Chapter 14
Recommender Systems

In this chapter we will be exploring **recommender systems**. We will take a look at how to utilize **matrix factorization** for the recommendation process, limitations of the approach, and solutions.

A recommender system is, in intuitive terms, an algorithm that takes n users and m items, and recommends users items that they will consume (i.e. products users will buy on Amazon, videos users will watch on YouTube). Typically, $n \gg m$, and there are many different ways that such a system can be implemented, a few of which we will briefly explore below.

14.1 Popularity

In this type of recommender system, the items recommendeed are purely what are the most popular at the moment. For example, if a vast majority of Netflix users have been watching the show "Squid Game" due to it's popularity, it will be one of the top shows recommended to other users. Though a system like this is easy to implement, it is prone to positive feedback loops and lacks personalization for individual users, whose interests may not always perfectly align with the most popular items.

14.2 User-User

We can typically do better than a simple popularity-based recommender system, and one of those such methods is with one based on nearest users. In other words, given some user u_i, compute some k nearest neighbors and recommend the items that the users nearest to them. The users and the features for each user are stored in an $n \times m$ matrix, with m representing some number of items that each user has interacted with. The interaction (i.e. liking a Youtube video, leaving a 1 star Amazon review) is recorded and then used as each user's features to calculate the distance between them. Similar users are then recommended products that users around them interacted positively with.

14.3 Item-Item

Another method is to recommend items that are commonly consumed in tandem. For example, if users who buy baby formula also often buy diapers on Amazon, then a user who is bought baby formula will be recommended to buy diapers.

14.3.1 Co-occurence Matrix

Definition 14.1: Co-occurence Matrix

A **co-occurence matrix** is a popular type of recommender system used to recommend data based on similarity.

Example(s)

Since a co-occurence matrix can recommend things based on similarity, a good application is product recommendation. If we have m products in our store, our co-occurence matrix C will be of size $m*m$, and C_{ij} is the number of people who bought products both i and j, where i is along the length of our matrix and j is along the width.

Note that we do have to normalize our data when using a co-occurence matrix, because we don't want an item to be falsely associated with all the other items in the matrix just because it is a popular item. We can normalize by using the $JaccardSimilarity$:

$$\text{Similarity between items } i \text{ and } j \; = \frac{\#\text{ purchases } i \textbf{ and } j}{\#\text{ purchases } i \textbf{ or } j} \tag{14.28}$$

14.3.2 Limitations of the Co-occurence Matrix

While the co-occurence matrix is good at personalizing to a user based on their purchase history, it leaves out some information that could be equally useful in predicting a user's purchasing behavior. For example, context (such as the time of day), the demographics of the users, and the product features are all data that are not captured by the co-occurence matrix but could have helped recommendation. Finally, a co-occurence matrix is also not scalable. As we start off with a fixed size for the matrix, once we add a new item, we come across a **cold start problem**.

Definition 14.2: Cold Start Problem

The **Cold Start problem** is an issue that we will repeatedly visit when discussing the limitations of recommender systems. In essence, this is the issue that occurs when a new item is added to our dataset, and the fact that this new item has no data on it to begin with yields the false computation that this item is not compatible with any other item in the dataset.

14.4 Feature Based

Feature based recommender systems are a possible solution to the cold start problem. They rely on features of the product (i.e. the genre/release year of a movie) and can be additionally enhanced with user specific features (i.e. age, gender identity) in order to provide recommendations. Now, when new users join or new products are listed, the system already has some information about the user or item intrinsically (i.e. user's age, movie's genre), thereby allowing us to solve the cold start problem.

Some weights $w_G \in \mathbb{R}^d$ are first defined for all users.

Then, the following linear model can be fitted where R is the total number of ratings. Note that we are trying to solve for $\hat{r}_v$, or in other words, the predicted rating that will be assigned to a product based on its features.

$$\hat{r}_v = w_G^T h(v) = \sum_{i=0}^{d} w_{G,i} h_i(v)$$

$$\hat{w}_G = \underset{w}{\text{argmin}} \frac{1}{R} \sum_v (w_G^T h(v) - r_v)^2 + \lambda ||w_G||$$

In order to personalize these results and instead solve for $\hat{r}_{u,v}$, the rating for a product personalized to the user, we can take the following two approaches.

14.4.1 Add User-Specific Features

Intuitively, we are just appending to our existing item-specific feature matrix, denoted $h(v)$ the user-specific features denoted $h(u)$. We can simply rewrite the equations above taking into consideration these features. Note that d now represents the total number of features across the item and user specific features, instead of just the item-specific features.

$$\hat{r}_{u,v} = w_G^T h(u,v) = \sum_{i=0}^{d} w_{G,i} h_i(u,v)$$
$$\hat{w}_G = \underset{w}{\operatorname{argmin}} \frac{1}{R} \sum_{u,v} (w_G^T h(u,v) - r_{u,v})^2 + \lambda ||w_G||$$

14.4.2 Linear Mixed Models

The Linear Mixed Model approach relies on the inclusion of a user-specified deviation from the global model. The intuition for this approach is that the global weights will be modified by a set of weights specific to each user denoted $\hat{w}_u$ to instead predict $\hat{r}_{u,v}$

$$\hat{r}_{u,v} = (\hat{w}_G + \hat{w}_u)^T h(v)$$

New users have their $\hat{w}_u$ initialized to the zero vector, but update over time based on the residuals of the global model or with Bayesian Update. While the latter is out of the scope of this class, the general process is that the vector is initialized with a probability distribution over user-specific deviations, then gets updated as more data is acquired.

14.5 Matrix Factorization

Definition 14.3: Matrix Factorization

Matrix factorization is another type of recommendation system. It utilizes ratings instead of similarities.

Example(s)

As matrix factorization utilizes ratings, a common application of this recommender system is movie recommendations.

Let's consider the movie recommendations example to dissect matrix factorization. First and foremost, note that for our dataset, we need a collection of ratings from a multitude of users for every movie. Not every user will rate every movie they watch, so our dataset is very **sparse.** In matrix factorization, we account for this by trying to **predict** what a given user would rate a movie based on other patterns in the data. This is virtually impossible to do with complete accuracy, because people can be unpredictable. So, we can only try our best by making assumptions about the dataset.
First, let's assume that there are k types of movies, and every movie belongs to at least one of those k types and every user enjoys at least one of those k types. Thus, we describe each movie in our dataset v with a feature vector R_v of length k, such that every value inside of R_v is how much v belongs into one of the k movie types. We can describe each user in our dataset u with a feature vector L_u of length k, such

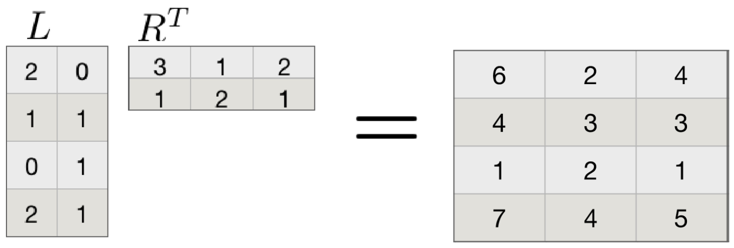

Figure 14.76: An example of a dot-product calculation of a user feature vector L and a movie feature vector R.

that every value inside of L_u is how much u enjoys each of the k movie types. Thus, our predicted rating $R\hat{ating}(u, v) = L_u \cdot R_v$, the dot product of movie v's features and user u's features.

Hence, the problem reduces down to a regression: We must find L and R such that when multipled, achieve predicted ratings that are close to the values that we have data for. Our quality metric is thus:

$$\hat{L}, \hat{R} = argmin_{L,R} \sum_{u,v:r?} (L_u \cdot R_v - r_{uv})^2 \tag{14.29}$$

$u, v : r?$ are the entries with known ratings.

Recall that in a regression problem, we use gradient descent to update all of our parameters at once. In this problem, since we have two unknowns $\hat{L}$ and $\hat{R}$, we will instead use **coordinate descent**, which is a similar optimizing technique to gradient descent except it updates one coordinate in our optimization at a time. We alternate between updating coordinates to simplify the computation for each round of optimization.

14.5.1 Coordinate Descent for Matrix Factorization

We will first begin by optimizing for L. One key insight is that **what one user prefers should have no influence on another user** so not only can we optimize for L by fixing R, we can also optimize for each user's feature vector L_u one at a time. This drasticallu simplifies the computation we have to do:

$$\text{for each user } \hat{L_u} = argmin_{L_u} \sum_{v \in V_u} (L_u \cdot R_v - r_{uv})^2 \tag{14.30}$$

where V_u are all the movies user u has rated, and $R_v - r_{uv}$ is fixed. Now, since everything in this formula except for L_u is fixed, we can treat this as a linear regression problem where $R_v - r_{uv}$ is an input and L_u is a coefficient we are intending to learn. Since this is now just linear regression, we can easily use gradient descent. The same logic goes for movies, so when we alternate to predicting $\hat{R}$, we can compute each R_v separately.

14.5.2 Using Results

Via matrix factorization, we can effectively use the movie features R to discover "topics" of a movie v. This is useful for categorizing movies into genres, or adding keywords to movies so that they show up with specific search queries. We can also use user features to discover "topic preferences" of a user u to recommend relevant movies to that user.

14.5.3 Limitations of Matrix Factorization

Although matrix factorization performs a more personalized computation for each user, it still does not capture context and fails to solve the cold start problem.

14.6 Blending Models: Featured Matrix Factorization

With matrix factorization, consider the scenario where a movie database adds a new user. As this user has no data yet, all of their preferences for every movie category starts off as 0. So, when trying to update the future preferences of this user, we fail to predict reasonably. The solution to this is to supplement matrix factorization with another machine learning model. We will define a feature vector for each movie that takes context into consideration, like the movie's genre, the year it was made, and maybe the director. Then, we will define a model that learns these features for *all* the users in the database. So, if we add a new user, instead of starting their preference for everything at 0, we just set it to be what movies seem to be popular with the entire population as a whole.

14.7 Evaluating Recommendations

Classification accuracy will not serve us well with recommendation systems since we do not care too much about what a user does not like, so it is not very practical to go through every single movie in the entire database and label it with a user's liking or disliking. Instead, we can measure **precision and recall** by assessing how many of our recommendations did the user actually like, and how many of the items that the user liked did we recommend.