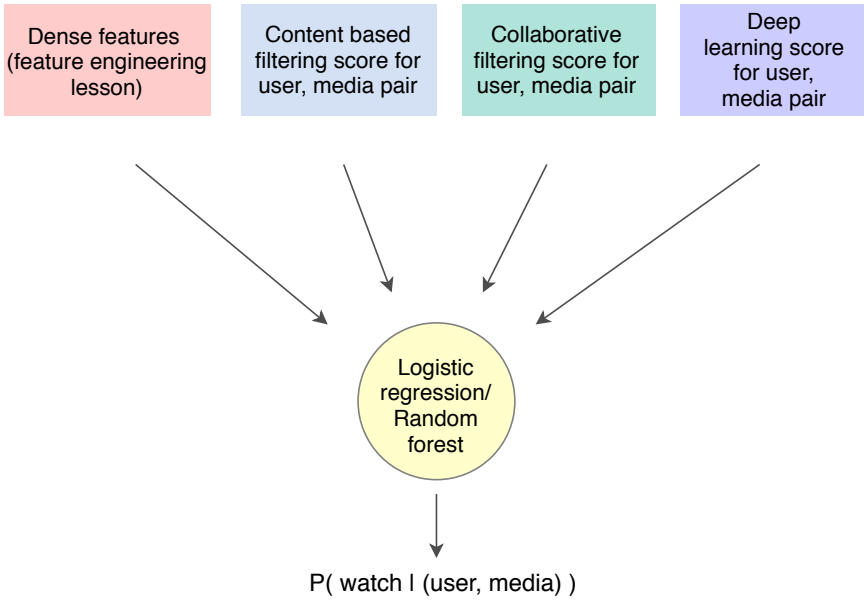
Deep learning should be able to learn through sparse features and outperform simplistic approach. Still, as we discussed in earlier problems, generalization with a deep NN model needs an order of magnitude more data (order of 100 of millions of training examples) and training capacity/time (100x more CPU cycles).

Approach 1: Logistic regression or random forest

There are multiple reasons that training a simplistic model might be the way to go. They are as follows:

- Training data is limited
- You have limited training and model evaluation capacity
- You want model explainability to really understand how the ML model is making its decision and show that to the end-user
- You require an initial baseline to see how far you can go in reducing our test set loss before you try more complex approaches

Along with other important features we discussed in the feature engineering section, output scores from different candidate selection algorithms are also a fairly important input consumed by the ranking models.



Approach 1

It is critical to minimize the test error and choose hyperparameters for training and regularization that gives us the best result on the test data.

Approach 2: Deep NN with sparse and dense features

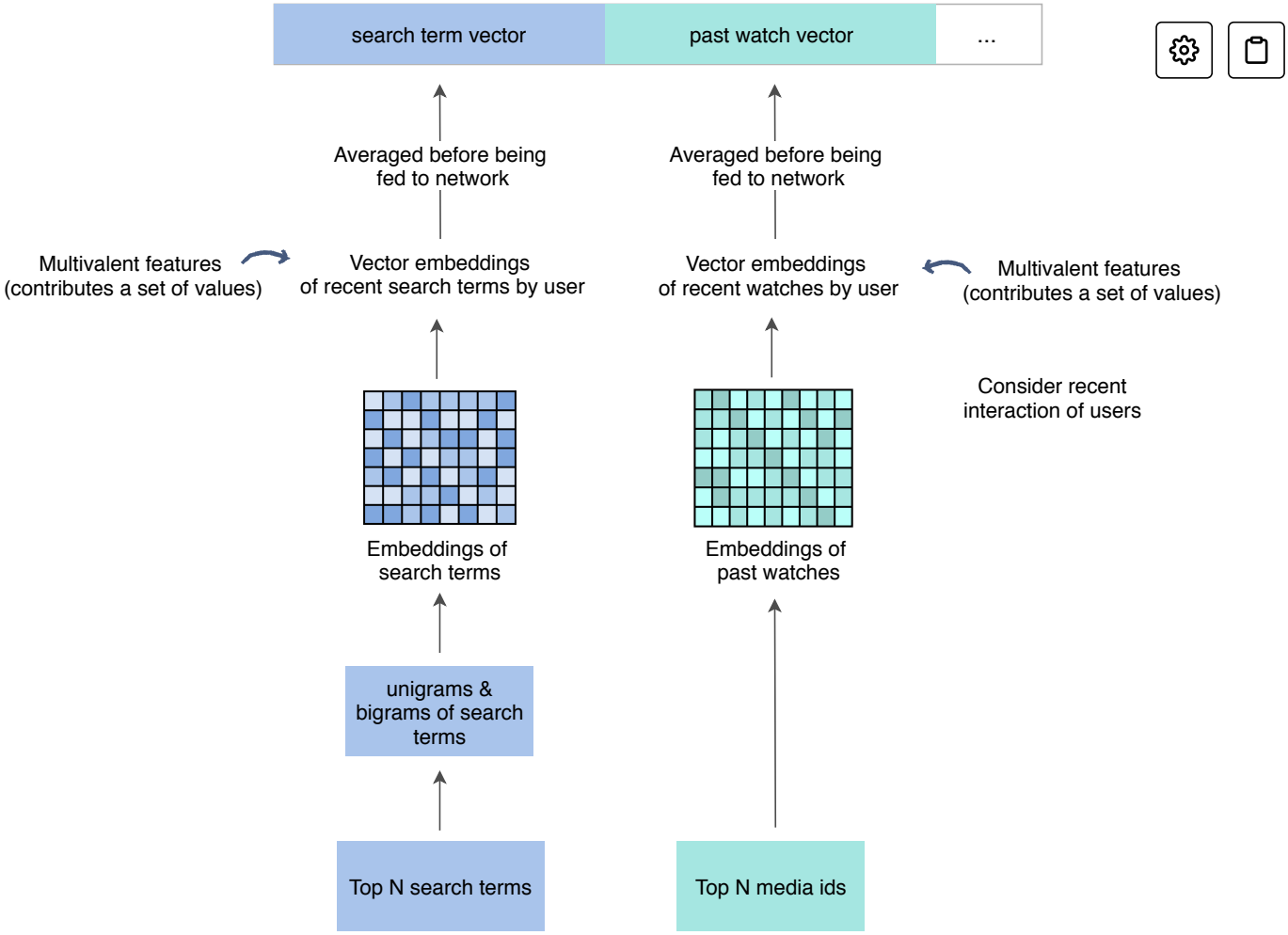
Another way to model this problem is to set up a deep NN. Some of the factors that were discussed in Approach 1 are now key requirements for training this deep NN model. They are as follows:

- Hundreds of millions of training examples should be available
- Having the capacity to evaluate these models in terms of capacity and model interpretability is not that critical.

It's important to call out that given the scale of Netflix, fulfilling the above requirements should not be a problem. Utilizing large scale data will definitely be able to outperform simplistic approaches discussed earlier.

Since the idea is that you want to predict whether the user will watch the media or not, you train a deep NN with sparse and dense features for this learning task. Two extremely powerful sparse features fed into such a network can be videos that the user has previously watched and the user's search terms. For these sparse features, you can set up the network to also learn media and search term embeddings as part of the learning task. These specialized embeddings for historical watches and search terms can be very powerful in predicting the *next watch idea* for a user. They will allow the model to personalize the recommendation ranking based on the user's recent interaction with media content on the platform.

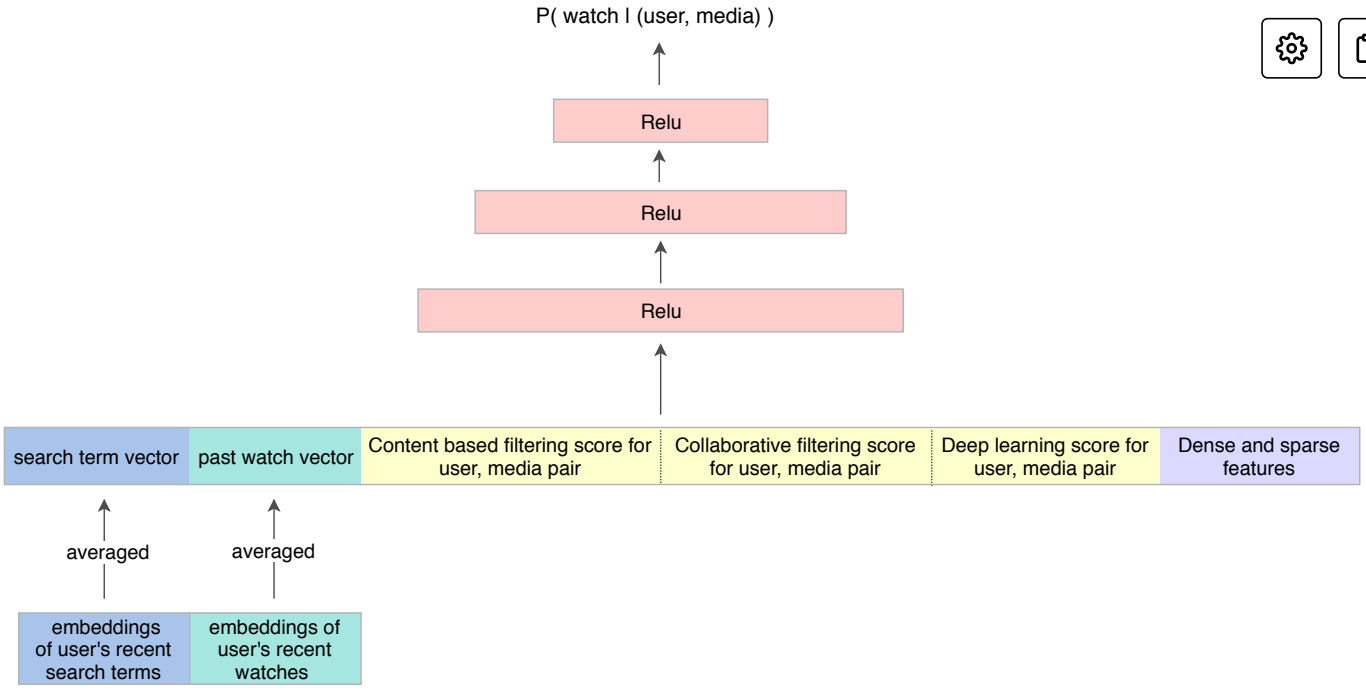
An important aspect here is that both search terms and historical watched content are list-wise features. You need to think about how to feed them in the network given that the size of the layers is fixed. You can use an approach similar to pooling layers in CNN (convolution neural networks) and simply average the historical watch id and search text term embeddings before feeding it into the network.



Generating features from historical watches and search terms

Click to expand: out of vocabulary searches / watches

One way to set up the network is shown below utilizing these sparse data embeddings and feeding them into the Neural network.



Training a deep neural network with embedding and other dense features

As shown above, you can set it up as multiple RELU units layered on top of the embeddings that you learned along with other features. The top layer will predict whether the media will be watched or not using logistic loss.

Network structure

How many layers should you setup? How many activation units should be used in each layer? The best answer to these questions is that you should start with 2-3 hidden layers with a RELU based activation unit and then play around with the numbers to see how this helps us reduce the test error. Generally, adding more layers and units helps initially, but its usefulness tapers off quickly. The computation and time cost would be higher relative to the drop in error rate.

Re-ranking

The top ten recommendations on the user's page are of great importance. After your system has given the watch probabilities and you have ranked the results accordingly, you may re-rank the results.

Re-ranking is done for various reasons, such as bringing diversity to the recommendations. Consider a scenario where all the top ten recommended movies are comedy. You might decide to keep only two of each genre in the top ten recommendations. This way, you would have five different genres for the user in the top recommendations.

If you are also considering past watches for the media recommendations, then re-ranking can help you. It prevents the recommendation list from being overwhelmed by previous watches by moving some previously watched media down the list of recommendations.