Lecture Notes on Recommender Systems

Michael D. Ekstrand, Boise State University

These notes summarize key points and definitions from CS 538, *Recommender Systems and Online Personalization*. Due to my improvisational teaching style in this class, these notes are sparse, but I hope they contain main of the most important points.

Table of Contents

If you find errors in this material, please e-mail michaelekstrand@boisestate.edu.

Resources

These notes are largely in outline format. For additional study, I recommend the following:

Kim Falk. 2019. *Practical Recommender Systems*. Manning, 432pp. ISBN 9781617292705.

Michael D. Ekstrand, John T. Riedl, and Joseph A. Konstan. 2011. Collaborative Filtering Recommender Systems. *Foundations and Trends® in Human-Computer Interaction* 4(2) (February 2011), 81–173. DOI 10.1561/1100000009.

Throughout these notes, I use the notation documented here:

Michael D. Ekstrand and Joseph A. Konstan. 2019. Recommender Systems Notation: Proposed Common Notation for Teaching and Research. *Computer Science Faculty Publications and Presentations* 177. Boise State University. arXiv:1902.01348 [cs.IR]. DOI 10.18122/cs_facpubs/177/boisestate.

Recommendation Fundamentals

What Is a Recommender System?

A recommender system is an algorithmic tool that recommends *items* to *users*.

Recommendations in Context

There are two ways that we think of context in recommender systems:

- How are the recommendations situated in the user's workflow? For example, presenting recommendations for additional items when the user is preparing to check out and complete their purchases is different from recommending related items on a product detail page.
- What are the user's particular situational needs when the recommendations are accessed or provided? This includes things like time, place, and who they are with.

Recommendation Tasks

There are many things we can use recommender systems for. Two key tasks:

- **Predict** – estimate how much will a given user like a particular item
- **Recommend** – provide one or more items for a user to (possibly) consume

Classical top-N recommendation computes recommendations by first predicting, and then recommending the items with the highest predicted preference.

This is not the only possible modality.

- Set recommenders produce a (possibly unranked) collection of items, that may be optimized for criteria beyond just "top prediction"
- Items with the highest predictions do not necessarily make the best set of recommendations.
- Streaming radio such as Pandora just plays the next item.

Recommendation Inputs

There are many different types of recommendation inputs:

- **Explicit feedback** – data provided by the user for the purpose of communicating preference, such as star ratings and thumbs up/down
- **Implicit feedback** – data gathered from other interactions that can be used to infer preference, such as purchases and clicks.

We can also reason about recommendation inputs based on the time at which they were provided:

- **Memory ratings** are provided sometime after the user experiences the item, and are based on the user's memory.
- **Consumption ratings** are provided when the user is experiencing the item.
- **Expectation ratings** are provided before the user has experienced the item, and is based on their expectation of their preference.

Any of these can be implicit or explicit.

Relation to Psychology
Psychology of preference gives us two major theories of preference. Roughly:

- **Articulated values** holds that we have (relatively stable) preferences for particular items, and when we express a preference we map the preference to the desired scale.
- **Basic values** holds that we have values that we place on different characteristics, along with memory of the characteristics of items; when we express a preference, we combine our memory of the item with our values for its characteristics, possibly adjusted for the current context, and map it to the desired scale.

Mathematical Representations
See the Notation paper (in Resources) for details on the mathematical notation.

Overview of Recommendation – Early Perspective
This paper covers a great deal of ground in recommender systems, anticipating many important lines of research that wouldn't be fully developed for another 10–15 years.

William Hill, Larry Stead, Mark Rosenstein, and George Furnas. 1995. Recommending and evaluating choices in a virtual community of use. In *CHI '95*, 194–201. DOI 10.1145/223904.223929

Mathematical Preliminaries

This section briefly reviews some of the mathematics required in this course, to supplement the notation document.

Linear Algebra

A **vector** $\vec{x} \in \mathbb{R}^n = x_1, \dots, x_n$ is an array of numbers.

A **matrix** $A \in \mathbb{R}^{m \times n}$ is an $m \times n$ array of numbers; in a programming language, we would call this array two-dimensional, but this terminology gets confused with using n to describe the dimensionality of a vector so we will avoid it. The item at row i and column j of A is denoted a_{ij}. The rows and columns of a matrix are vectors.

The **dot product** $\vec{x} \cdot \vec{y} = \sum_{i=1}^{n} x_i y_i$ is the sum of the elementwise products of two vectors of the same dimensionality.

The **Euclidean norm** $\|\vec{x}\|_2$ is the length of the vector; it is defined as $\sqrt{\sum_{i=1}^{n} x_i^2}$, the square root of the sum of the squares. The **Frobenius norm** $\|A\|_F$ of a matrix is the same, defined as $\sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2}$.

The **cosine** between two vectors is $\cos(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{\|\vec{x}\|_2 \|\vec{y}\|_2}$ is the cosine of the angle between the two vectors, and is a measure of similarity between items represented by vectors.

Non-Personalized Recommendation

Recommendations do not have to be personalized.

Means and the Bias Model

The simplest non-personalized *prediction* is the average rating:

$$s(i) = \bar{r}_i = \frac{\sum_{r_{ui} \in R_i} r_{ui}}{|R_i|}$$

Different users have different rating scales, so we can instead compute the *personalized mean* or *bias model*:

$$\begin{aligned} s(i|u) = b_{ui} &= \bar{r} + b_i + b_u \\ \bar{r} &= \frac{\sum_{r_{ui} \in R} r_{ui}}{|R|} \\ b_i &= \frac{\sum_{r_{ui} \in R_i} (r_{ui} - \bar{r})}{|R_i| + \alpha} \\ b_u &= \frac{\sum_{r_{ui} \in R_u} (r_{ui} - b_i - \bar{r})}{|R_u| + \alpha} \end{aligned}$$

If we want to score for a user or item we haven't seen before, then we can allow b_i or b_u to be 0.

The damping parameter α is a shrinking or regularization parameter to discourage the model from learning extreme average opinions based on little data. It is equivalent to assuming that each user or item has α average ratings; actual ratings will eventually pull this away.

Popularity

The simplest non-personalized *recommendation* is the Most Popular Item recommendation. This just recommends items rated or purchased by the most users.

If we want to score items using popularity, we can use the raw number of users; however, it is often more useful to use the *quantile* or the *popularity rank*: the most popular item is 1, least popular is 0.

We can also use popularity to estimate probabilities – if we know nothing else, the probability that a user purchased an item is $\Pr[i] \propto |U_i|$.

Time Decay: Hacker News and Reddit

See "How the Hacker News ranking algorithm works".

Contextualization: Association Rules

We can make recommendations more responsive by contextualizing them to items that the user is browsing. One way to do this is through *association rules*; if the user is browsing item j (called the *target* or *context* item), then we can talk about $s(i|j)$. The simplest way to do this is the *conditional probability* $s(i|j) = \Pr[i|j]$. This probability notation is shorthand for:

$$\Pr[i \in I_u | j \in I_u] = \frac{|U_j \cap U_i|}{|U_j|}$$

The formula for the empirical estimate of the conditional probability can be derived by algebra from the definition of conditional probability. We can then produce recommendations by selecting items with the highest conditional probability given the context item.

Philosophical Note. If it bothers you that we are using probability notation to discuss historical data, there are two different ways you can interpret the probabilistic statement to describe a random process; for this particular problem they result in equivalent mathematics:

- If new, unseen user comes to the site and purchases item j, what is the probability they will also purchase item i? In this case, the probability is taken over the process of randomly generating new users.
- If we randomly draw a user from our database, and they have purchased j, what is the probability they have also purchased i? In this case, the probability is taken over the process of drawing users uniformly at random from the set of users in the database.

Question. What is the drawback to this method? What happens when i is very popular?

We can make conditional probability better-behaved by using *lift*. The lift metric is no longer a probability, but is computed using one:

$$s(i|j) = \text{Lift}(i, j) = \frac{\Pr[i|j]}{\Pr[i]}$$

This answers the question "Given that the user has purchased j, how much does that *increase* our expectation that they will purchase i?".

Content-Based Filtering

Moving beyond basic popularity and probability statistics, we can look at the *content* of the items that we are thinking about recommending.

Types and Sources of Content Information

There are many possible sources of content. Some of them include:

- Textual item content (e.g. the text of a news article or research paper)
- Item metadata (title, authors, tags, dates, abstract or synopsis)
- Reviews (text *about* an item, such as movie reviews or reviews on a site like Amazon)
- Audio or visual content (e.g. the actual audio or video data for a song or movie)
- Transcripts and subtitles (text renderings of audio or video content)

Methods

Each of these types of data can be processed by techniques from relevant specialties of computer science, such as computer vision (for image analysis) and natural language processing (for text content). Information retrieval techniques are very useful for doing content-based recommendation with textual data.

Bag of Words and TF-IDF

One simple approach for processing textual is to use a *bag of words* approach, where we represent the text by the words it contains, usually just counting how often each word appears. We can do this with a pipeline like the following:

1. *Tokenize* the text to split it into individual words.
2. *Normalize* words so that we don't have multiple variants of the same word (e.g. convert to lowercase) (you don't always want to do this, it depends on your application!)
3. *Remove stop words* (e.g. 'the'), because they do not provide signal in bag-of-words settings. Again, you don't always want to do this.
4. *Stem* or *lemmatize* words to unify different tenses of the same word. Again, whether you want to do this depends on your application.
5. *Count* word occurrences to summarize each document as a *term vector*.

The result of this process is that an item or document is represented by a vector mapping words to the number of times those words appear. This is our *term frequency* vector, and it looks like this:

Word	Count
apple	10
pizza	5
fish	7

We can denote the number of times word w appears in item i with y_{iw}, and an item's term vector $\vec{y}_i$ is the vector of such words (if w does not appear in item i, then $y_{iw} = 0$; we often store these vectors sparsely, so we do not store the 0's). We also want to know the number of items that contain w at least once: $d_w = |\{i: y_{iw} > 0\}|$; this is called the *document frequency* of each term.

With these values, we can now compute the TF-IDF (term frequency – inverse document frequency) vector $\hat{\vec{y}}_i$ for each item with the following formula for its elements:

$$\hat{y}_{iw} = y_{iw} \log \frac{|I|}{d_w}$$

Question. Why do we do this? What happens to this equation if d_w increases but y_{iw} remains constant?

We can then compare two items by taking the cosine of their vectors:

$$s(i|j) = \cos\left(\hat{\vec{y}}_i, \hat{\vec{y}}_j\right) = \frac{\hat{\vec{y}}_i \cdot \hat{\vec{y}}_j}{\left\|\hat{\vec{y}}_i\right\|_2 \left\|\hat{\vec{y}}_j\right\|_2}$$

Nearest-Neighbor Collaborative Filtering

The core idea of *collaborative filtering* is to ignore the actual content of items and mine user-item interactions in order to predict unknown preferences. This means that the method does not depend on item characteristics or representations.

Item-based k-NN

We start with *item-based nearest-neighbor* collaborative filtering: to predict a user's preference for an item, we look at similar items they have rated in the past.

In order to do this, we need a notion of 'similarity' between items. One way is to compute the cosine between items' rating vectors: the vectors of users' ratings for those items. If we define $\vec{\hat{r}}_i$ such that $\hat{r}_{ui} = r_{ui} - \bar{r}_i$, then we can define the similarity w_{ij} between i and j:

$$w_{ij} = \cos(\vec{\hat{r}}_i, \vec{\hat{r}}_j)$$

Using these similarities, we can find a neighborhood $N(i|u) \subseteq I_u$, the k items most similar to i that have been rated by u, and use this to score:

$$s(i|u) = \frac{\sum_{j \in N(i|u)} w_{ij}(r_{uj} - \bar{r}_j)}{\sum_{j \in N(i|u)} |w_{ij}|} + \bar{r}_i$$

That is, our prediction of user u's rating for item i is the weighted average of the user's ratings or other items j.

Question. Why do we limit the number of neighbors? What happens if we use too many?

Unary or Binary Data

If R is a binary or unary matrix (values or 0 or 1), or even a count matrix, mean-centering (the $\hat{r}_{ui}$ normalization) and weighted average are not meaningful concepts. Therefore we either user raw rating vectors $\vec{r}_i$ or some other normalization of the matrix R, and can do a sum of similarities:

$$s(i|u) = \sum_{j \in N(i|u)} w_{ij}$$

Negative Similarities

We often limit similarities to only consider item pairs with similarities over some threshold $w_<$.

Question. Why? If no one likes i, what will generally be true about w_{ij}?

Question. How does the cosine above relate to the Pearson correlation?

Key Papers
Item-based k-NN is described in:

Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based Collaborative Filtering Recommendation Algorithms. In *Proceedings of the 10th International Conference on World Wide Web* (WWW '01), 285–295.

Mukund Deshpande and George Karypis. 2004. Item-based top-N Recommendation Algorithms. *ACM Transactions on Information Systems* 22, 1 (January 2004), 143–177.

User-based k-NN
An older method for collaborative filtering is to look at similar *users* instead of similar items. To score for a user, we will look at users $N(u|i) \subseteq U_i$ who have agreed with our target user's ratings in the past, and recommend things that they liked:

$$s(i|u) = \frac{\sum_{v \in N(u|i)} w_{uv}(r_{vi} - \bar{r}_v)}{\sum_{v \in N(u|i)} |w_{uv}|} + \bar{r}_u$$

Key Paper
User-based k-NN is described in:

Jonathan Herlocker, Joseph A. Konstan, and John Riedl. 2002. An Empirical Analysis of Design Choices in Neighborhood-Based Collaborative Filtering Algorithms. *Inf. Retr.* 5, 4 (October 2002), 287–310.

Matrix Factorization

One very common way of scoring items in collaborative filtering is *matrix factorization*.[1] The idea of matrix factorization is to decompose the ratings matrix R into two matrices, one for users and one for items, that represent users and items in a common lower-dimensional vector space. These representations are sometimes called *embeddings*. They take the original user or item rating vectors, in item or user space respectively (e.g. a user vector $\vec{r}_u$ is in $\mathbb{R}^{|I|}$), and map them into into a k-dimensional space; users and items are both represented by vectors in $\mathbb{R}^k$, where $k \ll |I|$.

This lower-dimensional space is called a *latent feature space*, and the dimensions are called *latent features*. The idea is that we can represent user preferences for items by learning user preferences for certain (indescribable) features possessed by items, and similarly we can represent items by their expression of those features. User preference for items is then estimated by combining user-feature preference with item-feature expression.

For a general survey of explicit-feedback matrix factorization, see:

> Y. Koren, R. Bell, and C. Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (August 2009), 30–37. DOI 10.1109/MC.2009.263

Singular Value Decomposition

One way to do this is through the *singular value decomposition*:

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

In this matrix, $P \in \mathbb{R}^{|U| \times k}$ is a matrix of user vectors and $Q \in \mathbb{R}^{|I| \times k}$ is a matrix of item vectors. Σ is a $k \times k$ diagonal matrix of *singular values*; these represent how relatively important each dimension in the latent feature space is. We *truncate* the decomposition by only keeping the k largest singular values and their corresponding columns in the user and item matrices.

In order to compute the singular value decomposition, we need to first prepare $\hat{R}$ to set up a problem that can be solved by an SVD solver. The major problem to solve is that the SVD is only defined over complete matrices, so we need to do something with the (many) missing values of R. A good way to handle this is to normalize the observed values by subtracting a suitable mean; the missing values can then be treated as 0 (now a neutral value, instead of a out-of-range low value) and a

[1] Authors differ in whether they consider matrix factorization to be collaborative filtering. I do, because in the form we are discussing it, it only uses the ratings matrix; other authors only apply the term 'collaborative filtering' to neighborhood methods.

normal sparse SVD solver, such as SciPy or Matlab's `svds` function. If we subtract the item mean, so $\hat{r}_{ui} = r_{ui} - \bar{r}_i$, then the SVD corresponds to the *principle component analysis*. Another reasonable decision is to subtract the bias model $\hat{r}_{ui} = r_{ui} - b_{ui}$.

With this decomposition, we can score an item by recombining features ($\vec{\sigma}$ is a vector of the diagonal of Σ, the singular values):

$$\begin{aligned} s(i|u) &= b_{ui} + \vec{p}_u \cdot \vec{\sigma} \cdot \vec{q}_i \\ &= b_{ui} + \sum_{f=1}^{k} p_{uf} \sigma_f q_{if} \end{aligned}$$

Key Paper

Badrul M. Sarwar, G. Karypis, Joseph Konstan, and John Riedl. 2002. Incremental Singular Value Decomposition Algorithms for Highly Scaleable Recommender Systems. In *Proceedings of the Fifth International Conference on Computer and Information Science* (ICCIT 2002).

Approximating SVD through Alternating Least Squares

The traditional SVD solver relies on all the values, and makes the assumption that missing values are 'neutral' (for whatever definition of neutral is implied by our choice of normalizing mean). We can bypass both of these issues through approximation procedures that learn P and Q matrices that minimize the error in predicting the ratings that we have. We still normalize before learning, but we no longer assuming missing values are actually 0.

Many practical matrix factorization models forgo the distinct Σ matrix, resulting in the following model (B is the matrix of bias model values):

$$R \approx B + PQ^{\mathrm{T}}$$

We can find good user and item matrices by solving the following optimization problem:

$$\underset{P,Q}{\operatorname{argmin}} \| R - B - PQ^{\mathrm{T}} \|_F^2$$

$\| \cdot \|_F$ is the Frobenius norm, a matrix generalization of the L_2 norm (or Euclidean distance); the square root of the sum of the squares of the inner matrix. The minimization is therefore finding P and Q that minimize the squared error of predicting r_{ui} with $s(i|u) = b_{ui} + \vec{p}_u \vec{q}_i$. We will augment this with a *regularization term* and per-user / per-item weightings to improve the solution quality and prevent overfitting. The regularization term makes the problem look like the following:

$$\underset{P,Q}{\operatorname{argmin}}\{\|R - B - PQ^{\mathrm{T}}\|_F^2 + \lambda(\|P\|_F^2 + \|Q\|_F^2)\}$$

We can solve this problem by alternating between solving for P and solving for Q, and each time solving a least squares problem. That is, for a given Q, we can find P that minimizes squared prediction error, and we can do the same to find Q given P. The method gets its name because we alternate between the user and item matrices, finding solving a least squares problem for each in turn. The following algorithm describes this, using the normal equations solution to the least squares problem:

```
    $Q \leftarrow$ random $|I| \times k$ matrix
    until done:
        solve $\left(Q_{I_u}^T Q_{I_u} + \lambda \vec{1}|R_u|\right)\vec{p}_u = Q_{I_u}^T \vec{r}_u$ for $\vec{p}_u$ for each $u \in U$ ($P$-step)
        solve $\left(P_{U_i}^T P_{U_i} + \lambda \vec{1}|R_i|\right)\vec{q}_i = P_{U_i}^T \vec{r}_i$ for $\vec{q}_i$ for each $i \in I$ ($Q$-step)
```

'Done' often means a fixed number of iterations, such as 15 or 20. The linear systems can be solved using the Cholesky decomposition. Since users are independent of each other, as are items, the P-step and Q-step can each be parallelized. This makes the method quite efficient in practice.

Key Paper

Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. 2008. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *Algorithmic Aspects in Information and Management*, 337–348. DOI 10.1007/978-3-540-68880-8_32

Approximate SVD through Featurewise Stochastic Gradient Descent (FunkSVD)

One of the earliest approximate SVD algorithms for recommendation is the FunkSVD algorithm. This algorithm minimizes the squared error by using stochastic gradient descent to optimize latent features one at a time. Its model — $R \approx B + PQ^{\mathrm{T}}$ — is identical to the one used in ALS, but the optimization technique used for estimating P and Q is different. Again, though, it solves the regularized minimization problem (λ is the strength of the regaulrization):

$$\underset{P,Q}{\operatorname{argmin}}\{\|R - B - PQ^{\mathrm{T}}\|_F^2 + \lambda(\|P\|_F^2 + \|Q\|_F^2)\}$$

The heart of FunkSVD is the *update rule*: given a rating r_{ui} and prediction $s(i|u) = b_{ui} + \vec{p}_u \cdot \vec{q}_i$, we can update p_{uf} and q_{if} by:

$$\begin{aligned}
\epsilon_{ui} &= r_{ui} - b_{ui} - \vec{p}_u \cdot \vec{q}_i \\
p'_{uf} &= p_{uf} + \eta\left(q_{if}\epsilon_{ui} - \lambda p_{uf}\right) \\
q'_{if} &= q_{if} + \eta\left(p_{uf}\epsilon_{ui} - \lambda q_{if}\right)
\end{aligned}$$

η is the *learning rate*, a small value (e.g. 0.001) that controls how quickly the algorithm moves through the search space. If the learning rate is too small, it will be very slow; if it is too large, it will jump around the search space and have difficulty converging. λ is the regularization strength, controlling how much the optimization process penalizes 'strong beliefs' (large values for user or item feature affinity/relevance).

These rules are obtained by taking the derivative of the error with respect to each parameter to optimize:

$$\begin{aligned}
\frac{\partial}{\partial p_{uf}}\epsilon_{ui}^2 &= \frac{\partial}{\partial p_{uf}}\left(r_{ui} - b_{ui} - \sum_{f'} p_{uf'}q_{if'}\right)^2 \\
&= 2q_{if}\epsilon_{ui}
\end{aligned}$$

The derivative with respect to q_{if} is equivalent. These derivatives (forming the *gradient* of gradient descent) result in the following algorithm:

```
$Q \leftarrow 0.1^{|I| \times k}$
$P \leftarrow 0.1^{|U| \times k}$
for $f \in 1 \ldots k$:
    until done (e.g. 100 iterations):
        for $r_{ui} \in R$:
            $\epsilon_{ui} \leftarrow r_{ui} - b_{ui} - \vec{p}_u \cdot \vec{q}_i$
            $p_{uf} \leftarrow p_{uf} + \eta(q_{if}\epsilon_{ui} - \lambda p_{uf})$
            $q_{if} \leftarrow q_{if} + \eta(p_{uf}\epsilon_{ui} - \lambda q_{if})$
```

Evaluation: Introduction and Fundamentals

How do we know whether these tools are any good?

Evaluation techniques broadly divide into three categories:

- Offline evaluations using existing data sets and experimental protocols from machine learning and information retrieval.
- Online experiments (often in the form of A/B tests) measuring user response to deployed recommender algorithms or interfaces.
- User studies gathering explicit user feedback on the recommendations and/or their surrounding user experience.

What is a Good Recommendation?

It depends on your context and time scale.

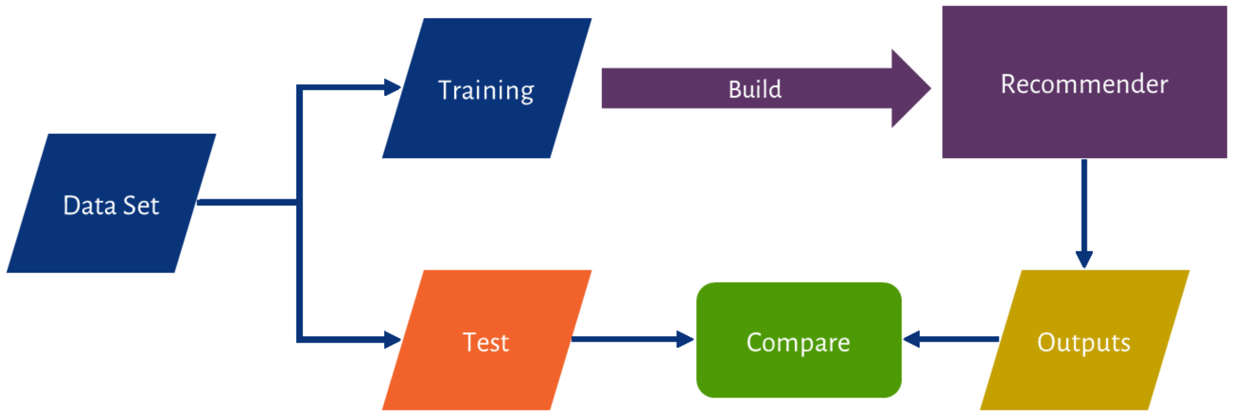
- Drives more sales
- Increases user satisfaction
- Increases lifetime customer value
- Meets user information needs
- Promotes social cohesion

Offline Evaluation

Offline evaluation protocols are derived from information retrieval and machine learning.

Note: the purpose of offline evaluation is to estimate the recommender's likely effects on future user behavior or satisfaction. An evaluation strategy is only as good as its ability to assess whether a recommender will achieve its desired goals when deployed in production.

Evaluation Structure



We do the following:

1. Split the data into training and test data.
2. Train the recommender model on the training data.
3. Generate recommendations or predictions for the users or user-item pairs in the test data.
4. Compare the recommendations or predictions with the test data to measure accuracy.

Sometimes we do this multiple times – a technique called *cross-validation* – in order to obtain more robust estimates.

Splitting Data

In classical supervised learning settings, we split data for evaluation or cross-validation simply by partitioning the instances into different test sets. In recommendation, however, it is not so simple. In particular, the most obvious instances (ratings) are not independent, but rather are grouped by user (and item). If we just partition ratings, we run into some possible problems:

- A user with few ratings may have no training ratings, because all their ratings are selected for the test set.

- Users with many ratings will show up in the test set more often, weighting the evaluation towards performing well for them.
- Likewise, items with many ratings will also show up in the test set more often.

There are therefore a few different approaches:

- **Rating-based splitting** works like supervised ML splitting, and has all of the above problems.
- **User-based splitting** considers each *user* and prepares test data for them. There are several ways to do this, including *hold-one-out* (each user gets one test rating), *fractional holdout* where a fraction of each user's ratings are put in the test data, and *hold-N-out*, where a fixed number of each user's ratings are in the test data. The training data consists of all other ratings. If this is done repeatedly, as in cross-validation, we partition the *users* into disjoint sets. For each set of users, the test data consists of the test items for those users, and the training data consists of the remaining items for those users plus *all* items from the other users in the data set.
- **Item-based splitting** works like user-based splitting but operates on items instead of users. It is not very common, but when done with *hold-N-out*, it can be useful for controlling for the problem of popularity bias by making each item be a 'correct' answer the same number of times regardless of its popularity.

Notation

In describing these metrics, we will use the following notation:

L_u	A recommendation list for user u
R_u^{test}	The test ratings for user u
I_u^{test}	The test items for user u
I_u^{train}	The training items for user u
$I_u^{\text{test+}}$	The items designated as *relevant* in user u's test data. When working with unary or interaction count data (e.g. records of clicks, purchases, or plays), this is typically all items for which we have data for the user. When using rating data, this is either all rated items (to simulate 'purchases' or 'views'), or all items the user 'likes' (gave a high rating to, i.e. $I_u^{\text{test+}} = \{i \in I_u^{\text{test}}: r_{ui} \geq 3\}$).

$I_u^{\text{test}-}$ The items designated as *irrelevant* in user u's test data. When using items with high ratings as relevant, then this is the items with low ratings. Sometimes this is empty, e.g. when we are treating all items the user purchased as relevant.

In addition, most evaluation protocols consider the items for which no relevance data is known ($I \setminus I_u^{\text{test}} \setminus I_u^{\text{train}}$) as irrelevant.

Measuring Predictive Accuracy

The easiest thing to measure in an offline evaluation is *prediction accuracy*: do the recommendation algorithm's predictions or scores match user-provided ratings in the test data?

Root Mean Squared Error

The typical way to measure predictive accuracy is through *root mean squared error* or RMSE. RMSE is computed as follows, where R^{test} is the set of test ratings:

$$\text{RMSE}(R^{\text{test}}, s) = \sqrt{\frac{\sum_{r_{ui} \in R^{\text{test}}} \left(r_{ui} - s(i|u)\right)^2}{|R^{\text{test}}|}}$$

When we compute RMSE over all of R^{test}, we get a global RMSE. We can also compute RMSE per user and average those results (R_u^{test} is the test ratings for user u):

$$\text{UserRMSE}(R^{\text{test}}, s) = \frac{\sum_{u \in U^{\text{test}}} \text{RMSE}(R_u^{\text{test}}, s)}{|U^{\text{test}}|}$$

Question. What are the effects of computing one of these instead of the other?

Mean Absolute Error

We can also compute prediction accuracy using the *mean absolute error*:

$$\text{MAE}(R^{\text{test}}, s) = \frac{\sum_{r_{ui} \in R^{\text{test}}} |r_{ui} - s(i|u)|}{|R^{\text{test}}|}$$

Question. What does RMSE do in terms of handling errors that MAE does not?

Measuring Recommendation Accuracy

We call them *recommender systems*, not *rating prediction systems*, so we might want to measure how they do at *recommending*. In an offline setting, this is usually done via 'top-*N*' metrics borrowed from machine learning and information retrieval.

The basic setup is this:

1. For each test user, build a *candidate set* containing the *test items* and zero or more *decoy items*. The most common candidate set is 'all items the user did not rate in the training data', ($I_u^{\text{cand}} = I \setminus I_u^{\text{train}}$).
2. Generate a recommendation list L of size N from the candidate set for each user.
3. Compare recommendations with the test data to compute a recommendation score.

Different metrics have different requirements from the test data, and different ways of interpreting the data available. Some require simply judgements of 'relevant' and 'not relevant'; others take advantage of relative ordering between items; others use a utility value.

Precision

Precision measures how good the recommender is at making its recommendations be things the user likes. It is computed as follows:

$$\text{P@N}(L) = \frac{|L \cap I_u^{\text{test+}}|}{|L|}$$

This answers the question 'of the recommended items, what fraction are known to be relevant?'

Recall and Hit Rate

Recall measures how good the recommender is at finding the items that the user has liked. It is computed as follows:

$$\text{R@N}(L) = \frac{|L \cap I_u^{\text{test}}|}{I_u^{\text{test+}}}$$

This answers the question 'of the items known to be relevant to the user, what fraction are recommended?'

When $|I_u^{\text{test+}}| > |L|$, there is not an established practice. This is not a problem in most recommendation evaluation protocols.

Hit rate is effectively binarized recall: 1 if the list contains at least one relevant item, and 0 otherwise.

Mean Reciprocal Rank

The *reciprocal rank* measures where the first relevant item appears in the list. It operationalizes a browsing model where the user scans the list from top to bottom and stops when they reach the first relevant or interesting item, and measures the value of the recommendations in such a model. It is computed as:

$$\mathrm{RR}(L) = \frac{1}{k}$$
$$\mathrm{MRR} = \sum_{u} \mathrm{RR}(L_u)$$

Where k is the rank of the first relevant item.

Discounted Cumulative Gain

Cumulative gain metrics attempt to estimate the 'gain' (or utility) of a recommendation list, premised on the idea that relevant items provide more utility if they are higher on the list. Given a rating or utility value r_{ui} (assumed to be 0 for unknown items), we can compute the discounted cumulative gain of list as follows:

$$\mathrm{DCG}(L) = \sum_{k=1}^{N} \frac{r_{ui_k}}{\mathrm{disc}(k)}$$
$$\mathrm{disc}(k) = \log_2 \max\{k, 2\}$$

The $\max\{k, 2\}$ operation is to prevent division by zero for the first item. The discount function $\mathrm{disc}(k)$ can be replaced with other discounts, such as a half-life decay function. Base-2 logarithmic discounting is the most common in current practice.

The DCG on its own, however, is not comparable between users. Users with different numbers of relevant items, or different total rating values, will have different maximum ratings; therefore, we compute *normalized discounted cumulative gain* (nDCG) to correct fot this:

$$\mathrm{nDCG}(L_u) = \frac{\mathrm{DCG}(L_u)}{\mathrm{DCG}\left(L_u^{\mathrm{ideal}}\right)}$$

L_u^{ideal} is the *ideal* recommendation list for u: the list that has all the items u likes at the top, in decreasing order of rating or utility. If $|I_u^{\mathrm{test+}}| > N$, this list is truncated to the N 'best' items. The resulting metric approximates the fraction of possible utility that a given list or ranking achieves.

Receiver Operating Characteristic and AUC

Another way to characterize the accuracy of a recommender is through the *receiver operating characteristic*. This is an analysis of the relationship of the true positive rate to the false positive rate. One way to think of it is this: as we go down the recommendation list, do we relevant or irrelevant items more quickly?

ROC curves are easy to compute for a single list or set of scores: order the results by decreasing threshold; for each threshold, compute the TPR and the FPR, and use that as a point in the plot.

Gunawardana and Shani (2009) identify the different ways of computing for multiple rankings, which is the common case for recommender evaluation as each user has their own recommendation list. We can either compute TPR and FPR at each length for each user, and average them by lengths; or we can compute a curve for each user, and average the curves. (There is also a third global method).

Question. What assumptions do each of these reflect?

With the ROC curve, we can also compute the *area under the curve*. A larger area corresponds to increased ability to discern between different lists. The area under the ROC curve is also equivalent to the probability of putting two items in the right order: if you randomly select two different items from the data set, the probability that the recommender put the relevant one before the irrelevant one is AUC.

Scikit-Learn provides ROC and AUC methods.

Key Papers

Asela Gunawardana and Guy Shani. 2009. A Survey of Accuracy Evaluation Metrics of Recommendation Tasks. *J. Mach. Learn. Res.* 10, (December 2009), 2935–2962. http://www.jmlr.org/papers/v10/gunawardana09a.html

Other Offline Metrics

Diversity is not the only thing that we can measure by running a recommender and reviewing its output. We can also look at things such as the *diversity* and *novelty* of recommendations.

Novelty

True novelty is difficult to measure, because we do not know what items the user is not familiar with. We can approximate it, however, using popularity statistics: more popular items are more likely to be familiar to the user.

One way to make this computationally useful is to *rank* items by popularity, with the most popular item being 1; $\text{nov}(i) = \text{rank}(i)$.

We can also use the number of items, e.g. $\text{nov}(i) \propto |U_i|^{-1}$.

Diversity
Diversity can be measured using some notion of similarity or distance between items. This is usually done with some external reference point or content data, such as tags or categories.

If we have a similarity function $\text{sim}(i, j)$, such as the cosine between tag vectors, we can compute the *intra-list similarity* of a recommendation list or user profile; this is the average similarity between pairs of items in the list:

$$\text{ILS}(L) = \frac{1}{|L|(|L| - 1)} \sum_{i,j \in L : i \neq j} \text{sim}(i, j)$$

A high intra-list similarity indicates that a list is not very diverse. If instead we have a distance function d, we can compute *intra-list distance*, where high distance is more diverse.

Coverage
Coverage is the ability of the recommender to score or recommend across the breadth of the database, or the fraction of items that might feasibly show up in a recommendation list.

Key Papers

Cai-Nicolas Ziegler, Sean McNee, Joseph A Konstan, and Georg Lausen. 2005. Improving Recommendation Lists through Topic Diversification. 22–32. DOI:https://doi.org/10.1145/1060745.1060754

Saúl Vargas and Pablo Castells. 2011. Rank and Relevance in Novelty and Diversity Metrics for Recommender Systems. In *RecSys '11*, 109–116. DOI:https://doi.org/10.1145/2043932.2043955

Problems with Offline Evaluation
Offline evaluation, while very useful, has a number of limitations.

General Limitations
In an offline evaluation, we are holding out historical data and testing the recommender's ability to predict or model that data. This entails a focus on what the user *has done*, and it does not test

the recommender algorithm's ability to prospectively affect *future* user behavior. This is true both for predictive accuracy evaluations and recommendation (top-*N*) evaluation.

Limitations Specific to Top-N Evaluation

Top-*N* evaluation has some additional limitations due to the fact that the test data is missing relevance judgements for most items.

- **Popularity bias** arises because popular items are more likely to be rated in general, which means that they are more likely to be a test rating. A recommender that just recommends popular items will do quite well, just because popularity is the right answer more often. The experimental protocol is therefore hindered in its assessment of the recommender's ability to actually model individual user preference. This would not be a problem if a user's likelihood to rate an item depended only on their preference for that item; but since it is a combination of preference *and* discovery or awareness, and popularity affects awareness, this is a problem in evaluation.
- **Missing relevance data** means that a recommender that is very good at finding items the user would like, but has never heard of — optimal serendipity — the evaluation procedure will not recognize or reward this quality, and will prefer a recommender that is effective at finding things the user has heard of.
- **Recommender infection** is similar to popularity bias, but arises because the data was collected from users interacting with a recommender system; items recommended by the system in use are more likely to appear in the data set.

Key Papers

Alejandro Bellogin, Pablo Castells, and Ivan Cantador. 2011. Precision-oriented Evaluation of Recommender Systems: An Algorithmic Comparison. In *RecSys '11*, 333–336. DOI:https://doi.org/10.1145/2043932.2043996

Michael D Ekstrand and Vaibhav Mahant. 2017. Sturgeon and the Cool Kids: Problems with Top-N Recommender Evaluation. Retrieved from https://aaai.org/ocs/index.php/FLAIRS/FLAIRS17/paper/viewPaper/15534

Allison J. B. Chaney, Brandon M. Stewart, and Barbara E. Engelhardt. 2018. How Algorithmic Confounding in Recommendation Systems Increases Homogeneity and Decreases Utility. In *Proceedings of the 12th ACM Conference on Recommender Systems* (RecSys '18), 224–232. DOI:https://doi.org/10.1145/3240323.3240370

Online Evaluation

There are, broadly speaking, three kinds of online evaluation:

- A/B tests
- User studies
- Multi-armed bandits

The last is not technically an evaluation technique, but rather is a method for directly finding optimal recommendation algorithms.

A/B Testing

A/B testing is a means for running a *randomized controlled trial* of your recommender system (or any other change to the system). The basic idea is this:

1. Randomly divide users into different (2 in the A/B case).
2. Give one the control (current system), and the other the treatment (new system, e.g. new recommender algorithm).
3. Measure key outcomes (sales, clicks, renewals, etc.) and compare performance using standard statistical techniques for evaluating controlled trials.

Notes on Structure

To do an A/B test well, you need to watch out for a number of things:

1. Run long enough to average over cyclic behavior (at least one week)
2. Throw away initial data, unless you are specifically looking to test novelty effects
3. Start slow and watch for problems, then ramp up
4. Have a good metric!

Key Papers and Resources

I highly recommend you read these, and especially watch the video!

Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M Henne. 2008. Controlled experiments on the web: survey and practical guide. *Data Min. Knowl. Discov.* 18, 1 (2008), 140–181. DOI:https://doi.org/10.1007/s10618-008-0114-1

Ron Kovahi. 2015. Online Controlled Experiments: Lessons from Running A/B/n Tests for 12 Years. In *KDD 2015*. Sydney, NSW, Australia. https://youtu.be/ZfhQ-fIg4EU

User Studies

User studies directly ask users to provide feedback on recommendations and their experiences. This is often in the form of survey questionnaires, asking users to rate things like their satisfaction or their perception of the diversity of the list.

User responses are necessary in order to measure subjective constructs such as satisfaction or perceptions. These measurements can be integrated with objective metrics, such as offline measures of diversity, and with behavioral observations such as user viewing or purchasing behavior to gain a holistic understanding of the system's capabilities.

To perform a user study well, it is not sufficient to ask individual questions about dimensions such as diversity, because individual user responses are quite noisy. Robust measurement of user perception requires that we ask *multiple* questions per target construct; it's best to make them as statements to which users respond on a 'Strongly Agree' to 'Strongly Disagree' Likert scale. For example, to measure Diversity, we might ask:

- This is a varied list of movies.
- The movies in this list are mostly the same. *(note: this question is reversed!)*
- This list of movies matches a variety of tastes.
- There are many different movies in this list to choose from.

We then use *factor analysis* to map the question responses to specific factors, estimating an overall 'diversity' score. If we have multiple constructs we want to relate to each other and/or to other variables such as user activity or experimental conditions, we use *structural equation modeling*.

Key Papers

Bart Knijnenburg, Martijn Willemsen, Zeno Gantner, Hakan Soncu, and Chris Newell. 2012. Explaining the user experience of recommender systems. *User Model. User-adapt Interact.* 22, 4–5 (October 2012), 441–504. DOI:https://doi.org/10.1007/s11257-011-9118-4

Learning to Rank

When we approximated the SVD, we used a machine-learning approach to learn a model that can *predict* ratings quite well. We then hoped that ranking by top-*N* would be a good way to turn those into recommendations.

But that is not the only way we can learn models to produce recommendations. We can also learn directly to *rank* items in an order consistent with user preferences. The fundamental machine learning paradigm – optimize an objective function – can apply to list ranking metrics, not just to prediction accuracy, like we saw with matrix factorization methods.

The fundamental difficulty for learning to rank is that optimizing a list accuracy function tends to be harder than optimizing estimates for individual items, in two fundamental ways: it is more computationally expensive to generate a list and measure it than to score an item, and the process of generating a list is not differentiable.

AUC to Ranking

One way to address these problems is to simplify the ranking problem. Instead of learning a model that can produce an entire rank of items, can we learn a model that, given two items, puts them in the correct order?

That is, can we learn a function $s(i|u)$ such that $s(i|u) > s(j|u)$ when $i \succ_u j$ – that is, user u prefers item i to item j?

The *area under the curve* we saw back in Offline Evaluation has a useful property: the AUC of a ranking is equivalent to the fraction of items that the ranking puts in the correct order, or equivalently, the probability that a randomly selected pair of items are in the correct order. We can change the learning process to look at *pairs* of items instead of individual items, and optimize a scoring function.

Bayesian Personalized Ranking

If $i \succ_u j$, then we want $s(i|u) > s(j|u)$, and therefore $s(i|u) - s(j|u) > 0$. The full details are in the paper, but we can accomplish this by maximizing the function:

$$\sum_{(u,i,j):i \succ_u j} \ln \sigma\big(s(i|u) - s(j|u)\big) - \mathrm{reg}(s)$$

This can be done for many different scoring functions; the common BPR-MF uses matrix factorization, so $s(i|u) = \vec{p}_u \cdot \vec{q}_i$.

The problem is therefore to find P and Q such that this function is maximized. For this model, the regularization term $\mathrm{reg}(s) = \lambda(\|P\|_F + \|Q\|_F)$.

Key Paper

Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian Personalized Ranking from Implicit Feedback. In *UAI '09*, 452–461. https://arxiv.org/abs/1205.2618