10
Rating-Based Collaborative Filtering: Algorithms and Evaluation

Daniel Kluver[1](✉), Michael D. Ekstrand[2], and Joseph A. Konstan[1]

[1] GroupLens Research, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, USA
{kluver,konstan}@cs.umn.edu

[2] People and Information Research Team (PIReT), Department of Computer Science, Boise State University, Boise, USA
michaelekstrand@boisestate.edu

Abstract. Recommender systems help users find information by recommending content that a user might not know about, but will hopefully like. Rating-based collaborative filtering recommender systems do this by finding patterns that are consistent across the ratings of other users. These patterns can be used on their own, or in conjunction with other forms of social information access to identify and recommend content that a user might like. This chapter reviews the concepts, algorithms, and means of evaluation that are at the core of collaborative filtering research and practice. While there are many recommendation algorithms, the ones we cover serve as the basis for much of past and present algorithm development. After presenting these algorithms we present examples of two more recent directions in recommendation algorithms: learning-to-rank and ensemble recommendation algorithms. We finish by describing how collaborative filtering algorithms can be evaluated, and listing available resources and datasets to support further experimentation. The goal of this chapter is to provide the basis of knowledge needed for readers to explore more advanced topics in recommendation.

1 Introduction

One problem with online collections is *information overload* - when presented with too much information people have trouble making informed decisions. While the tools for searching, visualizing, and navigating these large collections introduced in previous chapters of this book can help users find content, even these tools can be insufficient if an online collection is big enough, or if the user is unsure of exactly what content they are interested in. Ideally, a system should know what kind of items each user is interested in without ever being told. Then the system can focus on presenting each user only those items that they are most likely to be interested in.

This idea has led to a proliferation of strategies for helping users focus only on the items they will like. The most basic strategy is to focus on the most

© Springer International Publishing AG, part of Springer Nature 2018
P. Brusilovsky and D. He (Eds.): Social Information Access, LNCS 10100, pp. 344–390, 2018.
https://doi.org/10.1007/978-3-319-90092-6_10

popular items, or those that are reviewed most favorably by other users. While not personalized for a given user, these strategies can quickly guide users to the best content the system has to offer. It isn't even that hard to do basic personalization within these simple strategies. For example, if a system knows what genres of music a user tends to listen to, then the system can focus on presenting popular artists from that genre.

In the early 1990s, these strategies set the stage for *collaborative filtering recommendation systems*. The insight behind collaborative filtering recommender systems is that people have relative stable tastes. Therefore, if two people have agreed in the past they will likely continue to agree in the future. A key part of this insight – and the major difference between this and the personalization strategies that came before – is that it does not matter why two users agree. They could share tastes in books because they both like the same style of book binding, or they could share taste in movies due to a nuance of directing; a collaborative filtering recommender system does not care. So long as the two users continue agreeing, we can use the stated preferences of one user to predict the preferences of another. Since we need very little supplementary information, collaborative filtering algorithms are applicable in a wide range of possible circumstances.

Since the mid 1990s, collaborative filtering recommender systems have become very popular in industry. Companies like Amazon, Netflix, Google, Facebook and many others, have deployed collaborative filtering algorithms to help their users find things they would enjoy. The popularity of these deployments has pushed the field of recommender systems, leading to faster, more accurate recommender systems. These improvements have been coupled to changes in how we think about deploying collaborative filtering systems to support users. Collaborative filtering systems were originally seen as a filter which could separate the interesting items from the uninteresting ones, hence the term collaborative *filtering*. As the field has advanced, it become more common to think of these algorithms as recommending a short list of the best items for a user. Even if there are plenty of good items that go unrecommended, the a recommendation algorithm is doing its job if it's list contains the best of the best for a given user.

Like other forms of social navigation, collaborative filtering algorithms rely on the connections and patterns made possible by large bodies of users. Despite this, collaborative filtering algorithms are not typically social in the traditional sense: while one user's behavior does directly affect other users' experiences, this is rarely made clear to the users. Most recommender system users are blissfully unaware of how their actions benefit not only themselves, but other users with similar tastes.

This chapter describes the foundational collaborative filtering algorithms and methods for evaluating these algorithms for use in a given application. In particular we will focus on *rating based* algorithms, in which user preference is measured by numerical ratings. After presenting the most common algorithms for rating based collaborative filtering, we will present two more recent approaches, a learning to rank algorithm and ensemble methods. While these approaches are still popularly deployed today, many extensions and applications of the algorithms we describe go well beyond what we can present here. Later chapters

in this book are dedicated to recommendation algorithms that leverage social connections (Chap. 11 [44]), social tags (Chap. 12 [8]), user reviews (Chap. 13 [51]), implicit (non-rating) preference feedback (Chap. 14 [35]), and ways to recommend new social connections (Chap. 15 [25]). The goal of this chapter is to describe the foundational algorithms that are built upon in these later chapters. We also cover topics such as algorithm evaluation that are relevant throughout the following chapters.

1.1 Examples of Recommender Systems

There are many different ways recommendation algorithms can be incorporated into an online service. The most simple is the "streaming" style service, which is oriented around a stream of recommended content. Two examples of this are streaming music services like Pandora[1] and the Jester joke recommender[2]. Screenshots of these services is shown in Figs. 1 and 2. Both services share the same design: the user is presented with content (music or jokes). After each item the user is given the opportunity to evaluate the item. These evaluations influence the algorithm which then picks the next song or joke. This process repeats until the user leaves. Jester is known to use a collaborative filtering algorithm [24]. Interested readers can find more information about jester and even download a rating dataset for experimentation from the Jester web page. As a commercial product, less is known about Pandora's algorithm. However, it is reasonable to assume that they are using a hybrid algorithm that combines collaborative filtering information with their catalog of song metadata.

A quite different way to use recommendation algorithms can be seen in catalog based websites like MovieLens[3]. MovieLens is a movie recommender developed by the GroupLens research lab. On the surface MovieLens is similar to other movie catalog websites such as the Internet Movie DataBase (IMDB) or The Movie DataBase (TMDB). All three have pages dedicated to each movie detailing information about that movie and search features to help users find information about a given movie. MovieLens goes further, however, by employing a collaborative filtering algorithm. MovieLens encourages users to rate any movie they have seen, MovieLens then users these ratings to provide personalized predicted ratings which it shows alongside a movie's cover art in both movie search and detail pages. These predictions can help users rapidly decide if it is worth learning more about a movie. Users can also ask MovieLens to produce a list of recommended movies, with the top 8 most recommended movies for a user being centrally positioned on the MovieLens home page, this can be seen in Fig. 3.

A third common way to use recommendation algorithms is in e-commerce systems, perhaps the most notable being Amazon[4]. Amazon is an online store

[1] https://www.pandora.com/.
[2] http://eigentaste.berkeley.edu/.
[3] https://movielens.org/.
[4] https://www.amazon.com/.

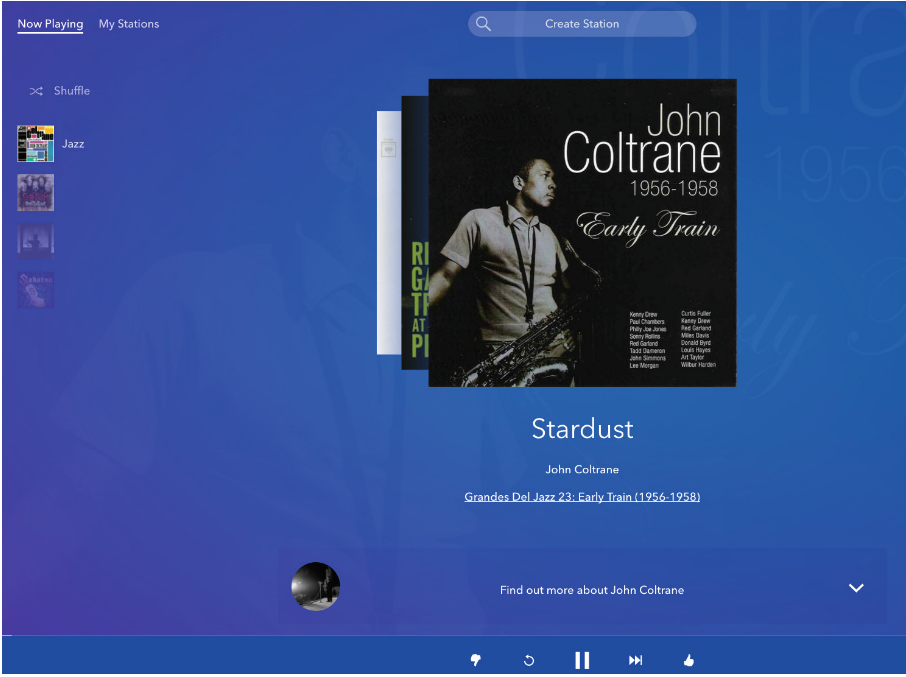


Fig. 1. Screenshot of the Pandora music streaming service

One day the first grade teacher was reading the story of the Three Little Pigs to her class. She came to the part of the story where the first pig was trying to accumulate the building materials for his home. She read, "...and so the pig went up to the man with the wheelbarrow full of straw and said, 'Pardon me sir, but may I have some of that straw to build my house?'"

The teacher paused then asked the class, "And what do you think that man said?"

One little boy raised his hand and said, "I know...he said, 'Holy Shit! A talking pig!'"

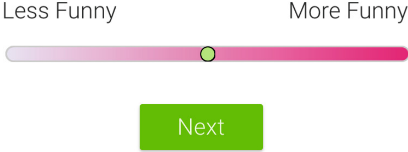


Fig. 2. Screenshot of the Jester joke recommender

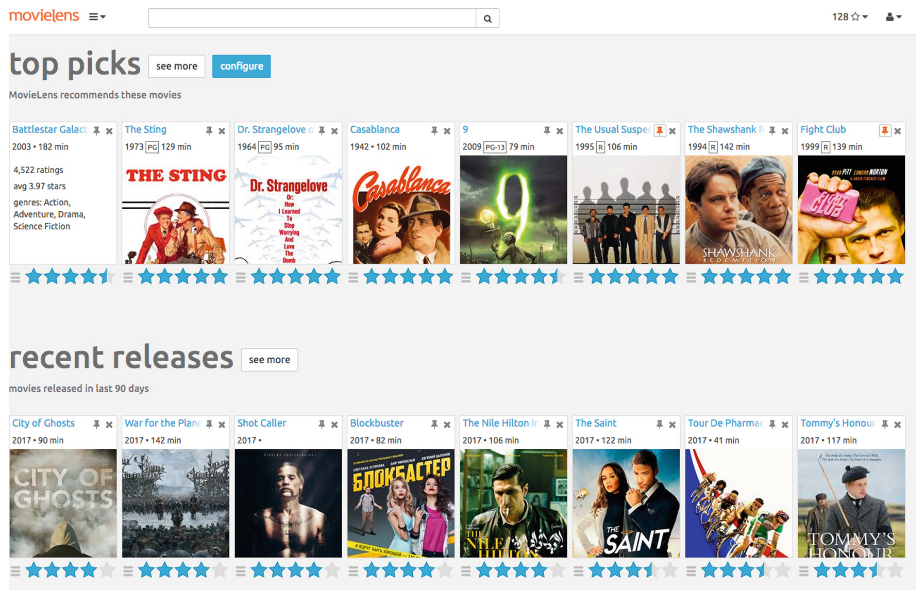


Fig. 3. Screenshot of the MovieLens home page

which started as a bookstore, but has since diversified to a general purpose online storefront. While the average user may not notice the recommendations in Amazon (or at the very least may think little of them) much of the Amazon storefront is determined based on recommendation algorithms. A screenshot of the Amazon main page for one author is shown in Fig. 4 with recommendation features highlighted. Since only a small proportion of users use reviews on Amazon it is likely that Amazon uses data beyond ratings in their collaborative filtering algorithm. Unlike MovieLens, getting information and recommendation is not the primary motivation of Amazon users. Therefore, while the basic interfaces may be similar, the way recommendations are used, and the algorithm properties that a system designer might look for, will be different.

As these examples show, recommendation algorithms can be useful in a wide range of situations. That said, there are some commonalities: each service has some way of learning what users like. In MovieLens and the streaming services users can explicitly rate how much they like a movie, joke, or song. In Amazon purchase records and browsing history can be used to infer user interests. Each service also has some way of suggesting one or more item to the user based on their recommendation algorithm. It will be helpful to keep these examples in mind as they will help anchor the more abstract algorithm details covered in this chapter to a specific context of use.

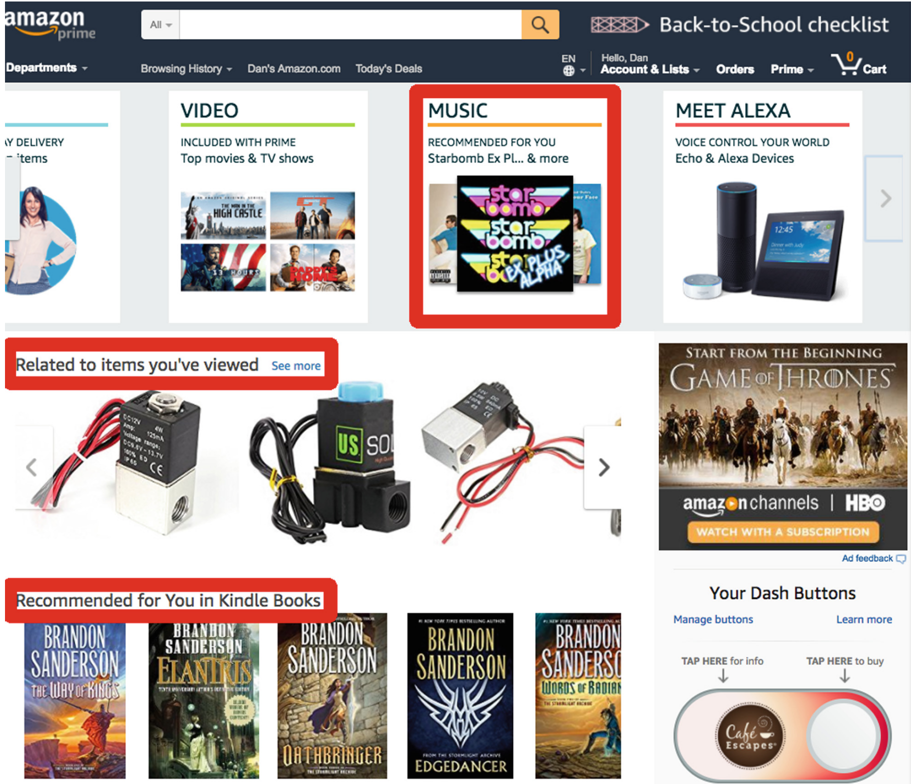


Fig. 4. Screenshot of the Amazon home page

1.2 A Note on the Organization of Recommendation Algorithms

Every year dozens of new recommendation algorithms are introduced. It should be no surprise, therefore, that there have been various attempts organizing these algorithms into a taxonomy or classification scheme for collaborative filtering algorithms. The purpose of any such organization is to allow better communication about how an algorithm works, and what other algorithms it is similar to, by describing where that algorithm is in a taxonomy.

To some degree these classifications have been useful; chapters in this book, for example, are organized based on important distinctions between different types of algorithms. Other distinctions that have been made are less useful, either because algorithms have advanced to the point where a distinction has no meaning, or because the classification itself has been used inconsistently. In this chapter we will restrict ourselves to categorizations that we feel are useful for communication. That said, we note that other works on recommender systems that a reader might explore are still organized under some of these traditional taxonomies. Therefore we will introduce some of these distinctions now so the

reader can be aware of them if they wish to read other resources on recommendation algorithms.

One important distinction that has been made between algorithms is between collaborative filtering algorithms (like those discussed in this chapter) and content-based algorithms. Collaborative filtering algorithms, as was described earlier, operate by finding patterns in user behavior that can be used to predict future behavior. The traditional example of this would be that two users tend to like the same things, therefore when one use likes something, we can predict the other user will as well. *Content based* algorithms, on the other hand, focus on relationships between users and the content they like. A traditional example of this would be an algorithm that learns which genres of music a user likes and recommends songs from that genre. While still meaningful, the line between collaborative and content based filtering has become somewhat blurry as modern algorithms have sought to combine the strengths of both algorithms. Readers can still expect to see this distinction made in new publications (including this one) as the algorithms that are both content based and collaborative filtering algorithms are still in the minority.

Within the specific range of collaborative filtering algorithms, the most common taxonomy separates so-called model-based algorithms and memory-based algorithms. The division was first made in a 1998 paper [9] where memory-based algorithms were defined as those that operate over the entire dataset, where model-based algorithms are those that use the dataset to estimate a model which can then be used for predictions. For recommender systems this split is problematic as many algorithms can be described sufficiently as a memory-based algorithm or a model-based algorithm depending on how the algorithm is optimized and deployed.

More recently this same distinction has been used more usefully to separate based on the basic design of an algorithm [66]. Model-based algorithms are those that use machine learning techniques to fit a parametrized model, while memory-based algorithms search through the training data to find similar examples (users or items). These examples are then aggregated to compute recommendations.

While still common we personally find this latter separation does not do a great job of communicating about the distinctions between algorithms. Therefore we will eschew this taxonomy and present algorithms grouped, and labeled, by their mathematical structure or motivation. In the next section we will cover the basic concepts and mathematical notation that will be used throughout this chapter. The section after that will describe *baseline algorithms*: simple algorithms which seek to capture broad trends in rating data. Section 4 will describe *nearest neighbor algorithms*: the group of algorithms that have historically been called memory based algorithms, which work by finding similar examples which are used in computing recommendations. Section 5 will describe *Matrix Factorization Algorithms*: a group of algorithms that share a common and powerful mathematical model inspired by matrix factorization. Section 6 will describe *Learning to Rank Algorithms*: algorithms that focus on ranking possible recommendations, instead of predicting what score a user will give a particular

item. Section 7 will briefly mention other groups of algorithms which we do not explore in depth: *graph based algorithms*, *linear regression based algorithms*, and *probabilistic algorithms*. Section 8 will describe *ensemble methods*: ways to combine multiple recommenders. Finally, Sect. 9 will explore metrics and evaluation procedures for collaborative filtering algorithms.

2 Concepts and Notation

In this section we will discuss the core concepts and mathematical notations that will be used in our discussion of recommendation algorithms (summarized in Table 1). The two most central objects in a recommendation system are the *users* the system recommends to and the *items* the system might recommend. These terms are purposely domain neutral as different domains often have domain specific terms for these concepts.

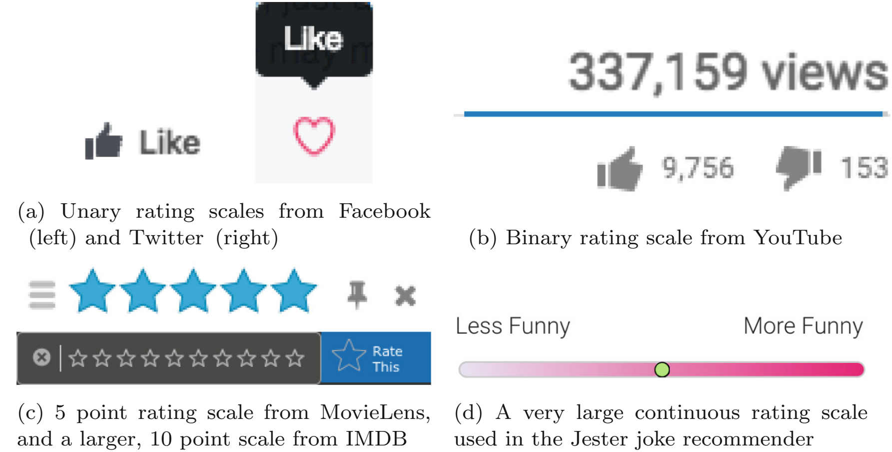
One *user* represents one independently tracked account for recommendation. Typically this represents one system account, and is assumed to represent one person's tastes. We will denote the set of all users as U with $u, v, w \in U$ being individual users from the set.

One *item* represents one independently tracked thing that can be recommended. In most systems its obvious what services or products should map to an item in the recommendation algorithm; in an e-commerce system like Amazon or Ebay, each product should an item. In a movie recommender each movie should be an item. In other domains there might be more uncertainty; in a music recommender should each song be an item (and recommended individually) or should each an album be an item? We will denote the set of all items as I with $i, j, k \in I$ being individual items from the set.

Most traditional collaborative filtering recommender systems are based on *ratings*: numeric measures of a user's preference on an item. Ratings are collected from users on a given *rating scale* such as the 1-to-5 star scale used in MovieLens,

Table 1. Summary of mathematical notation

Users	The set of all profiles in the system	U
	A profile in the system, usually one person	$u, v, w \in U$
Items	The collection of things being recommended	I
	A member of the collection of items	$i, j, k \in I$
Rating	A measure of a user's preference for an item	$r_{ui} \in R$
User's ratings	The set of all items rated by one user	I_u
	The vector of all ratings made by one user	$\mathbf{r}_u$
Item's ratings	The set of all users who have rated one item	U_i
	The vector of all ratings made on one item	$\mathbf{r}_i$
Score/Prediction	An algorithm's score assigned to a user and item	$S(u, i)$



(a) Unary rating scales from Facebook (left) and Twitter (right)

(b) Binary rating scale from YouTube

(c) 5 point rating scale from MovieLens, and a larger, 10 point scale from IMDB

(d) A very large continuous rating scale used in the Jester joke recommender

Fig. 5. Examples of various rating scales used in the wild. (https://www.facebook.com/, https://twitter.com/, https://www.youtube.com)

Amazon, and many other websites or the ten star scale used by IMDB[5] (see Fig. 5c for examples). Rating scales are almost always designed so that larger numbers indicate more preference: a user should like a movie they rated 5/5 stars more than they like a movie they rated 4/5 stars. The rating user u assigns to item i will be denoted r_{ui}.

While there are many different rating scales that have been used, the choice of rating scale is often not relevant for a recommendation algorithm. Some interfaces, however, deserve special consideration. Small scales which can only take one or two values such as a binary (thumbs up, thumbs down) scale (see Fig. 5b) or unary "like" scales (see Fig. 5a) may require special adaptions when applying algorithms designed with a larger scale in mind. For example, when working with the unary "like" scale it can be important to explicitly treat non-response as a form of rating feedback.

While the algorithms in this chapter are focused on rating-based approaches, it is important to understand that they only need a numeric measure of preference. In this way these algorithms can be used with for many different forms of preference feedback. Chapter 14 in this book [35] covers recommendation based on implicit feedback (measurements of user preference that are not ratings). One of the strategies covered are ways to convert common types of data into ratings that can be used with the algorithms in this chapter.

Recommender systems often organize the set of all ratings as a sparse *rating matrix* R, which is a $|U| \times |I|$ matrix. R only has values at $r_{ui} \in R$ where user u has rated item i; for all other pairs of u and i, r_{ui} is blank. The set of all items rated by a user u is $I_u \subset I$, the collection of all ratings by one user can also be expressed as a sparse vector $\mathbf{r}_u$. Similarly, the set of all users who have rated one item i is $U_i \subset U$, and the collection of these ratings can be expressed as a sparse vector $\mathbf{r}_i$.

[5] http://www.imdb.com/.

As the rating matrix R is sparsely observed, one of the ways collaborative filtering algorithms can be viewed is *matrix completion*. Matrix completion is the task of filling in the missing values in a sparse matrix. In the recommendation domain this is also called the *prediction task* as filling in unobserved ratings is equivalent to predicting what a target user u would rate an item i. Rating predictions can be used by users to quickly evaluate an unknown item: if the item is predicted highly it might be worth further consideration.

Like we described in MovieLens, one use of predictions is to sort items by predicted user preference. This leads to the view of a recommender algorithm as generating a ranking score. The *ranking task* is to generate a personalized ranking of the item set for each user. Any prediction algorithm can be used to generate rankings, but not all ranking algorithms produce scores that can be thought of as a prediction. Algorithms that focus specifically on the ranking task are known as *learning-to-rank* recommenders and will be discussed towards the end of this chapter.

Both prediction and ranking oriented algorithms produce a score for each user and item. Therefore, we will use the syntax $S(u, i)$ to represent the output of both types of algorithm. Almost every algorithm we discuss in this chapter will produce output as a score for each user and item.

One of the most interesting applications of collaborative filtering recommendation technology is the *recommendation task*. The *recommendation task* is to generate a small list of items that a target user is likely to want. The simplest approach to this is *Top-N recommendation* which takes the N highest ranked items by a ranking or prediction algorithm. More advanced approaches involve combining ranking scores with other factors to change the properties of the list of recommendations.

For each task there are associated metrics which can be used to evaluate the algorithm. Prediction algorithms can be evaluated by the accuracy of their prediction, and ranking algorithms can be compared to how users rank items. Recommendation algorithms are a particular challenge to evaluate, however, as users are sensitive to properties such as the diversity of the recommendations, or how novel the recommended items are. Evaluation is an important concept in the study of recommender systems, especially as some algorithms are partially defined by specific evaluation metrics. We will discuss evaluation approaches in more detail in Sect. 9.

3 Baseline Predictors

Before describing true collaborative filtering approaches, we will first discuss *baseline predictors*. Baseline predictors are the most simple approaches for rating. While a baseline predictor is rarely the primary prediction algorithm for a recommender system, the baseline algorithms do have their uses. Due to their simplicity, baseline predictors are often the most reliable algorithms in extreme conditions such as new users [37]. Because of this, baseline predictions are often used as a fallback algorithm in cases where a more advanced algorithm might fail.

Baseline predictions can also be used to establish a minimum standard of performance when comparing new algorithms and domains. Finally, baseline prediction algorithms are often incorporated into more advanced algorithms, allowing the advanced algorithms to focus on modeling deviations from the basic, expected patterns that are already well captured by a baseline prediction.

The most basic baseline is the global baseline, in which one value is taken as the prediction for every user and item $S(u,i) = \mu$. While any value of μ is possible, taking μ as the average rating minimizes prediction error and is the standard choice. The global baseline can be trivially improved by using a different constant for every item or user leading to the item baseline $S(u,i) = \mu_i$ and the user baseline $S(u,i) = \mu_u$ respectively. In the item baseline μ_i is an estimate of the item i's average rating. This allows the item baseline to captures differences between different items. In particular, some items are widely considered to be good, while others are generally considered to be bad. In the user baseline μ_u is an estimate of the user u's average rating. This allows the user baseline to capture differences between how users tend to use the rating scale. Because most rating scales are not well anchored, two users might use different rating values to express the same preference for an item.

This discussion leads us to the generic form of the baseline algorithm, the user-item bias model, given in Eq. 1.

$$S(u,i) = \mu + b_u + b_i \tag{1}$$

Equation 1 has three variables: μ, the average rating in the system; b_i, the item bias representing if an item is, on average, rated better or worse than average; and b_u the user bias representing whether the user tends to rate high or low on average. By combining all three baseline models we are able to simultaneously account for differences between users and items, albeit in a very naive way. This equation is sometimes referred to as the personalized mean baseline as it is *technically* a personalized prediction algorithm, even though the personalization is very minimal.

This model can be learned many ways [40], but are most easily learned with a series of averages, with μ being the average rating, b_i being the item's average rating after subtracting out μ and b_u being the user's average rating after subtracting out μ and b_i [21]. The following equations can be used to compute μ, b_i and b_u:

$$\mu = \frac{\sum_{r_{ui} \in R} r_{ui}}{|R|} \tag{2}$$

$$b_i = \frac{\sum_{u \in U_i} (r_{ui} - \mu)}{|U_i|} \tag{3}$$

$$b_u = \frac{\sum_{i \in I_u} (r_{ui} - b_i - \mu)}{|I_u|} \tag{4}$$

One problem that can lead to poor performance from the user-item baseline is when a user or item has very few ratings. Predictions made on only a few ratings can be very unreliable, especially if the prediction is extremely high or

low. One way to fix this is to introduce a damping term β to the numerator of the computation. Motivated by Bayesian statistics, this term will shrink the bias terms towards zero when the number of ratings for an item is small while having a negligible effect when the number of ratings is large.

$$b_i = \frac{\sum_{u \in U_i}(r_{ui} - \mu)}{|U_i| + \beta} \tag{5}$$

$$b_u = \frac{\sum_{i \in I_u}(r_{ui} - b_i - \mu)}{|I_u| + \beta} \tag{6}$$

Damping parameter values of 5 to 25 have been used in the past [22,37], but for best results β should be re-tuned for any given system.

4 Nearest Neighbor Algorithms

The first collaborative filtering algorithms were nearest neighbor algorithms. These algorithms work by finding similar items or users to the user or item we wish to make predictions for, and then uses ratings on these items, or by these users, to make a prediction. While newer algorithms have been designed, these algorithms are still in use in many live systems. The simplicity and flexibility of these basic approaches, combined with their competitive performance, makes them still important algorithms to understand.

Readers with a background in general machine learning approaches may be familiar with nearest-neighbor algorithms, as these algorithms are a standard technique in machine learning. That said, there are many important details in how recommender systems experts have deployed the nearest neighbor algorithm in the past. These details are the result of careful study in how to best predict ratings in the recommender system domain.

4.1 User-User

Historically, the first collaborative filtering algorithms were user-based nearest neighbors algorithm, sometimes called the user to user algorithm or user-user for short [58]. This is the most direct implementations of the idea behind collaborative filtering, simply find users who have agreed with the current user in the past and use their ratings to make predictions for the current user. User-based nearest neighbor algorithms were quite popular in early recommender systems, but they have fallen out of favor due to scalability concerns in systems with many users.

The first step of the user-user algorithm for a given user u and item i is to generate a set of users who are similar to u and have rated item i. The set of similar users is normally referred to as the user's neighborhood N_u, with the subset who have rated an item i being N_{ui}. Once we have the set of neighbors we can take a weighted average of their ratings on an item as the prediction. Therefore the most important detail in the user-based nearest neighbor algorithm is the similarity function $sim(u, v)$.

A natural and widely-used choice for $sim(u, v)$ is a measurement of the correlation between the ratings of the two users; usually, this takes the form of Pearson's r [27]:

$$sim(u, v) = \frac{\sum_i (r_{ui} - \mu_u)(r_{vi} - \mu_v)}{\sqrt{\sum_i (r_{ui} - \mu_u)^2}\sqrt{\sum_i (r_{vi} - \mu_v)^2}} \tag{7}$$

Alternatively, the rank correlation in the form of Spearman's ρ (the Pearson correlation of ranks rather than values) or Kendall's τ (based on the number of concordant and discordant pairs) can be used. In addition to statistical measures, vector space measures such as the cosine of the angle between the users' rating vectors can be used:

$$sim(u, v) = \frac{\mathbf{r}_u \cdot \mathbf{r}_v}{\|\mathbf{r}_u\|_2 \|\mathbf{r}_v\|_2} \tag{8}$$

In non-rating-based systems, the Jaccard coefficient between the two users' purchased items is a natural choice.

In most published work, Pearson correlation has produced better results than either rank-based or vector space similarity measures for rating-based systems [9, 27]. However, Pearson correlation does have a significant weakness for rating data: it ignores items that only one the users has rated. In the extreme, if two users have only rated two items in common their correlation would be 1. Unfortunately, its unlikely that two users who do not watch the same things would truly be that similar. In general, correlations based on a small number of common ratings trend artificially towards extreme values. While these similar neighbors may have high similarity scores, they often do not perform well as neighbors whose similarity scores are based on a larger number of ratings.

Significance weighting [27] addresses this problem by introducing a multiplier to reduce the measured similarities between between users who have not rated many of the same items. The significance weighting strategy is to multiply the similarity by $\frac{\min(|I_u \cap I_v|, T)}{T}$, where T is a threshold of "enough" co-rated items. This causes the similarity to linearly decrease between users with fewer than T common rated items. Past work has found $T = 50$ to be reasonable, with larger values showing no improvement [27].

There is a natural, parameter-free way to dampen similarity scores for users with few co-occurring items. If items a user has not rated are treated as having the user's average rating, rather than discarded, the Pearson correlation can be computed over all items. When computing with sparse vectors, this can be realized by subtracting each user's mean rating from their rating vectors, then comparing users by taking the cosine between their centered rating vectors and assuming missing values to be 0. This results in the following formula:

$$sim(u, v) = \frac{\hat{r_u} \cdot \hat{r_v}}{\|\hat{r_u}\|_2 \|\hat{r_v}\|_2} \tag{9}$$

$$= \frac{\sum_{i \in I_u \cap I_v} (r_{ui} - \mu_u)(r_{vi} - \mu_v)}{\sqrt{\sum_{i \in I_u} (r_{ui} - \mu_u)^2}\sqrt{\sum_{i \in I_v} (r_{vi} - \mu_v)^2}} \tag{10}$$

This is equivalent to the Pearson correlation, except that *all* of each users' ratings contributes to their term in the denominator, while only the common ratings are counted in the numerator (due to the normalized value for missing ratings being 0). The result is similar to significance weighting. Similarity scores are damped based on the fraction of rated items that are in common; if the users have 50% overlap in their rated items, their resulting similarity will be greater than the similarity between users with the same common ratings but only 20% overlap due to additional ratings of other items. This method has the advantage of being parameter-free, and has been seen to perform at least as well [15, 16, 20].

After picking a similarity function, the next step in predicting for a user u and item i is to compute the similarity between user u and every other user. For systems with many users approximations such as randomly sampling users [29] can improve performance, possibly at a tradeoff of prediction accuracy. Once similarities have been computed the system must choose a set of similar users N_u. Various approaches can be taken here, from using all users to a limited number or only those that are sufficiently similar. Past evaluations have suggested that using the 20 to 60 most similar users performs well and avoids excessive computations [27]. Additionally, by filtering the users the algorithm can avoid the noise that would be introduced by the lower quality neighbors.

Once the algorithm has a set of neighbors N_u the prediction for item i is simply the weighted average of the neighboring users' ratings. Direct averages, without the weighting term, have been used in the past, but tend to perform worse. Let N_{ui} refer to the subset of N_u containing all users in N_u who have rated item i.

$$S(u,i) = \frac{\sum_{v \in N_{ui}} sim(u,v) * r_{vi}}{\sum_{v \in N_{ui}} |sim(u,v)|} \tag{11}$$

These predictions can often be improved by incorporating basic normalization into the algorithm. For example, since users have different average ratings we can take a weighted average of the item's offset from the user's average rating.

$$S(u,i) = \mu_u + \frac{\sum_{v \in N_{ui}} sim(u,v) * (r_{vi} - \mu_v)}{\sum_{v \in N_{ui}} |sim(u,v)|} \tag{12}$$

More advanced normalization is also possible by using z score normalization in which all ratings are first reduced by the user's average rating, and then divided by the standard deviation in the users rating.

The user-based nearest neighbors approach tends to produce good predictions, but is often outperformed by newer algorithms. In this regard, the algorithm is listed here mostly for reference value. Other than its accuracy, one core issue with the performance of the user-based recommender is its slow predict time performance. Most modern recommender systems have a very large number of users which makes finding neighborhoods of users expensive. For good results neighborhood finding should be done online and cannot be extensively cached for performance improvements. This computation makes user-based nearest neighbors very slow for large scale recommender systems, but it remains viable option for a recommender system with many items but relatively few users.

4.2 Item-Item

The item-based nearest neighbor algorithm (sometimes called the item to item or item-item algorithm for short) is closely related to user-user [61]. Where user-user works by finding users similar to the given user and recommending items they liked, item-item finds items similar to the items the given user has previously liked and uses those to make recommendations. Instead of computing similarities between users, item-item computes similarities between items, and uses an average rating over the item neighborhood to make predictions. Unlike user-user, item-item is well suited to modern systems which have many more users than items. This allows item-item some key performance optimizations over user-user which we will address shortly.

To make a prediction for user u and item i item-item first computes a neighborhood of similar items N_{ui}. In practice it is common to limit this neighborhood to only the k most similar items. $k = 30$ is a common value from the academic research, however, different systems may require different settings for optimal performance [61]. Item-item then takes the weighted average of user u's ratings on the items in this neighborhood, and uses that as a prediction.

$$S(u,i) = \frac{\sum_{j \in N_{ui}} sim(i,j) * r_{uj}}{\sum_{j \in N_{ui}} |sim(i,j)|} \tag{13}$$

This equation can be enhanced by subtracting a baseline predictor from the ratings r_{ui} so the algorithm is only predicting deviation from baseline. If this is done, the baseline should be added back in after the fact to make a prediction. Note this will have no effect if the global or per-user baselines are used. The following equation shows how a baseline predictor could be subtracted, using $B(u,i)$ to represent the baseline

$$P_{u,i} = B(u,i) + \frac{\sum_{j \in N_{ui}} sim(i,j) * (r_{uj} - B(u,j))}{\sum_{j \in N_{ui}} |sim(i,j)|} \tag{14}$$

Generally item-item uses the same similarity functions as user-user, simply replacing the user ratings with item ratings.

The cosine similarity metric is the most popular similarity metric for item-item recommendation. Past work has shown that cosine similarity performs better than other traditionally studied similarity functions [61].

The key to getting the best quality predictions using cosine similarity is normalizing the ratings [61]. Evaluations have shown that subtracting the user's average rating from the ratings before computing similarity leads to substantially better recommendations. In practice we have found that subtracting the item baseline or the user-item baseline leads to improvements in performance [20].

$$sim(i,j) = \frac{\sum_{u \in U} (r_{ui} - B(u,i)) * (r_{uj} - B(u,j))}{\sqrt{\sum_{u \in U} (r_{ui} - B(u,i))^2} * \sqrt{\sum_{u \in U} (r_{uj} - B(u,j))^2}} \tag{15}$$

Both Pearson and Spearman similarities have been tried for item-item prediction, but do not tend to perform better than cosine similarity [61]. Just like

Pearson similarity for the user-user algorithm, signficance weighting can improve prediction quality when using Pearson similarity with item-item.

One key advantage of the item-item algorithm is that item similarities and neighborhoods can be shared between users. Since no information about the given user is used in computing the list of similar items, there is no reason that the values cannot be cached and re-used with other users. Furthermore, in systems where the set of users is much larger than the set of items, we would expect the average item to have very many ratings. Many ratings per items leads to relatively stable item similarity scores, meaning that these can be cached for a much larger amount of time than user-user similarity scores.

This insight has led to the common practice of precomputing the item-item similarity matrix. With a precomputed list of similar items the specific item neighborhood used for prediction can be found with a quick linear scan, using only information about the given user's past ratings. This speeds up predict-time computation drastically, making item-item more suitable for modern interactive systems than the user-user algorithm.

The cost of this speed-up is a regularly-occurring "model build" in which the similarity model is recomputed. This frequently is done nightly to ensure that new items are included in the model and are available to recommend. Since this model build is not interactive, it can run on a separate bank of machines from the live system and be scheduled to avoid peak system use.

The improvements from precomputing similarities can be made even larger by truncating the stored model. For reasonably large systems storing the whole item-item similarity matrix can take a lot of space. Many items have low to no similarity with all but a small percent of the system. Since these dissimilar items will almost never be used in an actual item neighborhood, there is no point to store them. By keeping only the most useful potential neighbors, the model size on disk and in memory can be reduced and predict time performance can be increased. Therefore it is common to keep only the n most similar items for any item in the model as "potential neighbors". Past work has shown that larger models do perform slightly better than smaller models, but that the advantages disappear after some point. In the original work on item-item, the point at which a larger model has no benefit is around 100 to 200 items [61]. Work based on larger datasets have also found larger models (500 items or more) to more effective; suggesting that, like all other parameters, the best value for the model size will vary from system to system [17,37].

With the various tweaks and optimizations the research community has found since item-item was first published, item-item can be a strong algorithm for recommendation. While it is slightly outperformed by newer algorithms, it is still very competitive when well tuned [18,19]. Furthermore, item-item is easier to implement, modify, and explain to users than most other recommendation algorithms. For these reasons, item-item is still a competitive algorithm for large scale recommender systems, and still sees modern deployment despite more recent, slightly more accurate, algorithms being offered [59].

Variants. Nearest-neighbor algorithms are the best-known approaches for collaborative filtering recommendation. Because of this, they have been modified in many interesting ways. One variant is an inversion of the user-user algorithm: the K-furthest neighbors algorithm by Said et al. [60] The K-furthest neighbors algorithm makes neighborhoods based on the least similar users, instead of the most similar users. The idea behind this is to enhance the diversity of the recommendations made. User evaluations comparing nearest neighbor recommendation to furthest neighbor recommendation shows that the two are relatively close in user satisfaction, even if the predictions made by nearest neighbor recommendation are much more accurate.

Another interesting variant to nearest neighbor recommendation is Bell and Koren's Jointly Derived Neighborhood Interpolation Weights approach [3]. The key insight of this approach is that the quality of the similarity function directly determines the quality of the user recommendation in a neighborhood model. Therefore, these similarity scores should be directly optimized, instead of relying on ad-hoc similarity metrics. One key advantage of this is that the similarity scores can be jointly optimized, which makes the algorithm more robust to interactions involving multiple neighbors. A similar approach has also been taken by Ning and Karypis' SLIM algorithm [49].

Many variants of nearest-neighbor algorithms use some external source of information to inform the similarity function used. One example of this is the trust aware recommendation framework [46], which re-weights user similarity scores based on an estimated degree of trust between two users. In this way the algorithm bases predictions on more trusted users. This same approach could be used with other forms of information such as content based similarity information.

5 Matrix Factorization Algorithms

Nearest-neighbor algorithms are good at capturing pairwise relationships between users or items. They cannot, however, take advantage of broader structure in the data, such as the idea that five different items share a common topic, or that a user's ratings can be explained by their interest in a particular feature. To explicitly represent this type of relationship requires a fundamentally different approach to recommendation. One such approach is the use of *latent feature models* such as the popular family of *matrix factorization* algorithms.

Rather than modeling individual relationships between users or items, latent feature models represent each user's preference for items in terms of an underlying set of k features. Each user can then be described in terms of their preference for each latent feature, and each item can be described in terms of its relevance to each feature. These item and user feature scores can then be combined to predict the user's preference for future items.

All matrix factorization algorithms encode each user's preference numerically in k-dimensional vectors $\mathbf{p}_u$ and each item's relevance to features in k-dimensional vectors $\mathbf{q}_i$. We will use p_{uf} to indicate the value representing a

user's preference for a given feature f and q_{if} to indicate the value representing an item's relevance to feature f. Once these vectors are computed, we can compute a user u's preference for a particular item i as the linear combination of the user feature vector and the item feature vector.

$$S(u,i) = \mathbf{p}_u \cdot \mathbf{q}_i = \sum_{f=1}^{k} p_{uf} q_{if} \tag{16}$$

Under this equation, $S(u,i)$ will be high if and only if those feature u prefers (with high scores is $\mathbf{p}_u$) are also those feature i is relevant to (with high scores in $\mathbf{q}_i$).

It is common to organize the vectors $\mathbf{p}_u$ into a $|U| \times k$ matrix named P and the vectors $\mathbf{q}_i$ into a $|I| \times f$ matrix named Q. This allows all scores for a given user to be computed in a single matrix operation $\mathbf{s}_u = \mathbf{p}_u \times Q^{\mathrm{T}}$. Likewise, all scores for every user can be computed as $S = P \times Q^{\mathrm{T}}$ These operations may be more efficient than repeatedly computing $S(u,i)$ in some linear algebra packages.

As with neighborhood based algorithms, this approach can easily be improved by directly accounting for a user's average rating and an item's average rating. This can be done as before, by normalizing ratings against a baseline predictor. However, it is much more common to introduce the bias terms directly into the model, and to learn these values simultaneously with learning matrices P and Q. This results in a biased matrix factorization model [40]:

$$S(u,i) = \mu + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i \tag{17}$$

The goal in matrix factorization algorithms is to find the vectors $\mathbf{p}_u$ and $\mathbf{q}_i$ (as well as extra terms like μ, b_u, b_i) that lead to the best scoring function for a given metric. One interesting difference between this and a nearest neighbor style algorithm is that the same core model and scoring equation algorithm can lead to many different algorithms depending on how $\mathbf{p}_u$ and $\mathbf{q}_i$ are learned. We will be presenting three approaches for learning $\mathbf{p}_u$ and $\mathbf{q}_i$, in this section we will see how to optimize $\mathbf{p}_u$ and $\mathbf{q}_i$ for prediction accuracy. In the following section we will address an algorithm that learns $\mathbf{p}_u$ and $\mathbf{q}_i$ to optimize how accurately the algorithm ranks pairs of items.

The algorithms we describe are a few of many possible matrix factorization algorithms. One of the interesting aspects of this model is that it has become a standard starting point for many novel algorithm modifications. SVD++ [40] and SVDFeature [12], for example, extend Eq. 17 by adding terms to incorporate implicit feedback and additional user or item feature information. By combining Eq. 17 with new terms to accommodate new data, and new ways of optimizing the model for different goals, many interesting algorithm variants are possible.

5.1 Training Matrix Decomposition Models With Singular Value Decomposition

One way to train a matrix factorization model for predictive accuracy, and the reason these are often called SVD algorithms, is with a truncated singular value decomposition (SVD) of the ratings matrix R.

$$R \approx P \Sigma Q^{\mathrm{T}} \tag{18}$$

Where P is an $|U| \times k$ matrix of user-feature preference scores, Q is an $|I| \times k$ matrix of item-feature relevance vectors, and Σ is a $k \times k$ diagonal matrix of global feature weights, called singular values. In a true algebraic SVD, P and Q are orthogonal, and this product is the best rank-k approximation for the original matrix R. This means that the matrices P and Q can be used to produce scores that are optimized to make accurate predictions of unknown ratings.

Singular value decompositions are not bounded to a particular k; the number of non-zero singular values will be equal to the rank of the matrix. However, we can truncate the decomposition by only retaining the k largest singular values and their corresponding columns of P and Q. This accomplishes two things: first, it greatly reduces the size of the model, and second, it reduces noise.

Ratings are known to contain both signal about user preferences and random noise [38]. If the ratings matrix is a combination of signal and noise, then consistent and useful signals will contribute primarily to the high-weight features while the random noise will primarily contribute to the lower wright features. For convenience, the columns of P and Q are often stored in a pre-weighted form so that Σ is not needed as a separate matrix. With this we see that the scoring function is simply $S(u, i) = \mathbf{p}_u \cdot \mathbf{q}_i$.

There are two important and related difficulties with the singular value decomposition for training a matrix factorization model. First, it is only defined over complete matrices, but most of R is unknown. In a rating-based system, this problem can be addressed by imputation, or assuming a default value (e.g. the item's mean rating) for unknown values [62]. If the ratings matrix is normalized by subtracting a baseline prediction before being decomposed, then the unknown values can be left as 0's and the normalized matrix can be directly decomposed with standard sparse matrix methods.

The second difficulty is that the process of computing a singular value decomposition is very computationally intensive and does not scale well to large matrices. Unlike with the first problem there is no natural solution to this. Because of this problem it is uncommon for matrix decomposition algorithms to operate based on a pure singular value decomposition.

Despite these limitations, using a singular value decomposition to compute $\mathbf{p}_u$ and $\mathbf{q}_i$ is still an easy way to build a basic collaborative filtering algorithm for experimentation. Optimized algorithms for computing matrix decompositions can be found in mathematical computing packages such as MATLAB. However, for the reasons mentioned above this is not a reasonable approach for production scale recommender systems.

5.2 Training Matrix Decomposition Models With Gradient Descent

The goal of matrix factorization is to produce an effective model of user preference. Therefore the algebraic structure (singular value decomposition) is more of a means to an end rather than the end itself. In practice it is common to sidestep the problems inherent in using a singular value decomposition and

instead directly optimize P and Q against some metric over our training data. This way we can simply ignore missing data opening up a range of speed-ups over a singular value decomposition. Simon Funk pioneered this approach to great affect by using gradient descent to train P and Q to optimize the popular mean squared error accuracy metric [22]. Similar algorithms are now a common strategy for latent factor style recommendation algorithms [41].

Since our goal is a matrix decomposition with minimal mean squared error, we can learn a decomposition by treating the problem as an optimization problem: learn matrices P $(m \times k)$ and Q $(n \times k)$ such that predicting the known ratings in R with the multiplication PQ^{T} has minimal (squared) error. As mean squared error is easily differentiable, optimization is normally done via either stochastic gradient descent or alternating least squares.

Stochastic gradient descent is a general purpose optimization approach used in machine learning to optimizing a mathematical model for a given loss function or metric, so long as the metric is easy to take the derivative of. First the computer starts with an arbitrary initial value for the model parameters, in this case the matrices P and Q as well as the bias terms. Then, it iterates through each training point, in our case a user, item and rating: (u, i, r_{ui}). Based on this point it computes an update to the model parameters that will reduce the error made on that training point. The specific update rules are derived by taking the derivative of an error function with respect to the training point, in this case the error function is the squared error. These updates are repeated many times until the algorithm converges upon a local optimum.

The update rules to train a biased matrix factorization model to minimize squared error using gradient descent are:

$$\epsilon_{ui} = \mu + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i - r_{ui} \tag{19}$$

$$\mu = \mu + \lambda(\epsilon_{ui} - \gamma\mu) \tag{20}$$

$$b_u = b_u + \lambda(\epsilon_{ui} - \gamma b_u) \tag{21}$$

$$b_i = b_i + \lambda(\epsilon_{ui} - \gamma b_i) \tag{22}$$

$$P_{uf} = P_{uf} + \lambda(\epsilon_{ui} * Q_{if} - \gamma P_{uf}) \tag{23}$$

$$Q_{if} = Q_{if} + \lambda(\epsilon_{ui} * P_{uf} - \gamma Q_{if}) \tag{24}$$

To apply these these update rules we first compute ϵ_{ui}, which represents the prediction error: $S(u, i) - r_{ui}$. This update rule should be applied to each of the k features. Then, the update for each variable should be computed before applying the updates.

The gradient descent process uses a learning rate λ that controls the rate of optimization (0.001 is a common value), and γ is a regularization term (0.02 is a common value). This regularization term penalizes excessively large user-feature and item-feature values to avoid overfitting. This update rule should be applied until some stopping condition is reached, the most common stopping condition being a specified number of iterations. To get the best performance, k, λ, and γ, and the stopping condition should be hand tuned using the evaluation methodologies discussed in Sect. 9.

Once learned, the set of variables μ, b_u, b_i, P and Q serves as the model for the algorithm. Given these values creating a prediction is as easy as applying Eq. 17. Like with item-item this model is normally computed offline. Traditionally the model is rebuilt daily or weekly (depending on how long it takes to rebuild a model and how actively new ratings, items, and users are added to the system). With this type of recommender model it is also possible to perform online updates [57] which allow a model to account for ratings, items, and users added after the model is built with a minimal loss of accuracy. In practice online and offline model updates can be combined to balance between complete optimization and interactive data use.

Matrix factorization approaches provide a memory-efficient and, at recommendation time, computationally efficient means of producing recommendations. Computing a score in a matrix factorization algorithm requires only $O(K)$ work (assuming that the factors are precomputed and stored for $O(1)$ lookup), this is true no matter how many items, users, or ratings are involved. Furthermore, by taking account of the underlying commonalities between users and items that are reflected in users' preferences and behaviors it makes very accurate predictions. Because of this matrix factorization algorithms are very popular and are one of the most common algorithms in the research literature.

6 Learning to Rank

As the original collaborative filtering algorithms focused on the prediction task, most of the research into the recommendation task has been designed around how we can use predictions to make good quality recommendations. The most common approach to doing this is also the most obvious: sort items by prediction. This approach is called the "Top-N recommendation", and is still used by systems like MovieLens and remains quite popular today. That said, other approaches have been developed for directly targeting the quality of a recommendation list.

Learning-to-rank algorithms are a recently popular class of algorithms from the broader field of machine learning. As their name suggests, these algorithms directly learn how to produce good rankings of items, instead of the indirect approach taken by the Top-N recommendation strategy. While there are several approaches, most learning-to-rank algorithms produce rankings by learning a ranking function. Like a prediction, a ranking function produces a score for each pairing of user and item. Unlike predictions, however, the ranking score has no deliberate relationship with rating values, and is only interesting for its ranked order.

Because learning-to-rank algorithms are designed around ranking and recommendation tasks, instead of prediction, they often outperform prediction algorithms at the recommendation task. However, because their output has no relation to the prediction task, they are incapable of making predictions. Many recommender systems do not display predicted rating to users; in such systems a learning-to-rank algorithm can lead to a much more useful recommender system.

The heart of most learning to rank algorithms is a specific way to define ranking or recommendation quality. Unlike prediction algorithms, where "accuracy" is easy to define, there are many ways to define ranking and recommendation quality. Furthermore, unlike prediction errors, ranking and recommendation errors are poorly suited for use in optimization. A small change to a model might lead to a small, but measurable prediction accuracy change but have no effect on the output ranking. Therefore the core work in many learning-to-rank algorithms is in designing easy-to-optimize measurements that approximate common ranking metrics. Once these new metrics are defined, standard optimization techniques can be applied to standard recommendation models such as the biased matrix factorization model from Eq. 17.

Learning-to-rank is an active area for research into recommender system algorithms with new algorithms being developed every year. To get a taste of this type of algorithm we will explain the *Bayesian Personalized Ranking* (BPR) algorithm [56]. First published in 2009, BPR is one of the earliest and most influential learning-to-rank algorithms for collaborative filtering recommendation. BPR will be discussed again in Chap. 14 in this book [35], as it was originally designed for implicit preference information. We discuss it here because it is trivial to modify for use with rating data, and the structure and development of the BPR algorithm serves as a good example of learning-to-rank algorithms in general.

6.1 BPR

BPR is a pairwise learning-to-rank algorithm, which means that it tries to predict which of two items a user will prefer. If BPR can accurately predict that a user will prefer one item over another we can use that prediction strategy to rank items and form recommendations. This approach has two advantages to prediction based training: first, BPR tends to produce better recommendations than prediction algorithms. Secondly, BPR can use a much wider range of training data. As long as we can deduce from user behavior that one item is preferred over another, we can use that as a training point.

BPR was originally designed for use with implicit unary forms of preference feedback, instead of ratings. For example, with unary data such as past purchases we can generate pairs by assuming that all purchased items are liked better than all other items. With a traditional ratings dataset we can generate training points by taking pairs of items that the user rated, but assigned different ratings to.

To predict that a user will prefer one item over another, BPR tries to learn a function $P(i >_u j)$ – the probability that user u prefers item i to item j. If $P(i >_u j) > 0.5$ then, according to the model, user u is more likely to prefer i over j than they are to prefer j over i. Therefore if $P(i >_u j) > 0.5$ we would want to rank item i above item j. There are many different functions that could be used for P, BPR uses the popular logistic function.

The logistic function allows us to shift focus from computing a probability to computing any number x_{uij} which represents a user's relative preference for i over j (or j over i if x_{uij} happens to be negative). The logistic transformation

then defines P as $P(i <_u j) = 1/(1 + e^{-x_{uij}})$. While we could build algorithms to directly optimize x_{uij} its easier to change variables one more time by defining $x_{uij} = S(u,i) - S(u,j)$ for some scoring function S. The way the logistic function works we have $P(i >_u j) > 0.5$ if and only if $S(u,i) > S(u,j)$. Furthermore, the probability P will be more confident (closer to 0 or 1) if $S(u,i)$ is substantially greater or smaller than $S(u,j)$. Therefore to optimize $P(i >_u j)$ for predictive accuracy we need to optimize S so that it ranks items correctly. For the same reason, once we train S we can use S directly for ranking.

Based on this formalization we arrive at the BPR optimization criteria, which is the function BPR seeks to optimize. The optimization criteria depends on some scoring function $S(u,i)$, and a collection of training points (u,i,j) which represent that u has expressed a preference for item i over item j. Given these, the optimization criteria is the product of the probability BPR assigns to each observed preference $P(i >_u j) = 1/(1 + e^{-(x_{ui} - x_{uj})})$. A good ranking function should maximize these probabilities, therefore we seek to maximize performance against this criteria. The full derivation of this, as well as the complete optimization criteria can be found the original BPR paper by Rendle et al. [56].

Almost any model can be used for the scoring function S. All that is required is that the derivative of S with respect to its model parameters can be found. Therefore any algorithm that can be trained for accuracy using gradient descent can also be trained using BPR's optimization criteria to effectively rank items. We will cover BPR-MF, which uses a matrix factorization model. In particular, we will give update rules for the non-biased matrix factorization seen in Eq. 16. Only minor modifications would be needed to derive update rules for a biased matrix factorization model.

As with all matrix factorization models we have two matrices P and Q representing user and item factor values which need to be optimized so that $S(u,i) = \mathbf{p}_u \cdot \mathbf{q}_i$ provides a good ranking. P and Q can be optimized for the BPR optimization criteria using stochastic gradient descent by applying the following update rules:

For a given training sample (u,i,j) representing the knowledge that user u prefers item i over item j:

$$\epsilon_{uij} = \frac{e^{-(S(u,i) - S(u,j))}}{1 + e^{-(S(u,i) - S(u,j))}} \tag{25}$$

$$P_{uf} = P_{uf} + \lambda\left(\epsilon_{uij} * (Q_{if} - Q_{jf}) + \gamma * P_{uf}\right) \tag{26}$$

$$Q_{if} = Q_{if} + \lambda\left(\epsilon_{uij} * P_{uf} + \gamma * Q_{if}\right) \tag{27}$$

$$Q_{jf} = Q_{jf} + \lambda\left(-\epsilon_{uij} * P_{uf} + \gamma * Q_{jf}\right) \tag{28}$$

Where λ is the learning rate and γ is the regularization term. Like the equations for updating a traditional matrix factorization algorithm, ϵ_{uij} in this equation represents the degree to which S does or does not correctly rank items i and j.

When training BPR algorithms, the order in which training points are taken can have a drastic impact on the rate at which the algorithm converges. Rendle et al. [56] showed that the naive approach of taking training points grouped by user can be orders of magnitude slower than an approach that takes randomly

chosen training points. Therefore, for simplicity, Rendle et al. recommend training the algorithm by selecting random training points (with replacement) and applying the update rules above. This process can be repeated until any preferred stopping condition, such as iteration count, has been reached.

Unsurprisingly, the BPR-MF algorithm is much better than classic algorithms at ordering pairs of items under the AUC metric. On other recommender metrics BPR-MF only shows modest improvements over traditional algorithms. The trade-off of this, however, is that BPR-MF, like most learning-to-rank algorithms, cannot make predictions. Theoretically, advanced techniques could be used to turn the ranking score into a prediction, however, in practice we find this does not lead to prediction improvements over prediction centered algorithms. Therefore for a website that uses both recommendations and predictions using separate algorithms for the two tasks might be essential.

7 Other Algorithms

The algorithms highlighted in this chapter provide an overview of the most influential and important recommender systems algorithms. While these few algorithms provide a basic grounding of most recommender algorithms, there are many more algorithms than covered in this chapter. Briefly, we want to mention a few other key approaches and techniques for generating personalized recommendations that have proved successful in the past.

7.1 Probabilistic Models

Probabilistic algorithms, such as those based on a Bayesian belief network, are a popular class of algorithms in the machine learning field. These algorithms have also seen increasing popularity in the recommender systems field recently. Many probabilistic algorithms are influenced by the PLSI (Probabilistic LSI) [30] and LDA (latent Dirichlet allocation) [7] algorithms. The basic structure of these models is to assume that there are k distinct clusters or profiles. Each profile has a distinct probability distribution over movies describing the movies that cluster tends to watch and, for each movie a probability distribution over ratings for that cluster. Instead of directly trying to cast a user into only one cluster, each user is a probabilistic mixture of all clusters [31]. This can be thought of as a type of latent feature model, each user has a value for each of k clusters (features) and each movie has a preference score associated with each of k clusters (features). One of the key advantages of these probabilistic models is that they are easier to update with new user or item information, due to the wealth of standard training approaches for probabilistic models [63].

7.2 Linear Regression Approaches

Many algorithms have incorporated linear regression techniques into their formulation. For example, the original work introducing the item-item algorithm

experimented with using regression techniques in addition to the similarity computations. For each pair of items a linear regression is used to find the best linear transformation between the two items. This transformation would then be applied to get an adjusted rating to be used in the weighted average. While this showed some promise for very sparse systems the idea showed little promise for more traditional recommender systems.

A more recent implementation of this idea is the Slope One recommendation algorithm [45]. In slope one we compute an average offset between all pairs of items. We then predict an item i by applying the offset to every other rating by that user, and performing a weighted average. The slope one algorithm has some popularity, especially as a reference algorithm, due to how simple it is to implement and motivate. Outside of nearest neighbor approaches, linear regression approaches are also a common way to combine multiple scoring functions together, or combine collaborative filtering output with other factors to create ensemble recommenders.

7.3 Graph-Based Approaches

Graph-based recommender algorithms leverage graph theoretic techniques and algorithm to build better recommender systems. Although uncommon, these algorithms have been a part of recommender systems research community since the early days [1]. In traditional recommender system websites like MovieLens or Netflix there is rarely a natural graph to consider, therefore these algorithms tend to impose a graph by connecting users to items they have rated. By also connecting items using content information you can use graph-based algorithms to combine content and collaborative filtering approaches [32,52].

Some services, however, have both recommendation features and social network features. In these websites it is natural to assume that a person to person connection is an indicator of trust. This leads to the set of trust based recommendation algorithms in which a person to person trust network is used as part of the recommendation process. Many of these algorithms use graph based propagation of trust as a core part of a recommendation algorithm [33,46].

8 Combining Algorithms

Most recommender system deployments do not directly tie the scores output by one of the above algorithms to their user interface. While these algorithms perform well, there usually are further improvements that can be made by combining the output of these algorithms with other algorithms or scores. Generally there are three reasons to perform these modifications: business logic, algorithm accuracy/precision, and recommendation quality. The first of these reasons – business logic – is both the most simple, and ubiquitous. Many recommender systems modify the output of their algorithm to serve business purposes such as “do not recommend items that are out of stock” or “promote items that are on sale”.

The second of these reasons – algorithm accuracy – leads researchers to develop and deploy ensemble algorithms. Ensemble algorithms are techniques for combining arbitrarily algorithms into one comprehensive final algorithm. The final algorithm normally performs better than any of its constituent algorithms independently. The design, development, and training of ensemble algorithms is a large topic in the broader machine learning field. As a comprehensive discussion of ensemble algorithms is out of scope for this chapter we will try to give a brief overview to the application of ensembles in the recommender systems field.

The final reason to combine algorithms – recommendation quality – is more complicated. Properties like novelty and diversity have a large impact on how well users like recommendations. These properties are very hard to deliberately induce in collaborative filtering algorithms as they can be in tension with recommending the best items to a user. Several algorithms have been developed to modify a recommendation algorithm's output specifically for these properties. While these algorithms are not ensembles in the traditional sense they are another way to moderate the behavior of a recommendation algorithm based on some other measure. These algorithms will be discussed after describing strategies for ensemble recommendation.

8.1 Ensemble Recommendation

The most basic approach to an ensemble algorithm is a simple weighted linear combination between two algorithms S_a and S_b

$$S(u,i) = \alpha + \beta_a * S_a(u,i) + \beta_b * S_b(u,i) \tag{30}$$

The simplest way to learn this is to have the developer directly specify the weightings. While this may sound naive there are several places where this can be appropriate, especially when the parameters are picked based on difficult to optimize metrics such as a user survey.

Linear Regression. A more attractive technique to train a linear model may be to use traditional linear regression to learn the best α and β parameters to optimize accuracy. You could imagine simple training S_a and S_b on all available training ratings and then using their predictions and the same training ratings to predict α and β. Unfortunately, this is not recommended – the core issue is that the same ratings should not be used when training the sub-algorithms and when training the ensemble as it leads to overfitting.

Ideally you want to train the ensemble on ratings that the sub-algorithms have not seen so that the ensemble is trained based on the out-of-sample error of each algorithm. The easiest way to do this would be to randomly hold out some small percent of training data (say 10%) and to use that withheld data to train the regression to minimize squared error. The problem with this approach to training a linear ensemble, however, is that it withholds a large amount of data, which might effect the overall algorithm performance.

Stacked Regression. An alternate approach, without this problem is Breiman's stacked regression algorithm [10]. Breiman's algorithm trains regression parameters by first producing a k subsets of the dataset by traditional (ratings based) crossfolding. Then each sub-algorithm is trained independently on each of the k folds of the algorithm. Due to the way crossfold validations are made this means that each of the original training ratings has an associated algorithm which has never seen that rating. When generating sub-algorithm predictions for any given training point (as needed to learn the linear regression) the algorithm which has never seen that training point is used. Once the linear regression has been trained the sub-algorithms should then be re-trained using the overall set of data. For a more comprehensive description of this procedure consult Breiman's original paper [10].

All ensemble algorithms work best when very different algorithms are being combined. So an ensemble between item-item (with cosine similarities) and item-item (with Pearson similarities) is unlikely to show the same improvement as an ensemble between item-item and a content-based algorithm [17].

In MovieLens, for example, we could imagine making a simple content-based algorithm by computing each user's average rating for each actor and director. While not necessarily the best algorithm, this content based algorithm would likely outperform a collaborative filtering algorithm for movies with very few ratings. Therefore, an ensemble of a content based algorithm and a collaborative filtering algorithm might show improved accuracy over a content based algorithm on its own.

This example does lead to one interesting observation: there are many cases where we would want to create an ensemble and we know the conditions when one algorithm might perform better than another. The actor-based recommender would be our best recommender when we have next to no ratings for a movie, as we get more ratings, however, we should trust the collaborative filtering algorithm more. A linear weighting scheme does not allow for this type of adjustment.

Feature Weighted Linear Stacking. The feature weighted linear stacking algorithm, introduced by Sill el al., is an extension to Breiman's linear stacked regression algorithm [64]. Feature weighted linear stacking allows the relative weights between algorithms to vary based on features like number of ratings on an item. This algorithm is most noteable for being very popular during the Netflix prize competition, a major collaborative filtering accuracy competition, where it was the key facet of the second best algorithm. In feature weighted linear stacking algorithms are linearly related as per Eq. 30, the difference is that the weights β are themselves a linear combination of the extra features. Sill et al. show that this model can be learned by solving a system of linear equations using any standard toolkit for solving systems of linear equations. Details of this solution, especially including information to assist in scaling are provided in the paper by Sill et al. [64].

While ensemble methods have provided much better predictive accuracy than single algorithm solutions, and could theoretically be applied to learning-to-rank problems as well, it should be noted that they can also become much more

complex than a traditional recommender. While it is easy to overlook technical complexity when designing an algorithm, technical complexity can be a significant barrier to actually deploying large ensembles in the field. Noteably, after receiving code for a 107-algorithm ensemble Netflix went on to only actually implement two of these sub-algorithms [2]. Ensemble methods can be much more complex and time invasive to keep up to date and can require much more processing when making predictions which can lead to slower responses to users. Therefore, when considering ensembles, especially very large ones, designers should consider if the improved algorithm accuracy is worth the increased system complexity.

8.2 Recommending for Novelty and Diversity

This brings us to the third reason that a recommender system might modify the scores output from a recommendation algorithm: to increase the quality of recommendations as reported by users. Research into recommendation systems has shown that selecting only good items is not enough to ensure that a user will find a recommendation useful [69,70]. Many other properties can effect how useful a recommendation is to a user. Two that have been shown to be important, and have been the focus of some research are novelty and diversity.

Novelty and diversity are properties of a recommendation that measure how the items relate to each other or the user. Novelty refers to how unexpected or unfamiliar the user is with their recommendations [53]. If a recommendation only contains obvious recommendations they are neither novel, nor useful at helping a user find new items. Diverse recommendations cover a large range of different items. One flaw with many recommender systems is that their recommendations are all very similar to each other, which limits how useful the recommendations are. For example, a top-8 recommendation consisting of only Harry Potter movies would neither be novel (as those movies are quite well known), nor would it be diverse (since the recommendation only represents a small niche of the user's presumably broader interests).

There have been several different approaches to modifying an algorithm's score to favor (or avoid) properties like novelty or diversity. We will focus on two broad strategies, the first is well suited to combining algorithms with item specific metrics such as how novel an item is, the second is well suited to metrics measured over the entire recommendation list. While these strategies have been pioneered for use with novelty and diversity, they can be applied with any metric. For example, when recommending library items, users may be disappointed by recommendations on item that have a waitlist and cannot be borrowed immediately. These strategies could be used to modify recommendations to favor items without a waitlist, increasing user satisfaction. Specific metrics for novelty and diversity will be discussed alongside algorithm evaluation in Sect. 9.4.

The most common way to combine algorithms with some metric to increase user satisfaction is to use a simple linear combination between the original algorithm score S_a and a some item level measurement of interest [6,67,69]. For example, the number of users who have rated an items $|U_i|$ (or its inverse $1/|U_i|$)

is normally measured to allow manipulating novelty. By blending the score from an algorithms with $1/|U_i|$ we can promote items that have fewer ratings and enhance the novelty of recommended items. These scores are normally used only when ranking, typically an unmodified rating prediction is still used even when the prediction is blended for recommendation.

The other major approach for modifying recommendations is an iterative re-ranking approach. In these approaches items are added to a recommendation set one at a time, with the ranking score recomputed after each step. This approach has two advantages. First, re-ranking allows hard constraints when selecting items. For example, the iterative function could reject any more than two movies by a given director. Secondly, the iterative approach allows measurements such as the average similarity of an item with the other items already chosen for recommending. This is often necessary when manipulating diversity, as diversity is a property of the recommendation, not one specific item. This was in fact the approach taken by the first paper to address diversity in recommendation [65]. The cost of this approach over re-scoring is slightly higher recommend time costs.

A primary example of the iterative re-ranking algorithm is the diversity adding algorithm introduced by Ziegler et al. [70]. To generate a top 10 recommendation list, this algorithm first picks the top 50 items for a user as candidate items. The size of the candidate set represents a trade off between run time cost and flexibility of the algorithm to find more diverse items. From the top 50 items, the best predicted item is immediately added to the recommendation. After that, for each remaining candidate the algorithm computes an sum of the similarity between that candidate and each item in the recommendation. The algorithm then sorts by inverse similarity sum to get a dissimilarity rank for each potential item. The overall item weight is then a weighted average of the prediction rank and the dissimilarity rank (with weights chosen beforehand). The item with the smallest score by this weight is added to the recommendation. This process repeats, with updated similarity scores, until the desired ten item recommendation list has been made.

Re-scoring and re-ranking algorithms are an easy way to promote certain properties in recommendations. Both algorithms can be relatively inexpensive to run, and can be added on top of an existing recommender. Additionally these approaches are very easy to re-use for a large variety of different recommendation metrics beyond just novelty and diversity.

9 Metrics and Evaluation

The last topic we will discuss in this chapter is how to evaluate a recommendation algorithm. This is an important concept in the recommender systems field as it gives us a way to compare and contrast multiple algorithms for a given problem. Given that there is no one "best" recommendation algorithm, it is important to have a way to compare algorithms and see which one will work best for a given purpose. This is true both when arguing that a new algorithm is better than previous algorithms, and when selecting algorithms for a recommender system.

One key application of evaluations in recommender systems is *parameter tuning*. Most recommendation algorithms have variables, called *parameters* which are not optimized as part of the algorithm and must be set by the system designer. Parameter tuning is the process of tweaking these parameters to get the best performing version of an algorithm. For example, when deploying a matrix factorization algorithm the system designer must choose the best number of features for their system. While we have tried to list reasonable starting points for each parameter of the algorithms we have discussed in this chapter, these are at best a guideline, and readers should carefully tune each parameter before relying on a recommendation algorithm.

In this section we will mostly focus on *Offline evaluation* methodologies. Offline evaluations can be done based only on a dataset and without direct intervention from users. This is opposed to *online evaluation* which is a term for evaluations done against actual users of the system, usually over the internet.

Offline evaluations are a common evaluation strategy from machine learning and information retrieval. In an offline evaluation we take the entire ratings dataset and split it into two pieces: a *training dataset* ($Train$), and a *test dataset* ($Test$); there are several ways of doing this, which we describe in more detail shortly. Algorithms are trained using only the ratings in the training set and asked to predict ratings, rank items, or make a recommendation for each user. These outputs are then evaluated based on the ratings in the test dataset using a variety of metrics to assess the algorithm's performance.

Just as there are several different goals for recommendation (prediction, ranking items, recommending items) there are many different ways to evaluate recommendations. Furthermore, while evaluating prediction quality may be straightforward (how well does the prediction match the rating), there are many different ways to evaluate if a ranking or recommendation is correct. Therefore, there are a great number of different *evaluation metrics* which score different aspects of recommendation quality. These can be broadly grouped into *prediction metrics*, which evaluate how well the algorithm serves as a predictor, *ranking quality metrics*, which focus on the ranking of items produced by the recommender, *decision support metrics*, which evaluate how well the algorithm separates good items from bad, and metrics of *novelty and diversity* that evaluate how novel and diverse the recommendations might appear to users. Depending on how an algorithm might be used metrics from one or all of these sections might be used in an evaluation, and performance on several metrics might need to be balanced when deciding on a best algorithm.

9.1 Prediction Metrics

The most basic measurement we can take of an algorithm is the fraction of items it can score, known as the *prediction coverage metric* or simply the *coverage metric* for short. The coverage metric is simply the percent of user item pairs in the whole system that can be predicted. In some evaluations coverage is only computed over the test set. Beyond convenience of computation, this modification focuses more explicitly on how often the algorithms cannot produce a score for

items that the user might be interested in (as evidenced by the user rating that item) [27]. This metric is of predominantly historic interest, as most modern algorithm deployments use a series of increasingly general baseline algorithms as a fallback strategy to ensure 100% coverage. That said, coverage can still be useful when comparing older algorithms, or looking at just one algorithm component in isolation.

Assuming that an algorithm is producing predictions, the next most obvious measurement question is how well its predictions match actual user ratings. To answer this we have two metrics *Mean Absolute Error* (MAE) and *Root Mean Squared Error* (RMSE). In the following two equations $Test$ is a set containing the test ratings r_{ui} and the associated users and items.

$$MAE(Test) = \frac{\sum_{(u,i,r_{ui}) \in Test} |S(u,i) - r_{ui}|}{|Test|} \tag{31}$$

$$RMSE(Test) = \sqrt{\frac{\sum_{(u,i,r_{ui}) \in Test} (S(u,i) - r_{ui})^2}{|Test|}} \tag{32}$$

Both MAE and RMSE measure the amount of error made when predicting for a user, and are on the same scale as the ratings. The biggest difference between these two metrics is that RMSE assigns a larger penalty to large prediction errors when compared with MAE. Since large prediction errors are likely to be the most problematic, RMSE is generally preferred.

Both the RMSE and MAE metrics measure accuracy on the same scale as predictions. This can be normalized to a uniform scale by dividing the metric value by the size of the rating scale ($maxRating - minRating$), yielding the normalized mean average error (nMAE) and normalized root mean squared error (nRMSE) metrics. This is rarely done, however, as comparisons across different recommender systems are often hard to correctly interpret do to differences in how users use those systems.

Prediction metrics can be computed for an entire system or individually for each user. RMSE, for example, can be computed for the system by averaging over all test ratings, or computed per user by averaging over each user's test ratings. Due to the differences between people, most algorithms work better for some users than they do for others. Measuring per-user error scores lets us understand and measure how accurate the system is for each user. By averaging the per user metric values we can then get a second measure of the overall system.

While the difference between per-user error and system-wide (or by rating) error may seem trivial it can be very important. Most deployed systems have power users who have rated many more items than the average user. Because they have more ratings these users tend to be overrepresented in the test set, leading to these users being given more weight when estimating system accuracy. Averaging the per user accuracy scores avoids this issue and allows the system to be evaluated based on its performance for all users.

9.2 Ranking Quality

A more advanced way of evaluating an algorithm is to ask if the way it orders or ranks items is consistent with user preferences. There are several ways of approaching this, the most basic being to simply compute the Spearman ρ or Pearson r correlation coefficients between the predicted ratings and test set ratings for a user. As noted by Herlocker et al. [27] Spearman's ρ is imperfect because it works poorly when the user rates many items at the same level. Additionally, both metrics assign equal importance to accuracy at the beginning of the list (the best items) and the end (the worst items). Realistically, however, we want a ranking metric that is most sensitive at the beginning of the ranking, where users are likely to look, and less sensitive to errors towards the end of the list, where users are unlikely to look.

A more elegant metric for evaluating ranking quality is called the *discounted cumulative gain metric* DCG. DCG tries to estimate the value a user will receive from a list. It does this by assuming each item gives a value represented by its rating in the test set, or no value if unrated. To make DCG focus on the beginning of the list, these values are discounted logarithmically by their rank, so the maximum gain of items later in the list is smaller than the potential gain early in the list. The DCG metric is defined as follows:

$$dcg(u, Rec) = \sum_{i \in Rec} \frac{r_{ui}}{max(1, log_b(k_i))} \tag{33}$$

Where k_i refers to the rank order of i, Rec is an ordered list of items representing an algorithms recommendation for the user u, and b is the base of the logarithm. While different values are possible, DCG is traditionally computed with $b = 2$. Other values of b have not been shown to yield meaningfully different results [36].

The DCG metric is almost always reported normalized as the *normalized discounted cumulative gain metric* nDCG. nDCG is simply the DCG value normalized to the 0–1 range by dividing by the "optimal" discounted cumulative gain value which would be given by any optimal ranking

$$ndcg(u) = \frac{dcg(u, prediction)}{dcg(u, ratings)} \tag{34}$$

A similar metric to this is the *half life utility metric* [9]. Half life utility uses a faster exponential discounting function. The half life utility is as follows:

$$HalfLife(u) = \sum_{i \in Rec} \frac{max(r_{ui} - d, 0)}{2^{(k_i - 1)/(\alpha - 1)}} \tag{35}$$

Half life utility has two parameters. The first is d which is a score that should represent the neutral rating value. A recommendation for an item with score d should neither help nor hurt the user, while any item rated above d should be good recommendations. d should also be used as the "default" rating value for

r_{ui} where a user does not have a rating for that item, In this way unrated items are assigned a value of 0. The α variable controls the speed of exponential decay and should be set so that an item at rank α has a roughly 50% chance of actually being seen by the user.

One common modification on these metrics is to absolutely limit the recommendation list size. For example, the nDCG@n metric is taken by computing the nDCG over the top-n recommendations only. Any item in the test set but not recommended is ignored in the computation. This is reasonable if you know there is a hard limit to how many items users can view, or if you only care about the beginning n elements of the list.

9.3 Decision Support Metrics

Another common approach to evaluating recommender systems is to use metrics from the information retrieval domain such as precision and recall [14]. These metrics treat the recommender as a classifier with the goal of separating good items from other items. For example, a good algorithm should only recommend good items (precision) and should be able to find all good items (recall). Decision support metrics give us a way of understanding how well the recommendation could support a user in deciding which items to consume.

The basic workflow of all decision support metrics is to first perform a recommendation, then compare that recommendation against a previously selected "relevant item set". The relevant item set represents those items that we know to be good items to recommend to a user. You then count how many of the recommended items were relevant and how many were not.

For items in a larger scale system, a choice needs to be made about which items to consider relevant. The easiest choice would be to take all rated items in the test set as good items, which evaluates an algorithm on its ability to select items that the user will see. More useful, however, is the practice of choosing a cutoff such as four out of five stars at which we consider a recommendation 'good enough'. Best practices recommend testing with multiple similar cutoffs to ensure that results are robust across various choices for defining relevant items. Evaluation results that favor one algorithm with items rated 4 and above as "good", but another algorithm if 4.5 and above are "good", deserve more careful consideration.

Once this decision has been made there remains an issue of how to treat the remaining "non relevant" items. In traditional information retrieval work it is often reasonable to assume that every item that is not known to be relevant can be considered not relevant, and therefore bad to recommend. This assertion is much less reasonable in the recommender system domain, while some of these items are known to be rated poorly, many more have simply never been rated. There are likely many good items for each user that has not been rated and would therefore be considered not relevant. It has been argued that not having complete knowledge of which items a user would like may make these metrics inaccurate or suffer from a bias [4,13,28]. Ultimately, this problem has not been

solved, and most evaluations settle for the assumption most non-rated items are not good to recommend, and the evaluation bias caused by this will be minor.

Related to the above issue of how to treat not relevant items is the question of how to compute recommendations. There have been various different approaches taken, and these have been shown to lead to different outcomes in the evaluation [5]. Commonly recommendations are done by taking the top-n predicted items, in which case these metrics would be labeled with that n such as *precision*@20 for precision computed over the top-20 list. n should be picked to match interface practices, so if only eight items are shown to a user, algorithms should be evaluated by their *precision*@8.

The other important consideration is which candidate set of items the recommendations should be over. Many different candidate set options have been used, but the most common options are either all items, or the relevant item set plus a random subsample of not relevant items. Some work has used the set of items that the user rated in the test set as a recommendation candidate set; while this does avoid the issue of how to treat unrated items, this evaluation methodology also provides different results which are believed to be less indicative to user satisfaction with a recommender [5].

Once the set of good items has been picked and recommendations have been generated, the next step is to compute a *confusion matrix* for each user. A confusion matrix is a two by two matrix counting how many of the relevant items were recommended, how many of the relevant items were not recommended, and so forth (Table 2).

Table 2. A confusion matrix

	Good items	Not good items
Recommended items	True positives (tp)	False positives (fp)
Other items	False negatives (fn)	True negatives (tn)

There are several metrics to compute based on this given confusion matrix.

- *precision* - $\frac{tp}{tp+fp}$ - The percent of recommended items that are good
- *recall* (also known as *sensitivity*) - $\frac{tp}{tp+fn}$ - The percent of good items that are recommended
- *false positive rate* - $\frac{fp}{fp+tn}$ - the percent of not good items that are recommended
- *specificity* - $\frac{tn}{fp+tn}$ - The percent of not good items that are not recommended

These metrics, especially precision and recall, are traditionally reported and analyzed together. This is because precision and recall tend to have an inverse relationship. An algorithm can optimize for precision by making very few recommendations, but doing that would lead to a low recall. Likewise, an algorithm might get high recall by making very many recommendations, but this would

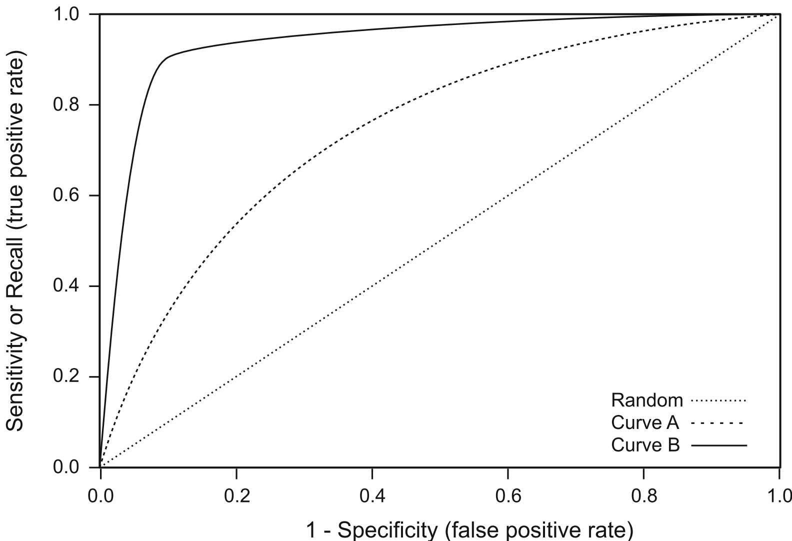


Fig. 6. An example ROC curve. Image used with permission from [21].

lead to low precision. An ideal algorithm will therefore want to balance these two properties finding a balance that recommends predominantly good items, and recommends almost all of the good items. To make finding this balance easier researchers often look at the *F-score*, which is the harmonic mean between precision and recall

$$F = \frac{2 * precision * recall}{precision + recall} \tag{36}$$

Another way to summarize this information is with the use of an ROC curve. An *ROC curve* is a plot of the recall on the y-axis against the false positive rate on the x-axis. An example ROC curve is given in Fig. 6. The ROC curve will have one point for every recommendation list length from recommending only one item to recommending all items. When the recommendation list is small, we expect a small false positive rate but also a small recall (hitting the point (0, 0)). Alternatively, when all items are recommended the false positive rate and recall will both be 1. Therefore the ROC curve normally connects the point $(0, 0)$ to $(1, 1)$. The ROC curve can be used to evaluate an algorithm broadly, with a good algorithm approaching the point (0, 1) which means it has almost no false positives, while still recalling almost every good item. To make this property easier to compare numerically it is also common to compute the area under the ROC curve, referred to as the AUC metric. It has been pointed out [56] that the AUC metric also measures the fraction of pairs of items that are ranked correctly.

As mentioned earlier, it is often incorrect to assume that items that are not rated highly by a user are by definition bad items to recommend. That does not mean, however, that there are no bad recommendations, just like we can say

highly rated (4 or more stars out of 5) items are clearly good, we can say poorly rated items (one or two stars out of five) are clearly bad. Using this insight, we can define the *fallout* metric. In a typical information retrieval evaluation fallout is the same as the false positive rate. In a recommender system evaluation, however, we can explicitly focus on how often bad items are recommended, and compute the percent of recommended items that are known to be bad. If one algorithm has a significantly higher fallout than another we can assume that it is making significant mistakes at a higher rate, and should be avoided.

One issue with the precision metric is that, while it rewards an algorithm for recommending good items, it does not care where those items are in a recommendation. Generally, we want good items recommended as early in the list as possible. To evaluate this we can use the *mean average precision metric* (MAP), which is the mean of the *average precision* over every user. The average precision metrics takes the average of the precision at each of the relevant items in the recommendation. If an item is not recommended then it contributes a precision of 0.

$$MAP@N = \frac{\sum_{u \in U} averagePrecision@N(u)}{|U|} \tag{37}$$

$$averagePrecision@N(u) = \frac{1}{|goodItems|} \sum_{i \in goodItems} precision@rank(i) \tag{38}$$

By taking the mean average precision we place more importance on the early items in the list than the later items, as the first item is used in all N precision computations, while the last one is only used in one.

Another approach for checking that good items are early in the recommendation is the *mean reciprocal rank metric* (MRR). Instead of looking at how many good items or bad items an algorithm returns, mean reciprocal rank looks at how many items the user has to consider before finding a good item. For any user, their rank ($rank_u$) is the position in the recommendation of the first relevant item. Based on this we can take reciprocal rank as $1/rank_u$ and mean reciprocal rank is the average reciprocal rank over all users. A larger mean reciprocal rank means that the average user should have to look at fewer items before finding an item they will enjoy.

$$MRR = \frac{\sum_{u \in U} 1/rank_u}{|U|} \tag{39}$$

9.4 Novelty and Diversity

There are several other properties of a recommendation that can be measured. The most commonly discussed are novelty and diversity. These properties are believed to be very important in determining whether a user will find a set of recommendations useful, even if they are unrelated to the pure quality of the recommendation. Understanding the effect these properties have on user satisfaction is still one of the ongoing directions in recommender systems research [18].

It is important to compare these metrics along with other metrics, such as accuracy or decision support metrics, as large values for these metrics are often seen along with large losses in quality. At the extreme a random recommender would have very high novelty and diversity, but would score bad on all other metrics. Generally speaking, good algorithms are those that increase novelty or diversity without meaningfully decreasing other measures of quality.

Novelty refers to how unexpected or unfamiliar the user is with their recommendations [53]. Recommendations that mostly contain familiar items are not considered novel recommendations. Since the goal of a recommender is to help its users find items they would not otherwise see, we expect that a good recommender should have higher novelty. The most obvious, and common, way to estimate novelty offline is to rely on some estimate of how well known an item is. The count of users who have rated an item, referred to as the item's popularity, is commonly used for this. More popular items are assumed to be better-known and less novel to recommend [11].

Diverse recommendations cover a large range of different items. One flaw with many recommender systems is that they focus too heavily on some small set of items for a given user [34]. So knowing that a user liked a movie from the Star Wars franchise, for example, might lead the algorithm to only recommend science fiction to a user, even if that user likes adventure films in general. The most common way of understanding how diverse a set of recommendations is, is to measure the total or average similarity between all pairs of items in the recommendation list [68,70]. This inter-list similarity (ILS) measure can be seen as an inverse of diversity, the more similar recommended items are, the less diverse the recommendation is. Ideally a similarity function that is based on the item itself, or item meta-data is used, as it allows diversity to be measured independent of properties of the ratings and predictions. Where sim is the similarity function, the diversity of a recommendation list Rec can be defined as:

$$ILS(Rec) = \frac{2}{|Rec| * (|Rec| - 1)} \sum_{i \in Rec} \sum_{j \neq i \in Rec} sim(i, j) \tag{40}$$

9.5 Structuring an Offline Evaluation

The heart of a good offline evaluation is how the train and test datasets are generated. Without a good process for splitting train and test datasets, the evaluation can fail to produce results, produce misleading results, or produce statistically insignificant results. To avoid this, simple standard approaches for generating train and test datasets have been developed. The standard approach to producing train and test datasets is user-based K-fold crossfolding.

In user-based K-fold crossfolding the users are split into K groups, where typical values of K are 5 or 10. For each of the K groups we generate a new train and test dataset split. For dataset split n, all users not in fold n are considered train users and all their ratings are allocated to the training dataset. The users in group n are then test users, and their ratings are split so that some can be in the train dataset (to inform the algorithm about that user's tastes) and the rest

go to the test dataset. Typically either a constant number of ratings, such as ten per user, or a constant fraction, such as ten percent of ratings per user, are allocated for testing. Testing items can either be chosen randomly, or the most recent items can be chosen to emphasize the importance of the order in which a user makes ratings.

User-based K-fold crossfolding has several benefits. First, by performing a user based evaluation we know that each user will be evaluated once and only once. This ensures that our conclusions give equal weight to each user, with no user evaluated more or less than the others. Secondly, by ensuring that there are a large number of training users we know that we are evaluating the algorithm under a reasonably realistic condition, with a reasonable amount of training data. Finally, through replication we can measure statistical confidence around our metric values, as each train and test can be considered relatively independent. With user-based crossfolding it is common to treat each user as its own independent sample of the per-user error when computing statistical significance.

There are several other approaches for structuring an offline evaluation that have been used in the past. These approaches are typically designed to focus the evaluation on a single factor, or to support new and interesting metrics. For example, Kluver and Konstan used a modified crossfold strategy to look at how the algorithm changes as it learns more about users [37]. The key insight in this strategy is to initially perform a crossfold where the maximum number of training points is retained. New test sets with fewer training points per user can then be made by subsampling the training set, but leaving the test alone. By keeping the test set constant across different sample sizes, biases related to the size of the test set are avoided and a fair comparison can be drawn between algorithm properties at different number of training ratings for a user.

Another interesting approach is the temporal evaluation [42,43]. Temporal evaluations have been used to look at properties of the recommender system over time as the collaborative filtering changes. In a temporal evaluation the dataset is split into N equal sized temporal windows. This allows $N-1$ evaluations to be done for each window after the first where the train set is all windows before the given window, and the test set is the target window. Applying normal metrics this way can give you an understanding of how an algorithm might behave over time in a real deployment [42]. Temporal analysis also allows for interesting new metrics such as temporal diversity [43] that measure properties of how frequently recommendations change over time.

9.6 Online Evaluations

Not every comparison can be done without users. While offline evaluations are good for rapid development, online testing is needed to truly understand how a given system's users will respond to a given algorithm. Because the users are a central part of the recommender system, an algorithm that works well in an offline evaluation may have unexpected properties when it interacts with real users. Since it is these properties that determine which algorithm makes users most happy, we recommend that offline evaluation be used to choose a small set of algorithms and tunings that are then compared using a final online evaluation.

There are several ways to perform an online evaluation, each with its own benefits and drawbacks:

Lab Study - In a lab study users are brought into a computer lab and asked to go through a series of steps. These steps may involve using a real version of the website, an interactive survey, or an in depth interview. An example lab study might bring users into the lab so eye tracking can be performed to evaluate how well a recommender chooses eye catching content for the home page. Lab studies give the experimenter a large amount of control over what the user does. The drawback of this flexibility, however, is that lab studies often do not create a realistic environment for how a recommender system might be used. Additionally, lab studies are typically limited in how many users they can involve as they often require space and supervision from an experimenter.

Virtual Lab Study - Virtual lab studies are similar to lab studies but are performed entirely online and without the direct supervision of the experimenter. This deployment trades some of the control of a lab study for much larger scale, as virtual lab studies can involve many more users in the same amount of time as a lab study. These normally take the form of purpose built web services that interact with the recommender and guide the user through a series of actions and questions. While almost any form of data about user behavior and preference can be used with a virtual lab study, surveys are particularly popular. An example virtual lab study might guide users through rating on several different interfaces and then survey users to find out which they prefered. A well designed survey can be easy for a user to complete remotely, and very informative about how users evaluate the recommender system.

Online Field Study - Online field studies focus on studying how people use a deployed recommender system. Generally there are two approaches to online field studies. In the first, existing log data from a deployed system is used to try to understand how users have been behaving on the system. For example, rating data from a system could be analyzed to understand how often users enter a rating under a current system.

Alternatively, a change can be made to a deployed system to answer a specific question. New forms of logging could be introduced, or new experimental features deployed for a trail period. This can allow answering more specific questions about user behavior. For example, a book recommender system might set the goal that 80% or more of users find a book within 5 min of accessing the service. Logging could be added to the system to allow measuring how often users find books, and in how long so that performance can be directly compared with this goal.

Online field studies are ideal for understanding how a system is used or for measuring progress against some goal for the recommender system as a whole. Online field studies are not as suited for comparing different options. Likewise, online field studies are limited by their connection to a deployed live system, which might preclude studying possibly disruptive changes. Finally, online field studies often do not allow asking follow-up questions without resorting to a

secondary evaluation, for example, while a researcher might know what users do in a situation, they will not be able to ask why.

A-B Test - A-B tests can be seen as an extension of an online field study. In an A-B test two or more versions of an algorithm or interface are deployed to a given website, with any user seeing only one of these versions. By tracking the behavior of these users, a researcher can identify differences in how users interact with the algorithm in a realistic setting. An example A-B test might deploy two algorithms to a recommender service for two months and then look at user retention rates. If one algorithm leads to fewer users having another visit, then we can say that algorithm is likely worse. Being an experimental extension to online field studies, A-B tests share the same weaknesses: it can be hard to understand what is causing the results that it finds.

Within these options virtual lab studies are most common when the goal is to understand why users perceive algorithms differently, and A-B tests are preferred when the goal is simply to pick the "best" algorithm by one or more user behavior metrics.

Online evaluations are a complicated subject, and the description here only scratches the surface. Several texts are available that go into depth on the various ways to design a user experiment to answer any number of questions, including those of algorithm performance. In particular, we recommend the book "Ways of knowing in HCI" [50], which contains in depth coverage of a broad range of research methods which are applicable in this scenario. With that said, there are some considerations that are specific to recommender systems which we will discuss.

Since all recommender systems require a history of user data to work with, ensuring that this information is available is important when considering the design of a study. This decision is closely tied to how participants for the study will be recruited. If recruiting from an existing recommender system a lab study or virtual lab study can simply request user account information and use that to access ratings. Better yet, in a virtual lab study, links to the study could be pre-generated with a user-specific code so that users do not need to log in.

If recruitment for a lab study does not come from an existing recommender system, than typically the first stage of the study is to collect enough ratings to make useful recommendations. As this is the same problem faced when onboarding users to a recommender system this is a well studied problem. The common solution is to present users with a list of items to rate [23, 54, 55]. There are many different ways to pick which order to show items in to optimize how useful the ratings are for recommending, the time that it takes to get a certain number of ratings, or both. For a good review of this field of work see the 2011 study by Golbandi et al. [23], which also includes an example of an adaptive solution to collecting useful user ratings. If time is not a major factor, however, a design where users search to pick items to represent their tastes may have some benefits [48].

Just as with offline evaluations, comparing reasonable algorithms is essential to producing useful results. Starting with offline evaluation methodologies can be a good way to make sure only the best algorithms are compared. When comparing novel algorithms, it can often help to include a baseline algorithm as a comparison point whose behavior is relatively well known.

When using a survey, often as part of a virtual lab study, it is important to choose well written questions. Some questions can be ambiguous making it hard to assign meaning to their answers. A survey question is useless to an experimenter if many users do not understand it. To support experimenters, several researchers have put together frameworks to support online, user centered evaluations of recommender systems [39,53]. These frameworks split the broader subject of how well the algorithm works into smaller factors tuned for specific properties of a recommender system. Each of these factors are also associated with well designed survey questions which are known to accurately capture a user's opinion. When possible we recommend using one of these frameworks as a resource when designing an online evaluation.

9.7 Resources for Algorithm Evaluation

There are plenty of resources available to help explore ratings-based collaborative filtering recommender systems. In particular, there are open source recommender algorithm implementations, and rating datasets available for non-commercial use. These resources have been developed in an effort to help reduce the cost of research and development in collaborative filtering.

In recent years there has been a push in the recommender systems community to support reproducible recommender systems research. One major result of this call has been several open source collections of recommender systems algorithms such as LensKit[6], Mahout[7], and MyMediaLight[8]. By publishing working code for an algorithm the researchers can ensure that every detail of an algorithm is public, and that two people comparing the same algorithm don't get different results due to implementation details. More importantly, however, these toolkits allow programmers who are not recommender systems experts to use and learn about recommender systems algorithms and benefit from the work of the research community.

The other noteworthy resource for exploring and evaluating recommender systems is the availability of public ratings datasets. These datasets are released by live recommender systems to allow people without direct access to a recommender service and its user base to participate in recommender system development. Some notable examples of datasets are the MovieLens movie rating datasets[9], described in detail in Harper and Konstan's 2015 paper [26], the Jester joke rating dataset[10] described in the 2001 paper by Goldberg et al. [24], the Book-Crossing book rating dataset[11] introduced in 2005 paper by Ziegler et al. [70] and the Amazon product rating and review dataset[12] first introduced in

[6] https://lenskit.org/.
[7] https://mahout.apache.org/.
[8] http://mymedialite.net/.
[9] available at http://grouplens.org/datasets/movielens/.
[10] available (with updates) at http://eigentaste.berkeley.edu/dataset/.
[11] available at http://www2.informatik.uni-freiburg.de/~cziegler/BX/.
[12] available at http://jmcauley.ucsd.edu/data/amazon/.

McAuley and Leskovec's 2013 paper [47]. These datasets, and many more available online, are available for anyone to download and use to learn about, and experiment with, recommender system algorithms. Most of the recommender systems toolkits have code for loading these datasets and performing offline evaluations already built.

While most directly applicable towards offline evaluations, these resources can also help with online evaluations. Open source libraries can be used to quickly prototype recommender systems either for incorporation in an existing live system or for a lab or virtual lab study. Likewise, research datasets can be used to power a collaborative filtering algorithm for use in a lab or virtual lab study design in which new ratings will be collected from research participants. This can allow anyone with access to research participants to perform online evaluations to deeply understand how users react to collaborative filtering technologies.

10 Conclusions

In this chapter we have presented the central concepts, algorithms, and means of evaluation in ratings based collaborative filtering. While recommendation systems construed broadly is still an active area of research, research on pure ratings-based collaborative filtering algorithms is becoming more rare. Indeed, it is increasingly rare to see a new pure ratings-based algorithm make a significant improvement in offline evaluations. Instead, research into collaborative filtering recommender systems has started focusing on new problems and new sources of information. Many of these more recent directions for collaborative filtering research have become the basis for the future chapters in this book.

The next five chapters of this book will explore more advanced techniques for recommendation based on different forms of information and recommendation tasks. Taken as a whole, these chapters serve as an excellent introduction to the state of the art in collaborative filtering recommender systems. Chapter 11 will explore recommendation based on online social networking [44], Chap. 12 will explore recommendation based user volunteered tags [8], Chap. 13 will explore recommendation based on publicly shared user opinions on sites like Amazon or Twitter [51], Chap. 14 will explore recommendation based on implicit feedback [35], and finally, Chap. 15 will explore how to use a recommender algorithms to recommend person-to-person connections in online social websites [25].

References

1. Aggarwal, C.C., Wolf, J.L., Wu, K.L., Yu, P.S.: Horting hatches an egg: a new graph-theoretic approach to collaborative filtering. In: Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 1999, pp. 201–212. ACM. (1999). https://doi.org/10.1145/312129.312230
2. Amatriain, X., Basilico, J.: Netflix recommendations: beyond the 5 stars (part 1). http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html

3. Bell, R.M., Koren, Y.: Scalable collaborative filtering with jointly derived neighborhood interpolation weights. In: Proceedings of the 2007 Seventh IEEE International Conference on Data Mining, ICDM 2007, pp. 43–52. IEEE Computer Society (2007). https://doi.org/10.1109/ICDM.2007.90
4. Bellogin, A.: Performance prediction and evaluation in recommender systems: an information retrieval perspective. Ph.D. thesis. Universidad Autnoma de Madrid (2012)
5. Bellogin, A., Castells, P., Cantador, I.: Precision-oriented evaluation of recommender systems: an algorithmic comparison. In: Proceedings of the Fifth ACM Conference on Recommender Systems, RecSys 2011, pp. 333–336. ACM (2011). https://doi.org/10.1145/2043932.2043996
6. Bieganski, P., Konstan, J., Riedl, J.: System, method and article of manufacture for making serendipity-weighted recommendations to a user, 25 December 2001. US Patent 6,334,127
7. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent Dirichlet allocation. J. Mach. Learn. Res. 3, 993–1022. http://dl.acm.org/citation.cfm?id=944919.944937
8. Bogers, T.: Tag-based recommendation. In: Brusilovsky, P., He, D. (eds.) Social Information Access. LNCS, vol. 10100, pp. 441–479. Springer, Cham (2018)
9. Breese, J.S., Heckerman, D., Kadie, C.: Empirical analysis of predictive algorithms for collaborative filtering. Technical report MSR-TR-98-12, Microsoft Research, May 1998. http://research.microsoft.com/apps/pubs/default.aspx?id=69656
10. Breiman, L.: Stacked regressions. Mach. Learn. **24**(1), 49–64 (1996). https://doi.org/10.1023/A:1018046112532
11. Celma, Ò., Herrera, P.: A new approach to evaluating novel recommendations. In: Proceedings of the 2008 ACM Conference on Recommender Systems, RecSys 2008, pp. 179–186. ACM (2008). https://doi.org/10.1145/1454008.1454038
12. Chen, T., Zhang, W., Lu, Q., Chen, K., Zheng, Z., Yu, Y.: SVDFeature: a toolkit for feature-based collaborative filtering. J. Mach. Learn. Res. **13**(1), 3619–3622 (2012). http://dl.acm.org/citation.cfm?id=2503308.2503357
13. Cremonesi, P., Garzottto, F., Turrin, R.: User effort vs. accuracy in rating-based elicitation. In: Proceedings of the Sixth ACM Conference on Recommender Systems, RecSys 2012, pp. 27–34. ACM (2012). https://doi.org/10.1145/2365952.2365963
14. Cremonesi, P., Koren, Y., Turrin, R.: Performance of recommender algorithms on top-n recommendation tasks. In: Proceedings of the Fourth ACM Conference on Recommender Systems RecSys 2010, pp. 39–46. ACM (2010). https://doi.org/10.1145/1864708.1864721
15. Ekstrand, M.: Similarity functions for user-user collaborative filtering. http://grouplens.org/blog/similarity-functions-for-user-user-collaborative-filtering/
16. Ekstrand, M.: Similarity functions in item-item CF. https://md.ekstrandom.net/blog/2015/06/item-similarity/
17. Ekstrand, M., Riedl, J.: When recommenders fail: predicting recommender failure for algorithm selection and combination. In: Proceedings of the Sixth ACM Conference on Recommender Systems, RecSys 2012, pp. 233–236. ACM (2012). https://doi.org/10.1145/2365952.2366002
18. Ekstrand, M.D., Harper, F.M., Willemsen, M.C., Konstan, J.A.: User perception of differences in recommender algorithms. In: Proceedings of the 8th ACM Conference on Recommender Systems, RecSys 2014, pp. 161–168. ACM (2014). https://doi.org/10.1145/2645710.2645737

19. Ekstrand, M.D., Kluver, D., Harper, F.M., Konstan, J.A.: Letting users choose recommender algorithms: an experimental study. In: Proceedings of the 9th ACM Conference on Recommender Systems, RecSys 2015, pp. 11–18. ACM (2015). https://doi.org/10.1145/2792838.2800195
20. Ekstrand, M.D., Ludwig, M., Konstan, J.A., Riedl, J.T.: Rethinking the recommender research ecosystem: reproducibility, openness, and LensKit. In: Proceedings of the Fifth ACM Conference on Recommender Systems, RecSys 2011, pp. 133–140. ACM (2011). https://doi.org/10.1145/2043932.2043958
21. Ekstrand, M.D., Riedl, J.T., Konstan, J.A.: Collaborative filtering recommender systems. Found. Trends Hum.-Comput. Interact. **4**(2), 81–173 (2011). https://doi.org/10.1561/1100000009
22. Funk, S.: Netflix update: try this at home. http://sifter.org/~simon/journal/20061211.html
23. Golbandi, N., Koren, Y., Lempel, R.: Adaptive bootstrapping of recommender systems using decision trees. In: Proceedings of the Fourth ACM International Conference on Web Search and Data Mining, WSDM 2011, pp. 595–604. ACM (2011). https://doi.org/10.1145/1935826.1935910
24. Goldberg, K., Roeder, T., Gupta, D., Perkins, C.: Eigentaste: a constant time collaborative filtering algorithm. Inf. Retrieval **4**(2), 133–151 (2001). https://doi.org/10.1023/A:1011419012209
25. Guy, I.: People recommendation on social media. In: Brusilovsky, P., He, D. (eds.) Social Information Access. LNCS, vol. 10100, pp. 570–623. Springer, Cham (2018)
26. Harper, F.M., Konstan, J.A.: The MovieLens datasets: history and context. ACM Trans. Interact. Intell. Syst. **5**(4), 19:1–19:19 (2015). https://doi.org/10.1145/2827872
27. Herlocker, J., Konstan, J.A., Riedl, J.: An empirical analysis of design choices in neighborhood-based collaborative filtering algorithms. Inf. Retrieval **5**(4), 287–310 (2002). https://doi.org/10.1023/A:1020443909834
28. Herlocker, J.L., Konstan, J.A., Terveen, L.G., Riedl, J.T.: Evaluating collaborative filtering recommender systems. ACM Trans. Inf. Syst. **22**(1), 5–53 (2004). https://doi.org/10.1145/963770.963772
29. Hill, W., Stead, L., Rosenstein, M., Furnas, G.: Recommending and evaluating choices in a virtual community of use. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 1995, pp. 194–201. ACM Press/Addison-Wesley Publishing Co. (1995). https://doi.org/10.1145/223904.223929
30. Hofmann, T.: Probabilistic latent semantic indexing. In: Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 1999, pp. 50–57. ACM (1999). https://doi.org/10.1145/312624.312649
31. Hofmann, T.: Latent semantic models for collaborative filtering. ACM Trans. Inf. Syst. **22**(1), 89–115 (2004). https://doi.org/10.1145/963770.963774
32. Huang, Z., Chung, W., Chen, H.: A graph model for E-commerce recommender systems. J. Am. Soc. Inf. Sci. Technol. **55**(3), 259–274 (2004). https://doi.org/10.1002/asi.10372
33. Jamali, M., Ester, M.: TrustWalker: a random walk model for combining trust-based and item-based recommendation. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2009, pp. 397–406. ACM (2009). https://doi.org/10.1145/1557019.1557067

34. Jannach, D., Lerche, L., Gedikli, F., Bonnin, G.: What recommenders recommend – an analysis of accuracy, popularity, and sales diversity effects. In: Carberry, S., Weibelzahl, S., Micarelli, A., Semeraro, G. (eds.) UMAP 2013. LNCS, vol. 7899, pp. 25–37. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38844-6_3
35. Jannach, D., Lerche, L., Zanker, M.: Recommending based on implicit feedback. In: Brusilovsky, P., He, D. (eds.) Social Information Access. LNCS, vol. 10100, pp. 510–569. Springer, Cham (2018)
36. Järvelin, K., Kekäläinen, J.: Cumulated gain-based evaluation of IR techniques. ACM Trans. Inf. Syst. **20**(4), 422–446 (2002). https://doi.org/10.1145/582415.582418
37. Kluver, D., Konstan, J.A.: Evaluating recommender behavior for new users. In: Proceedings of the 8th ACM Conference on Recommender Systems, RecSys 2014, pp. 121–128. ACM (2014). https://doi.org/10.1145/2645710.2645742
38. Kluver, D., Nguyen, T.T., Ekstrand, M., Sen, S., Riedl, J.: How many bits per rating? In: Proceedings of the Sixth ACM Conference on Recommender Systems, RecSys 2012, pp. 99–106. ACM (2012). https://doi.org/10.1145/2365952.2365974
39. Knijnenburg, B., Willemsen, M., Gantner, Z., Soncu, H., Newell, C.: Explaining the user experience of recommender systems. User Model. User-Adap. Interact. **22**(4), 441–504. https://doi.org/10.1007/s11257-011-9118-4
40. Koren, Y.: Factorization meets the neighborhood: a multifaceted collaborative filtering model. In: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2008, pp. 426–434. ACM. https://doi.org/10.1145/1401890.1401944
41. Koren, Y., Bell, R., Volinsky, C.: Matrix factorization techniques for recommender systems. Computer **42**(8), 30–37 (2009). https://doi.org/10.1109/MC.2009.263
42. Lathia, N., Hailes, S., Capra, L.: Temporal collaborative filtering with adaptive neighbourhoods. In: Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2009, pp. 796–797. ACM (2009). https://doi.org/10.1145/1571941.1572133
43. Lathia, N., Hailes, S., Capra, L., Amatriain, X.: Temporal diversity in recommender systems. In: Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2010, pp. 210–217. ACM (2010). https://doi.org/10.1145/1835449.1835486
44. Lee, D., Brusilovsky, P.: Recommendations based on social links. In: Brusilovsky, P., He, D. (eds.) Social Information Access. LNCS, vol. 10100, pp. 391–440. Springer, Cham (2018)
45. Lemire, D., Maclachlan, A.: Slope one predictors for online rating-based collaborative filtering. In: Proceedings of SIAM Data Mining (SDM 2005) (2005). https://arxiv.org/abs/cs/0702144
46. Massa, P., Avesani, P.: Trust-aware recommender systems. In: Proceedings of the 2007 ACM Conference on Recommender Systems, RecSys 2007, pp. 17–24. ACM (2007). https://doi.org/10.1145/1297231.1297235
47. McAuley, J., Leskovec, J.: Hidden factors and hidden topics: Understanding rating dimensions with review text. In: Proceedings of the 7th ACM Conference on Recommender Systems, pp. 165–172. RecSys 2013. ACM (2013). https://doi.org/10.1145/2507157.2507163
48. McNee, S.M., Lam, S.K., Konstan, J.A., Riedl, J.: Interfaces for eliciting new user preferences in recommender systems. In: Brusilovsky, P., Corbett, A., de Rosis, F. (eds.) UM 2003. LNCS (LNAI), vol. 2702, pp. 178–187. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44963-9_24

49. Ning, X., Karypis, G.: Slim: Sparse linear methods for top-n recommender systems. In: Proceedings of the IEEE 11th International Conference on Data Mining, ICDM 2011, pp. 497–506, December 2011. https://doi.org/10.1109/ICDM.2011.134
50. Olson, J.S., Kellogg, W.A. (eds.): Ways of Knowing in HCI. Springer, New York (2014). https://doi.org/10.1007/978-1-4939-0378-8
51. O'Mahoney, M., Smyth, B.: From opinions to recommendations. In: Brusilovsky, P., He, D. (eds.) Social Information Access. LNCS, vol. 10100, pp. 480–509. Springer, Cham (2018)
52. Phuong, N.D., Thang, L.Q., Phuong, T.M.: A graph-based method for combining collaborative and content-based filtering. In: Ho, T.-B., Zhou, Z.-H. (eds.) PRICAI 2008. LNCS (LNAI), vol. 5351, pp. 859–869. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89197-0_80
53. Pu, P., Chen, L., Hu, R.: A user-centric evaluation framework for recommender systems. In: Proceedings of the Fifth ACM Conference on Recommender Systems, RecSys 2011, pp. 157–164. ACM (2011). https://doi.org/10.1145/2043932.2043962
54. Rashid, A.M., Albert, I., Cosley, D., Lam, S.K., McNee, S.M., Konstan, J.A., Riedl, J.: Getting to know you: learning new user preferences in recommender systems. In: Proceedings of the 7th International Conference on Intelligent User Interfaces, IUI 2002, pp. 127–134. ACM (2002). https://doi.org/10.1145/502716.502737
55. Rashid, A.M., Karypis, G., Riedl, J.: Learning preferences of new users in recommender systems: an information theoretic approach. ACM SIGKDD Explor. Newslett. **10**(2), 90–100 (2008). https://doi.org/10.1145/1540276.1540302
56. Rendle, S., Freudenthaler, C., Gantner, Z., Schmidt-Thieme, L.: BPR: Bayesian personalized ranking from implicit feedback. In: Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI 2009, pp. 452–461. AUAI Press. http://dl.acm.org/citation.cfm?id=1795114.1795167
57. Rendle, S., Schmidt-Thieme, L.: Online-updating regularized kernel matrix factorization models for large-scale recommender systems. In: Proceedings of the 2008 ACM Conference on Recommender Systems, RecSys 2008, pp. 251–258. ACM (2008). https://doi.org/10.1145/1454008.1454047
58. Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., Riedl, J.: GroupLens: an open architecture for collaborative filtering of netnews. In: Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work, CSCW 1994, pp. 175–186. ACM (1994). https://doi.org/10.1145/192844.192905
59. Rogers, S.K.: Item-to-item recommendations at Pinterest. In: Proceedings of the 10th ACM Conference on Recommender Systems, RecSys 2016, pp. 393–393. ACM (2016). https://doi.org/10.1145/2959100.2959130
60. Said, A., Fields, B., Jain, B.J., Albayrak, S.: User-centric evaluation of a k-furthest neighbor collaborative filtering recommender algorithm. In: Proceedings of the 2013 Conference on Computer Supported Cooperative Work, CSCW 2013, pp. 1399–1408. ACM (2013). https://doi.org/10.1145/2441776.2441933
61. Sarwar, B., Karypis, G., Konstan, J., Riedl, J.: Item-based collaborative filtering recommendation algorithms. In: Proceedings of the 10th International Conference on World Wide Web, WWW 2001, pp. 285–295. ACM. https://doi.org/10.1145/371920.372071
62. Sarwar, B.M., Karypis, G., Konstan, J.A., Riedl, J.T.: Application of dimensionality reduction in recommender system - a case study. In: WebKDD 2000. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.8381
63. Shan, H., Banerjee, A.: Generalized probabilistic matrix factorizations for collaborative filtering. In: IEEE International Conference on Data Mining, pp. 1025–1030. IEEE Computer Society (2010). https://doi.org/10.1109/ICDM.2010.116

64. Sill, J., Takacs, G., Mackey, L., Lin, D.: Feature-weighted linear stacking. arXiv:0911.0460 [cs], http://arxiv.org/abs/0911.0460
65. Smyth, B., McClave, P.: Similarity vs. diversity. In: Aha, D.W., Watson, I. (eds.) ICCBR 2001. LNCS (LNAI), vol. 2080, pp. 347–361. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44593-5_25
66. Su, X., Khoshgoftaar, T.M.: A survey of collaborative filtering techniques. Adv. Artif. Intell. (2009). http://www.hindawi.com/journals/aai/2009/421425/abs/
67. Weng, L.T., Xu, Y., Li, Y., Nayak, R.: Improving recommendation novelty based on topic taxonomy. In: 2007 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology Workshops, pp. 115–118 (2007)
68. Zhang, M., Hurley, N.: Avoiding monotony: improving the diversity of recommendation lists. In: Proceedings of the 2008 ACM Conference on Recommender Systems, RecSys 2008, pp. 123–130. ACM (2008). https://doi.org/10.1145/1454008.1454030
69. Zhang, Y.C., Saghdha, D.Ò., Quercia, D., Jambor, T.: Auralist: introducing serendipity into music recommendation. In: Proceedings of the Fifth ACM International Conference on Web Search and Data Mining, WSDM 2012, pp. 13–22. ACM (2012). https://doi.org/10.1145/2124295.2124300
70. Ziegler, C.N., McNee, S.M., Konstan, J.A., Lausen, G.: Improving recommendation lists through topic diversification. In: Proceedings of the 14th International Conference on World Wide Web, WWW 2005, pp. 22–32. ACM (2005). https://doi.org/10.1145/1060745.1060754