Prototyping a Recommender System Step by Step Part 1: KNN Item-Based Collaborative Filtering

Kevin Liao Follow
Nov 10, 2018 · 9 min read

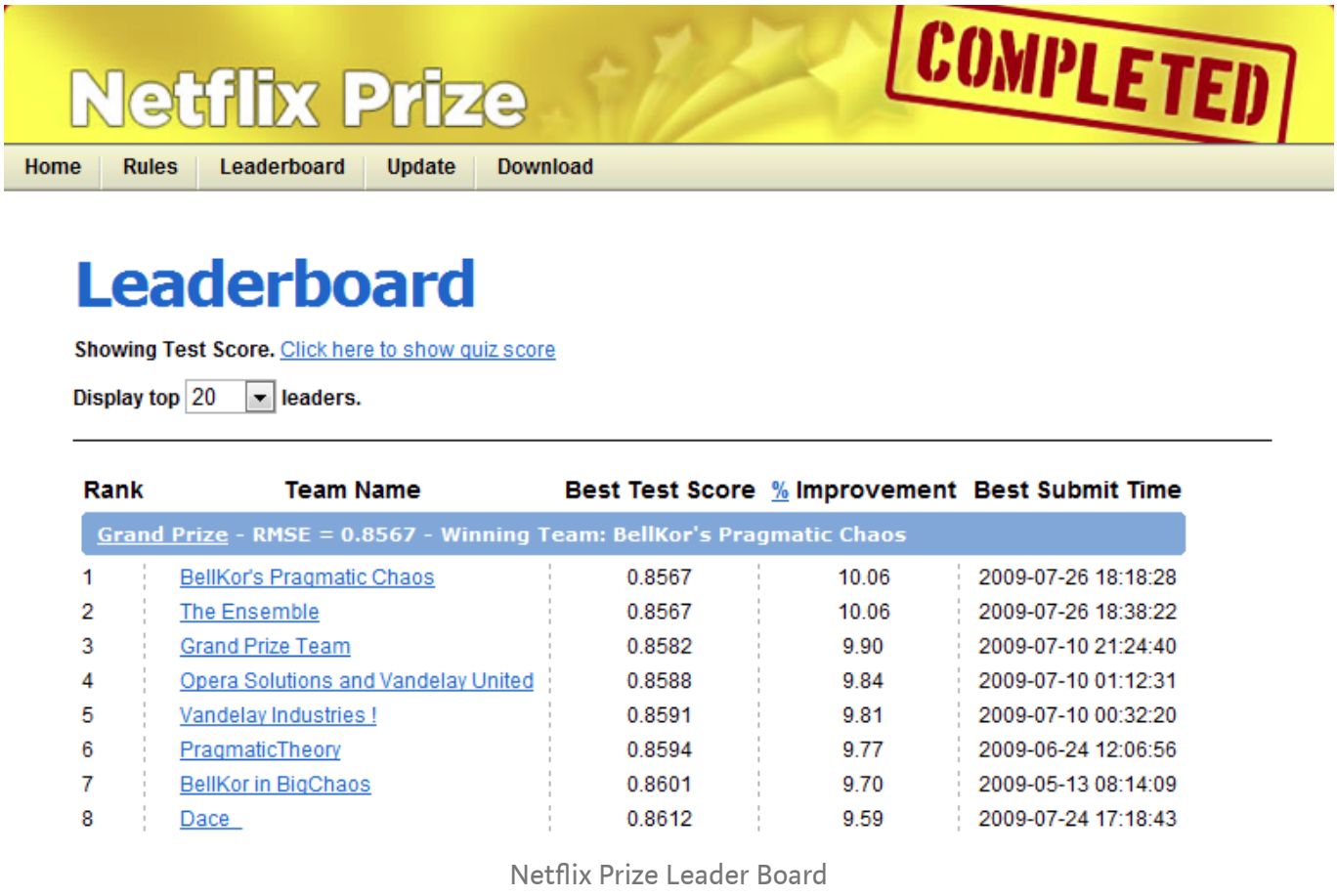
Movie Recommender Systems

Part 2 of *recommender systems* can be found *here*

Recommender Systems

Most internet products we use today are powered by recommender systems. Youtube, Netflix, Amazon, Pinterest, and long list of other internet products all rely on recommender systems to filter millions of contents and make personalized

recommendations to their users. Recommender systems are well-studied and proven to provide tremendous values to internet businesses and their consumers. In fact, I was shock at the news that Netflix awarded a $1 million prize to a developer team in 2009, for an algorithm that increased the accuracy of the company's recommendation system by 10%.

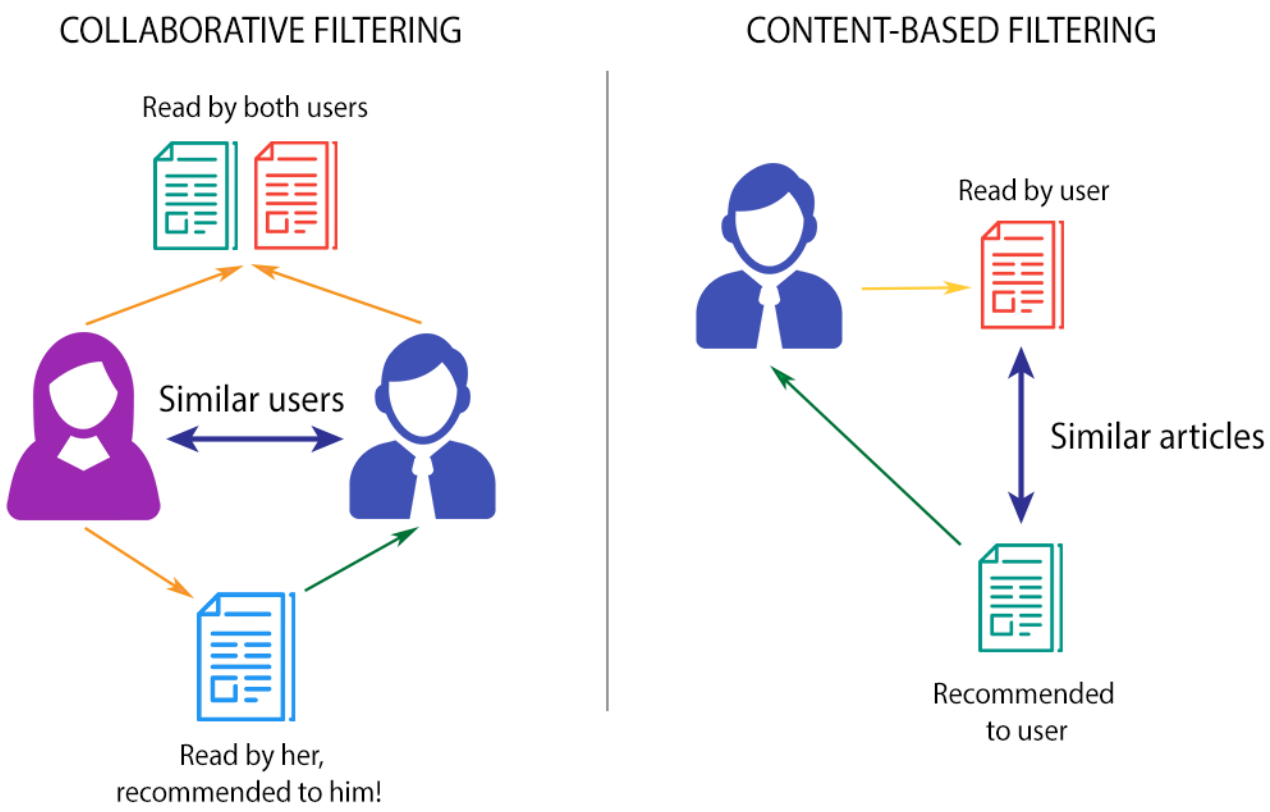


Netflix Prize Leader Board

Although recommender systems are the secret source for those multi-billion businesses, prototyping a recommender system can be very low cost and doesn't require a team of scientists. You can actually develop your own personalized recommender for yourself. It only takes some basic machine learning techniques and implementations in Python. In this post, we will start from scratch and walk through the process of how to prototype a minimum viable movie recommender.

Approaches

Recommender systems can be loosely broken down into three categories: **content based systems**, **collaborative filtering systems**, and **hybrid systems** (which use a combination of the other two).



An Overview of Recommendation Systems

Content based approach utilizes a series of discrete characteristics of an item in order to recommend additional items with similar properties.

Collaborative filtering approach builds a model from a user's past behaviors (items previously purchased or selected and/or numerical ratings given to those items) as well as similar decisions made by other users. This model is then used to predict items (or ratings for items) that the user may have an interest in.

Hybrid approach combines the previous two approaches. Most businesses probably use hybrid approach in their production recommender systems.

In this post, we are going to start with the most common approach, **collaborative filtering**, with a simple vanilla version. In the future posts, we will develop more advanced and sophisticated methods to further improve our recommender's performance and scalability.

Let's Build A Movie Recommender

I love watching movies so I decided to build a movie recommender. It will be so cool to see how well my recommender knows my movie preferences. We will go over our movie datasets, ML model choices, how to evaluate our recommender, and lastly I will give some pros and cons about this approach.

Data

Sometimes it can be very hard to find a good dataset to start with. However, I still encourage you to find interesting datasets and build your own recommender. I found there are some good datasets on this *page*. Besides building a movie recommender, building a food or dating recommender can be very fun too. Just a thought!

To build a movie recommender, I choose MovieLens Datasets. It contains 27,753,444 ratings and 1,108,997 tag applications across 58,098 movies. These data were created by 283,228 users between January 09, 1995 and September 26, 2018. The ratings are on a scale from 1 to 5.

We will use only two files from MovieLens datasets: `ratings.csv` and `movies.csv`. Ratings data provides the ratings of movies given by users. There are three fields in each row: `['userId', 'movieId', 'rating']`. Each row can be seen as a record of interaction between a user and a movie. Movies data provides a movie title and genres for each `'movieId'` in Ratings data.

```
import os
import pandas as pd

# configure file path
data_path = os.path.join(os.environ['DATA_PATH'], 'MovieLens')
movies_filename = 'movies.csv'
ratings_filename = 'ratings.csv'

# read data
df_movies = pd.read_csv(
    os.path.join(data_path, movies_filename),
    usecols=['movieId', 'title'],
    dtype={'movieId': 'int32', 'title': 'str'})

df_ratings = pd.read_csv(
    os.path.join(data_path, ratings_filename),
    usecols=['userId', 'movieId', 'rating'],
```

```
    dtype={'userId': 'int32', 'movieId': 'int32', 'rating':
'float32'})
```

Let's take a quick look at the two datasets: Movies and Ratings

```
In [7]: df_movies.head()
```

Out[7]:

	movieId	title
0	1	Toy Story (1995)
1	2	Jumanji (1995)
2	3	Grumpier Old Men (1995)
3	4	Waiting to Exhale (1995)
4	5	Father of the Bride Part II (1995)

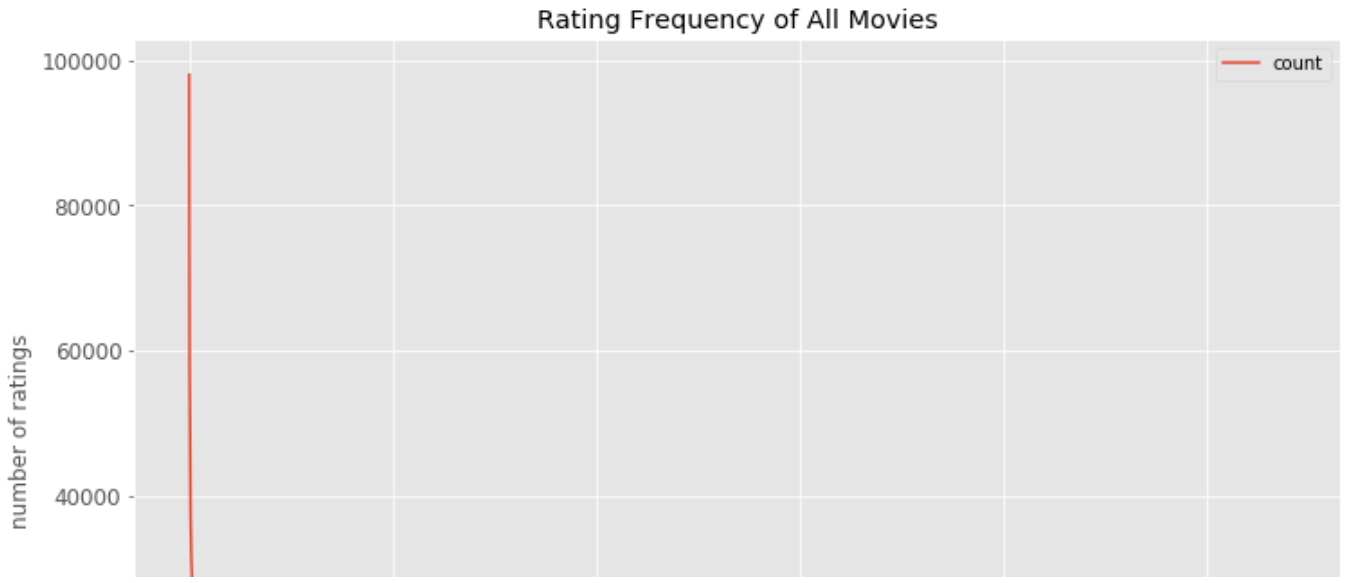
```
In [8]: df_ratings.head()
```

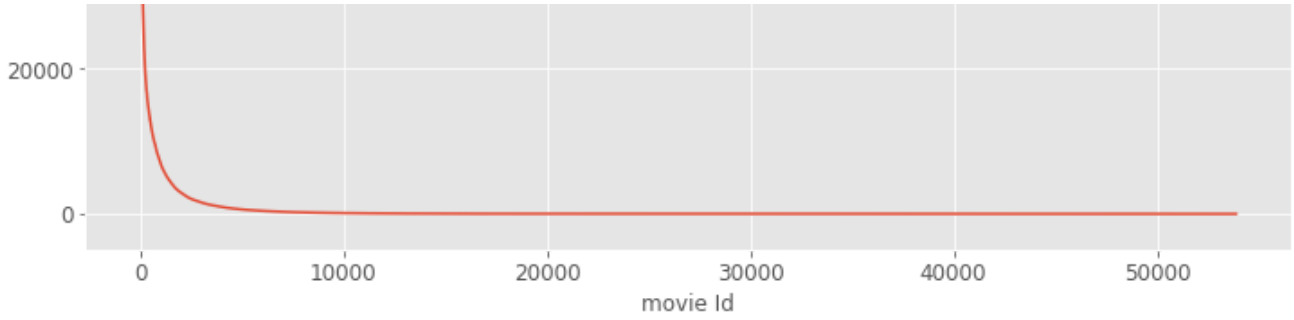
Out[8]:

	userId	movieId	rating
0	1	307	3.5
1	1	481	3.5
2	1	1091	1.5
3	1	1257	4.5
4	1	1449	4.5

Data Filtering

In a real world setting, data collected from explicit feedbacks like movie ratings can be very sparse and data points are mostly collected from very popular items (movies) and highly engaged users. Large amount of less known items (movies) don't have ratings at all. Let's see plot the distribution of movie rating frequency.





Rating Frequency is a "long tail" distribution. Only a small fraction of the items are rated frequently. Such items are referred to as popular items. The vast majority of items are rated rarely.

If we zoom in or plot it on a log scale, we can find out that only about 13,500 out of 58,098 movies received ratings by more than 100 users and the majority rest are much less known with little or no user-interactions. These sparse ratings are less predictable for most users and highly sensitive to an individual person who loves the obscure movie, which makes the pattern very noisy.

Most models make recommendations based on user rating patterns. To remove noisy pattern and avoid "memory error" due to large datasets, we will filter our dataframe of ratings to only popular movies. After filtering, we are left with 13,500 movies in the Ratings data, which is enough for a recommendation model.

Modeling

Collaborative filtering systems use the actions of users to recommend other movies. In general, they can either be user-based or item-based. **Item based approach** is usually preferred over **user-based approach**. User-based approach is often harder to scale because of the dynamic nature of users, whereas items usually don't change much, and item based approach often can be computed offline and served without constantly re-training.

To implement an **item based collaborative filtering,** KNN is a perfect go-to model and also a very good baseline for recommender system development. But what is the KNN? **KNN** is a **non-parametric, lazy** learning method. It uses a database in which the data points are separated into several clusters to make inference for new samples.

KNN does not make any assumptions on the underlying data distribution but it relies on **item feature similarity**. When KNN makes inference about a movie, KNN will calculate

the “distance” between the target movie and every other movie in its database, then it ranks its distances and returns the top K nearest neighbor movies as the most similar movie recommendations.

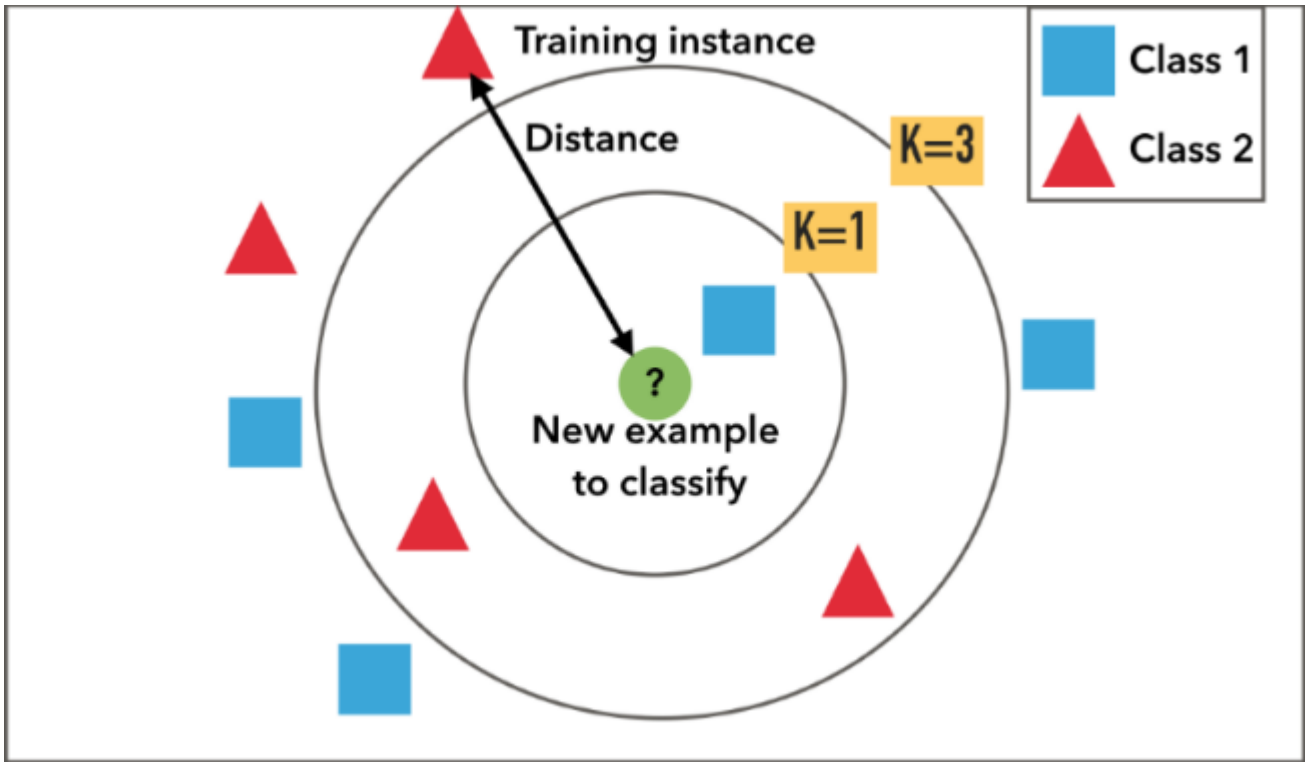


Illustration of how KNN makes classification about new sample

Wait, but how do we feed the dataframe of ratings into a KNN model? First, we need to transform the dataframe of ratings into a proper format that can be consumed by a KNN model. We want the data to be in an `m x n` array, where `m` is the number of movies and `n` is the number of users. To reshape dataframe of ratings, we’ll `pivot` the dataframe to the wide format with movies as rows and users as columns. Then we’ll fill the missing observations with `0` s since we’re going to be performing linear algebra operations (calculating distances between vectors). Let’s call this new dataframe a “dataframe of movie features”.

Our dataframe of movie features is an extremely sparse matrix with a shape of `13,500 x 113,291`. We definitely don't want to feed the entire data with mostly `0` s in `float32` datatype to KNN. For more efficient calculation and less memory footprint, we need to transform the values of the dataframe into a **scipy sparse matrix**.

```
from scipy.sparse import csr_matrix
```

```
# pivot ratings into movie features
df_movie_features = df_ratings.pivot(
    index='movieId',
    columns='userId',
    values='rating'
).fillna(0)

# convert dataframe of movie features to scipy sparse matrix
mat_movie_features = csr_matrix(df_movie_features.values)
```

```
In [31]: df_movie_features.head(5)
```

Out[31]:

userId	4	5	10	14	15	18	19	26	31	34	...	283199	283204	283206	283208	283210	283215	283219	283222	283224	283228
movieId																					
1	4.0	0.0	5.0	4.5	4.0	0.0	0.0	0.0	5.0	0.0	...	5.0	0.0	0.0	4.5	0.0	4.0	4.0	0.0	0.0	4.5
2	4.0	0.0	0.0	4.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	4.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0	4.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0	0.0	0.0

Now our training data has a very high dimensionality. KNN's performance will suffer from **curse of dimensionality** if it uses "euclidean distance" in its objective function. **Euclidean distance** is unhelpful in high dimensions because all vectors are almost equidistant to the search query vector (target movie's features). Instead, we will use **cosine similarity** for nearest neighbor search. There is also another popular approach to handle nearest neighbor search in high dimensional data, **locality sensitive hashing**, which we won't cover in this post.

Let's Make Some Movie Recommendations

After we preprocessed the data and transformed the dataframe of ratings into a movie features scipy sparse matrix, we need to configure our KNN model with proper hyper-params:

```
from sklearn.neighbors import NearestNeighbors

model_knn = NearestNeighbors(metric='cosine', algorithm='brute',
n_neighbors=20, n_jobs=-1)
```

Finally we can make some movie recommendations for ourselves. Let's implement a `make_recommendations` method in our KNN recommender.

```python
def make_recommendations(self, fav_movie, n_recommendations):
    """
    make top n movie recommendations
    Parameters
    ----------
    fav_movie: str, name of user input movie
    n_recommendations: int, top n recommendations
    """
    # get data
    movie_user_mat_sparse, hashmap = self._prep_data()
    # get recommendations
    raw_recommends = self._inference(
        self.model, movie_user_mat_sparse, hashmap,
        fav_movie, n_recommendations)
    # print results
    reverse_hashmap = {v: k for k, v in hashmap.items()}
    print('Recommendations for {}:'.format(fav_movie))
    for i, (idx, dist) in enumerate(raw_recommends):
        print('{0}: {1}, with distance '
              'of {2}'.format(i+1, reverse_hashmap[idx], dist))
```

make recommendations knn.py hosted with ♡ by GitHub view raw

snippet of our final step in KNN recommender's implementation

This snippet demos our `make_recommendations` method in our recommender's implementation. Please find the detailed source code for recommender application in my **GitHub Repo**.

If you go to my source code page, you see that I put together a KNN recommender system in a script as a small python application. I parameterized my recommender application and exposed two options, `movie_name` and `top_n`, for users to play around. Now I want to ask my recommender to propose top 10 most similar movies to "Iron Man". So we can run below bash commend inside terminal (linux/mac): (instruction of commands can be found *here*)

```
python src/knn_recommender.py --movie_name "Iron Man" --top_n 10
```

```
You have input movie: Iron Man
Found possible matches in our database: ['Iron Man (2008)', 'Iron Man 3 (2013)', 'Iron Man 2 (2010)']

Recommendation system start to make inference
```

```
......

It took my system 1.39s to make inference

Recommendations for Iron Man:
1: Bourne Ultimatum, The (2007), with distance of 0.42213231325149536
2: Sherlock Holmes (2009), with distance of 0.4194795489311218
3: Inception (2010), with distance of 0.3934664726257324
4: Avatar (2009), with distance of 0.3836246728897095
5: WALL·E (2008), with distance of 0.3835691213607788
6: Star Trek (2009), with distance of 0.37533360719680786
7: Batman Begins (2005), with distance of 0.37033611536026
8: Iron Man 2 (2010), with distance of 0.3703038692474365
9: Avengers, The (2012), with distance of 0.35819315910339355
10: Dark Knight, The (2008), with distance of 0.3013307452201843
```

List of movie recommendations based on my favorite movie: "Iron Man"

Hooray!! Our recommender system actually works!! Now we have our own movie recommender.

Some Thoughts

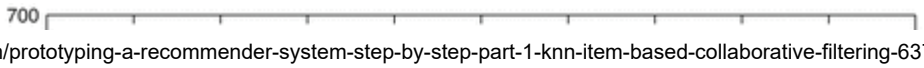
At first glance, my recommender seems very impressive. I love all of its recommended movies. But if we really think about them, they were all very popular movies during the same time period as "Iron Man" in 2008. It's very likely that people who watched "Iron Man" at that time probably also watched some of the recommended movies. This list of movies are not only popular during the same era but also share very similar genres and topics.

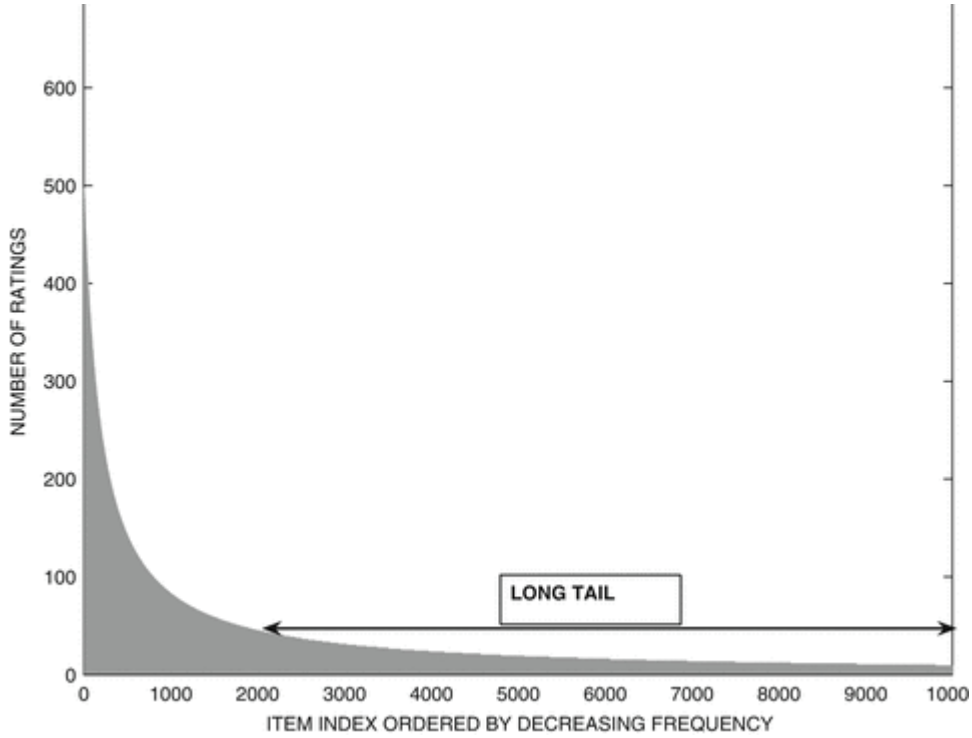
As a movie lover, I am probably looking for movies that I haven't watched or movies with different topics. Recommending movies with diverse topics allow users to explore different tastes and keeps user engaged with the recommender product. On the other hand, lack of diversity will make users get bored and less engaged with the product.

So we just effectively identify there are two shortcomings in **item based collaborative filtering**:

1. popularity bias: recommender is prone to recommender popular items
2. item cold-start problem: recommender fails to recommend new or less-known items because items have either none or very little interactions

Recall the plot of distribution of movie rating frequency:





"Long Tail" Property in Rating Frequency Distribution

Only a very small fraction of movies have lots of user interactions while a "long-tail" of movies don't have. In a business setting, high-frequency items tend to be relatively competitive items with little profit for the merchant. On the other hand, the lower frequency items have larger profit margins.

How do we improve our movie recommender system to solve above two shortcomings? We will cover a more sophisticated method to improve movie recommender in next post: **Prototyping a Recommender System Step by Step Part 2: Alternating Least Square (ALS) Matrix Factorization in Collaborative Filtering**

Summary

In this post, we briefly covered three approaches in recommender system: content-based, collaborative filtering, and hybrid. We learned about how to prototype an **item based collaborative filtering** in **KNN** with only a few steps! The **Jupyter Notebook version** for this blog post can be found *here*. If you want to play around my **source code**, you can find it *here*.

In my next post, we will cover more advanced topics in recommender systems and

leverage **Spark** to build a scalable recommender system. Stay tuned! Until then, have fun with machine learning and recommenders!

. . .

Like what you read? Checkout more data science / machine learning projects at my Github: ***Kevin's Data Science Portfolio***

Machine Learning Data Science Python Recommendation System Movies