Recommendation Systems in the Real world

An overview of the process of designing and building a recommendation system pipeline.

Parul Pandey
May 17, 2019 · 9 min read

Photo by Pixabay from Pexels

Too few choices are bad but too many choices can lead to paralysis

*Have you heard about the famous **Jam Experiment**? In 2000, psychologists Sheena Iyengar and Mark Lepper from Columbia and Stanford University presented a study based on their field experiment. On a regular day, consumers shopping at an upscale grocery store at a local food market were presented with a tasting booth which displayed 24 varieties of Jam. On some other day, the same booth displayed only 6 varieties of Jam. The*

experiment was being conducted to adjudge which booth would garner more sales and it was assumed that more varieties of jam would fetch more people to the counter thereby getting more business. However, a strange phenomenon was observed. Whereas the counter with 24 jams generated more interest, their conversion to sales was pretty low(about 10 times lower) as compared to the 6 jams counter.

Image Source: The Paradox of Choice

So what just happened? Well, it appears that a lot of choices does seem appealing but this choice overload may sometime prove to be confusing and hampering for the customers. **So even if the online stores have access to millions of items, without a good recommendation system in place, these choices can do more harm than good.**

. . .

In my last article on Recommender Systems, we had an overview of the remarkable world of Recommended systems. Let us now go a little deeper and understand its architecture and various terminologies associated with the Recommender Systems.

The Remarkable world of Recommender Systems

An overview of the Recommendation systems and how they provide an effective form of targeted marketing.

towardsdatascience.com

towardsdatascience.com

. . .

Terminology & Architecture

Let's look at some important terms which are associated with Recommender systems.

Items/Documents

These are the entities that are recommended by the system like movies on Netflix, videos on Youtube and songs on Spotify.

Query/Context

The system utilises some information to recommend the above items and this information constitutes the query. Queries can further be a combination of the following:

- **User Information** which may include user id or items with which the user has previously interacted.
- **Some additional context** like the user's device, user's location etc.

Embedding

Embeddings are a way to represent a categorical feature as a continuous-valued feature. In other words, an embedding is a translation of a high-dimensional vector into a low-dimensional space called an embedding space. In this case, queries or items to recommend have to be mapped to the embedding space. **Many recommendation systems rely on learning an appropriate embedding representation of the queries and items.**

Here is a great resource on Recommender system which is worth a read. I have kind of summarised it above but you can study it in detail and it gives a holistic view of the recommendations especially from Google's point of view.

Introduction | Recommendation Systems | Google Developers

An introduction to recommendation systems in machine learning

developers.google.com

. . .

Architectural Overview

A common architecture of Recommender Systems comprises of the following three essential components:

1. Candidate Generation

This is the first stage of the Recommender Systems and takes events from the user's past activity as input and retrieves a small subset (hundreds) of videos from a large corpus. There are mainly two common candidate generation approaches:

- **Content-Based Filtering**

Content-based filtering involves recommending items based on the attributes of the items themselves. The system recommends items similar to what a user has liked in the past.

- **Collaborative Filtering**

Collaborative filtering relies on the user-item interaction and relies on the concept that similar users like similar things eg Customers who bought this item also bought this.

2. Scoring

This constitutes the second stage where another model further ranks and scores the candidates usually on a scale of 10. For instance, in the case of Youtube, the ranking network accomplishes this task by assigning a score to each video according to the desired objective function using a rich set of features describing the video and user. The highest scoring videos are presented to the user, ranked by their score.

3. Re-ranking

In the third stage, the system takes into account additional constraints to ensure diversity, freshness, and fairness. For instance, the system removes the content which

has been explicitly disliked by the user earlier and also takes into account any fresh item on the site.

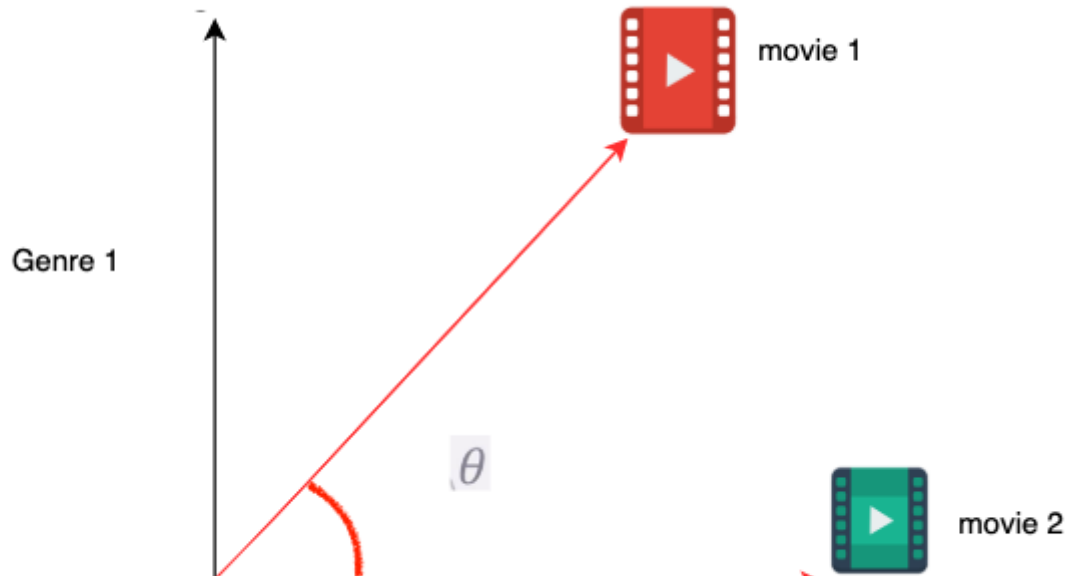
The overall structure of a typical recommendation system

. . .

Similarity Measures

How do you identify whether an item is similar to another? It turns out that both content-based and collaborative filtering techniques employ some kind of similarity metrics. Let's look at two such metrics.

Consider two movies — movie1 and movie 2 belonging to two different genres. Let's plot the movies on a 2D graph assigning a value of zero if the movie doesn't belong to a genre and 1 if a movie belongs to the genre.



Here movie 1(1,1) belongs to both the genres 1 and 2 while movie 2 only belongs to genre 2(1,0). These positions can be thought of as vectors and the angle between these vectors tells a lot about their similarity.

Cosine Similarity

It is the cosine of the angle between the two vectors, `similarity(movie1,movie2) = cos(movie1,movie2) = cos 45` which is around 0.7. Cosine Similarity of 1 denotes the highest similarity while a cosine similarity value of zero denotes no similarity.

Dot product

The dot product of two vectors is the cosine of the angle multiplied by the product of norms i.e `similarity(movie1,movie2) = ||movie1|| ||movie 2|| cos(movie1,movie2).`

. . .

Recommender Pipeline

A typical recommender system pipeline consists of the following five phases:

A typical recommender system pipeline

Let's say we are building a **movie recommender system**. The system has no prior knowledge of the users or the movies but only the interactions that users have with the movies through rating given by them. Here is a dataframe which consists of movie ID, user ID and the ratings of the movie.

user_id	movie_id	rating
2	439	4.0

10	368	4.5
14	114	5.0
19	371	1.0
2	371	3.0

Movie rating Dataframe

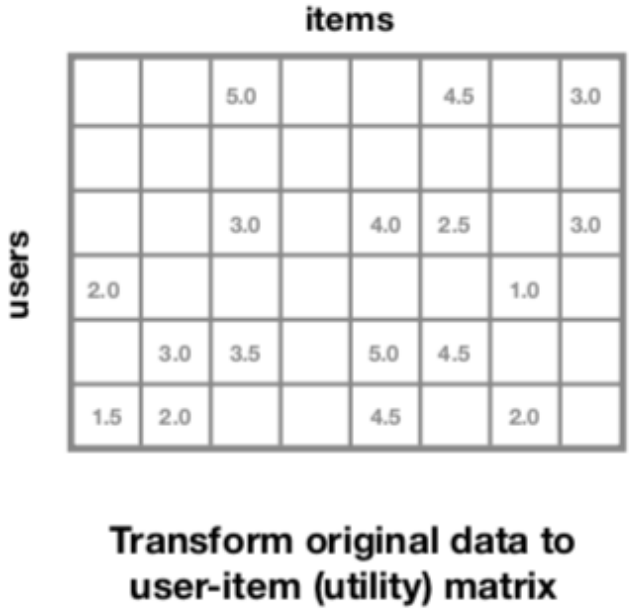
Since we only have the ratings with us and nothing else, we shall be using collaborative filtering for our Recommender system.

1. Pre-Processing

- **Utility matrix conversion**

We need to first transform the movie rating dataframe into an user-item matrix, also called a **utility matrix.**

user_id	movie_id	rating
2	439	4.0
10	368	4.5
14	114	5.0
19	371	1.0
2	371	3.0
19	114	4.5
3	439	3.5
54	421	2.0
32	114	3.0
10	369	1.0



Source

Every cell of the matrix is populated by the ratings that the user has given for the movie. This matrix is typically represented as a **scipy sparse matrix** since many of the cells are empty due to the absence of any rating for that particular movie. Collaborative filtering doesn't work well if the data is sparse so we need to calculate the sparsity of the matrix.

$$\text{sparsity} = \frac{\text{No of ratings}}{\text{Total no. of elements}}$$

If the sparsity value comes out to be around 0.5 or more, then collaborative filtering might not be the best solution. Another important point to note here is that the empty cells actually represent new users and new movies. Therefore, if there is a high proportion of new users then again we might think of using some other recommender methods like content-based filtering or hybrid filtering.

- **Normalization**

There will always be users who are overly positive(always leave a 4 or 5 rating) or overly negative(rate everything as 1 or 2). Therefore we need to normalise the ratings to account for the user and item bias. This can be done by taking the Mean Normalisation.

$$b_{ui} = \mu + b_i + b_u$$

user-item rating bias — global avg — item's avg rating — user's avg rating

Source: Normalisation the Ratings

2. Model Training

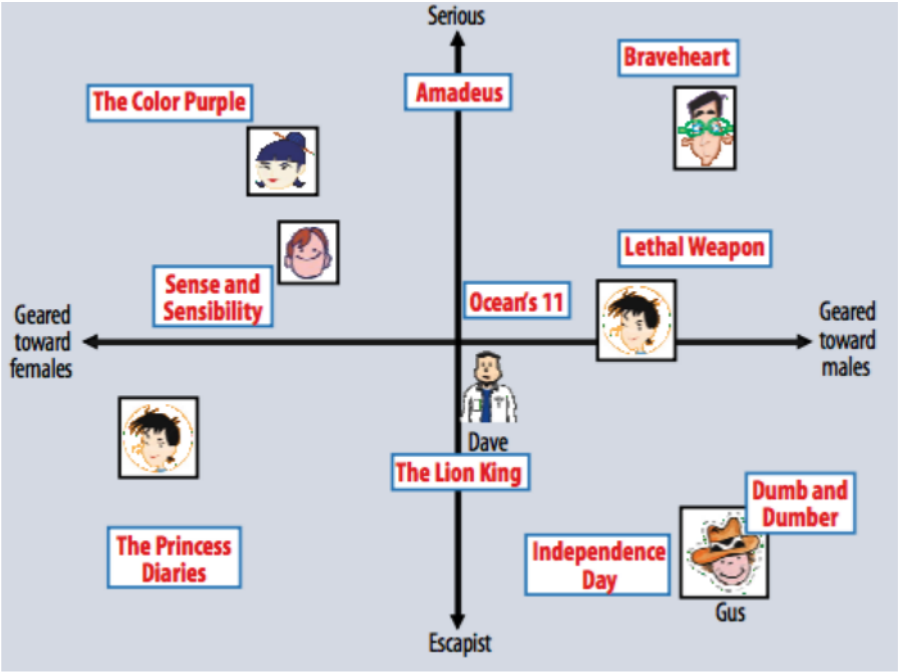
After the data has been pre-processed we need to start the model building process. **Matrix Factorisation** is a commonly used technique in collaborative filtering although there are other methods also like **Neighbourhood methods**. Here are the steps involved:

- **Factorize the user-item matrix to get 2 latent factor matrices — user-factor matrix and item-factor matrix.**

The user ratings are features of the movies that are generated by humans. These features are directly observable things that we assume are important. However, there are also a certain set of features which are not directly observable but are also

important in rating predictions. These set of hidden features are called **Latent features**.

- Assume that both movies and users live in some **low-dimensional space** describing their properties
- **Recommend** a movie based on its **proximity** to the user in the latent space



Figures from Koren et al. (2009)

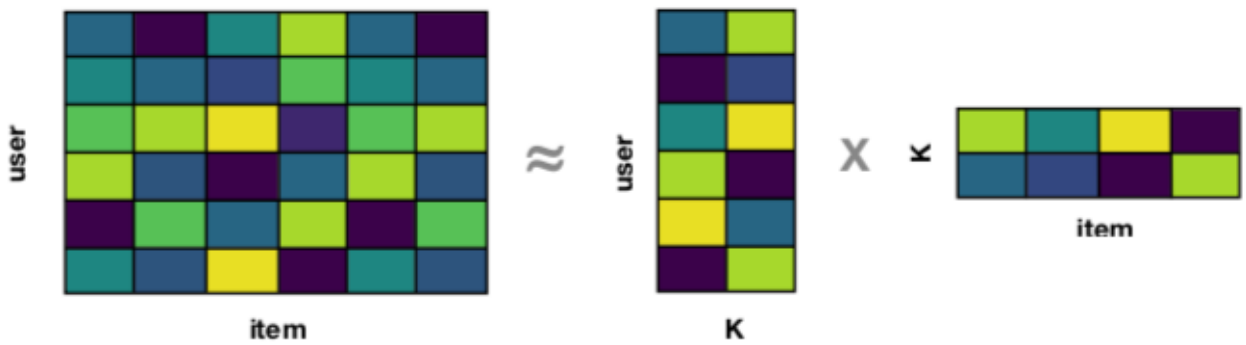
23

A simplified illustration of the latent factor approach

The Latent Features can be thought of as features that underlie the interactions between users and items. Essentially, we do not explicitly know what each latent feature represents but it can be assumed that one feature might represent that a user likes a comedy movie and another latent feature could represent that user likes animation movie and so on.

- **Predict missing ratings from the inner product of these two latent matrices.**

$$X_{mn} \approx P_{mk} \times Q_{nk}^{T} = \hat{X}$$



Source

Latent factors here are represented by **K.** This reconstructed matrix populates the empty cells in the original user-item matrix and so the unknown ratings are now known.

But how do we implement the Matrix Factorisation shown above? Well, it turns out that there are a number of ways of doing that by using one of the methods below:

- **Alternating Least Squares(ALS)**
- **Stochastic Gradient Descent(SGD)**
- **Singular Value Decomposition(SVD)**

3. Hyperparameter Optimisation

Before tuning the parameters we need to pick up an evaluation metric. A popular evaluation metric for recommenders is **Precision at K** which looks at the top k recommendations and calculates what proportion of those recommendations were actually relevant to a user.

Therefore, our goal is to find the parameters that give the best **precision at K** or any other evaluation metric that one wants to optimize. Once the parameters are found, we can re-train our model to get our predicted ratings and we can use these results to generate our recommendations.

4. Post Processing

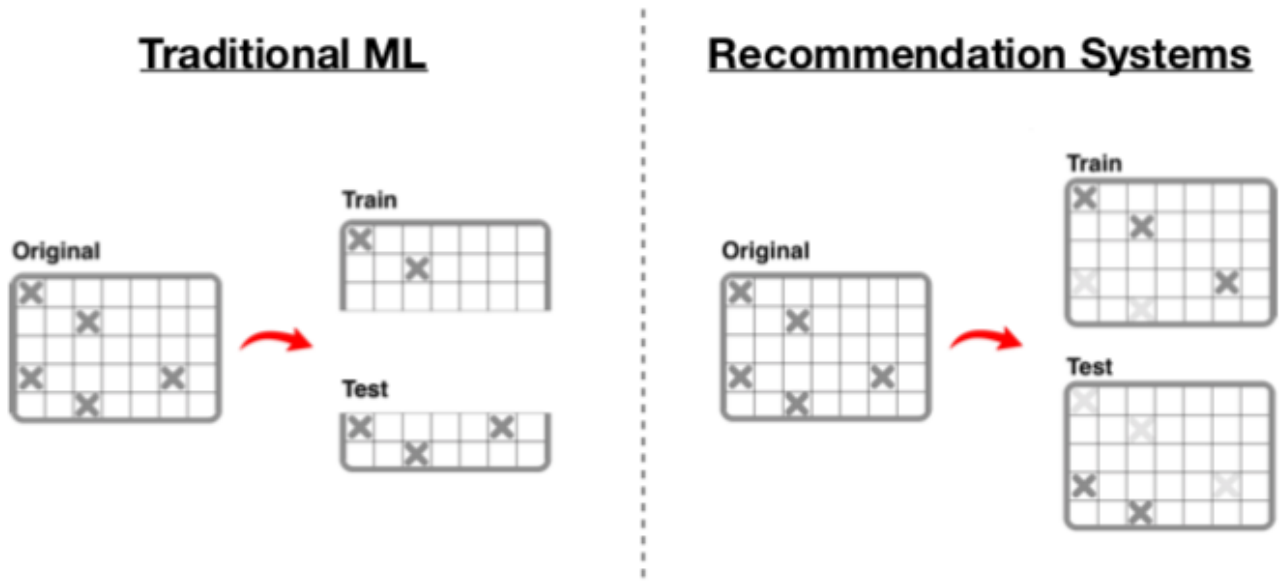
We can then sort all of the predicted ratings and get the top N recommendations for the user. We would also want to exclude or filter out items that a user has already interacted with before. In the case of movies, there is no point in recommending a movie that a user has previously watched or disliked earlier.

5. Evaluation

We have already covered this before but let's talk in a bit more detail here. The best way to evaluate any recommender system is to test it out in the wild. Techniques like **A/B testing** is the best since one can get actual feedback from real users. However, if that's not possible, then we have to resort to some offline evaluation.

In traditional machine learning, we split our original dataset to create a training set and a validation set. This, however, doesn't work for recommender models since the model won't work if we train all of our data on a separate user population and validate

it on another. So for recommenders, we actually mask some of the known ratings in the matrix randomly. We then predict these masked ratings through machine learning and then compare the predicted rating with the actual rating.



Evaluating recommenders Offline

Earlier we talked about Precision as an evaluation metric. Here are some of the others that can be used.

$$RMSE = \sqrt{\frac{\Sigma_{i=1}^{N}(y - \hat{y})^2}{N}} \qquad precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN} \qquad F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

. . .

Python Libraries

A number of Python libraries are available that are specifically created for recommendation purposes. Here are the most popular ones:

- **Surprise**: A Python scikit building and analyzing recommender systems.
- **Implicit**: Fast Python Collaborative Filtering for Implicit Datasets.
- **LightFM**: Python implementation of a number of popular recommendation algorithms for both implicit and explicit feedback.
- **pyspark.mlib.recommendation**: Apache Spark's Machine Learning API.

. . .

Conclusion

In this article, we discussed the importance of recommendations in a way of narrowing down our choices. We also walked through the process of designing and building a recommendation system pipeline. Python actually makes this process simpler by giving access to a host of specialised libraries for the purpose. Try using one to build your own personalised recommendation engine.

. . .

References

- An introduction to recommendation systems in machine learning
- How to Design and Build a Recommendation System Pipeline in Python (Jill Cates)

Machine Learning Data Science Recommender Systems Towards Data Science

Artificial Intelligence

About Help Legal