Tutorial on Application-oriented Evaluation of Recommendation Systems

Guy Shani [a]
[a] *Department of Information System Engineering Ben Gurion University of the Negev E-mail: shanigu@bgu.ac.il*

Asela Gunawardana [b]
[b] *Microsoft Research E-mail: aselag@microsoft.com*

Recommendation systems are now widely used in many commercial applications. This tutorial focuses on the evaluation of such systems, from an application-oriented view. The tutorial recommends best practices, suggests a protocol for the evaluation process, and reviews a set of metrics that can be evaluated. The practices in this paper are motivated by similar procedures in nearby areas, such as machine learning, and information retrieval.

Keywords: Recommender Systems, Evaluation

1. Introduction

Recommendation systems [18] — systems that actively suggest items to users — can now be found in many applications. For example CNN[1] presents at the bottom of each article a set of other stories that the user might be interested in, given the current article, under the title “We recommend”. They also present a set of “Most popular” stories, which can also be considered as a recommendation for articles.

Another well-known example can be found in Amazon[2] where the system presents the user with various recommended items under different categories, such as “New for You” or “Recommendations for You in Books”. These recommendations are typically computed using the set of items that the user has previously bought in Amazon. Upon adding an item to the shopping basket, Amazon presents a list of recommended items under the title “Customers Who Bought This Item Also Bought”. The title hints at the popular recommendation approach that computes a set of similar users (in this case, people who have bought the same item), and recommends additional items that these similar users have purchased.

A third example is the Netflix video rental and streaming service, which incorporates several types of recommendations — including suggesting additional movies once a movie is added to the watching queue, and predicting how many stars would a user assign to a movie that she has not yet watched, e.g., when viewing new releases.

In all these applications, a recommendation system is used within an existing application, be it a news website, an e-commerce retailer, or an online video streaming service. Indeed, in most cases the recommendation system has an assisting role within an application, and it is rarely a stand alone application by itself.

There is vast literature on various methods for constructing recommendations systems, starting with the general approach, such as Collaborative Filtering [2] vs. Content-Based recommendations [17], through specific algorithms, such as decision trees [2], SVD [9], or naive Bayes classifiers [16], and on to the setting of specific parameters. While some of these choices are dictated by the application and the available data, some can be chosen by the application designers so as to meet the application goals.

This tutorial focuses on best practices for evaluating recommendation systems with a strong emphasis on applications. That is, we suggest methods for an application designer to better understand how the choices that they make will affect the behavior of their application.

This can help the application designer answer several application oriented questions:

– Does the recommendation help the user of the application? For example, can Netflix identify good movies, can Amazon recommend a book that the user would enjoy reading, or can CNN recommend additional interesting stories.

[1] www.cnn.com
[2] www.amazon.com

– Does the recommendation help the application? For example, will users remain Netflix subscribers, will they buy more books on Amazon, or will they remain longer in the CNN website, (and hence view more ads)?

To answer these questions, the application designer must first identify the designated goal of the recommendation system within the application. Such goals may be to reduce user effort in reaching items of interest, increasing user time on the website, increasing the total sales, or increasing revenue. Once the goal has been established, we can formalize concrete measurable metrics appropriate for the specific goal and its intended outcome. We will later discuss appropriate measures for different goals. Finally we can create experiments that evaluate these measures. These experiments can be executed over various recommendation approaches alternatives, in order to understand how our choices over alternatives affect the achievement of these goals.

We call this the selection problem in recommendation systems, i.e. the identification of appropriate alternative for achieving a set of goals and tasks, given certain limitations and constraints. More specifically, the selection problem is about choosing the most appropriate system from a set of candidates. An underlying assumption of the selection problem is that there is no "silver bullet." That is, no single alternative is the most accurate, fastest, cheapest and in general the most appropriate choice, independently of the application and its goals. We assume that different applications with different goals require different choices to be made and different approaches to be used to meet the application needs.

Clearly, the selection problem is not the only interesting problem in recommendation system development. Many researchers publish papers that compare their algorithm of choice with other state-of-the-art algorithms in order to understand the strengths and weaknesses of a given algorithm. Such evaluations also have commercial value. Consider for example the booming market of recommendation providers who offer recommendation services to various businesses. These providers have the benefit of specializing in recommendation systems and have expertise in tailoring solutions to applications. The provider may have a set of pre-constructed parameterized systems that they must adjust to the needs of the current business purchasing their services. Thus, the provider must know in advance which approach is suitable to the characteristics of the different applications. Such providers might want to experiment with their pre-made algorithms on various datasets with no specific application in mind, in order to better understand these characteristics. While in this paper we do not directly discuss such evaluation scenarios, most of our suggestions can also be applied there.

2. Collaborative Filtering and Content-Based Algorithms

There are two dominant approaches for computing recommendations for the *active user* — the user that is currently interacting with the application and the recommender system. First, the *collaborative filtering* approach [2] identifies a neighborhood of users that are similar to the active user. This set of neighbors is based on the similarity of observed preferences between these users and the active user. Then, items that were preferred by users in the neighborhood are recommended to the active user. In this approach, the system only has access to the item identifiers, and no additional information about items is available. For example, websites that present recommendations titled "users who preferred this item also prefer" typically use some type of collaborative filtering algorithm.

A second popular approach is the *content-based* recommendation [17]. In this approach, the system has access to a set of item features. The system then learns the user preferences over features, and uses these computed preferences to recommend new items with similar features. Such recommendations are typically titled "similar items".

Each approach has advantages and disadvantages, and a multitude of algorithms from each family, as well as a number of hybrid approaches have been suggested. This paper, though, makes no distinction between the underlying recommendation algorithms when evaluating their performance. Just as users should not need to take into account the details of the underlying algorithm when using the resulting recommendations, it is inappropriate to select different evaluation metrics for different recommendation approaches. In fact, doing so would make it difficult to decide which approach to employ in a particular application.

3. Goals and Tasks

As we have explained above, the first step in creating an evaluation procedure is to properly define the goals

Fig. 1. Book recommendation in Amazon.

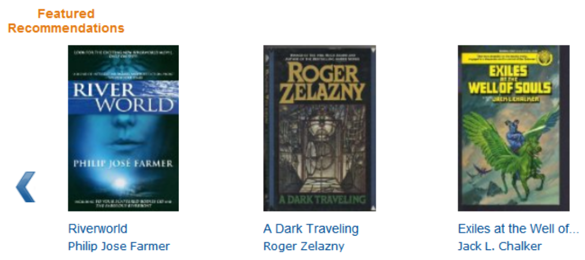
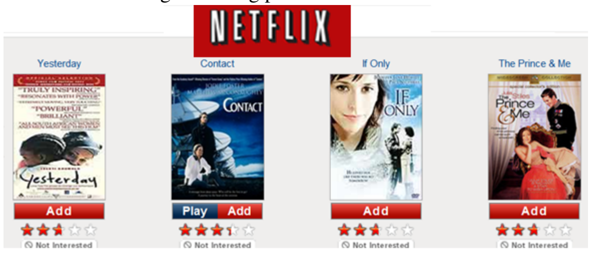


Fig. 2. Rating prediction in Netflix.



and the tasks of the application for which the recommendation system is constructed. We now delve deeper into the definition of goals and tasks.

We consider goals to be the high level intentions of the application designer. For example, the goal of the system can be to improve the user experience while reading articles, to help the user find electronic gadgets of interest, or to increase the e-commerce website profitability.

Tasks, on the other hand, should be quantitative and measurable. They are the manifestation of the goals into specific optimization quantities. For example, when the goal is to improve the user experience, the task can be to minimize the number of clicks before an interesting article is found. When the goal is to increase the website profitability, the task may be to maximize the number of items or the net profit in each separate transaction.

Given the defined task, the evaluation must focus on measuring success on the given task [6]. It is hence crucial to first define the task, and only then define the metric that will be measured.

We can identify three popular tasks in recommendation system literature and applications. The first is the *recommendation task* where the system needs to recommend a set of items that the user will choose (see, e.g., Figure 1). This is also often known as Top-N recommendations. In this task the system must choose a finite set of recommended items that must be displayed to the user. The number of items is typically limited by the application designer, and usually only 3 or 5 items can be presented. In this scenario we may want to optimize the precision of the set, i.e. the number of items within the set that the user will choose, but we discuss other alternatives later on.

A second common task the the *utility optimization task* where the system must optimize some utility, such as net profit, or time on the website. In many case, from the commercial application designer point of view, this is the preferred task, because it translates directly to the profit that the recommendation system generates, and hence its value for the application. It is important to note that while system utility is typically easy to define and measure, user utility is more difficult to handle. This is because the utility of different users is difficult to compare and aggregate. For example, when one alternative increases utility for user u_1 but decreases it for u_2, while a second alternative increases utility for u_2 and decreases it for u_1, it is difficult to aggregate the change in utility to decide which alternative is better.

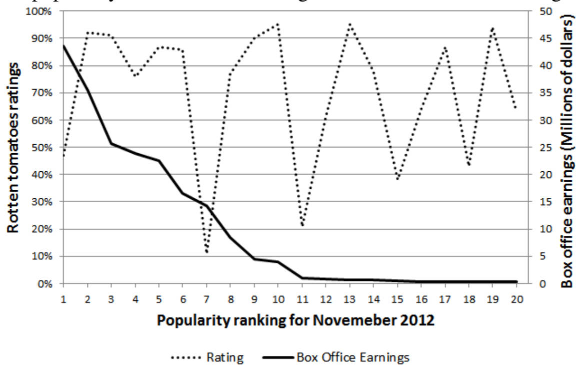
A third common task is the *rating prediction task* where the system predicts the rating that the system will assign to a given item (see, e.g., Figure 2). An obvious example is the prediction of the number of stars that a user will give to a video in Netflix. One might argue that prediction of ratings is not truly a recommendation system task, because there is no intentional influence on the the users, on the display of additional information.

On the other hand, a prediction for a high rating can certainly be interpreted as a suggestion to use the item. Either way, a huge amount of the recommendation system literature has been devoted to the prediction of ratings (see, e.g., [5,15,19,13,20,11]), and thus this is clearly of interest to this tutorial.

It is also sometimes argued that all these tasks area actually identical. Most specifically, one can argue that to construct a good set N of recommended items, we can simply predict the ratings of all items, and then put the N items with the highest predicted rating into the recommended set. The underlying assumption behind this approach is that ratings and preferences are identical, i.e., that a user will prefer to choose items with high ratings.

This assumption is clearly inaccurate in many domains. For example, a user may agree that a Porsche is a great car, and would certainly assign it a higher rating than, say, a Ford Focus. The user may not, however, choose to purchase the Porsche because it is too pricey for her. Even in domains where all items have

Fig. 3. DVD rental behavior, showing the lack of correlation between the popularity of rentals, their ratings, and their box office earnings.



an equivalent price, such as video rentals, users may still not choose the highly rated items. To demonstrate this behavior, we have collected DVD rental behavior from the Rotten Tomatoes website[3]. Figure 3 plots the ratings and the profits of the top 20 rentals in decreasing order of rental popularity for the month of August, 2012. As we can see, the ratings correlate poorly with the popularity in rentals or the profits. The ratings are high for the first two movies, then drops sharply, and then increases again. It is clear that people in many cases choose movies with low ratings, such as the movies in position 3 through 11 (except for 9) over highly rated movies, such as the ones on position 14 and on.

Thus, tasks are typically different and require different metrics to measure success in achieving them. We will later review various success metrics associated with different tasks.

4. Evaluation Process

We will now explain three different evaluation methods, namely offline testing, user studies, and online testing. We will review the advantages and disadvantages of each such method, and explain how they interact. We will then explain the complete evaluation process that we suggests, using the three evaluation methods.

4.1. Online Testing

In this method the recommendation system is evaluated directly within the application with real users. We typically compare two or more systems, or perhaps the application with and without the system. Users working with the application get assigned to a single alternative, in an *all-between* setting (also known as AB-testing). It is important to assign users to alternatives uniformly at random, so as to avoid inherent biases in different user populations.

This is the typical setting used by many search engines and typically known as *flights* [8], where the engineers of the search engine create a slight adjustment of, say, the ranking algorithm, and direct a portion of the traffic to this adjusted ranking engine for a short while (e.g., a day). Then, they compare the performance of the original and the adjusted algorithms and decide whether the adjustment is beneficial.

The advantages of this method is that its results are the most trustworthy, because they are gathered from real users who are, in many cases, oblivious to the ongoing experiment. The performance is also measured directly within the application and is hence most accurate, and takes all different parameters of the modification into account.

On the other hand, this approach can only be used when we have a working application with a reasonable number of users to experiment with. This approach cannot be used when a recommendation system is integrated in an application in its development phase.

It is also quite costly to compare many alternatives this way. Assuming that we need a reasonable amount of traffic in every adjustment, and given that typically a few hours are needed at the least for gathering sufficient data, clearly no more than a handful of alternatives can be evaluated this way.

Finally, there is a risk of having a negative impact on real users, such as degrading their experience with the application, or their trust in the system. Clearly, it is difficult to imagine an application willing to suffer a loss of business due to experiments over a recommendation system.

Thus, we must ensure, prior to testing the system online, that it is reasonable.

4.2. User Studies

To know whether an alternative is good enough to be tested online, we can run a user study. In such a setting we collect a group of test subjects, ask them to work with the application, observe their behavior, and ask them directly about their experience with the system [1].

[3]`www.rottentomatoes.com`

The test subjects are typically volunteers who are paid to try one or more alternative systems. The study can be done in a controlled environment, such as a lab, and it is possible to observe and record the behavior of the subjects during the test. We can also ask the subjects questions, for example concerning their expertise with the application before the test, or whether the recommendation system was helpful after the test. Subjects can also be asked to provide demographic information that can be used to relate their opinions to their background. For example, it might be that people with an extensive background in software engineering will be more willing to try complicated user interfaces.

It is crucial in such settings to find test subjects who represent the true user population of the system as closely as possible, to avoid biases in the results. For example, if typical users of the system come from all age groups, and the user study is done only using engineering students, who are comfortable with advanced software, the results might show a willingness to use the recommendation system that would not be replicated when installing the recommendation system in the real application. In many cases, however, getting a uniform sample of the true user population to participate in a user study is difficult, because most users are too busy, and user studies pay too little, to be appealing for regular working people.

In user studies, it is beneficial to hide the true purpose of the experiment from the subjects, to avoid people's natural subconscious tendency to meet the expectations of the study. It is also important to randomize the order of using the alternatives in a within subject setting so as to avoid biases towards the first (or last) alternative that is shown.

The advantages of user studies are the ability to compare various alternatives for each subject (known as the *within subjects* setting). We can ask users directly which alternative they preferred. We can also collect qualitative information this way, such as subject preferences, in addition to quantitative information that is collected in all methods. This is also the only method that can provide reliable explanations as to why one system was better than the other, by either asking the subjects, or observing the way they behave. This information can later be used to improve the alternatives.

The main problem with user studies is that they are unable to capture the real user behavior when accomplishing their true goals. For example, people who go to an e-commerce website to purchase a laptop may behave significantly different than people who are asked to find a reasonable laptop in a user study. In general, it is difficult to simulate well in user studies the way that users behave when spending their real resources, such as money, time, attention, effort, and so forth.

Finally, user studies are expensive, because the subjects are typically paid, and evaluation in a controlled environment, with a professional team to conduct the experiment and collect observations is also expensive. Thus, in this setting we can, again, compare no more than a handful of alternatives.

4.3. Offline Testing

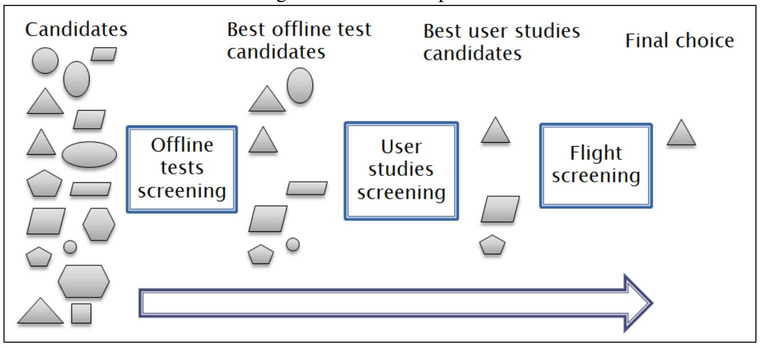
Testing with real people is always expensive, and when these are real users it is also risky for the application. Thus, when working with real people we will always be restricted to experimenting with only a handful of alternatives. Still, we need a method to experiment with many alternatives, especially for tuning the parameters of the algorithms. Simulations of real users are an obvious choice for cheap and rapid experiments.

In order for the results computed in simulations to be trustworthy, it is important for the simulation to be as closely related to the behavior of the real users as possible. In machine learning, such simulations are constructed by, e.g., taking a set of items labeled by real users, hiding their labels, and predicting the hidden labels, thus simulating the behavior of users when labeling the items. Indeed, this method is reasonable when we're interested in rating prediction, which is one of the tasks suggested above.

However, when we are concerned with the recommendation task, this simulation process is less appropriate. Specifically, recommendations affect user behavior. For example, we would expect a strong recommendation to purchase a book on Amazon to increase its overall purchases. We would expect an article that is recommended in CNN to have more readers, and so forth. This is the main goal of recommender systems and if this assumption is void, then application designers would not have invested money in constructing recommendation systems.

This assumption can be stated as follows, given that the (prior) probability of choosing an item i is p, the (posterior) probability of choosing i given a recommendation for it must be at least p. Still, it is difficult to estimate the effect of the recommendation on p. For example, it may be that for unpopular items, the effect might be huge, while for very popular items, a recommendation can be almost useless. Thus, it is difficult to know in simulation what is the true success of the recommendation system when using such simula-

Fig. 4. The selection process.



tions. Currently, most evaluation methods simply assume that the probability of choosing i given the recommendation remains p (i.e. that p of the user population will choose i), and thus, at the very least, provide a lower bound over the true performance of the system within the application.

The standard simulation procedure uses an existing dataset of interactions of users with the application, typically using a fixed existing system (as in the case of Movielens and Netflix) or without any recommendation system.

It is popular to take generic datasets, such as the Netflix prize dataset[4], or the Movielens dataset[5], and experiment with them. It is however important to note that it would be difficult to draw reliable conclusions from domains that are significantly different than the application that we care about.

We typically take the interactions dataset and split it into two — a training set, which is used as input to the recommendation algorithm, and a test set, which is used to measure the performance of the algorithm. Such a division allows us to model the true state of the system that tries to predict the results of new interactions (simulated by the test set), and to discredit algorithms that only learn to memorize observed interactions.

In splitting the dataset we name three alternatives:

– Split on time: perhaps the most reliable method is to choose a time point, and split the interactions into those occurring prior to the chosen time point (the training set), and those occurring afterwards (the test set). This is the closest simulation to the true state of the system within the application, that knows all interactions prior to the current time, and knows nothing about the future.
– Split on order: when for some reason splitting on time cannot be achieved, such as when the dataset does not contain reliable timestamps, it is reasonable to split the interactions of each user somewhere along the sequence of interactions. That is, given the ordered sequence of interactions, we would choose a random point along this sequence, and take all the interactions prior to that sequence as our training set, and all the interactions afterwards as our test set.
– Split uniformly: finally, when no information on time or order is given, we can split the dataset uniformly. Here we take the n interactions of each user, and pick a number $k \in [0, n]$. We then pick k random interactions and add them to the training set, when the rest of the interactions are added to the test set.

Other methods exist for splitting the dataset, such as using a fixed number k of interactions in the training set for all users (known as "given k"), or adding a fixed number k of interactions to the test set for all users (known as "all but k"). While such evaluations may shed light on the performance of the algorithms in these special cases, it is difficult to draw conclusions from such experiments as to the behavior of the algorithm within the real system.

Some people also suggest to use the so-called model-based simulations [14], where some generative model is used to produce artificial interactions of users with the system. While such models can capture complex behavior, and cheaply generate as many data as needed, it is crucial to be certain that the model truly simulates the real behavior of users. Otherwise, it is possible that we will achieve a performance in the experiments that will be significantly different from the one observed within the real application.

[4] `www.netflixprize.com`
[5] `www.grouplens.org/node/73`

4.4. Combining Offline tests, User Studies, and Online Tests

Looking on the advantages and disadvantages of the three approaches, an obvious process comes to mind:

1. Begin with offline tests. Use as many alternatives as possible, i.e., different approaches (content-based and collaborative filtering, for example), different algorithms from each approach, various parameter setting for algorithms, and so forth. Select the top m_1 performing alternatives to continue to the next step.
2. Use user studies with these m_1 candidates. Obviously, m_1 must be small enough so that we can afford these user studies. Choose the top m_2 alternatives to continue to the next step.
3. Use online tests to pick the best of the m_2 candidates. This would be the alternative that will be incorporated into the application.

This process is illustrated in Figure 4.

5. Measuring Performance

In this section we will briefly review the most popular performance metrics in recommender systems. For a more exhaustive list of measurable attributes, please see [22]. We begin this section with some general guidelines for measuring performance, and then delve into particular attributes of performance that can be evaluated.

As we explained above, it is important to first decide about the goals and the tasks of the recommendation engine within the application. Then, we can conclude which attributes are of importance for fulfilling the specified tasks, and measure how well does the system perform on these attributes. In many cases we may care about more than a single attribute, and it is typically the case that in order to improve upon one attribute, we may be forced to reduce performance on another attribute. It is crucial to be aware of these tradeoffs, and explicitly measure them. To do that, we typically compute the dependent attributes on a number of parameter settings, in order to draw a curve of the tradeoff. We will later focus on a few popular tradeoffs, such as the precision-recall tradeoff.

For all other parameters of the system that are not measured and optimized, we have to be careful not to change their values between different alternatives. We have to maintain the ceteris paribus principle (all else being equal), so as to be able to safely conclude that one system was better than the other due to, e.g., the parameter that we are currently tuning, and not because of some other parameter. For example, if we wish to modify the way that the system computes relevant items to recommend, it is important to evaluate all alternatives using the same user interface.

In addition, in order to make reliable conclusions from our experiments, it is important to compute the statistical significance of our results. More specifically, in the selection problem that we're interested in, we typically say that system A is superior to system B because it achieved a higher score on some test. To be able to claim such superiority, we need to be able to convince ourselves that the probability that A achieved a better score by luck is low. This is typically done by running an appropriate statistical test that computes a p-value, which is the probability that the observed scores were achieved by luck, and do not represent the true behavior of the system. For more information, please see [22].

Perhaps the simplest method to select one alternative from a list of candidates, is to ask users which system they preferred in a between-subjects setting [7]. In this case, all the tradeoffs between the various relevant attributes are implicitly considered by the user. There are a few disadvantages to this approach; First, this must be done using people, which requires an expensive evaluation. Second, it is difficult to conclude why one alterative did better, and as such, it is difficult to further improve the system. If we ask more specific questions, such as whether one system was easier to use, or whether one system was more accurate, then we essentially fall back to asking about specific attributes, which can be measured directly.

5.1. Measuring Accuracy

Perhaps the most popular attribute that we care about with recommendation systems is the accuracy of the results. This is appropriate for the two most common tasks that we listed above — the rating prediction task and the recommendation task. We will now review metrics for each of these two tasks.

In rating prediction, we typically predict a rating value $\hat{r}_{ui}$ that a user u will assign to an item i, and then observed the true value r_{ui} either through simulations or by getting some feedback from users. We can then compute the prediction error $|\hat{r}_{ui} - r_{ui}|$. A popular way to aggregate the errors over a set of predictions is by computing the root of the mean square error (RMSE) using:

$$RMSE = \sqrt{\frac{1}{|J|} \sum_{(u,i)\in J} (\hat{r}_{ui} - r_{ui})^2} \quad (1)$$

where J is the set of user-item pairs for which we are making predictions and observing the actual values. The preferred system is the one that minimizes the RMSE over J.

Example 1. *To demonstrate this, we ran an experiment over the MovieLens[6] dataset that contains ratings of users for movies. The task is to predict the missing ratings that users will assign movies.*

We split the dataset into a training set (85% *of the data) and a test set (*J*) using the uniform split, because the dataset does not contain meaningful time-stamps or interaction order information.*

We used 4 alternative rating prediction algorithms — memory-based neighborhood algorithms using the Pearson correlation or the Cosine metric for user-user similarity [2,4], an SVD model [11,10], and a random predictor, based on the rating distribution of the user. We evaluate these candidates using an RMSE score to measure the accuracy of the predicted ratings of the different candidates. The results can be seen in Table 1. In this case, the neighborhood-based algorithm using the Pearson correlation metric had the lowest RMSE and was hence the winner.

Table 1

RMSE results for predicting user ratings by different methods over the MovieLens dataset.

Method	RMSE
Pearson	0.9288
SVD	0.938
Cosine	0.9547
Random	1.3491

As we have discussed above, it is important to use a statistical test to make reliable conclusions. The simplest test that can be used is the sign test[3,12], and we will demonstrate here how it is used. The sign test is used in a paired setting, where each alternative is evaluated over the same set J. We can then compare the results that each system achieved and conclude a "winner", e.g., the system with the lower RMSE. In this case, we compute the RMSE of each user u and count the number of times that alternative A achieved a lower RMSE than alternative B, denoted n_A. If we are interested in a pure winner, i.e., that A would achieve a strictly better RMSE than B, then draws should count against A, that is, they should not be counted in n_A. If we are interested in the case where A should be no worse than B, then draws should be counted in n_A.

Let n be the number of users in J for which the predictions were made. We can now compute the probability that we will observe at least n_A times that system A got a better score than system B under the null hypothesis that the two systems are equal using:

$$p = (0.5)^n \sum_{i=n_A}^{n} \frac{n!}{i!(n-i)!} \quad (2)$$

when this p-value is below some predefined value (typically, 0.05) we can say that the null hypothesis that the two system have an equal performance is rejected.

Example 2. *Coming back to our rating prediction task over the MovieLens dataset in the previous example, we now wish to check whether the the candidate ranking is significant. That is, we want to know how sure are we, given the relatively small differences in RMSE, whether we can confidently claim that, e.g., Pearson achieves a better RMSE than SVD. We therefore compute the sign test pairwise* p*-values as explained above. We count for each pair of methods,* A *and* B *(say, Pearson and SVD), for how many users did* A *achieve a better (lower) score than* B*. We then use Equation 2 to compute the* p*-value for the pair of the methods. The results are reported in Table 2. Observe that we have computed only results comparing an algorithm that achieved a better RMSE to an algorithm that achieved a worse RMSE.*

From the table we can see that for Pearson vs. SVD, the p*-value is above* 0.05*, and therefore we say that the results are not significant. In order to achieve significance, we can test with more users, which may reduce the* p*-value. For SVD vs. Cosine, the* p*-value is below* 0.05*, and we can hence decide that SVD is significantly better than Cosine. Random prediction method is significantly worse than all other methods.*

We may now decide, given the RMSE and the sign-test results to select both the Pearson candidate and the SVD candidate to be compared in a within-users user study. In such a study, volunteers may observe rating predictions from both algorithms and comment upon their perceived accuracy. Following the user-study we can flight the two alternatives in a system like Netflix or MovieLens that presents users with movie rating pre-

[6]http://www.grouplens.org/node/73

dictions. In an AB-testing procedure we can measure online how users agree or fix the predicted ratings of each candidate. Then, we can make an informed decision whether to use SVD or Pearson to compute the predicted ratings that are shown to users in our application.

Table 2

Pair-wise p-values computed using the sign-test for the experiment reported in Table 1. Each method is compared to methods that achieved worse scores.

	Random	Cosine	SVD
Pearson	< 0.001	< 0.001	0.08
SVD	< 0.001	0.011	X
Cosine	< 0.001	X	X

In the recommendation task the user is presented with a list of recommendations, and may choose to use one or more of the recommended items, or use any other item. The possible results are hence:

	Recommended	Not recommended
Used	True-positive	False-negative
Not used	False-positive	True-negative

We can count how many items fall into each of the cells of this confusion matrix. Then, we can now compute the precision of the recommended items which is the portion of the successful recommended items out of the list of recommended items:

$$Precision = \frac{\#tp}{\#tp + \#fp} \tag{3}$$

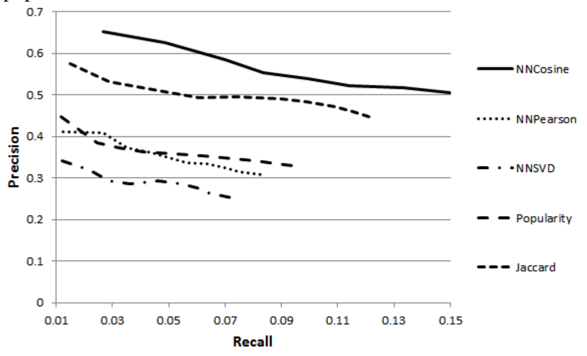
When the number of items to be presented to the user, N, if fixed, we typically optimize the precision, which is called Precision@N. When we wish to decide which value of N to use, we need to tradeoff precision with recall, which is the portion of successful recommendations out of all the items that the user has used:

$$Recall = \frac{\#tp}{\#tp + \#fn} \tag{4}$$

To visualize this tradeoff we can draw a precision-recall curve using varying values of N.

An alternative whose curve dominates all other curves everywhere is typically deemed better. In most cases, however, different alternatives dominate different segments of the curve, and thus we need to fist decide on a reasonable range of N, and then focus on the segment containing this segment. In some cases people use a summary of the curve such as the area under the curve, or the so-called F-score, but the implications of such summaries over the application are difficult to foretell.

Fig. 5. Precision-Recall curves for various algorithms over the MovieLens dataset. NN denotes user-user nearest neighbor approach using various similarity metrics (Pearson, Cosine, and SVD), Jaccard denotes item-item recommendations using the Jaccard coefficient, and Popularity is the fixed recommendation list of the most popular items.



Example 3. *To demonstrate the computation of the precision-recall curve, we use the MovieLens dataset again, this time ignoring the ratings and predicting which movies will a user rate. We thus simulate a recommendation task where the system should present to the user a list of movies that she would like to rate.*

We compare here 5 alternatives — 3 nearest neighbor approaches (denoted NN), where users are recommended items that were popular within their neighborhood. The neighborhood computation in the various alternatives uses different similarity metrics — Person, Cosine, and SVD. We also compare the fixed recommendation list of the most popular items overall, and an item-item recommender using the Jaccard coefficient to compute item-item similarities, and a max aggregation operator [13].

The results can be seen in Figure 5. In this example, the NNCosine method dominates all others, while NN-Pearson and the Popularity approach dominate each other in different regions.

As we have explained above, different metrics denote different preferences. As Gunawardana and Shani [6] show, choosing an inadequate metric may lead to choosing the wrong alternative. It is thus important to choose the metric that is most suitable for the task that we care about.

5.2. *Utility-based Accuracy*

For the third task, the utility-maximization scenario, we wish to select the list of the recommendations with the highest expected utility. We can measure instead the average utility of the list as:

$$R_u = \sum_j utility(r_{ui_j} utility(j) \quad (5)$$

where the sum is upon the items in the list, $utility(r_{ui})$ is the utility of recommending item i to user u. For example, we can say that this utility is the net profit of the item i if u has bought i and 0 otherwise. Another alternative [21] is to use $utility(r_{ui}) = -log(popularity(i)$ when i is used and 0 otherwise, capturing the amount of information in recommending item i.

The last part, $utility(j)$ denotes the utility of presenting items down the recommendation list. For example, [2] suggest an exponential decay decrease using $utility(j) = 2^{-\frac{j}{\alpha}}$, where α is the so called half-life parameter, i.e., the location in the list that the user will choose with probability half.

We can continue to compute in retrospect the utility of the optimal list — the list of recommended items that we would have presented, had we known what the user will choose, denoted R^*_u. Then, we can compute the R score, which is the portion of the optimal utility achieved by the alternative:

$$R = 100 \frac{\sum_u R_u}{\sum_u R^*_u} \quad (6)$$

5.3. Evaluating Other Properties

It is easy to believe that accuracy is important, and that users prefer a system that is more accurate. Measuring accuracy is also well understood, and is similar to evaluation in other areas, such as machine learning and information retrieval.

For other properties, it is somewhat less trivial to be convinced of their importance. For example, is the diversity of the list of recommendations important? Is it better to have a less accurate list with more diverse items? So far, we have pursued an application oriented evaluation, where the important attributes are dictated by the application designer. Sometimes, however, the application designer may also require help in identifying the set of important attributes. This can be achieved by running user studies, where various attributes are being evaluated. For example, we can present to users lists of items with varying diversity and ask users which list looks better, thus concluding whether users notice and prefer diverse lists.

We can also use such user studies to identify which attributes are more important to users, and conclude some importance ordering, or even some weighting of attributes.

Interested readers can find in [22] a list of other attributes and suggestions of how they can be measured in experiments.

6. Conclusion

To conclude, we have presented in this tutorial an application-oriented approach to the evaluation of recommender systems. We suggest to first identify the goals that the application designer sets for the recommender system within the application. Then, we must identify quantifiable and measurable tasks that fulfil these goals. Then, we need to identify which attributes influence the achievement of these tasks, and the tradeoffs between these attributes. Finally, we need to decide on metrics that measure improvement on these attributes, and design experiments to evaluate these metrics.

We focus here on the selection problem, where there are initially a large set of alternatives, or candidates, and we need to pick one alternative to be used in the actual application. To achieve this, we begin with a large set of candidates and filter them using offline tests, which are conducted using simulations of interactions of users with the recommendation system, typically computed using previous interactions of users with the application without the recommendation system. Then, the best candidates can be further screened using user studies, where groups of users evaluate several alternatives in a within-subject setting and answer questions in a controlled environment. Finally, the most promising candidates are tested online within the real application in a between-subjects setting, and the best candidate is identified.

References

[1] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters.* Wiley, New York, 1978.

[2] John S. Breese, David Heckerman, and Carl Myers Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *UAI*, pages 43–52, 1998.

[3] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *J. Mach. Learn. Res.*, 7:1–30, 2006.

[4] Christian Desrosiers and George Karypis. A comprehensive survey of neighborhood-based recommendation methods. In Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors, *Recommender Systems Handbook*, pages 107–144. Springer, 2011.

[5] N. Good, J. B. Schafer, J. Konstan, A. Borchers, B. Sarwar, J. Herlocker, and J. Riedl. Combining collaborative filtering with personal agents for better recommendations. In *1999 Conference of the American Association of Artificial Intelligence (AAAI)*, pages 439–446, 1999.

[6] Asela Gunawardana and Guy Shani. A survey of accuracy evaluation metrics of recommendation tasks. *Journal of Machine Learning Research*, 10:2935–2962, 2009.

[7] Rong Hu and Pearl Pu. A comparative user study on rating vs. personality quiz based preference elicitation methods. In *IUI 0́9: Proceedings of the 13th international conference on Intelligent user interfaces*, pages 367–372, New York, NY, USA, 2009. ACM.

[8] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M. Henne. Controlled experiments on the web: survey and practical guide. *Data Min. Knowl. Discov.*, 18(1):140–181, 2009.

[9] Yehuda Koren. Factor in the neighbors: Scalable and accurate collaborative filtering. *TKDD*, 4(1), 2010.

[10] Yehuda Koren and Robert M. Bell. Advances in collaborative filtering. In Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors, *Recommender Systems Handbook*, pages 145–186. Springer, 2011.

[11] Yehuda Koren, Robert M. Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.

[12] E. L. Lehmann and Joseph P. Romano. *Testing statistical hypotheses*. Springer Texts in Statistics. Springer, New York, third edition, 2005.

[13] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

[14] Tariq Mahmood and Francesco Ricci. Learning and adaptivity in interactive recommender systems. In *ICEC '07: Proceedings of the ninth international conference on Electronic commerce*, pages 75–84, New York, NY, USA, 2007. ACM.

[15] Benjamin M. Marlin. Modeling user rating profiles for collaborative filtering. In *NIPS*, 2003.

[16] Raymond J. Mooney and Loriene Roy. Content-based book recommending using learning for text categorization. In *Proceedings of the fifth ACM conference on Digital libraries*, DL '00, pages 195–204, New York, NY, USA, 2000. ACM.

[17] Michael J. Pazzani and Daniel Billsus. Content-based recommendation systems. In *The Adaptive Web*, pages 325–341, 2007.

[18] Paul Resnick and Hal R. Varian. Recommender systems. *Commun. ACM*, 40(3):56–58, 1997.

[19] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, WWW '01, pages 285–295, New York, NY, USA, 2001. ACM.

[20] J. Ben Schafer, Dan Frankowski, Jonathan L. Herlocker, and Shilad Sen. Collaborative filtering recommender systems. In *The Adaptive Web*, pages 291–324, 2007.

[21] Guy Shani, David Maxwell Chickering, and Christopher Meek. Mining recommendations from the web. In *RecSys '08: Proceedings of the 2008 ACM Conference on Recommender Systems*, pages 35–42, 2008.

[22] Guy Shani and Asela Gunawardana. Evaluating recommendation systems. In *Recommender Systems Handbook*, pages 257–297. Springer, 2011.