Capstone Project 2
Recommender Engine Using a Last.fm Music Dataset

Objective and goal

Recommender systems have been around since at least 1992. Today we see different flavours of recommenders, deployed across different verticals:

- Amazon
- Netflix
- Hulu
- Spotify
- Facebook
- Last.fm.

What recommender systems do is they predict whether or not a user will like an item they have not seen. The system does this either using the user's history of rating(content based filtering) or the user's similarity to other users (collaborative filtering). We will explain these filtering methods in more detail below. The goal of this project is to build a recommender system using collaborative filtering methods.

Introduction to Recommender Systems

Recommendation systems changed the way inanimate websites communicate with their users. Rather than providing a static experience in which users search for and potentially buy products, recommender systems increase interaction to provide a richer experience. Recommender systems identify recommendations autonomously for individual users based on past purchases and searches, and on other users' behavior. This article introduces you to recommender systems and the algorithms that they implement.

Basic approaches

Most recommender systems take either of two basic approaches: collaborative filtering or content-based filtering. Other approaches (such as hybrid approaches) also exist.

Collaborative filtering

Collaborative filtering arrives at a recommendation that's based on a model of prior user behavior. The model can be constructed solely from a single user's behavior or — more effectively — also from the behavior of other users who have similar traits. When it takes other users' behavior into account, collaborative filtering uses group knowledge to form a recommendation based on like users. In essence, recommendations are based on an automatic collaboration of multiple users and filtered on those who exhibit similar preferences or behaviors.

For example, suppose you're building a website to recommend blogs. By using the information from many users who subscribe to and read blogs, you can group those users based on their preferences. For example, you can group together users who read several of the same blogs. From this information, you identify the most popular blogs that are read by that group. Then — for a particular user in the group — you recommend the most popular blog that he or she neither reads nor subscribes to.

In the table in Figure 1, a set of blogs forms the rows, and the columns define the users. The intersection of blog and user contains the number of articles read by that user of that blog. By clustering the users based on their reading habits (for example, by using a *nearest-neighbor* algorithm), you can see two clusters of two users each. Note the similarities in the reading habits of the members of each cluster: Marc and Elise, who both read several articles about Linux® and cloud computing, form Cluster 1. In Cluster 2 reside Megan and Jill, who both read several articles about Java™ and agile.

Blogs	Marc	Megan	Elise	Jill
Linux	13	3	11	-
OpenSource	10	-	-	3
Cloud Computing	6	1	9	-
Java Technology	-	6	-	9
Agile	-	7	1	8
		Articles read per user		
Cluster	1	2	1	2

Figure 1: Simple example of collaborative filtering

Now you can identify some differences within each cluster and make meaningful recommendations. In Cluster 1, Marc read 10 open source blog articles, and Elise read none; Elise read one agile blog, and Marc read none. In Figure 1, then, one recommendation for Elise is the open source blog. No recommendations can be made for Marc because the small difference

between him and Elise in agile blog reads would likely be filtered away. In Cluster 2, Jill read three open source blogs, and Elise read none; Elise read 11 Linux blogs, and Jill read none. Cluster 2, then, carries a pair of recommendations: the Linux blog for Jill and the open source blog for Megan.
Another way to view these relationships is based on their similarities and differences, as illustrated in the Venn diagram in Figure 2. The similarities define (based on the particular algorithm used) how to group users who have similar interests. The differences are opportunities that can be used for recommendation — applied through a filter of popularity, for example.

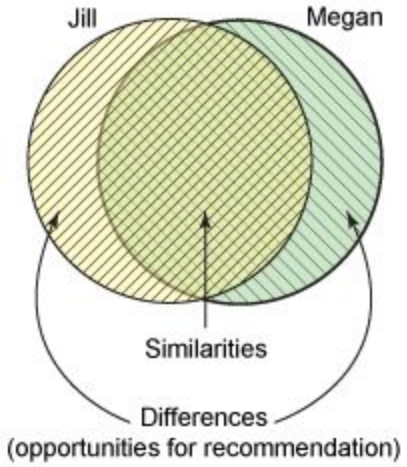


Figure 2: Similarities and differences used in collaborative filtering

Although Figure 2 is a simplification (and suffers from a sparseness of data by using only two samples), it's a convenient representation.

Content-based filtering

Content-based filtering constructs a recommendation on the basis of a user's behavior. For example, this approach might use historical browsing information, such as which blogs the user reads and the characteristics of those blogs. If a user commonly reads articles about Linux or is likely to leave comments on blogs about software engineering, content-based filtering can use this history to identify and recommend similar content (articles on Linux or other blogs about software engineering). This content can be manually defined or automatically extracted based on other similarity methods.
Returning to Figure 1, focus on the user Elise. If you use a blog ranking that specifies that users who read about Linux might also enjoy reading about open source and cloud computing, you can easily recommend — on the basis of her current reading habits — that Elise read about open

source. This approach, illustrated in Figure 3, relies solely on content that a single user accesses, not on the behavior of other users in the system.

Figure 3: Ranked differences used in content-based filtering

The Venn diagram in Figure 2 also applies here: If one side is the user Elise and the other a ranked set of similar blogs, the similarities are ignored (because those blogs were already read by Elise), and the ranked differences are the opportunities for recommendation.

Hybrids

Hybrid approaches that combine collaborative and content-based filtering are also increasing the efficiency (and complexity) of recommender systems. A simple example of a hybrid system could use the approaches shown in Figure 1 and Figure 3. Incorporating the results of collaborative and content-based filtering creates the potential for a more accurate recommendation. The hybrid approach could also be used to address collaborative filtering that starts with sparse data — known as *cold start*— by enabling the results to be weighted initially toward content-based filtering, then shifting the weight toward collaborative filtering as the available user data set matures.

Algorithms to determine similarity

As demonstrated by the winning approach for the Netflix prize, many algorithmic approaches are available for recommendation engines. Results can differ based on the problem the algorithm is designed to solve or the relationships that are present in the data. Many of the algorithms come from the field of machine learning, a subfield of artificial intelligence that produces algorithms for learning, prediction, and decision-making.

Pearson correlation

Similarity between two users (and their attributes, such as articles read from a collection of blogs) can be accurately calculated with the *Pearson correlation.*This algorithm measures the linear dependence between two variables (or users) as a function of their attributes. But it doesn't calculate this measure over the entire population of users. Instead, the population must be filtered down to *neighborhoods* based on a higher-level similarity metric, such as reading similar blogs. The Pearson correlation, which is widely used in research, is a popular algorithm for collaborative filtering.

Clustering algorithms

Clustering algorithms are a form of unsupervised learning that can find structure in a set of seemingly random (or unlabeled) data. In general, they work by identifying similarities among items, such as blog readers, by calculating their distance from other items in a *feature space.* (Features in a feature space could represent the number of articles read in a set of blogs.) The number of independent features defines the dimensionality of the space. If items are "close" together, they can be joined in a cluster.

Many clustering algorithms exist. The simplest one is *k*-means, which partitions items into *k* clusters. Initially, the items are randomly placed into clusters. Then, a *centroid* (or *center*) is calculated for each cluster as a function of its members. Each item's distance from the centroids is then checked. If an item is found to be closer to another cluster, it's moved to that cluster. Centroids are recalculated each time all item distances are checked. When stability is reached (that is, when no items move during an iteration), the set is properly clustered, and the algorithm ends.

Calculating the distance between two objects can be difficult to visualize. One common method is to treat each item as a multidimensional vector and calculate the distance by using the Euclidean algorithm.

Other clustering variants include the Adaptive Resonance Theory (ART) family, Fuzzy C-means, and Expectation-Maximization (probabilistic clustering), to name a few.

Other algorithms

Many algorithms — and an even larger set of variations of those algorithms — exist for recommendation engines. Some that have been used successfully include:

- **Bayesian Belief Nets**, which can be visualized as a directed acyclic graph, with arcs representing the associated probabilities among the variables.

- **Markov chains**, which take a similar approach to Bayesian Belief Nets but treat the recommendation problem as sequential optimization instead of simply prediction.
- **Rocchio classification** (developed with the Vector Space Model), which exploits feedback of the item relevance to improve recommendation accuracy.

Building a recommender engine using an artist rating dataset from Last.fm

Dataset

Last.fm provides a dataset for music recommendations. For each user in the dataset it contains a list of their top most listened to artists including the number of times those artists were played. There are a total of **92,834** user-artist ratings in the dataset. It also includes user applied tags which could be used to build a content vector.

Last.fm's data is aggregated, so some of the information (about specific songs, or the time at which someone is listening to music) is lost. However, it is the only dataset in the sample that has information about the social network of the people in it. More detailed dataset description and variable definitions can be found at the link below:

http://files.grouplens.org/datasets/hetrec2011/hetrec2011-lastfm-readme.txt

Below is a screenshot of the first 5 rows in the main dataset (user_artists):

- Number of unique users: **1,892**
- Number of unique artists: **17,631**
- Number of ratings (i.e. total number of rows): **92,834**

The 'weight' variable is the number of times a user has listened to an artist. We will use this variable as a measure of how much the particular user liked the artist. We will also use this variable to define the final 'rating' variable. More on this later.

	userID	artistID	weight
0	2	51	13883
1	2	52	11690
2	2	53	11351
3	2	54	10300
4	2	55	8983

Methodology

We will take the following approach to build the engine and evaluate its performance:

- Establish a baseline performance by taking the average of other users' ratings for a particular item when predicting a user's rating on that item
- Further develop the algorithm by using pearson correlation coefficient to assess similarity of users. When it comes to predicting a particular user's rating, we only average the ratings of the users who are more similar to the one we are predicting the rating of. Pearson correlation coefficient between two vectors is defined as:

$$pearson(x,y) = \frac{(x-\bar{x}).(y-\bar{y})}{\sqrt{(x-\bar{x}).(x-\bar{x}) * (y-\bar{y}).(y-\bar{y})}}$$

- Build a 3rd engine using the same method as above except that we use the cosine method to calculate similarity between two vectors. The cosine between two vectors is defined as:

$$cosine(x,y) = \frac{(x.y)}{\sqrt{(x.x) * (y.y)}}$$

Data Wrangling and Cleaning

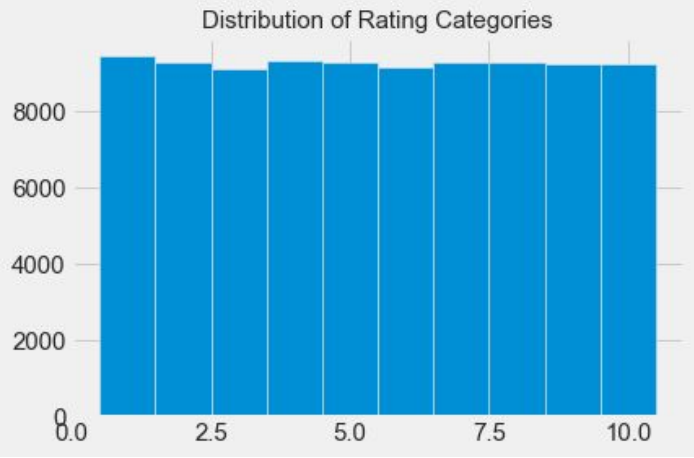
After obtaining the data we examine the information in the 'user_artists' dataset, as below, and we see that there are no missing values. All of the userID, artistID and weight values are populated. So no further cleaning was needed.

Python output of the .info() method:

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 92834 entries, 0 to 92833
Data columns (total 3 columns):
userID 92834 non-null int64
artistID 92834 non-null int64
weight 92834 non-null int64
dtypes: int64(3)
memory usage: 2.1 MB

Rating System

The 'weight' variable in user_artists dataset represents the number of times a user has listened to an artist. It can be interpreted as how much the user liked the artist. But since predicting the exact number of times a user would listen to an artist is not of particular interest, we categorize the 'weight' variable into 10 categories, with cut points being the 10th, 20th, …, 100th percentiles. We then treat the resulting 10 categories as our rating categories. Below is a histogram of the 92,834 rating values:



Generating 80-20 train/test sets

We do this by putting 20% of each user's ratings in the test set and the remaining 80% in train set. The reason for this is to have as much of the users as possible in both train and test sets. However, this would not be fully possible since some users have only rated one artist. For these cases the user would either be placed in train set or test set.

- Size of the train set: **74,241** rows
- Size of the test set: **18,593** rows

Evaluation Criterion

We will produce different versions of a collaborative filter and our criterion to compare the performance of each against others is Root-mean-squared-error (RMSE) defined as:

$$RMSE = \sqrt{\frac{\sum(y-\hat{y})^2}{n}}$$

3 Versions of the engine

We used 3 different algorithms for this recommender engine. One algorithm to establish the baseline performance and 2 other algorithms which were further developed:

- **Baseline algorithm:** For each user-artist pair in the test set, this algorithm predicted the rating by taking the *average* of other users' ratings for the same artist in the train set. This algorithm yielded a RMSE of **2.95**.

- **Using Pearson correlation:** For each user-artist pair in the test set, this algorithm uses pearson correlation coefficient to calculate the similarity between the user in question and other users in the train set. It then predicts the rating by taking the *weighted average* of the ratings from users who had a positive (i.e. between 0 and 1) correlation coefficient. The calculated pearson correlation would serve as weight in averaging. This algorithm yielded a RMSE of **2.83**.

- **Using cosine similarity:** For each user-artist pair in the test set, this algorithm uses cosine similarity to calculate the similarity between the user in question and other users in the train set. It then predicts the rating by taking the *weighted average* of the ratings from users who had a positive cosine similarity value. The calculated cosine similarity would serve as weight in averaging. This algorithm yielded a RMSE of **2.95**.

Conclusion

Given the above RMSE values, we would use the collaborative filter with pearson correlation coefficient since it has the lowest RMSE value.

Next steps

- Users of last.fm can connect by becoming friends in the system and our data includes these friendship relationships. One could consider adding friends of a particular user as additional features. This way having common friends can make two users more similar. This addition was not easy to implement in this project due to computer resource limitations. The algorithms which we already employed each took more than 2 hours to run.
- Fastai approaches are also common approaches for recommender systems which can be considered as further development.