1

Introduction

Recommender systems (or recommendation systems) are computer programs that recommend the "best" items to users in different contexts. The notion of a best match is typically obtained by optimizing for objectives like total clicks, total revenue, and total sales. Such systems are ubiquitous on the web and form an integral part of our daily lives. Examples include product recommendations to users on an e-commerce site to maximize sales; content recommendations to users visiting a news site to maximize total clicks; movie recommendations to maximize user engagement and increase subscriptions; or job recommendations on a professional network site to maximize job applications. Input to these algorithms typically consists of information about users, items, contexts, and feedback that is obtained when users interact with items.

Figure 1.1 shows an example of a typical web application that is powered by a recommender system. A user uses a web browser to visit a web page. The browser then submits an HTTP request to the web server that hosts the page. To serve recommendations on the page (e.g., popular news stories on a news portal page), the web server makes a call to a recommendation service that retrieves a set of items and renders them on the web page. Such a service typically performs a large number of different types of computations to select the best items. These computations are often a hybrid of both offline and real-time computations, but they must adhere to strict efficiency requirements to ensure quick page load time (typically hundreds of milliseconds). Once the page loads, the user may interact with items through actions like clicks, likes, or shares. Data obtained through such interactions provide a feedback loop to update the parameters of the underlying recommendation algorithm and to improve the performance of the algorithm for future user visits. The frequency of such parameter updates depends on the application. For instance, if items are time sensitive or ephemeral, as in the case of news recommendations, parameter updates must be done frequently (e.g., every few minutes). For

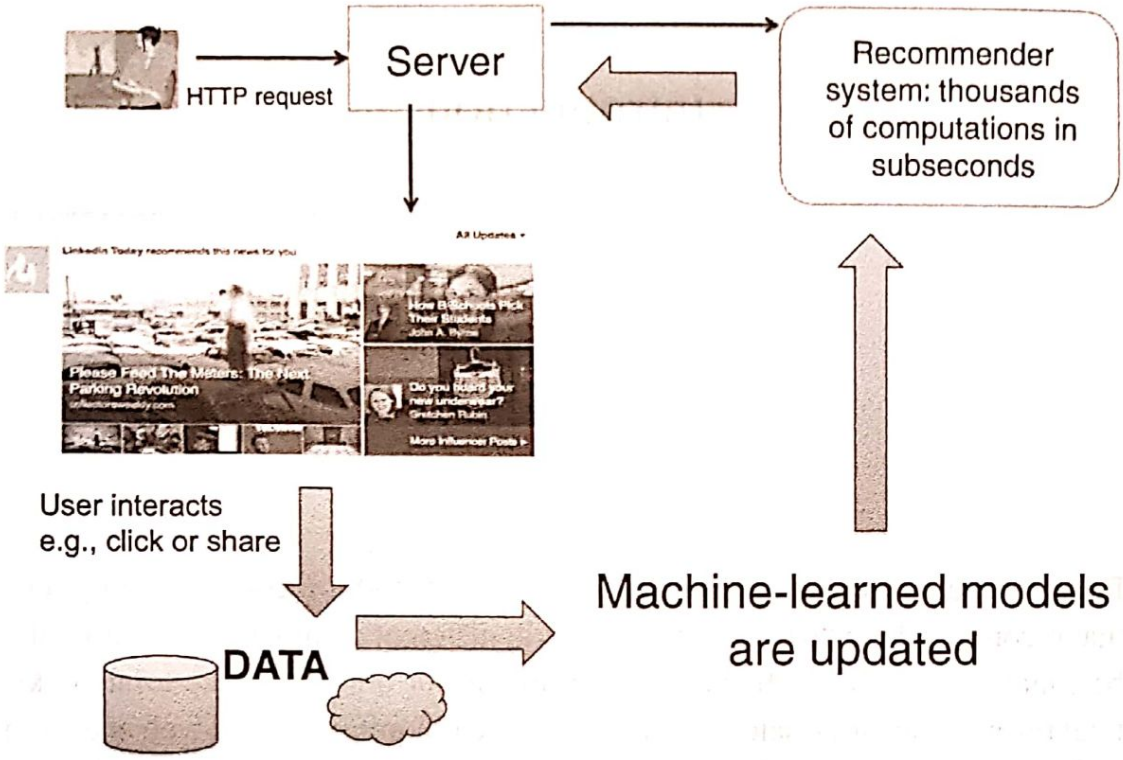

Figure 1.1. A typical recommender system.

other applications where items have a relatively longer lifetime (e.g., movie recommendations), parameter updates can happen less frequently (e.g., daily) without significant degradation in overall performance.

Algorithms that facilitate selection of best items are crucial to the success of recommender systems. This book provides a comprehensive description of statistical and machine learning methods on which we believe such algorithms should be based. For the sake of simplicity, we loosely refer to these algorithms as recommender systems throughout this book, but note that they only represent one component (albeit a crucial one) of the end-to-end process required to serve items to users in a scalable fashion.

1.1 Overview of Recommender Systems for Web Applications

Before developing a recommender system, it is important to consider the following questions.

- *What input signals are available?* When building machine-learned models of what items a user is likely to interact with in a given context, we can draw on many signals, including the content and source of each item; a

user's interest profile (reflecting both long-term interests based on prior visits and short-term interests as reflected in the current session); a user's declared information, such as demographics; and "popularity" indicators such as observed *click-through rates* or CTRs (the fraction of time in which the item is clicked on when a link to it is presented to users) and extent of social sharing (e.g., the number of times the item is tweeted, shared, or liked).

- *What objective(s) to optimize for?* There are many objectives a website could choose to optimize for, including near-term objectives, such as clicks, revenue, or positive explicit ratings by users, and long-term metrics, such as increased time spent on the site, higher return and user retention rates, increase in social actions, or increase in subscriptions.

Different recommendation algorithms need to be developed based on the answers to these questions.

1.1.1 Algorithmic Techniques

In general, a recommender system needs algorithmic techniques to address the following four tasks:

- *Content filtering and understanding*. We need to have sound techniques to filter out low-quality content from the *item pool* (i.e., the set of candidate items). Recommending low-quality content hurts user experience and the brand image of the website. The definition of low quality depends on the application. For a news recommendation problem, salacious content could be considered low quality by reputed publishers. An e-commerce site may not sell items from certain sellers with a low reputation rating. Defining and flagging low-quality content is typically a complex process that is addressed through a combination of methods, such as editorial labeling, crowdsourcing, and machine learning methods like classification. In addition to filtering low-quality content, it is important to analyze and understand the content of items that pass the quality bar. Creating item profiles (e.g., feature vectors) that capture the content with high fidelity is an effective approach. Features can be constructed using a variety of approaches, such as bag-of-words, phrase extraction, entity extraction, and topic extraction.
- *User profile modeling*. We also need to create user profiles that reflect the items that the users are likely to consume. These profiles could be based on demographics, user identity information submitted at the time of registration, social network information, or behavioral information about users.

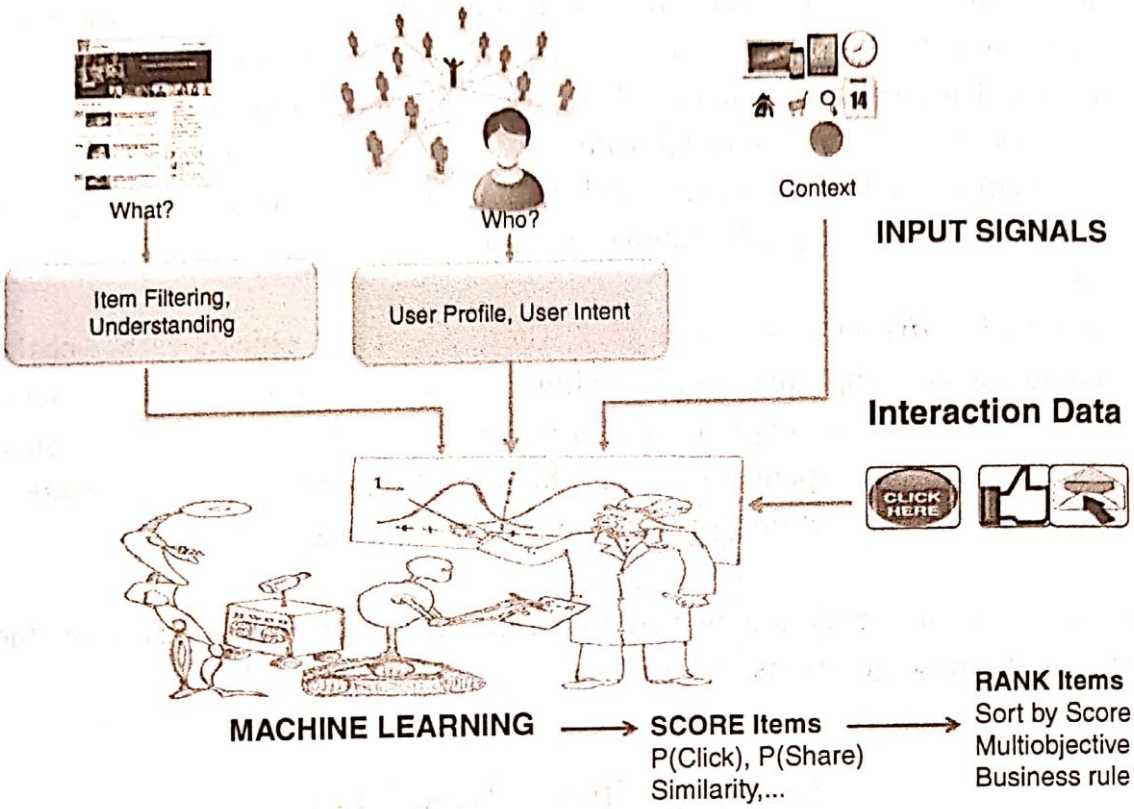

Figure 1.2. Overview of recommender system.

- *Scoring*. On the basis of user and item profiles, a scoring function needs to be designed to estimate the likely future "value" (e.g., CTRs, semantic relevance to the user's current goal, or expected revenue) of showing an item to a user in a given context (e.g., the page the user is viewing, the device being used, and the current location).
- *Ranking*. Finally, we need a mechanism to select a ranked list of items to recommend so as to maximize the expected value of the chosen objective function. In the simplest scenario, ranking may consist of sorting items based on a single score, such as the CTR of each item. However, in practice, ranking is more involved and is a blend of different considerations, such as semantic relevance, scores quantifying various utility measures, or diversity and business rules to ensure good user experience and preserve the brand image.

Figure 1.2 illustrates how the previously described algorithmic components are related. Input signals based on user information, item information, and historical user-item interaction data are used by machine-learned statistical models to produce scores that quantify users' affinity to items. The scores are combined by the ranking module to produce a sorted list of items based on descending order of priority obtained by considering single or multiple objectives.

Content filtering and understanding techniques depend to a large extent on the types of items to be recommended. For example, techniques for processing text are quite different from those used to process images. We do not intend to cover all such techniques, but we provide a brief review in Chapter 2. We also do not intend to cover a large variety of techniques for generating user profiles. However, we describe techniques that automatically "learn" both user and item profiles from historical user-item interaction data and that can also incorporate any existing profile information produced by some other existing techniques in a seamless fashion.

1.1.2 Metrics to Optimize

Among the considerations important to determining appropriate solutions for web recommendation problems, the first and foremost is to ascertain the metric(s) we want to optimize. In many applications, there is a single metric to optimize, for example, maximizing the total clicks or total revenue or total sales in a given time period. However, some applications may require simultaneous optimization of multiple metrics, for example, maximizing total clicks on content links subject to constraints on downstream engagement. An example constraint could be to ensure that the number of *bounce clicks* (clicks that do not materialize into a read) is less than some threshold. We may also want to balance other considerations, such as diversity (ensuring a user sees a range of topics over time) and serendipity (ensuring that we do not overfit our recommendations to the user, thereby limiting the discovery of new interests) to optimize long-term user experience.

Given the definition of metrics to optimize, the second consideration is to define scores that serve as the input to the optimization problem. For instance, if the goal is to maximize the total clicks, CTR is a good measure of the value of an item to a user. In the case of multiple objectives, one may have to use multiple scores, such as CTR and expected time spent. Statistical methods that can estimate the scores in a reliable fashion have to be developed. This is a nontrivial task that requires careful consideration. Once score estimates are available, they are combined in the ranking module based on the optimization problem under consideration.

1.1.3 The Explore-Exploit Trade-off

Reliably estimating scores is a fundamental statistical challenge in recommender systems. This often involves estimating expected rates of some positive response, such as click rate, explicit rating, share rate (probability of sharing an

item), or like rate (probability of clicking the "like" button associated with an item). The expected response rates can be weighted according to the utility (or value) of each possible response. This provides a principled approach for ranking items based on expected utility. Response rates (appropriately weighted) are the primary scoring functions we consider in this book.

To accurately estimate response rates for each candidate item, we could *explore* each item by displaying it to some number of user visits to collect response data on all items in a timely manner. Then, we can *exploit* the items with high response rate estimates to optimize our objectives. However, exploration has an opportunity cost of not showing items that are empirically better (based on the data collected so far); balancing these two aspects constitutes the explore-exploit trade-off.

Explore-exploit is one of the main themes of this book. We provide an introduction in Chapter 3 and discuss the technical details in Chapter 6. The methods described in Chapters 7 and 8 are also developed to address this issue.

1.1.4 Evaluation of Recommender Systems

To understand whether a recommender system achieves its objectives, it is important to evaluate its performance at different stages during the development cycle. From the perspective of evaluation, we divide the development of a recommendation algorithm into two phases:

- *Predeployment phase* includes steps before the algorithm is deployed online to serve some fraction of user visits to the website. During this phase, we use past data to evaluate the performance of the algorithm. The evaluation is limited because it is *offline*; users' responses to items recommended by the algorithm are not available.
- *Postdeployment phase* starts when we deploy the algorithm *online* to serve users. It consists primarily of online bucket tests (also called A/B experiments) to measure appropriate metrics. Although this is far more close to reality, there is a cost to running such tests. A typical approach is to filter out algorithms with poor performance based on offline evaluation in the predeployment stage.

Different evaluation methods are used to evaluate various components of a recommender system:

- *Evaluation of scoring*. Scoring is usually done through statistical methods that predict how a user would respond to an item. Prediction accuracy is often used to measure the performance of such statistical methods. For example,

if a statistical method is used to predict the numeric rating that a user would give to an item, we can use the absolute difference between the predicted rating and the true rating, averaged across users, to measure the error of the statistical method. The inverse of error is accuracy. Other ways of measuring accuracy are described in Section 4.1.2.

- *Evaluation of ranking*. The goal of ranking is to optimize for the objectives of a recommender system. In the postdeployment stage, we can evaluate a recommendation algorithm by directly computing the metrics of interest (e.g., CTR and time spent on recommended items) using data collected from online experiments. In Section 4.2, we discuss how to set up the experiments and analyze the results properly. However, in the predevelopment stage, we do not have data from users served by the algorithm – estimating the performance of the algorithm offline to mimic its online behavior is challenging. In Sections 4.3 and 4.4, we describe a couple of approaches to addressing this challenge.

1.1.5 Recommendation and Search: Push versus Pull

To set the scope of this book, we note that user intent is an important factor that differentiates various web applications. If the intent of a user is explicit and strong (e.g., query in web search), the problem of finding or "recommending" items that match the user's intent can be solved through a *pull* model – by retrieving items that are relevant to the explicit information needs of the user. However, in many recommendation scenarios, such explicit intent information is not available; at best, it can be inferred to some extent. In such cases, it is typical to follow the *push* model, where the system pushes information to the user – the goal is to serve items that are likely to engage the user.

Actual recommendation problems encountered in practice fall somewhere in the continuum of pull versus push. For instance, recommending news articles on a web portal is predominantly through a push model because explicit user intent is generally unavailable. Once the user starts reading an article, the system can recommend news stories related to the topic of the article that the user is reading, which provides some explicit intent information. Such a related news recommender system is usually based on a mix of pull and push models; we retrieve articles that are topically related to the article that the user is currently reading and then rank them to maximize user engagement.

We do not focus much on applications, such as web search, that require mostly a pull model and rely heavily on methods that estimate semantic similarity between a query and an item. Our focus is more on applications where

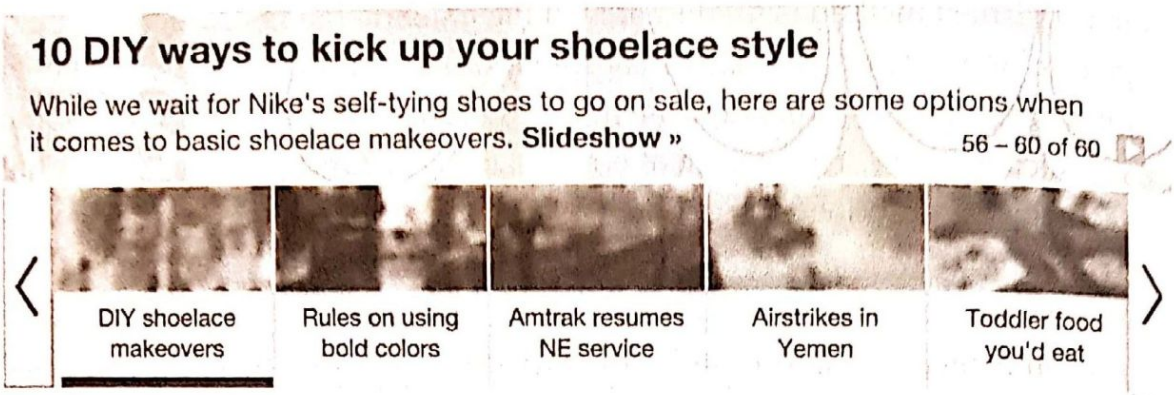

Figure 1.3. Today Module on the Yahoo! front page.

user intent is relatively weak and it is important to score items for each user based on response rates estimated through previous user-item interactions.

1.2 A Simple Scoring Model: Most-Popular Recommendation

To illustrate the basic idea of scoring, we consider the problem of recommending the most popular item (i.e., the item with the highest CTR) on a single slot of a web page to all users to maximize the total number of clicks. Although simple, this problem of *most-popular recommendation* includes the basic ingredients of item recommendation and also provides a strong baseline for the more sophisticated techniques that we describe in later chapters. We assume the number of items in the item pool is small relative to the number of visits and clicks. We do not make any assumption on the composition of the item pool; new items may get introduced and old ones may disappear over time.

Our example application is recommending new stories on the Today Module of Yahoo! front page (Figure 1.3 shows a snapshot). This application is used throughout the book for the purposes of illustration. The module is a panel with several slots, where each slot displays an item (i.e., story) selected from an item pool consisting of several items that are created through editorial oversight. For simplicity and ease of exposition, we focus on maximizing clicks on the single most prominent slot of the module, which gets a large fraction of the clicks.

Let p_{it} denote the instantaneous CTR of item i at time t. If we knew p_{it} for each candidate item i, we could simply serve the item with the highest instantaneous CTR to all user visits that occur at the given time point t. In other words, we select item $i_t^* = \arg\max_i p_{it}$ for visits at time t. However, instantaneous CTRs are not known; they have to be estimated from data. Let $\hat{p}_{it}$ denote the estimated CTR from data. Is it enough to serve the item with the highest estimated

instantaneous CTR? Mathematically, is $\hat{i}_t^* = \arg\max_i \hat{p}_{it}$ a good approximation to i_t^*? This is clearly not always true because the statistical variance in the estimates varies across items. For instance, suppose there are two items and $\hat{p}_{1t} \sim \mathcal{D}$(mean=0.01, var=.005), $\hat{p}_{2t} \sim \mathcal{D}$(mean=0.015, var=.001), where $\mathcal{D}$ denotes some probability distribution that is approximately normal. Then, $P(\hat{p}_{1t} > \hat{p}_{2t}) = .47$, that is, there is a 47 percent chance of selecting the first item, although it is inferior to the second one. This happens because the variance in estimating the CTR of the first item is significantly larger than the variance in estimating the second because of the smaller sample size. Thus, it is clear that the naive scheme of selecting the item with the highest estimated CTR is likely to produce false positives (not selecting the true best item) in practice. Are there other schemes that can reduce such false positives on average? The answer is in the affirmative: there exist several such schemes (also called explore-exploit schemes) that perform better than the greedy scheme of selecting the item with the highest estimated CTR, especially in scenarios where CTR estimates have significant statistical variance when first introduced into the item pool.

The simplest explore-exploit scheme is to allocate a small fraction of visits chosen at random to a *randomized* serving scheme, which serves each item in the item pool with equal probability (1 / total number of items) for visits in this fraction. We refer to this fraction of visits as the *random bucket* of visits. Data collected from such a random bucket are used to estimate instantaneous CTRs of items, and visits outside the random bucket are served with the item that has the highest estimated CTR. We refer to the fraction of visits outside the random bucket as the *serving bucket*. The main idea here is to estimate the CTRs through a randomized design, which smooths out the sample size and disparity in variance across items. The random bucket also avoids the item "starvation" problem because every item in the item pool is guaranteed to receive a reasonable sample on a continuous basis. Instantaneous CTRs can be estimated using data in the random bucket through time series methods like moving averages and dynamic state-space models.

Figure 1.4 shows the CTR curves of items in the Today Module for a period of two days (the curves have been smoothed). Each curve represents the CTR of an item over time, which is estimated based on data collected from the random bucket. As is evident from the figure, the CTR of each item changes over time, and the lifetimes of items are usually short (from a few hours to a day). It is thus imperative to continuously update the CTR estimates of each item to adapt to changing CTR trends by giving more weight to recent data. A simple state-space model that can perform such smoothing for a given item entails running an exponentially weighted moving average (EWMA) separately for

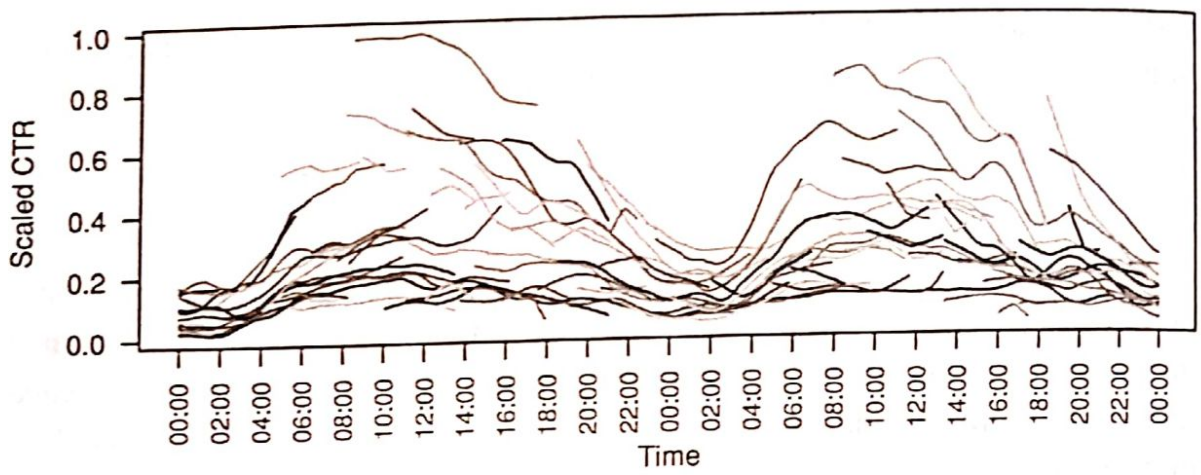

Figure 1.4. CTR curves of items in the Yahoo! Today Module over a period of two days. The y-axis is linearly scaled to conceal the actual CTR numbers.

clicks and views;[1] the ratio estimator provides the instantaneous item CTR. More specifically, we update the estimates after every ten-minute interval for this application. We chose ten minutes as our interval size because it takes time to get data from web servers to distributed computing clusters for offline processing. In general, keeping the interval size small ensures faster reaction to changes but requires more infrastructure cost. But reducing the interval size also reduces the sample size available per interval and may not give bang for the buck for the additional infrastructure investment. In our experience, we have not found significant gains in choosing interval sizes that are smaller than the interval size that would receive five clicks per item on average.

If the EWMA estimators for clicks and views for item i at the end of the tth interval are denoted by α_{it} and γ_{it}, respectively, they are given by

$$\begin{aligned} \alpha_{it} &= c_{it} + \delta\alpha_{i,t-1} \\ \gamma_{it} &= n_{it} + \delta\gamma_{i,t-1}, \end{aligned} \tag{1.1}$$

where c_{it} and n_{it} denote the number of clicks and views in the tth interval for item i, respectively, and $\delta \in [0, 1]$ is the EWMA smoothing parameter that can be selected via cross-validation to minimize predictive accuracy (typically, $[.9, 1]$ is an appropriate range to search over). Note that $\delta = 1$ corresponds to an estimator that uses cumulative CTR with no time decay (i.e., the importance of the observed clicks and views does not decay over time), and $\delta = 0$ uses only the data in the current interval. We could interpret α and γ as pseudo-clicks and pseudo-views associated with an item. To complete the specification, we initialize $\alpha_{i0} = p_{i0}\gamma_{i0}$, where p_{i0} is an initial estimate of item i's CTR and γ_{i0} is the number of pseudo-views, which reflects the degree of confidence in the initial CTR estimate. Typically, in the absence of other information, we could

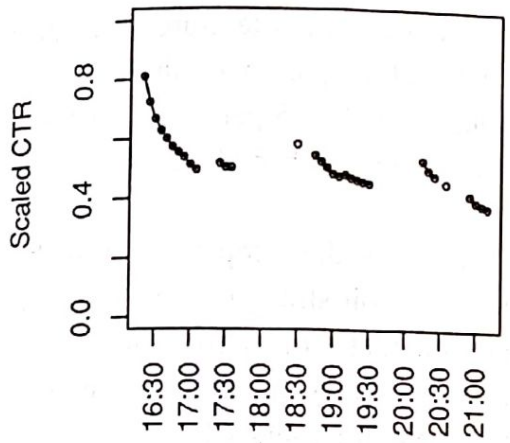

Figure 1.5. CTR of an item decays as a result of repeatedly showing the item.

set p_{i0} to some global system average and γ_{i0} to a small number like unity (or set γ_{i0} such that $p_{i0}\gamma_{i0}$ is unity).

Another important aspect that has to be kept in mind is the impact of repeatedly showing the same item to a given user. This is more likely to happen in the serving bucket and for users who visit more frequently. Figure 1.5 shows the relative CTR decay when the same item is shown to a user multiple times despite no prior clicks by the user. There is a significant drop in CTR of an item when it is shown to the same user multiple times. Hence the most-popular algorithm to display items in the serving bucket has to be modified to "discount" item CTRs based on the number of previous views by a user. In other words, the CTR of item i at time t for a user u is given by $\hat{p}_{it} f(v_{iu})$, where $\hat{p}_{it}$ is the estimated instantaneous CTR of item i using data in the random bucket and $f(v_{iu})$ is the discount applied based on the number of previous views v_{iu} on item i by user u. Typically, $f(v)$ can be estimated empirically for each value of v; one could also fit a parametric function like exponential decay.

We summarize our simple most-popular algorithm to maximize CTR as follows:

1. Create a small random bucket that shows an item at random to each visit.
2. Estimate the instantaneous CTR of each item through classical time series models using data in the random bucket. The CTRs are updated by collecting data in time intervals of some duration (e.g., ten minutes for time-sensitive applications).
3. Calculate the discounting factor to discourage repeated views of an item for each user based on prior exposure to the item by the user.
4. For each visit in the serving bucket, serve the item with the highest discounted CTR.

The scheme of using a random bucket to explore every candidate item and recommending the items with the highest estimated CTRs in the serving bucket

is also called the ϵ-greedy explore-exploit scheme in the literature. The value of ϵ (i.e., the size of the random bucket) is a tweaking parameter that has to be chosen empirically. In practice, we have found that 1 to 5 percent works well when the number of clicks is large relative to the number of items, but it is application dependent.

When the number of items becomes large and/or the sample size available per item shrinks, more parsimonious explore-exploit strategies than ϵ-greedy are necessary. Although most-popular recommendation is a good first step in practice for several applications, popularity may vary by user segments. It is also desirable to personalize recommendations for users who visit the site frequently. The main challenge in dealing with these issues is data sparseness. Solutions to these issues are the focus of this book.

Exercises

1. Think of your favorite product that is powered by recommendation algorithms. How would you formulate the problem if you were to build this system from scratch using the framework described in this chapter?
2. Write down the probabilistic state-space model that provides the solution in Exercise 1.1. Also derive the variance of the estimate using the probabilistic model. Then derive the expression for the probability of an item being the best among a pool of K items. Can we use this expression to create a better explore-exploit scheme?

2

Classical Methods

A recommender system seeks to recommend a small set of items for each user visit to optimize one or more objectives in a given context. In this chapter, we review classical methods for making such recommendations.

In general, recommendations are made based on

- what we know about the items (Section 2.1)
- what we know about the user (Section 2.2)
- how users interacted with items (e.g., clicked or not) in the past

We represent information about a user i as a vector of user *features* and information about an item j as a vector of item features (examples of such feature vectors are given later). In this chapter, we review classical methods to compute a score $\hat{y}_{ij}$ for each (user i, item j) pair. The scores $\hat{y}_{ij}$ are usually computed in one of the following ways:

- similarity between feature vectors of user i and item j (Section 2.3)
- user i's past responses to items that are "similar" to item j and/or how other users "similar" to user i responded to item j in the past (Section 2.4)
- a combination of both of the preceding (Section 2.5)

Some methods leverage predefined similarity measures, whereas others seek to learn the similarity measure directly from data.

Classical methods serve as strong baselines for the more sophisticated methods described in later chapters. In this book, we only provide an overview of classical methods. For more details on classical methods, see Adomavicius and Tuzhilin (2005), Jannach et al. (2010), and Ricci et al. (2011).

Notation. We use i and j to denote a user and an item, respectively. Let y_{ij} denote the *rating* that user i gives to item j. Here we use the term *rating* in a general sense: it can be an explicit rating, such as a 5-star rating (on a movie,

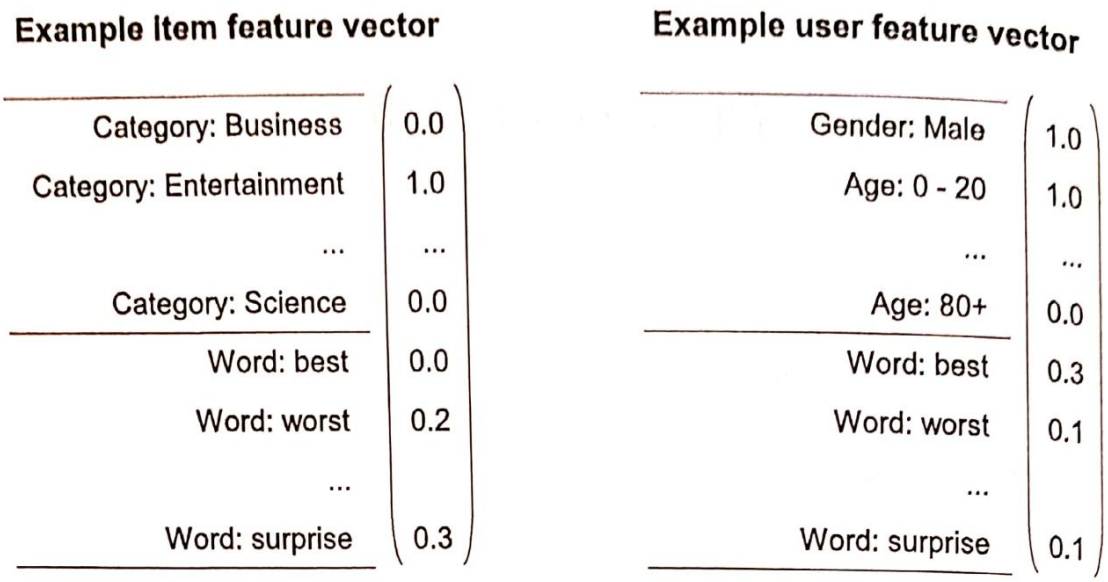

Figure 2.1. Example feature vectors. The meaning of each dimension is noted to the left of the vector.

a book, or a product), or an implicit rating, such as a click (on a recommended item). We use $Y = \{y_{ij} : \text{user } i \text{ rated item } j\}$ to denote a set of observed ratings and call it a *rating matrix*. Notice that many entries of this rating matrix are unobserved. We let $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ denote the vector of features of user i and the vector of features of item j, respectively. Although we use symbol $\boldsymbol{x}$ to denote both user and item feature vectors, it is important to note that $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ may consist of different features and may have different dimensionality. Figure 2.1 shows an example of an item feature vector $\boldsymbol{x}_j$ and an example of user feature vector $\boldsymbol{x}_i$. In the next section, we discuss how to construct such feature vectors.

2.1 Item Characterization

There are several ways to construct an item feature vector $\boldsymbol{x}_j$. The kinds of features that can be extracted depend on the nature of the item. A feature vector associated with a document can be very different from that of an image. We do not attempt to provide a detailed description of all the kinds of features that one might consider for various items. Instead, we give examples of commonly used item features for popular web recommender problems. However, as long as information about items can be represented as a vector of numbers for each item, the statistical methods described in this book can be applied.

Commonly used methods to create feature vectors for items are categorization (Section 2.1.1), bag of words (Section 2.1.2), and topic modeling (Section 2.1.3). We then discuss other item features in Section 2.1.4.

2.1.1 Categorization

In many application settings, items can be classified into a predefined taxonomy. For example, news articles on a website like Yahoo! News are usually classified into a news taxonomy that has "U.S.," "World," "Business," "Entertainment," "Sports," "Tech," "Politics," "Science," "Health," and so on, as first-level categories. Within these categories we may have subcategories. For instance, within the category "U.S.," there are subcategories "Education," "Religion," "Crimes and Trials," and so on. Similarly, product items on an e-commerce site are usually classified into a product taxonomy. For example, Amazon.com has "Books," "Movies, Music & Games," "Electronics & Computers," "Home, Garden & Tools," and so on, as first-level categories. Within the category "Books," there are subcategories such as "Arts & Photography," "Biographies & Memoirs," and "Business & Investing"; within the subcategory "Arts & Photography" are "Architecture Art & Design," "Business of Art," "Collections, Catalogs & Exhibitions," "Decorative Arts & Design," and so on; and within the sub-subcategory "Architecture Art & Design" are "Buildings," "Decoration & Ornament," "History," and "Individual Architects & Firms." An item may also belong to multiple categories with different degrees of membership.

The categorization of an item into a set of categories in the taxonomy can be done in several ways. The simplest method is, of course, through human labels. Many news websites categorize articles through an editorial process. For products, sellers usually categorize their items to facilitate discovery by potential buyers. Such a human labeling process works well with a small number of items; it is expensive for recommender problems where a large number of items are introduced at a rapid rate. When human labels are not available for all items, statistical methods can be used to automatically classify items into appropriate nodes in taxonomies. Such statistical methods learn a model from a set of labeled examples for each category using standard supervised learning methods. For a description of general supervised learning methods, see Hastie et al. (2009) and Mitchell (1997). For a survey of automated text categorization methods, see Sebastiani (2002).

Although classifying items into categories in a taxonomy is typically done to help users browse a website more efficiently to find items of interest, it may also provide useful semantic information about items. A common way of constructing a feature vector from this item-category relation is to define a vector space such that each dimension corresponds to a category. The ℓth dimension of the feature vector $\boldsymbol{x}_j$ of item j is 1 if item j belongs to the ℓth category; otherwise, it is 0. The top portion of the item feature vector in Figure 2.1 represents category features, where the example item belongs to the

"Entertainment" category. When information about the degree of membership to a category is available (usually when statistical methods are used to categorize items), the binary values (0 or 1) in the item feature vector can be replaced by these membership scores. It is also common to normalize the feature vector so that $||\boldsymbol{x}_j||_1 = 1$ or $||\boldsymbol{x}_j||_2 = 1$. However, the benefits of normalization are application – dependent and require empirical evaluation.

2.1.2 Bag of Words

For items that have associated text, the bag-of-words vector space model (Salton et al., 1975) is commonly used to construct item features. Even in applications where the primary content of an item may not be textual, there is usually some text associated with each item. For example, product items almost always have some textual descriptions; multimedia items usually have titles and sometimes textual descriptions and tags.

In the bag-of-words vector space model, a high-dimensional vector space is created by treating each word that appears in a reference *item corpus* as a dimension, where an item corpus is a large collection of typical items that may appear in the system. Each item j is represented as a point $\boldsymbol{x}_j$ in this bag-of-words vector space. We let $x_{j,\ell}$ denote the value of the ℓth dimension of vector $\boldsymbol{x}_j$. Three commonly used methods to map an item into this vector space are as follows:

1. *Unweighted version*: $x_{j,\ell} = 1$ if item j contains the ℓth word; otherwise, $x_{j,\ell} = 0$.
2. *Term frequency (TF) version*: We let $TF(j, \ell)$ denote the number of times the ℓth word appears in item j, then $x_{j,\ell} = TF\ (j, \ell)$.
3. *Term frequency – inverse document frequency (TF-IDF) version*: We let $DF(\ell) =$ the fraction of items (in a reference item corpus) that contain the ℓth word, and $IDF(\ell) = \log(1/DF(\ell))$, which represents how unique word ℓ is when it is used to describe an item. Then, $x_{j,\ell}$ is defined as $TF(j, \ell) \cdot IDF(\ell)$.

Finally, each vector $\boldsymbol{x}_j$ is normalized so that $||\boldsymbol{x}_j||_2 = 1$. In information retrieval, the inner product of a pair of normalized item vectors (which equals the cosine of the angle between the two vectors) provides a similarity measure between the two items. The bottom portion of the item feature vector in Figure 2.1 represents such bag-of-words features.

Sparse Format. Note that $\boldsymbol{x}_j$ is usually a sparse vector with zeros on a large fraction of dimensions. We can save memory by not storing all of the zeros.

Dense Format

Dimension	Vector
1	0.0
2	0.8
3	0.0
4	0.0
5	0.6
6	0.0

Sparse Format

Index	Value
2	0.8
5	0.6

Figure 2.2. Example vector in dense and sparse formats.

One way to store a sparse vector is by keeping a list of (index, value) pairs such that (i, v) is in the list if the ith dimension of the vector has value v, which is nonzero. Figure 2.2 shows an example vector in both dense and sparse formats. When reading the book, we can think of $\boldsymbol{x}_j$ as a high-dimensional vector with many zeros, with the understanding that $\boldsymbol{x}_j$ is actually stored in memory using a much smaller amount of space than the dimensionality of the vector.

Phrases and Entities. Although we refer to those features that capture textual information as "bag of words," we do not need to restrict ourselves to individual words when describing items with associated text. It is common to extend the vector space by including two-contiguous-words (bi-grams) or even three-contiguous-words (tri-grams) that correspond to phrases as additional dimensions. It is also often beneficial to focus on features corresponding to named entities such as people, organizations or locations. For a survey of named-entity recognition methods, see Nadeau and Sekine (2007).

Reducing Dimensionality. Even without considering phrases, the total number of words in an item corpus can be large, with several of them typically appearing in a small number of items. Also, it is typical for each item to contain only a small fraction of the total words in the corpus. Because the eventual goal is to use item feature vectors to estimate scores, increased dimensionality and data sparseness may add more noise relative to signal-to-score estimates. It is often useful to augment items with richer related information and/or to reduce noise by using dimensionality reduction procedures. We describe a few commonly used methods to reduce sparsity and dimensionality. These are mostly unsupervised methods that provide useful representations to be used as

inputs to various supervised score estimation tasks in general. Direct methods to obtain representations that are optimal for an application-specific supervised learning problem are considered in Chapter 7 and Section 8.1.

1. *Synonym expansion*: A simple but useful method for expanding words in an item is by adding synonyms. For example, if an item contains the word "hardworking," we can also add synonyms like "dedicated" and "diligent" to the bag of words of the item so that the corresponding feature vector would have nonzero values on the dimensions corresponding to "dedicated" and "diligent." Synonyms can be found in thesauri or lexical databases such as the WordNet by Princeton University (2010).
2. *Feature selection*: Not all words are useful. Feature selection methods (Guyon and Elisseeff, 2003) can be applied to select top-K informative words. A very simple selection method is to prune words that appear too often or too infrequently, for example, by only considering words that appear in at least N items. This simple method often reduces the dimensionality and noise in the problem.
3. *Singular value decomposition*: Another way of reducing sparsity and dimensionality is through singular value decomposition (SVD). (See Golub and Van Loan, 2013, for details). Let $\boldsymbol{X}$ denote the $m \times n$ matrix such that the jth row is the feature vector $\boldsymbol{x}_j$ of each item j, where m is the number of items and n is the number of dimensions of a feature vector. The SVD of $\boldsymbol{X}$ is

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}', \tag{2.1}$$

where $\boldsymbol{U}$ is an $m \times m$ orthogonal matrix, $\boldsymbol{V}$ is a $n \times n$ orthogonal matrix, and $\boldsymbol{\Sigma}$ is a rectangular diagonal matrix with descending nonnegative singular values on the diagonal (we have at most $\min(m, n)$ nonzero diagonal entries). To reduce dimensionality, we can choose to keep the top-d ($d \ll \min(m, n)$) projections corresponding to the largest d singular values. Let $\boldsymbol{\Sigma}_d$ denote the resulting $d \times d$ diagonal matrix obtained by retaining only the largest d singular values. Let $\boldsymbol{U}_d$ and $\boldsymbol{V}_d$ be the $m \times d$ and $n \times d$ matrices produced by keeping only the first d columns of $\boldsymbol{U}$ and $\boldsymbol{V}$, respectively. By the Eckart-Young theorem, $\boldsymbol{U}_d\boldsymbol{\Sigma}_d\boldsymbol{V}_d'$ is the *best* rank d matrix that approximates $\boldsymbol{X}$ in terms of minimizing the Frobenius norm (sum of squared element-wise differences) between the two matrices. Because $\boldsymbol{V}'$ is an orthonormal transformation (that only rotates the vector space), we can replace the original $m \times n$ feature matrix $\boldsymbol{X}$ with the new $m \times d$ feature matrix $\boldsymbol{U}_d\boldsymbol{\Sigma}_d = \boldsymbol{X}\boldsymbol{V}_d$; that is, we replace the original n-dimensional feature vector $\boldsymbol{x}_j$ for item j

with the new d-dimensional feature vector $V_d' x_j$, where V_d' projects x_j from an n-dimensional vector space into a d-dimensional one.

4. *Random projection*: Notice that the SVD method is based on a linear projection from a high-dimensional space to a low-dimensional one. A simple and sometimes effective alternative representation is to use a random linear projection (e.g., Bingham and Mannila, 2001). Let R_d be a $n \times d$ matrix with entries drawn from the standard normal distribution. Then, we use $R_d' x_j$ as our new d-dimensional feature vector for item j. The main idea is due to the Johnson-Lindenstrauss lemma that if d is sufficiently large, the Euclidean distance between two points in the original vector space is approximately preserved in the new d-dimensional vector space.

Empirical evaluations are usually required to determine which representations work best for a particular application setting. It is useful to construct multiple such representations and use them as inputs to supervised learning tasks for computing scores. New methods based on deep learning (e.g., Bengio et al., 2003) that can learn more non-linear representations is another alternative that may work well for certain applications. Because most modern-day supervised learning procedures use regularization techniques like L_2 or L_1 norm to avoid overfitting, they are effective at selecting the most informative features from a reasonable pool of input features. We defer the description of regularization to the end of Section 2.3.2.

2.1.3 Topic Modeling

In addition to hand-crafted categories, taxonomies, and the low-level bag-of-words representation of items, recent research has made good progress on unsupervised clustering of text-based items. Although several unsupervised clustering methods for documents are available, the latent Dirichlet allocation (LDA) model proposed by Blei et al. (2003) has emerged as a method of choice. Based on textual content, it assigns a membership score for each (item j, topic k) pair, which represents the probability that item j is about topic k. Note that topics here can be thought of as clusters.

The LDA model describes how each occurrence of words in each item is generated. It assumes that there are K topics in total in a corpus of items, where K is a prespecified number. Words and items are connected through these topics. Each topic is represented as a multinomial probability mass function over all of the words in the corpus. Let W denote the number of unique words in the corpus. The probability mass function of topic k can be represented as a W-dimensional vector Φ_k of nonnegative numbers on the simplex (i.e.,

the numbers in the vector sum to 1) such that the wth element in the vector represents the probability of seeing word w in an item on topic k; that is, $\Phi_{k,w} = \Pr(\text{word } w \mid \text{topic } k)$. Each item j is represented as a multinomial probability mass function over topics, which is a K-dimensional vector θ_j such that the kth element in the vector represents the probability that item j is on topic k; that is, $\theta_{j,k} = \Pr(\text{topic } k \mid \text{item } j)$. If Φ_k and θ_j are given, a word in item j can be generated by first sampling a topic k from θ_j and then, given the topic k, sampling a word from Φ_k. To complete the Bayesian specification of the model, we add conjugate Dirichlet priors to the two multinomial probability mass vectors, Φ_k and θ_j. The full specification of the generative model that prescribes how the words in each item are generated is as follows:

- For each topic k, draw a W-dimensional probability mass vector Φ_k from a Dirichlet prior with hyperparameter η.
- Then, for each item j,
  - first, draw a K-dimensional probability mass vector θ_j from a Dirichlet prior with hyper-parameter λ
  - then, for each occurrence of a word in item j,
    - draw a topic k from the probability mass vector θ_j
    - draw a word w from the probability mass vector Φ_k corresponding to the topic k just picked

The preceding process describes how each occurrence of a word in each item is generated based on the LDA model. Now, given a set of items with their associated words as the observed data, we would like to estimate the parameters of the model. The posterior distribution of θ_j for each item j and the posterior distribution of Φ_k for each topic k can be estimated through variational approximation (Blei et al., 2003) or Gibbs sampling (Griffiths and Steyvers, 2004). The posterior mean of θ_j is the Bayesian estimator of the probabilities that item j is about different topics, while the posterior mean of Φ_k helps to interpret each topic k. By looking at the top-n words with the highest probability mass in Φ_k, many studies reported meaningful topics found in various article collections. We refer interested readers to Blei et al. (2003) and Griffiths and Steyvers (2004) for a detailed exposition of LDA. In Chapter 9, we discuss how to extend the LDA model to simultaneously model users' interactions with items and the topics of items with a detailed description of a parameter estimation method based on Gibbs sampling.

2.1.4 Other Item Features

Before we move on to user features, we briefly discuss some other types of item features that are available in different recommender problems. The following

list is by no means exhaustive, and each application may have its own unique features:

- *Source*: The source of an item (e.g., author and publisher) can be a useful feature if users tend to prefer some sources more than others.
- *Location*: In some applications, items may be tagged with geographical locations. For example, pictures taken using a cell phone can be easily tagged with the location, and product items may be tagged with the locations of the stores that sell them. Such location information is important for applications of geographical interest. A location may be represented as two numbers, longitude and latitude, or as a node in a location taxonomy (with levels like Country, State/Province, Town).
- *Image features*: When items contain images or video clips, image features can provide useful information for recommendation. There is a large body of literature on this topic. See Datta et al. (2008) and Deselaers et al. (2008) for examples.
- *Audio features*: Similarly, when items contain audio clips, audio features are potentially useful. See Fu et al. (2011) and Mitrović et al. (2010) for examples.

Beyond the common ones, item features are usually application specific. Identification of good item features requires domain knowledge, experience, and insights into the application.

2.2 User Characterization

We now describe a number of methods for creating user features x_i for each user i. In general, user features can be derived from declared profiles (Section 2.2.1), users' past interaction with content (Section 2.2.2), and other user-related information available in a recommender system (Section 2.2.3).

2.2.1 Declared Profile

In many application settings, users provide basic information about themselves and sometimes even declare their interests in various topics when they sign up for services. It is common for the following user-declared features to be available in recommender systems:

- *Demographics*: As part of the registration process for a service, users are usually asked to provide their age, gender, profession, educational level, location, and other demographic information. Although some users do not

provide all demographic information, many users do. It is likely that, for example, users with different genders, in different age groups or living in different locations, have different item preferences. Thus, it is usually useful to consider demographic features in recommender systems.

- *Declared interests*: Some recommender systems allow users to declare their interests by selecting from a set of predefined categories or topics or providing a set of keywords. Although many users do not bother to declare their interests, for users who do, these declared interests may serve as important features to provide better item recommendations. For recommender systems that allow users to declare interests, one important design problem is how to make this interest elicitation process natural, effortless, and even fun.

Feature vectors of user-declared information can be constructed in a way similar to item features. Categorical features such as gender, profession, education, and interest categories can be handled in a way similar to that described in Section 2.1.1. Keyword-based features can be handled by similar methods described in Section 2.1.2. Numerical features such as age can be treated as a single dimension in the feature vector or can be discretized into a number of bins (e.g., age groups), which are then treated as categories. Figure 2.1 (right) shows an example of a user feature vector.

2.2.2 Content-Based Profile

For users who have interacted with items in the past, one way of creating a feature vector for a given user is by aggregating the feature vector of items with which the user has interacted (A similar strategy can also be used to construct additional item features.) Let $\mathcal{J}_i$ denote the set of items that user i interacted with in the past. The kind of interaction depends on the application setting and may consist of clicking, sharing, reading (for article recommendation), buying (for product recommendation), or authoring (in systems where regular users also produce content items, e.g., comments). Recall that $\boldsymbol{x}_j$ denotes the feature vector of item j. The content-based feature vector of user i is then given by

$$\boldsymbol{x}_i = F(\{\boldsymbol{x}_j : j \in \mathcal{J}_i\}), \tag{2.2}$$

where F is an aggregator function that takes a set of vectors and returns a vector. One common choice of F is the average function. A simple extension is to make F a weighted average by giving more weight to items with which the user has interacted recently.

The set $\mathcal{J}_i$ of items with which user i interacted in the past can include items that are not candidates for recommendation. For example, a movie

recommender system can use movie-related news articles that a user clicked on, read, and/or made comments on to construct content-based user features for the user, even though the recommender system does not recommend news articles or comments to users.

2.2.3 Other User Features

In addition to user-declared information and content-based user features, a recommender system may also have access to other kinds of user features. We list a few examples:

- *Current location*: Users may not always be at the declared locations that they provided when they signed up for a service. The current location of the user can usually be inferred based on the IP (Internet Protocol) address of the user's device or exactly determined if the user's device is equipped with GPS. This user location information is important when recommending geosensitive items such as stores and restaurants, for example, in an iPhone app used in a car.
- *Usage-based features*: Statistics on how a user interacts with a website (for which we are designing the recommender system) also provide a useful characterization of the user. Examples include the frequency of user visits to the website (e.g., number of times per month) and the frequency with which the user uses different devices, services, applications, or components of the website.
- *Search history*: If a recommender system is designed for a website that has a search capability, the search history of a user provides valuable information about the user's interests and recent intent. For example, if a user recently searched for an organization, then news articles about the organization may be of interest to the user. Usually the search history of a user can be represented using a bag-of-words vector in a way similar to that described in Section 2.1.2.
- *Item set*: A set of items in which a user showed interest (e.g., clicked, shared, liked) in the past can also be used to construct useful features. A feature vector can be constructed in a way that is similar to a bag-of-words approach whereby each item can be interpreted as "word."

2.3 Feature-Based Methods

Given user features and item features, one common scoring approach is to score items for a given user by designing a scoring function $s(x_i, x_j)$ that measures

the affinity between user i and item j based on their feature vectors. Then, recommendations can be made to a user by ranking items based on their scores. The scoring function can be obtained by using either unsupervised methods (Section 2.3.1) or supervised methods (Section 2.3.2).

In general, finding good features is an elaborate task, but when done properly, it leads to significant improvements in performance. In Section 2.4.3, we also describe methods that automatically "learn" feature vectors for users and items based on past user-item interaction data. We shall call such "learned" features *factors* to distinguish them from the regular features generated without supervised learning.

2.3.1 Unsupervised Methods

Scoring functions for unsupervised methods are typically based on some similarity measure between a user feature vector $\boldsymbol{x}_i$ and an item feature vector $\boldsymbol{x}_j$. There are several ways to measure similarity between two vectors. We start with a simple setting where $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are points in the same vector space; that is, both users and items are represented using the same set of features. A common example where this occurs is when both users and items are represented as bags of words from the same corpus. When items have associated text, it is natural to represent them as bags of words (as described in Section 2.1.2). The bag-of-words feature representation for a user in this case can be obtained from a content-based profile (see Section 2.2.2). Cosine similarity is a common choice of the scoring function:

$$s(\boldsymbol{x}_i, \boldsymbol{x}_j) = \frac{\boldsymbol{x}_i' \boldsymbol{x}_j}{||\boldsymbol{x}_i|| \cdot ||\boldsymbol{x}_j||}, \tag{2.3}$$

where $\boldsymbol{x}_i'$ is the transpose of column vector $\boldsymbol{x}_i$ and $\boldsymbol{x}_i' \boldsymbol{x}_j$ is the inner product of the two vectors. Other commonly used similarity functions for bag-of-words features include Okapi BM25 (Robertson et al., 1995), which is a popular TF-IDF-based similarity function used in information retrieval. For binary features, Jaccard similarity (Jaccard, 1901) is often used. The Jaccard similarity between two sets is the size of the intersection divided by the size of the union.

When $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are normalized such that $||\boldsymbol{x}_i||_2 = ||\boldsymbol{x}_j||_2 = 1$, the inner product $\boldsymbol{x}_i' \boldsymbol{x}_j = \sum_k x_{i,k} x_{j,k}$ is the cosine similarity between the two vectors, where $x_{i,k}$ and $x_{j,k}$ are the values on the kth dimension of vectors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. A simple extension is to associate different weights to dimensions; that is, $s(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i' \boldsymbol{A} \boldsymbol{x}_j = \sum_k a_{kk} x_{i,k} x_{j,k}$, where $\boldsymbol{A}$ is a diagonal matrix and a_{kk}, the value of entry (k, k) of matrix $\boldsymbol{A}$, is the weight for the kth dimension. A

$$\boldsymbol{x}_i = \begin{pmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \end{pmatrix} \qquad \boldsymbol{x}_j = \begin{pmatrix} x_{j1} \\ x_{j2} \end{pmatrix} \qquad A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{33} \end{pmatrix}$$

$$\boldsymbol{x}_i \boldsymbol{x}_j' = \begin{pmatrix} x_{i1}x_{j1} & x_{i1}x_{j2} \\ x_{i2}x_{j1} & x_{i2}x_{j2} \\ x_{i3}x_{j1} & x_{i3}x_{j2} \end{pmatrix} \qquad \boldsymbol{x}_{ij} = \begin{pmatrix} x_{i1}x_{j1} \\ x_{i2}x_{j1} \\ x_{i3}x_{j1} \\ x_{i1}x_{j2} \\ x_{i2}x_{j2} \\ x_{i3}x_{j2} \end{pmatrix} \qquad \boldsymbol{\beta} = \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{12} \\ a_{22} \\ a_{32} \end{pmatrix}$$

Figure 2.3. Correspondence between the bilinear form and the regular linear form: $\boldsymbol{x}_i' A \boldsymbol{x}_j = \boldsymbol{x}_{ij}' \boldsymbol{\beta}$.

further extension is to allow $\boldsymbol{A}$ to be a full matrix; that is, $s(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i' A \boldsymbol{x}_j = \sum_{k\ell} a_{k\ell} x_{i,k} x_{j,\ell}$, where $a_{k\ell}$ intuitively prescribes the similarity between the kth dimension of the user feature space and the ℓth dimension of the item feature space. With this extension, $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ can consist of different features and have different numbers of dimensions. However, how should we specify the matrix A? It can be difficult when $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are high dimensional. This provides us with motivation to use supervised methods.

For succinctness, we will mostly use x_{ij} in later chapters. Extension from x_{ij} to $x_{ij}^{(k)}$ is straightforward.

2.4 Collaborative Filtering

The ratings that a user gives to different items is usually indicative of his or her preferences. Two users who rate items similarly are likely to have similar taste in items. Based on this intuition, for a given user i, we can obtain a set of other users that are similar to i in terms of their rating behavior. The score of an item j for user i can then be obtained by using the average rating given to j by users who are similar to i. Such an approach usually predicts how much a user prefers an item based only on users' past ratings on items and does not rely on any user or item features. This approach treats item ratings by users as a collaborative process in which users help one another to identify interesting items (although they are not aware of the collaboration), thus called *collaborative filtering*.[1]

2.4.1 Methods Based on User-User Similarity

We start with collaborative filtering methods that predict the rating (or score) s_{ij} that user i would give to an unrated item j based on the ratings that other users similar to user i have given to item j. One common choice of the scoring function is the average of the ratings of the similar users; this can also be a weighted average that assigns larger weights to users who are more similar to user i.

Let $\mathcal{I}_j(i)$ denote the set of users who rated item j and are similar to user i. We discuss how to construct this set later in this section. Let $w(i, \ell)$ denote the weight we assign to the rating of user ℓ when it is used to predict how user i would rate item j. Let $\bar{y}_{i\cdot}$ denote the average rating of user i. Given these notations, the predicted rating s_{ij} that user i would give item j is

$$s_{ij} = \bar{y}_{i\cdot} + \frac{\sum_{\ell \in \mathcal{I}_j(i)} w(i, \ell)\,(y_{\ell j} - \bar{y}_{\ell\cdot})}{\sum_{\ell \in \mathcal{I}_j(i)} |w(i, \ell)|}, \tag{2.17}$$

which averages over the "centered" ratings to mitigate users' individual rating bias (because the same rating value can represent different degrees of satisfaction for different users). In addition to centering, we can further standardize a user's ratings by dividing the centered rating by the standard deviation of the user's ratings. See Herlocker et al. (1999) for an example.

Similarity Functions. One popular choice for the similarity function between users is the Pearson correlation as used in Resnick et al. (1994), where the similarity between user i and ℓ is defined as

$$sim(i, \ell) = \frac{\sum_{j \in \mathcal{J}_{i\ell}} (y_{ij} - \bar{y}_{i\cdot})(y_{\ell j} - \bar{y}_{\ell\cdot})}{\sqrt{\sum_{j \in \mathcal{J}_{i\ell}} (y_{ij} - \bar{y}_{i\cdot})^2} \sqrt{\sum_{j \in \mathcal{J}_{i\ell}} (y_{\ell j} - \bar{y}_{\ell\cdot})^2}}. \tag{2.18}$$

$\mathcal{J}_{i\ell}$ denotes the set of items rated by both users i and ℓ. Notice that correlations can be negative. We may choose to set negative correlations to zero. See Desrosiers and Karypis (2011) for more similarity functions.

Neighborhood Selection. The set $\mathcal{I}_j(i)$ of similar users can be constructed in several different ways. We can choose simply to include all users who rated item j and use the similarity between users i and ℓ to define the weight $w(i, \ell)$ when predicting ratings for user i. For items rated by many users, such averaging over many users may incur a high computational cost. Other choices include selecting the top n users who are most similar to user i or selecting users whose similarity to user i passes some threshold. Empirical evaluation is typically needed to make a good choice for a given application.

Weighting. The most common weighting method is to set $w(i, \ell) = sim(i, \ell)$. When $sim(i, \ell)$ is computed based on a small set $\mathcal{J}_{i\ell}$ of items rated both by user i and ℓ, this can become unreliable due to small sample size. One way of addressing this small sample size problem is to give unreliable similarity values lower weights. For example, Herlocker et al. (1999) used the formula

$$w(i, \ell) = \min\{|\mathcal{J}_{i\ell}|/\alpha \, , \, 1\} \cdot sim(i, \ell) \tag{2.19}$$

and found that $\alpha = 50$ gave the best results on their data. When $\mathcal{I}_j(i)$ selects only the top n most similar users, we can also set $w(i, \ell) = 1$ to perform an unweighted average of the ratings from the most similar users. Again, empirical evaluation is often needed to make a good choice.

2.4.2 Methods Based on Item-Item Similarity

In Section 2.4.1, we discussed how to measure similarity between users. We can also leverage similarity between items to predict scores. In this case, we predict the ratings that user i would give to item j by averaging the user's own ratings on items that are similar to item j.

Let $\mathcal{J}_i(j)$ denote the set of items that were rated by user i and are similar to item j. Let $w(j, \ell)$ denote the weight we assign to the rating that user i gives

item ℓ when we use it to predict user i's rating on item j. Let $\bar{y}_{\cdot j}$ denote the average rating on item j. The predicted rating s_{ij} that user i would give item j is

$$s_{ij} = \bar{y}_{\cdot j} + \frac{\sum_{\ell \in \mathcal{J}_i(j)} w(j, \ell)\,(y_{i\ell} - \bar{y}_{\cdot \ell})}{\sum_{\ell \in \mathcal{J}_i(j)} |w(j, \ell)|}. \tag{2.20}$$

$\mathcal{J}_i(j)$ and $w(j, \ell)$ can be determined in a way similar to the method described in Section 2.4.1.

4

Evaluation Methods

After developing statistical methods for recommender systems, it is important to evaluate their performance to assess performance metrics in different application settings. Broadly speaking, there are two kinds of evaluation, depending on whether a recommendation algorithm, or more precisely, the model used in the algorithm, has been deployed to serve users:

1. *Predeployment offline evaluation*: A new model must show strong signs of performance improvement over existing baselines before being deployed to serve real users. To ascertain the potential of a new model before testing it on real user visits, we compute various performance measures on retrospective (historical) data. We refer to this as *offline evaluation*. To perform such offline evaluation, we need to log data that record past user-item interactions in the system. Model comparison is performed by computing various offline metrics based on such data.
2. *Postdeployment online evaluation*: Once the model performance looks promising based on offline metrics, we test it on a small fraction of real user visits. We refer to this as *online evaluation*. To perform online evaluation, it is typical to run randomized experiments online. A randomized experiment, also referred to as an A/B test or a bucket test in web applications, compares a new method to an existing baseline. It is conducted by assigning two random user or visit populations to the *treatment bucket* and the *control bucket*, respectively. The treatment bucket is typically smaller than the control because it serves users according to the new recommendation model that is being tested, whereas the control bucket serves users using the status quo. After running such a *bucket test* for a certain time period, we gauge model performance by comparing metrics that are computed using data collected from the corresponding buckets.

In this chapter, we describe several ways to measure the performance of recommendation models and discuss their strengths and weaknesses. We start

in Section 4.1 with traditional offline evaluation metrics that measure out-of-sample predictive accuracy on retrospective ratings data. Our use of the term *rating* is generic and refers to both explicit ratings like star ratings on movies and implicit ratings (also called responses) like clicks on recommended items (we use ratings and responses interchangeably). In Section 4.2, we discuss online evaluation methods, describing both performance metrics and how to set up online bucket tests in a proper way. Improving online performance of a model is our ultimate goal; however, performing online evaluation has an associated cost because a poor model could significantly impact user experience. Can we use offline evaluation to approximate online performance metrics? Unfortunately, the answer is not always affirmative. In Sections 4.3 and 4.4, we describe two offline methods that can bridge this gap in some scenarios through simulation and replay.

4.1 Traditional Offline Evaluation

For recommendation models that predict user i's rating on item j, because they are predictive models, one natural evaluation metric is to measure out-of-sample predictive accuracy for *unseen ratings*, that is, ratings on unseen or unobserved user-item pairs. For other recommendation models such as the unsupervised methods discussed in Section 2.3.1, the goal is not to predict ratings. Metrics that measure a model's ranking performance are useful for these models. In fact, such ranking metrics are useful in a much broader setting, because the goal of many recommendation problems (no matter whether predictive models or unsupervised methods are used) is to rank items according to some score – we do not necessarily have to accurately predict users' absolute ratings, as long as we can appropriately rank items on a relative scale.

Offline evaluation methods are based on users' ratings on items observed in the past. We use the term *observed ratings* to distinguish them from unseen ratings and predicted ratings. To measure a model's out-of-sample accuracy or ranking performance, we first need appropriate methods to split the observed ratings into a training set and a test set and use the ratings in the test set to "simulate" unseen ratings for various application settings. After introducing data-splitting methods, we describe commonly used accuracy metrics in Section 4.1.2 and commonly used ranking metrics in Section 4.1.3.

4.1.1 Data-Splitting Methods

In this section, we describe ways of splitting the observed ratings into two parts: a training set and a test set. All unknown model parameters must be estimated

using only data in the training set. The estimated model is used to compute test set accuracy by aggregating predicted ratings or rankings in the test set. For ease of exposition, in the rest of this section, we assume that we want to compute an accuracy metric. Applying the following data-splitting methods to computation of ranking metrics is straightforward. Because we have not yet precisely defined accuracy and ranking metrics, for now, we can think of them as functions that return a number that measures the performance of a recommendation model based on a training set and a test set.

Splitting Methods. Although splitting data into training and test sets is routinely used to assess model performance for supervised learning tasks, methods to perform this split depend on the properties of our recommendation model. In the following, we describe a number of ways to split the data, and we discuss their properties.

- *Random splitting*: We randomly select P percent of the observed ratings to create the training set and the remaining $(100 - P)$ percent observed ratings from to create the test set. Random splitting is a standard way to measure the predictive accuracy of a statistical model. However, it may not be appropriate for estimating the expected performance of a recommendation model in a real system. In particular, old ratings of a user (or an item) may be included in the test set, whereas new ratings of the user (or the item) may be in the training set. In this case, some models may end up using future ratings to predict past ratings, which does not correspond to any real application setting. However, if we only want to measure the predictive accuracy of a statistical model, then random splitting can be a suitable method. One advantage of random splitting is that we can easily perform random splitting multiple times and compute the variance of the accuracy numbers obtained from different training-test splits.
- *Time-based splitting*: If we record the time when a user rates an item, we can split the data by using observed ratings before a certain time point to form the training set and the rest to form the test set. Time-based splitting eliminates the problem of using future data to predict the past and is perhaps a better splitting method for understanding the performance of a model in a real application setting. Unlike random splitting, we cannot use time-based splitting to generate multiple training-test splits with the same percentage of training (or test) data because each splitting time point corresponds to a unique training-test split. Nonetheless, bootstrap sampling (Efron and Tibshirani, 1993) can be used to estimate the variance of model accuracy for this splitting method. For example, after creating a training-test split, we can

obtain multiple versions of training and test data by random sampling with replacement from the original training set and test set.

- *User-based splitting*: If the goal is to understand the accuracy of a model on *new users* who have not yet rated any item, we can randomly select P percent users and their ratings to form the training set, and the ratings by the remaining $(100 - P)$ percent users form the test set. User-based splitting simulates a scenario where users in the test set do not have any past ratings that a model can use. Thus, the test set accuracy measures the ability of the model to predict ratings for new users. Similar to the random splitting method, we can easily generate multiple training-test splits to assess the variance of model accuracy for this splitting method.
- *Item-based splitting*: If the goal is to understand the accuracy of a model on *new items* that have not yet received any ratings, we can split items into a set of training items and a set of test items. All the ratings on training items belong to the training set, and all the ratings on test items belong to the test set. Again, multiple training-test splits can be easily generated.

Cross-validation. When we need to generate multiple training-test splits, n-fold cross-validation is a useful method for ensuring that each observed rating is used exactly once in the computation of test set accuracy. We describe cross-validation for random splitting and note that the method applies to user-based splitting and item-based splitting in a similar manner. The n-fold cross-validation works as follows:

- We randomly split the observed ratings into n roughly equally sized partitions.
- For k from 1 to n, do the following:
  - Treat the kth partition as the test set and the union of the other $n - 1$ partitions as the training set.
  - Train the model using the training set and compute the accuracy metric using the test set.
- Average the n partition-specific accuracy numbers to obtain the final accuracy estimate. Variance or standard deviation can also be computed from the n partition-specific accuracy numbers.

Tuning Set. It is common for a model-fitting algorithm to have a few *tuning parameters* that are not estimated by the algorithm but important inputs to the algorithm. These tuning parameters often affect how the fitting algorithm determines regular model parameters. For example, regularization parameters

(e.g., λ in Equation 2.15 and λ_1 and λ_2 in Equation 2.24) are usually treated as tuning parameters. The number of latent dimensions in matrix factorization is another example. The step size (e.g., α in Equation 2.29) in the SGD method is yet another one. In general, it is not a good practice to compare many different settings of tuning parameters based on the accuracy on the test set and report the test set accuracy of the best setting. This best test set accuracy obtained after estimating the tuning parameter using the test set will always be optimistic and overestimate the actual accuracy of the model on unseen ratings. Consider the simple example of a model that predicts ratings randomly. If we run this model several times on the test set and report the best test set accuracy over all trials, the model performance will get better with increasing number of trials. Each trial here can be thought of as a tuning parameter setting that has no real effect on model performance at all. If the test set is large and the number of tuning parameter settings is small, this overestimation may not be a serious issue. Nevertheless, it is a good practice to further split the training set into two parts. We call one part the tuning set and still call the other part the training set. Now, the training set is used to determine model parameters, and the tuning set is used to select a good setting of tuning parameters. Finally, after fixing all unknown quantities obtained without using the test set, we apply the model with the best setting to the test set and measure its accuracy. The process works as follows:

- For each setting s of tuning parameters, we do the following:
  - Fit the model using tuning parameter setting s and the training set.
  - Measure the accuracy of the fitted model using the tuning set.
- Let s^* denote the best setting of tuning parameters that has the highest tuning set accuracy.
- Fit the model using tuning parameter setting s^* and the training set (or the union of the training set and tuning set).
- Measure the accuracy of the fitted model using the test set.

Cross-validation is not used in the preceding process. It is straightforward to apply cross-validation to create training-tuning splits and find the best setting of tuning parameters.

4.1.2 Accuracy Metrics

Let Ω^{test} denote the set of (user i, item j) pairs in the test set. Recall that y_{ij} denotes the observed rating that user i gives to item j and $\hat{y}_{ij}$ denotes the predicted rating by a model. Accuracy can be measured in a variety of ways.

- *Root mean squared error*: For numeric ratings, root mean squared error (RMSE) between predicted ratings and observed ratings is a frequently used accuracy metric:

$$\text{RMSE} = \sqrt{\frac{\sum_{(i,j)\in\Omega^{\text{test}}}(y_{ij} - \hat{y}_{ij})^2}{|\Omega^{\text{test}}|}}. \tag{4.1}$$

  The fact that RMSE was chosen as the accuracy metric for the Netflix contest has made it particularly popular in recent years.
- *Mean absolute error*: Another frequently used metric for numeric ratings is mean absolute error (MAE):

$$\text{MAE} = \frac{\sum_{(i,j)\in\Omega^{\text{test}}} |y_{ij} - \hat{y}_{ij}|}{|\Omega^{\text{test}}|}. \tag{4.2}$$

- *Normalized L_p norm*: MAE and RMSE are two special cases of the normalized L_p norm with $p = 1$ and 2, respectively:

$$\text{Normalized } L_p \text{ norm} = \left(\frac{\sum_{(i,j)\in\Omega^{\text{test}}} |y_{ij} - \hat{y}_{ij}|^p}{|\Omega^{\text{test}}|}\right)^{1/p}. \tag{4.3}$$

  Larger values of p put more penalty on (user, item) pairs with larger errors. At one extreme, $L_\infty = \max_{(i,j)\in\Omega^{\text{test}}} |y_{ij} - \hat{y}_{ij}|$. In practice, MAE and RMSE are widely used.
- *Log-likelihood*: For a model that predicts the probability that a user would respond to an item positively (e.g., click or not), the log-likelihood of the model on the test set (not the training set) is a useful accuracy metric. Let $\hat{p}_{ij}$ denote the predicted probability that user i responds to item j positively and $y_{ij} \in \{1, 0\}$ denote whether user i actually responds to item j positively. Then, we have

$$\begin{aligned}\text{Log-likelihood} &= \sum_{(i,j)\in\Omega^{\text{test}}} \log \Pr(y_{ij} \mid \hat{p}_{ij}) \\ &= \sum_{(i,j)\in\Omega^{\text{test}}} y_{ij} \log(\hat{p}_{ij}) + (1 - y_{ij}) \log(1 - \hat{p}_{ij}).\end{aligned} \tag{4.4}$$

Accuracy metrics are basic evaluation measures for predictive models. When comparing a new predictive model with an existing one, ensuring that the new model is more accurate than the old one on retrospective data is a good start. However, we should not rely solely on accuracy metrics because they have the following limitations:

- *Unsuitable for measuring ranking performance*: In most systems, only the top few items are recommended to users. Accuracy on items that are likely to

be ranked higher is relatively more important than accuracy on lower-ranked items. For instance, in a binary response problem, large errors in estimating low response rates may not affect the performance by much because there could be a large margin between items with low and high response rates. Classical accuracy metrics do not weight errors appropriately to take the ordering of items into consideration.

- *Difficult to translate accuracy improvements to real system performance improvements*: Being able to improve model accuracy is typically an encouraging sign. However, offline accuracy improvements do not necessarily translate to performance improvements in real online systems. For example, it is difficult to say how much we can expect a recommender system to improve if we improve a model's RMSE from 0.9 to 0.8 or how many more additional clicks we can obtain on recommended items if we improve the model's log-likelihood by 10 percent.

4.1.3 Ranking Metrics

As discussed before, recommendation problems are usually ranking problems. It is useful to evaluate a model based on how well it can rank highly rated items relative to the rest. Let s_{ij} denote the score that the model gives to a (user i, item j) pair. As long as the scores can be used to order items according to their affinities to a user, the actual score values need not be on the same scale as the ratings. As before, we use y_{ij} to denote the observed rating that user i gives to item j and split the observed ratings (more precisely, user-item pairs for which we have observed ratings) into a training set and a test set and measure ranking metrics using the test set.

We first consider *global ranking metrics*, where the entire test set is ranked according to the predicted scores from a model and we measure the extent to which user-item pairs with high ratings are ranked above those with low ratings. Then, we discuss *local ranking metrics*, where we rank items for each individual user separately and measure the extent to which highly rated items are ranked above those with lower ratings for that user, and then average across users.

Global Ranking Metrics. We start with problem settings where the observed ratings y_{ij} are binary (e.g., clicks) or can be made binary (e.g., treating rating values above a threshold as positive and the rest as negative). Let Ω_+^{test} denote the test (user i, item j) pairs with positive ratings and Ω_-^{test} denote those with negative ratings. Given a threshold θ, we call (i, j) pairs with score $s_{ij} > \theta$ the *predicted positive* pairs, and (i, j) pairs with score $s_{ij} \leq \theta$ the *predicted*

Table 4.1. *Definitions of TP, TN, FP, and FN*

$TP(\theta)$	$= \lvert\{(i,j) \in \Omega_+^{\text{test}} : s_{ij} > \theta\}\rvert$, which is the number of true positive user-item pairs
$TN(\theta)$	$= \lvert\{(i,j) \in \Omega_-^{\text{test}} : s_{ij} \leq \theta\}\rvert$, which is the number of true negative user-item pairs
$FP(\theta)$	$= \lvert\{(i,j) \in \Omega_-^{\text{test}} : s_{ij} > \theta\}\rvert$, which is the number of false positive user-item pairs
$FN(\theta)$	$= \lvert\{(i,j) \in \Omega_+^{\text{test}} : s_{ij} \leq \theta\}\rvert$, which is the number of false negative user-item pairs

negative pairs. We define $TP(\theta)$, $TN(\theta)$, $FP(\theta)$, and $FN(\theta)$ in Table 4.1. A few commonly used metrics based on these quantities are as follows:

- *Precision-recall (P-R) curve*: The P-R curve of a model is a two-dimensional curve generated by varying θ from negative infinity to positive infinity. A point on the curve for a given θ is (Recall(θ), Precision(θ)), where

$$\begin{aligned} \text{Precision}(\theta) &= \frac{TP(\theta)}{TP(\theta) + FP(\theta)} \\ \text{Recall}(\theta) &= \frac{TP(\theta)}{TP(\theta) + FN(\theta)}. \end{aligned} \tag{4.5}$$

- *Receiver operating characteristic curve*: The receiver operating characteristic (ROC) curve of a model is also a two-dimensional curve generated by varying θ from negative infinity to positive infinity. The point on the curve for a given θ is ($FPR(\theta)$, $TPR(\theta)$), where

$$\begin{aligned} TPR(\theta) &= \frac{TP(\theta)}{TP(\theta) + FN(\theta)} \quad \text{(i.e., true positive rate)} \\ FPR(\theta) &= \frac{FP(\theta)}{FP(\theta) + TN(\theta)} \quad \text{(i.e., false positive rate).} \end{aligned} \tag{4.6}$$

  It is easy to see that for a *random model* that outputs random s_{ij} scores, the ROC curve is a straight line from (0,0) to (1,1).
- *Area under the ROC curve (AUC)*: P-R curves and ROC curves are both two-dimensional plots. Sometimes it is useful to summarize the performance of a model using a single number. AUC is a commonly used metric that summarizes an ROC curve by computing the area under the curve. The range is from 0 to 1 – the higher, the better. A random model has an AUC of 0.5.

Another way to measure the ranking performance of a model is rank correlation, which compares a ranked list of test user-item pairs sorted by model scores s_{ij} to the *ground-truth* ranked list of test user-item pairs sorted by the observed ratings y_{ij}. The more similar the two ranked lists are, the better the performance of the model is. Two commonly used rank correlation metrics for measuring the similarity of two ranked lists are Spearman's ρ and Kendall's τ:

- *Spearman's* ρ is the Pearson correlation between rank values of elements in the two lists. Let s_{ij}^* and y_{ij}^* denote the ranks of (user i, item j) pair in the ranked list according to s_{ij} and the ranked list according to y_{ij}, respectively. For example, if s_{ij} is the kth highest score among all user-item pairs in the test set, then $s_{ij}^* = k$. Ties are handled in the following way. If there are n user-item pairs with the same rating (or score) value y and there are m user-item pairs with rating (or score) values $> y$, then ranks of those n user-item pairs are all equal to the average rank = $\sum_{i=m+1}^{m+n} i/n$. Let $\bar{s}^*$ and $\bar{y}^*$ denote the average of s_{ij}^* and the average of y_{ij}^* over all test (i, j) pairs, respectively. Then, we compute the Pearson correlation coefficient as follows:

$$\frac{\sum_{(i,j)\in\Omega^{\text{test}}}(s_{ij}^* - \bar{s}^*)(y_{ij}^* - \bar{y}^*)}{\sqrt{\sum_{(i,j)\in\Omega^{\text{test}}}(s_{ij}^* - \bar{s}^*)^2}\sqrt{\sum_{(i,j)\in\Omega^{\text{test}}}(y_{ij}^* - \bar{y}^*)^2}}. \tag{4.7}$$

- *Kendall's* τ measures the propensity that two user-item pairs would be ordered in the same way in the two ranked lists. Consider any two user-item pairs in the test set: (i_1, j_1) and (i_2, j_2). We define two indicator functions as follows:

$$\begin{aligned}&\text{Concordant}((i_1, j_1), (i_2, j_2))\\ &= I((s_{i_1 j_1} > s_{i_2 j_2} \text{ and } y_{i_1 j_1} > y_{i_2 j_2}) \text{ or } (s_{i_1 j_1} < s_{i_2 j_2} \text{ and } y_{i_1 j_1} < y_{i_2 j_2}))\\ &\text{Discordant}((i_1, j_1), (i_2, j_2))\\ &= I((s_{i_1 j_1} > s_{i_2 j_2} \text{ and } y_{i_1 j_1} < y_{i_2 j_2}) \text{ or } (s_{i_1 j_1} < s_{i_2 j_2} \text{ and } y_{i_1 j_1} > y_{i_2 j_2})).\end{aligned}$$

Let n_c and n_d denote the numbers of concordant and discordant pairs:

$$\begin{aligned}n_c &= \frac{1}{2}\sum_{(i_1,j_1)\in\Omega^{\text{test}}}\sum_{(i_2,j_2)\in\Omega^{\text{test}}}\text{Concordant}((i_1, j_1), (i_2, j_2))\\ n_d &= \frac{1}{2}\sum_{(i_1,j_1)\in\Omega^{\text{test}}}\sum_{(i_2,j_2)\in\Omega^{\text{test}}}\text{Discordant}((i_1, j_1), (i_2, j_2)).\end{aligned} \tag{4.8}$$

Then, Kendall's τ is defined as follows:

$$\tau = \frac{n_c - n_d}{n(n-1)/2}, \tag{4.9}$$

where $n = |\Omega^{\text{test}}|$ is the total number of test user-item pairs. Rank correlation metrics naturally handle nonbinary ratings. Several variants to deal with ties are also available (e.g., τ_b, τ_c).

Local Ranking Metrics. Let $\mathcal{J}_i^{test}$ denote the set of items rated by user i. We first compute a ranking metric for each user i based on $\mathcal{J}_i^{test}$ and then average over all users. We focus on binary ratings; otherwise, we either convert multivalue ratings to binary ones based on a threshold or compute average rank correlation over users. A number of commonly used metrics are as follows:

- *Precision at rank K (P@K)*: For each user i, we rank items in $\mathcal{J}_i^{test}$ according to their scores (from high to low) predicted by a model. P@K is the fraction of positive items among the first K items. After computing P@K for each user, we average those numbers over all users. Usually, we consider a few K values (e.g., 1, 3, 5). A good model should outperform baseline models consistently for all K values.
- *Mean average precision*: One way to summarize P@K for all K values is the average precision, which is computed as follows. As before, we rank items in $\mathcal{J}_i^{test}$ according to the predicted scores, for each user i. Average precision is defined as the average of P@K over only the rank positions K at which the items have positive ratings. Mean average precision (MAP) is then the mean of average precisions over all users.
- *Normalized discounted cumulative gain (nDCG)*: Again, for each user i, we rank items in $\mathcal{J}_i^{test}$ according to the predicted scores. Let $p_i(k) = 1$ if the item at rank position k has a positive rating by user i; otherwise, $p_i(k) = 0$. Let $n_i = |\mathcal{J}_i^{test}|$ and n_i^+ denote the number of items in $\mathcal{J}_i^{test}$ that are rated positively by user i. Then, discounted cumulative gain (DCG) is defined as follows:

$$\text{DCG}_i = p_i(1) + \sum_{k=2}^{n_i} \frac{p_i(k)}{\log_2 k}. \tag{4.10}$$

nDCG is DCG normalized by the maximal achievable DCG value for user i:

$$\text{nDCG}_i = \frac{\text{DCG}_i}{1 + \sum_{k=2}^{n_i^+} \frac{1}{\log_2 k}}, \tag{4.11}$$

where the denominator is the maximal achievable DCG value for user i and is 1 if $n_i^+ = 1$. Finally, we average nDCG_i over all users i who have at least one positive rating.

Remarks. Most ranking metrics were originally defined to measure the performance of information retrieval (IR) systems. In those settings, the goal is to measure whether an IR model can rank documents "relevant" to a given query higher than the documents "irrelevant" to the query. Usually, a set of queries is sampled based on the goal of an IR task, and for each query, a set of documents is sampled according to a desired distribution. Then, human evaluators judge whether a document is relevant or irrelevant to the query.

When applying global ranking metrics to a recommendation model, there is no notion of a query and no clear correspondence to the usage in IR. In fact, global ranking metrics obtained by evaluating ratings on (user, item) pairs do not directly measure the ability of a ranking model to rank items for each user; instead, they should be treated in a way similar to accuracy metrics in a classification task in supervised learning. Hence, all the limitations of accuracy metrics also apply to global ranking metrics.

For local ranking metrics, each user corresponds to a query, the items rated by the user in the test data correspond to documents, and a positive rating corresponds to a "relevant" judgment. Local ranking metrics are useful for measuring the ability of a model to rank items for users. However, they are limited by selection bias and by difficulties in translating ranking metrics to online performance.

Unlike a properly defined IR task where documents are sampled according to certain desired distribution, the set of items used to compute a local ranking metric are subject to selection bias. For explicit ratings, users select the items they want to rate. In many systems, users are more likely to rate items they like. Thus, the item distribution of the test data (which consists of rated items, many of which are liked by users) can be quite different from the item distribution obtained with a new model used to serve real users (where most of the items are unseen by the user before). For implicit ratings, such as clicks on recommended items, the test data are usually collected from an existing recommender system. In this case, the model used by the system during the data collection period decides the item distribution for the test data. If a new model that is to be tested tends to select items that do not appear in the test data, we cannot measure its performance accurately.

Measuring online model performance is the ultimate goal of experimentation. Ranking metrics provide a useful indicator of whether a new model may rank items better than an old model. However, it is difficult to use offline metrics to obtain performance gains for online user engagement metrics like clicks on recommended items. For instance, it is difficult to predict the online performance gain that corresponds to a 10 percent improvement in a ranking metric.