Recommender Systems

Introduction

> "Example is not the main thing in influencing others. It is the only thing." Albert Schweitzer

Today, we regularly interact with recommender systems using digital services and applications, from online retailers to movie and TV services and news aggregators. These are algorithms that make suggestions about what a user may like, such as a specific movie. They are functions that take information about a user's preferences as input to generate a prediction about the rating a user would give of the items under evaluation and predict how they would rank a set of items individually or as a bundle. Recommender systems collect, curate, and act upon vast personal data. Inevitably, they shape individual experiences of digital environments and social interactions.

The basic principle of recommendation methods is based on the assumption (or fact?) that significant dependencies exist between the user and the items. For example, a user interested in a historical Second World War documentary is more likely to be interested in another historical documentary or an educational program rather than an action movie. In many cases, various categories of items may show significant correlations, which can be leveraged to make more accurate recommendations based on this similarity.

Alternatively, the dependencies may be present at the finer granularity of individual items rather than categories. These dependencies can be learned in a data-driven approach from the ratings matrix, and the resulting model is used to make predictions for target users. The larger the number of rated items that are available for a user, the easier it is to make robust predictions about the future behavior of the user. A variety of learning models can be used to accomplish this task. For example, the collective buying or rating behavior of various users can be leveraged to create cohorts of similar users interested in similar products. The interests and actions of these cohorts can be leveraged to make recommendations to individual members of these cohorts.

These above approaches can be referred to as neighborhood models or referred to as collaborative filtering. The term *collaborative filtering* refers to the use of ratings from multiple users in a collaborative way to predict missing ratings. In practice, recommender systems can be more complex and data-enriched. For example, in content-based recommender systems, the content plays a primary role in the recommendation process, in which the ratings of users and the attribute descriptions of items are leveraged to make predictions. The basic idea is that user interests can be modeled based on properties (or attributes) of the items they have rated or accessed in the past. A different framework is knowledge-based systems, in which users interactively specify their interests, and the user specification is combined with domain knowledge to provide recommendations. In advanced models, contextual data, such as temporal information, external knowledge, location information, social information, or network information, can also be used.

In this module, we will build a recommender system on the MovieLens dataset, which is composed of five sub-datasets of (1.) movies, (2.) movie ratings, (3.) movie tags, (4.) tags to describe movies by users, and (5.) movie tag scores.

Distance/Similarity Measures

A distance metric, $d : \mathbf{X} \times \mathbf{X} \longrightarrow [0, \infty) \in \mathbb{R}^+$, satisfies the following:

Property	Name
$d(x,y) \geq 0$	non-negative
$d(x,y) = 0 \iff x = y$	identity
$d(x,y) = d(y,x)$	symmetry
$d(x,y) \leq d(x,z) + d(z,y)$	triangle inequality

Define d_{ij} = the distance between the user i and j, where there are N users, M movies, and $\mathbf{X}$ as the movie ratings of users as $x \in \mathbb{R}^{N \text{x} M}$. Define I_i as the set of movies user i rated, where $i \in \{1, 2, \ldots, N\}$ and $I_i \in \mathbb{P}(\{1, 2, \ldots, M\})$.

Name	Distance
Jaccard index	$d_{ij} = \frac{x_i \cap x_j}{x_i \cup x_j}$
Cosine θ	$d_{ij} = \cos(\theta) = \frac{x_i \cdot x_j}{\|x_i\|\|x_j\|}$
Pearson	$d_{ij} = \frac{\sum_{k \in I_i \cap I_j}(x_{ik}-\mu_i)(x_{jk}-\mu_j)}{\sqrt{\sum_{k \in I_i \cap I_j}(x_{ik}-\mu_i)^2}\sqrt{\sum_{k \in I_i \cap I_j}(x_{jk}-\mu_j)^2}}$
Manhattan, L_1	$d_{ij} = \sum_{k \in I_i \cap I_j} \|x_{ik} - x_{jk}\|$
Euclidean, L_2	$d_{ij} = \sqrt{\sum_{k \in I_i \cap I_j}(x_{ik} - x_{jk})^2}$
Chebyshev	$d_{ij} = \max_{k \in I_i \cap I_j} \|x_{ik} - x_{jk}\|$

The following table can be used as a guideline to apply distance metrics.

Data	Example	Similarity
Unary (only ♡) Transactions of movies liked	u$_1$ ♡ mov$_2$ u$_2$ likes mov$_1$ u$_3$ likes mov$_3$	Jaccard
Binary (♡, ♠) Likes and dislikes	u$_1$ ♡ mov$_2$ u$_2$ ♠ mov$_1$ u$_2$ dislikes mov$_2$ u$_3$ likes mov$_3$	Jaccard
Quantitative and Ordered ratings	u$_1$ ⋆⋆⋆ to mov$_2$ u$_2$ 5-star to mov$_1$ u$_2$ 1-star to mov$_2$ u$_3$ 3-star to mov$_3$	Pearson Cosine

---

A Note on Representation Learning

Representation learning is about learning useful features or representations of data that simplify the process of downstream tasks such as classification, regression, or clustering. Unlike traditional feature engineering, where features are manually designed, representation learning automates this process, often uncovering high-dimensional, non-linear structures in data. Techniques such as autoencoders, convolutional neural networks, and transformers are strong in extracting meaningful features from raw inputs like images, text, and audio. By leveraging these learned representations, models become more robust, generalizable, and effective at handling complex data.

Representation learning uses methods from linear algebra, optimization, and probability theory. The process is the mapping of raw data $x \in \mathbb{R}^n$ to a latent space $z \in \mathbb{R}^m$ (where typically $m < n$), preserving information essential for the task. This is achieved by solving optimization problems like minimizing a loss function, $\mathcal{L}(x, z)$, which measures reconstruction error or task-specific performance. Linear transformations (e.g., matrix multiplication) combined with non-linear activations (e.g., ReLU or sigmoid) help model complex relationships. In a probabilistic setting, methods like Variational Autoencoders (VAEs) use the Kullback-Leibler divergence to enforce prior constraints on latent variables. Similarly, the interplay of eigenvalues and eigenvectors in techniques like PCA and singular value decomposition (SVD) reveals data's principal components, underscoring the core mathematical principles of representation learning.

Representation learning can provide the mathematical foundation for recommender systems by enabling the decomposition and transformation of user-item interaction data into a shared latent space. Formally, given a user-item interaction matrix $R \in \mathbb{R}^{m \times n}$, where m and n represent the number of users and items, respectively, matrix factorization techniques approximate R as the product of two low-dimensional matrices: $U \in \mathbb{R}^{m \times k}$ and $V \in \mathbb{R}^{n \times k}$, where $k \ll \min(m, n)$. The rows of U and V represent latent embeddings for users and items, respectively. The predicted interaction between a user i and item j is computed as $\hat{R}_{ij} = U_i \cdot V_j^\top$. Advanced methods like neural collaborative filtering extend this framework by using deep neural networks to learn non-linear mappings of user and item features into the latent space. Similarly, graph-based methods utilize representation learning to encode user-item interactions as edge weights in a graph and learn embeddings via techniques such as GNNs. These mathematical formulations enable recommender systems to uncover latent structures in the data, predict missing entries, and generalize to new interactions.

Dataset and Feature Exploration

- Examine the dataset, do sanity checks, extract indices
    - Original movie dataset is exported from a database

- Extract Genres
- Limit the number of users `USERS_N` for faster processing
- Use a threshold `RATINGS_N_MIN` to ease the pre-processing
    - movies with little information may not help
- `Ratings` matrix, `N` users, and `M` movies are stored in an order 2 tensor for faster processing
- Implement distance metrics
- Inspect distance metrics on users
    - What is the feature vector?

Let's see how we use distance/similarity metrics to find and rate movies.

In [3]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.dpi"] = 72
import numpy as np
import pandas as pd
import re
from collections import defaultdict
```

In [4]:
```
# A subset of the dataset to shorten the pipeline processing
USERS_N = 1000
```

In [5]:
```
# read datasets
df_movies = pd.read_csv('/EP_datasets/movielens_movie.csv')
df_ratings = pd.read_csv('/EP_datasets/movielens_rating.csv')
df_tag = pd.read_csv('/EP_datasets/movielens_tag.csv')

df_genome_tags = pd.read_csv('/EP_datasets/movielens_genome_tags.csv
df_genome_scores = pd.read_csv('/EP_datasets/movielens_genome_scores
```

In [6]:
```
df_movies.head()
```

Out[6]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

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

Out[7]:

	userId	movieId	rating	timestamp
0	1	2	3.5	2005-04-02 23:53:47
1	1	29	3.5	2005-04-02 23:31:16
2	1	32	3.5	2005-04-02 23:33:39
3	1	47	3.5	2005-04-02 23:32:07
4	1	50	3.5	2005-04-02 23:29:40

In [8]:

```
df_tag.head()
```

Out[8]:

	userId	movieId	tag	timestamp
0	18	4141	Mark Waters	2009-04-24 18:19:40
1	65	208	dark hero	2013-05-10 01:41:18
2	65	353	dark hero	2013-05-10 01:41:19
3	65	521	noir thriller	2013-05-10 01:39:43
4	65	592	dark hero	2013-05-10 01:41:18

In [9]:

```
df_genome_tags.head()
```

Out[9]:

	tagId	tag
0	1	007
1	2	007 (series)
2	3	18th century
3	4	1920s
4	5	1930s

In [10]:

```
df_genome_scores.head()
```

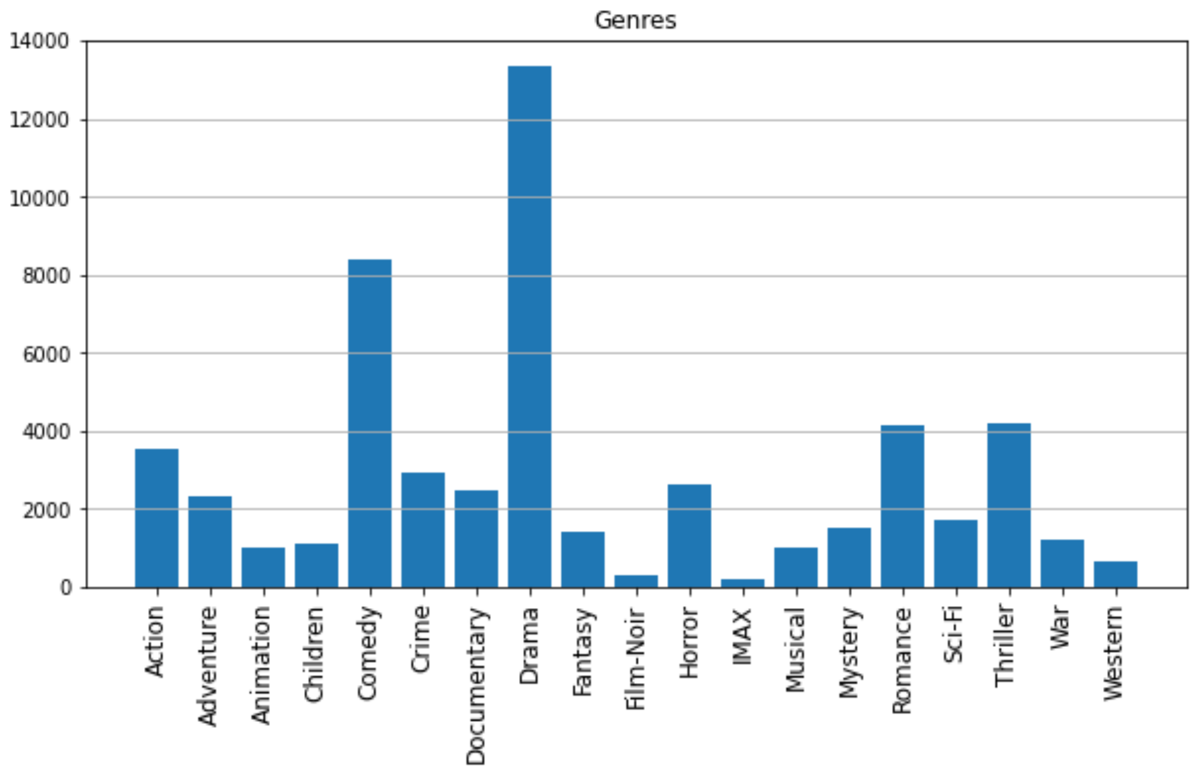
Out[10]:

	movieId	tagId	relevance
0	1	1	0.02500
1	1	2	0.02500
2	1	3	0.05775
3	1	4	0.09675
4	1	5	0.14675

In [11]:
```
print(f"Number of unique movies= {len(df_movies)}")
print(f"Number of unique users= {len(df_ratings.userId.unique())}")

print(f"Number of unique rated movies= {len(df_ratings.movieId.uniqu

# MovieLens rating system
print(f"Number of tags= {len(df_genome_tags)}")
print(f"Number of tagger users= {len(df_tag.userId.unique())} (who c
print(f"Number of created unrefined tags= {len(df_tag.tag.unique())}

Genres = defaultdict(int)
for genre in df_movies.genres:
    if 'no' not in genre:
        m = re.findall(r"([A-Za-z-]+)(?=\||$)", genre)
        if m:
            for gen in m:
                Genres[gen] += 1

print(f"Number of genres= {len(Genres.keys())}")
```

```
Number of unique movies= 27278
Number of unique users= 138493
Number of unique rated movies= 26744 (not all movies are rated by user
s)
Number of tags= 1128
Number of tagger users= 7801 (who created/assigned tags)
Number of created unrefined tags= 38644 (created tags, not-refined tag
s)
Number of genres= 19
```

In [12]:
```
# genres
gens = sorted(Genres.keys())

plt.figure(figsize=(10,5), dpi=72)
plt.bar(gens, [Genres[g] for g in gens])
plt.xticks(rotation=90, fontsize=12)
plt.grid(axis='y')
plt.title('Genres')
plt.show()
```



In [13]:
```
# explore ratings data
Rating_Min, Rating_Max = df_ratings.rating.min(), df_ratings.rating.r
print(Rating_Min, Rating_Max)
```

```
0.5 5.0
```

In [14]:
```
# movies index
movies = df_ratings.movieId.unique()
MoviesIndex = {mov:i for i, mov in enumerate(movies)}
MoviesIndexRev = {MoviesIndex[mov]:mov for mov in MoviesIndex.keys()

print(len(MoviesIndex.keys()), len(MoviesIndexRev.keys()))
```

```
26744 26744
```

In [15]:
```
%%time

# rating matrix N-by-M, users-by-movies

# N = len(df_ratings.userId.unique())
N = USERS_N  # Full problem as above
M = len(MoviesIndex.keys())
Ratings = np.zeros((N,M), dtype=np.float32)

for _, row in df_ratings.iterrows():
    if row.userId > USERS_N:  # Shorten the problem dataset
        break
    Ratings[row.userId-1, MoviesIndex[row.movieId]] = row.rating
```

```
CPU times: total: 4.28 s
Wall time: 4.28 s
```

In [16]:
```
# remove movies that have min number of ratings from the user list
RATINGS_N_MIN= 20

moviesnull = np.where(np.count_nonzero(Ratings, axis=0) <= RATINGS_N
moviesnull = [MoviesIndexRev[_] for _ in moviesnull]

print(len(moviesnull))
```

```
25038
```

In [17]:
```
# movies index
movies = list(set(df_ratings.movieId.unique().tolist()) - set(movies
MoviesIndex = {mov:i for i, mov in enumerate(movies)}
MoviesIndexRev = {MoviesIndex[mov]:mov for mov in MoviesIndex.keys()

print(len(MoviesIndex.keys()), len(MoviesIndexRev.keys()))
```

```
1706 1706
```

In [18]:
```
# update the rating matrix
M = len(MoviesIndex.keys())
Ratings = np.zeros((N,M), dtype=np.float32)

for _, row in df_ratings.iterrows():
    if row.userId > USERS_N:  # Shorten the problem dataset
        break
    if row.movieId in MoviesIndex:  # Skip low count rated movies
        Ratings[row.userId-1, MoviesIndex[row.movieId]] = row.rating

print(Ratings.shape)  # number of users, number of movies
```

```
(1000, 1706)
```

In [19]:
```
# sanity
moviesnull = np.where(np.count_nonzero(Ratings, axis=0) <= RATINGS_N

print(len(moviesnull))
```

```
0
```

In [18]:
```
# distance metrics

def sim_jaccard(_a:np.ndarray, _b:np.ndarray)->float:
    ix = np.where((_a>0.) & (_b>0.))
    ixd = np.where((_a>0.) | (_b>0.))
    return ix[0].shape[0] / ixd[0].shape[0]

def sim_costeta(_a:np.ndarray, _b:np.ndarray)->float:
    from numpy.linalg import norm
    assert _a.shape == _b.shape, 'require vectors same shape'
    return np.dot(_a, _b) / (norm(_a)*norm(_b))

def sim_pearson(_a:np.ndarray, _b:np.ndarray)->float:
    ix = np.where((_a>0.) & (_b>0.))
    if ix[0].shape[0] == 0:
        return 0.
    _ua, _ub = np.mean(_a[ix]), np.mean(_b[ix])
    num = (_a[ix]-_ua) @ (_b[ix]-_ub).T
    den  = np.sqrt(np.sum((_a[ix]-_ua)**2)) * np.sqrt(np.sum((_b[ix]-
    if den == 0.:
        return 0.
    return num/den

def dist_L1(_a:np.ndarray, _b:np.ndarray)->float:
    ix = np.where((_a>0.) & (_b>0.))
    if ix[0].shape[0] == 0:
        return 1.
    return np.sum(np.abs(_a[ix]-_b[ix])) / ix[0].shape[0]

def dist_L2(_a:np.ndarray, _b:np.ndarray)->float:
    ix = np.where((_a>0.) & (_b>0.))
    if ix[0].shape[0] == 0:
        return 1.
    return np.sqrt(np.sum((_a[ix]-_b[ix])**2)) / ix[0].shape[0]

def dist_cheby(_a:np.ndarray, _b:np.ndarray)->float:
    ix = np.where((_a>0.) & (_b>0.))
    if ix[0].shape[0] == 0:
        return 1.
    return np.max(np.abs(_a[ix]-_b[ix])) / (Rating_Max-Rating_Min)

DistanceMetrics = [(sim_jaccard, 'Jaccard (sim)'), (sim_costeta, 'Co
                   (dist_L1, 'L1 (dist)'), (dist_L2, 'L2 (dist)'), (
```

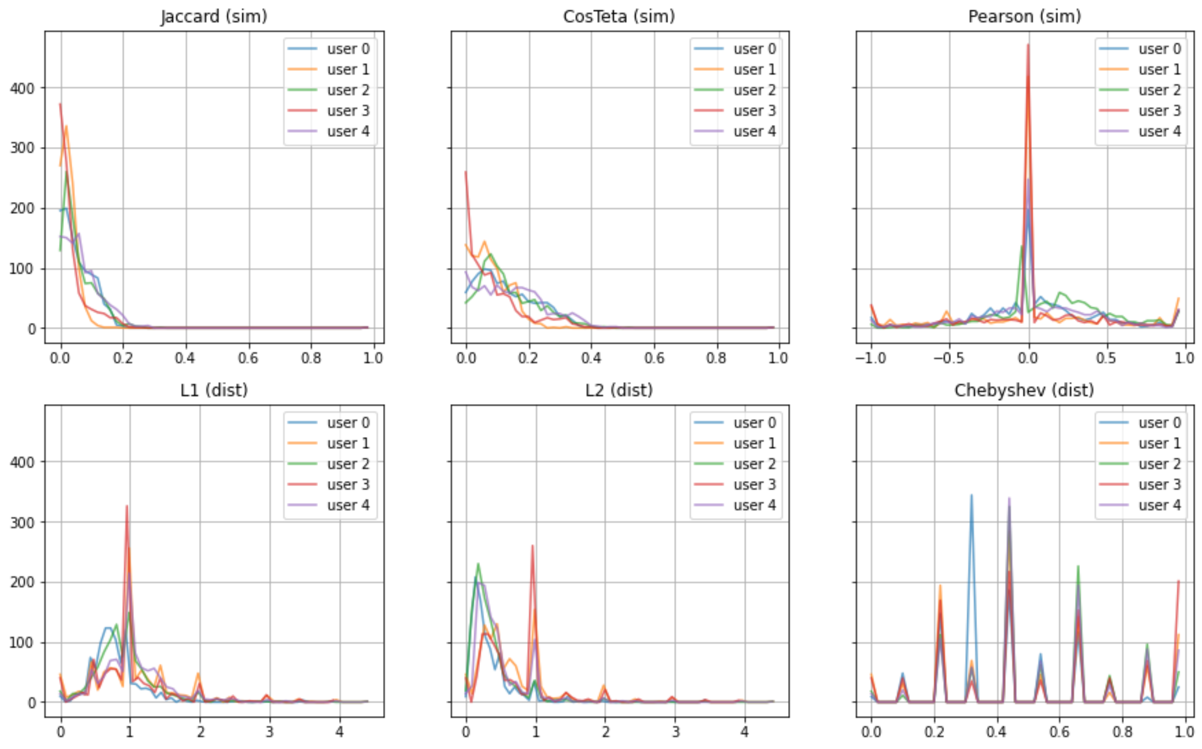
In [19]:

```
# distances
USER_A, USER_B = 0, 1

x1, x2 = Ratings[USER_A], Ratings[USER_B]

for metric in DistanceMetrics:
    print(f'{metric[1]:<17} {metric[0](x1, x2):.3f}')
```

```
Jaccard (sim)     0.053
CosTeta (sim)     0.133
Pearson (sim)     -0.069
L1 (dist)         1.167
L2 (dist)         0.412
Chebyshev (dist)  0.444
```

In [20]:

```
Users = [0, 1, 2, 3, 4]

# example, user Users to other users histogram for these six distanc
fig, ax = plt.subplots(nrows=2, ncols=3, sharex='none', sharey='all'

for i, metric in enumerate(DistanceMetrics):
    for u in Users:
        d = [metric[0](Ratings[u], Ratings[_]) for _ in range(N)]
        h = np.histogram(d, bins=50)
        ax.flatten()[i].plot(h[1][:-1], h[0], label=f'user {u:d}', a
    ax.flatten()[i].set_title(metric[1])
    ax.flatten()[i].legend(loc='upper right')
    ax.flatten()[i].grid()
```



Collaborative Filtering

Online retailers, bookshops, and libraries sometimes have recommendations saying, "If you liked this popular book X, then you may also try this (not-so-much-popular) book Y." These remarks are directed toward a large group of people and often work well, and they are similar to a filtered chart. A more robust approach is to create individualized content lists or, at least, lists for small groups of like-minded users. A database of preferences for items by users to predict additional topics or products a new user might like.

User-based Neighborhood-based Filtering

1. For a particular user, compute the similarity to the *other users*
2. Order *users* by similarity
3. Select a *neighborhood of users* close to the *active user*
4. Compute the predicted rating from that neighborhood

Item-based Filtering

1. For a particular item, compute the similarity to the *other items*
2. Order *items* by similarity
3. Select a *neighborhood of items* close to the *active item*
4. Compute the predicted rating from that neighborhood

The `Ratings` matrix will be used in both filters to compute similarities.

In an alternative approach, the item similarities might be computed using the **tags** and **tag scoring**. This would be equivalent to **content-based** approach. Note that tags are computed by another pipeline or annotated/tagged by experts/users.

User-based Example

- Find out what a user would score a particular movie that he/she did not score (or watch)
  - Similarity will be based on users
- Use `sim_costeta` for similarity (can try other ones)
- The distance matrix is `N`-by-`N`
- Always use `float32` for speed

In [21]:
```
# User-based filtering
OUR_USER= 0
OUR_ITEM= 0  # pick an item which our user did not rate
OUR_USER_NEIGHBORHOOD= 50  # number of users similar to our user
```

In [22]:
```
# users who rated our item
ix_users = np.where(Ratings[:,OUR_ITEM]>0.)[0]
```

In [23]:
```
%%time

# generate a distance matrix - user
Dist_u = np.zeros((N,N), dtype=np.float32)
for i in range(N):
    for j in range(N):
        Dist_u[i,j] = sim_costeta(Ratings[i,:], Ratings[j,:])
```

```
CPU times: total: 3.83 s
Wall time: 3.83 s
```

In [24]:
```
# Pearson generates -1 to +1
d = np.abs(Dist_u[OUR_USER,ix_users])
ix_best = np.argsort(-d)[:OUR_USER_NEIGHBORHOOD]
```

```
users = ix_users[ix_best]
```

In [25]:
```
Dist_u[OUR_USER,users]
```

Out[25]:
```
array([0.40457195, 0.376088  , 0.3636198 , 0.36297017, 0.36068535,
       0.35957903, 0.35941103, 0.35755163, 0.3565974 , 0.34926307,
       0.34772184, 0.3475802 , 0.34442246, 0.34433424, 0.34246457,
       0.3385418 , 0.33742315, 0.3373577 , 0.3327701 , 0.33230612,
       0.3305398 , 0.3281719 , 0.3254474 , 0.32429686, 0.32400984,
       0.32379663, 0.32297403, 0.3224957 , 0.32155728, 0.32003078,
       0.3195294 , 0.31799585, 0.31655583, 0.31534153, 0.31273744,
       0.31241196, 0.3090288 , 0.30858234, 0.3075587 , 0.3062176 ,
       0.30284712, 0.30217868, 0.30106026, 0.29852498, 0.29817006,
       0.29561612, 0.29506028, 0.2946949 , 0.2922781 , 0.2906266 ],
      dtype=float32)
```

In [26]:
```
ratings = np.array([Ratings[_,OUR_ITEM] for _ in users])

print(f'The user not rated the item= {Ratings[OUR_USER, OUR_ITEM]:.1
print(f'Average rating in user-neighborhood= {ratings.mean():.1f}')
```

```
The user not rated the item= 0.0
Average rating in user-neighborhood= 3.9
```

Outcome: The user did not rate the item. Other users in the neighborhood of this user rated the item with an average computed above.

Item-based Example

- Find out what a user would score a particular movie that he/she did not score (or watch)
  - Similarity will be based on items
- Use `sim_costeta` for similarity (can try other ones)
- The distance matrix is `M`-by-`M`
- Always use `float32` for speed

In [27]:
```
# Item-based filtering
OUR_USER= 0
OUR_ITEM= 0  # pick an item which our user did not rate
OUR_ITEM_NEIGHBORHOOD= 50  # number of items similar to our item
```

In [28]:
```
# users who rated our item
users = np.where(Ratings[:,OUR_ITEM]>0.)[0]
```

In [29]:
```
%%time
```

```
# generate a distance matrix - item
Dist_i = np.zeros((M,M), dtype=np.float32)
for i in range(M):
    for j in range(M):
        Dist_i[i,j] = sim_costeta(Ratings[:,i], Ratings[:,j])
```

```
CPU times: total: 16.6 s
Wall time: 16.6 s
```

In [30]:
```
d = np.abs(Dist_i[OUR_ITEM,:])
ix_best = np.argsort(-d)[:OUR_ITEM_NEIGHBORHOOD]
items = ix_best
```

In [31]:
```
Dist_i[OUR_ITEM,items]
```

Out[31]:
```
array([1.        , 0.5504396 , 0.5502418 , 0.5399177 , 0.53434914,
       0.5339682 , 0.5261372 , 0.52247614, 0.5207451 , 0.5182745 ,
       0.5171993 , 0.5162483 , 0.5151485 , 0.51244044, 0.5092254 ,
       0.5065545 , 0.49518365, 0.49036434, 0.48994136, 0.48688045,
       0.48679808, 0.4855667 , 0.48526853, 0.4845558 , 0.48364452,
       0.4789724 , 0.47775424, 0.47728276, 0.47648945, 0.47547597,
       0.4751349 , 0.47399274, 0.47177908, 0.4688636 , 0.46806997,
       0.46561712, 0.46376756, 0.46283498, 0.46212852, 0.45594773,
       0.4524766 , 0.4489189 , 0.44841185, 0.44607133, 0.44498888,
       0.44414526, 0.44198734, 0.44049805, 0.4375241 , 0.43522343],
      dtype=float32)
```

In [32]:
```
ratings = np.array([Ratings[users,_] for _ in items])

print(f'The user not rated the item= {Ratings[OUR_USER, OUR_ITEM]:.1
print(f'Average rating in item-neighborhood= {ratings.mean():.1f}')
```

```
The user not rated the item= 0.0
Average rating in item-neighborhood= 1.8
```

Outcome: The user did not rate the item. Other items in the neighborhood of this item has the average score as above.

Content-based Approach: Cluster the Items

Item category can help a user select a relevant item. In addition, the user selections and scores can guide the recommender system in predicting what category a user likes more than the others. Creating item genres or item categories can be accomplished via clustering item features. These features can also be some augmented information provided by experts (or even movie critics). In this example, we can use movie titles (plots, information extracted from the Web, or even parsing movie scripts) to create a movie feature vector. Note that in our dataset, there are also tags that capture this kind of information.

Use the following two resources:

1. `gensim` GloVe 📚 **Gensim** = "Generate Similar" is a topic modeling library to implement Latent Semantic Methods, and it is licensed under **GNU LGPLv2.1 license**
2. GloVe embedding model with feature size `M=100` using 🧩 **https://nlp.stanford.edu/projects/glove/**

`gensim` library can download GloVe word embeddings to be used as word vectors in text processing. A movie is represented by its title and the average word vectors it constitutes (Word2Vec is another method to represent a text phrase).

The GloVe project is an embedding approach trained by very large corpora, specifically indexed words consistently, to improve sharing across problems and domains. The selected embedding is `M=100` dimensions, 400K words.

The number of genres might be a good starting point as the number of clusters is expected.

Let's represent each movie by its average word embedding extracted from its title and cluster.

Important: We need to ensure the word is in the Glove dictionary - use lowercase words, etc.

In [33]:
```
import gensim.downloader

print(f'gensim version= {gensim.__version__}')
Glove = gensim.downloader.load('glove-wiki-gigaword-100')

GLOVE_M = 100
```

```
gensim version= 4.3.0
```

In [34]:
```
# restart movie index due to filtering as in the previous Cells
movies = sorted(df_ratings.movieId.unique())
MoviesIndex = {mov:i for i, mov in enumerate(movies)}
MoviesIndexRev = {MoviesIndex[mov]:mov for mov in MoviesIndex.keys()

M_titles = len(MoviesIndex.keys())
```

In [35]:
```
# populate titles, extract tokens
```

```
Titles, TitleGenres, TitlesParsed = [None]*M_titles, [None]*M_titles
for _, row in df_movies.iterrows():
    if row.movieId in MoviesIndex:
        Titles[MoviesIndex[row.movieId]] = row.title
        TitleGenres[MoviesIndex[row.movieId]] = re.search(r'[\w\-]+'

for _, title in enumerate(Titles):
    TitlesParsed[_] = re.findall(r'[a-z]+', title.lower())

# sanity
print(Titles[:5])
print(TitlesParsed[:5])
print(TitleGenres[:5])
```

```
['Toy Story (1995)', 'Jumanji (1995)', 'Grumpier Old Men (1995)', 'Wait
ing to Exhale (1995)', 'Father of the Bride Part II (1995)']
[['toy', 'story'], ['jumanji'], ['grumpier', 'old', 'men'], ['waiting',
'to', 'exhale'], ['father', 'of', 'the', 'bride', 'part', 'ii']]
['Adventure', 'Adventure', 'Comedy', 'Comedy', 'Comedy']
```

In [36]:
```
# take the average word vectors to represent titles in GloVe embeddi
X = np.zeros((M_titles, GLOVE_M), dtype=np.float32)
for _, title in enumerate(TitlesParsed):
    v = []
    if len(title) == 0:
        # movie title empty after parsing, setting to zero vector
        X[_] = np.zeros(GLOVE_M, dtype=np.float32)
    else:
        for w in title:
            v += [Glove[w]] if w in Glove else [np.zeros(GLOVE_M, dt
        X[_] = np.mean(np.array(v), axis=0)  # average
```

In [37]:
```
from sklearn.cluster import KMeans

# use number of genres to cluster
K = len(Genres.keys())
Clusters = KMeans(n_clusters=K, n_init=10).fit_predict(X)
```

In [38]:
```
# Print a few movie titles and their cluster IDs in a genre
GENRE = 'Horror'
n_results = 20
for _, g in enumerate(TitleGenres):
    if g == GENRE and n_results > 0:
        print(f'{Clusters[_]:2d}  {Titles[_]:s}')
        n_results -= 1
```

```
 4  Lord of Illusions (1995)
10  Species (1995)
11  Castle Freak (1995)
18  Relative Fear (1994)
 4  Tales from the Crypt Presents: Demon Knight (1995)
 8  Village of the Damned (1995)
 8  Fear, The (1995)
 8  In the Mouth of Madness (1995)
13  Body Snatchers (1993)
 4  Puppet Masters, The (1994)
 2  301, 302 (301/302) (1995)
12  Cemetery Man (Dellamorte Dellamore) (1994)
13  Thinner (1996)
 8  Spirits of the Dead (1968)
12  Eyes Without a Face (Yeux sans visage, Les) (1959)
 4  Relic, The (1997)
 4  Halloween: The Curse of Michael Myers (Halloween 6: The Curse of Mi
chael Myers) (1995)
 8  Night of the Living Dead (1968)
 8  Children of the Corn IV: The Gathering (1996)
 8  Fog, The (1980)
```

In [39]:
```python
# Print a few movie titles and their cluster IDs in a genre
GENRE = 'War'
n_results = 20
for _, g in enumerate(TitleGenres):
    if g == GENRE and n_results > 0:
        print(f'{Clusters[_]:2d}  {Titles[_]:s}')
        n_results -= 1
```

```
16  Pharaoh's Army (1995)
 8  Walk in the Sun, A (1945)
 4  Prisoner of the Mountains (Kavkazsky plennik) (1996)
13  Pork Chop Hill (1959)
 5  Run Silent Run Deep (1958)
 8  Cross of Iron (1977)
 4  Devil's Brigade, The (1968)
 9  Story of G.I. Joe (1945)
14  Anzio (1968)
 8  Lion of the Desert (1980)
16  Attack Force Z (a.k.a. The Z Men) (Z-tzu te kung tui) (1982)
18  Objective, Burma! (1945)
 8  Battle of Britain (1969)
12  Duel at Diablo (1966)
16  Murphy's War (1971)
12  Plainsman, The (1937)
 8  In Enemy Hands (2004)
 1  Bataan (1943)
 0  Villa Rides! (1968)
 4  Battle for Haditha (2007)
```

Question: How do you assess the performance of this clustering? Do you expect above clustering will work on titles for other genres?

Matrix Factorization Techniques

Matrix factorization is a core technique in recommender systems, widely used for collaborative filtering. Decomposing the large, sparse user-item interaction matrix `Ratings` (R) into the product of two lower-dimensional matrices: a user latent feature matrix U and an item latent feature matrix V. These latent matrices capture hidden factors influencing user preferences and item characteristics. The goal is to approximate R by minimizing the reconstruction error, typically using optimization techniques such as gradient descent or alternating least squares (ALS). By projecting both users and items into the same latent space, matrix factorization enables efficient prediction of unknown interactions, thereby personalizing recommendations. Extensions of matrix factorization, like singular value decomposition (SVD) and non-negative matrix factorization (NMF), are adapted to enhance interpretability, sparsity, or scalability, making the approach suitable for real-world applications with high number (e.g., millions) of users and items.

Singular Value Decomposition

SVD is a mathematical technique used to factorize a matrix into three components: $A = U \sum V^\top$, where A is the original matrix, U and V are orthogonal matrices, and $\sum$ is a diagonal matrix of singular values. In recommender systems, SVD is commonly applied to decompose the user-item interaction matrix and extract latent factors representing users and items. By retaining only the top-k singular values, we can reduce dimensionality and capture the most significant features, improving generalization and computational efficiency. The scipy package `sparse.linalg` has an implementation of SVD, named `svds`.

Given a user-item matrix R of dimensions $N \times M$:

- U: $N \times N$ orthogonal matrix (left singular vectors)
- $\sum$: $N \times M$ diagonal matrix (singular values
- $V^\top$: $M \times M$ orthogonal matrix (right singular vectors)

The approximation is achieved by truncating $\sum$ to rank k, resulting in $R \approx U_k \sum_k V_K^\top$.

Alternating Least Squares

ALS is a matrix factorization technique specifically designed for sparse data, like the `Ratings` matrix above, often used in collaborative filtering. It minimizes the reconstruction error of the user-item interaction matrix by iteratively solving least squares optimization problems. The algorithm alternates between fixing user or item factors and solving for the other, ensuring computational efficiency. The Apache Spark package `pyspark` has an implementation of ALS.

Given R, ALS approximates it as $R \approx U \cdot V^\top$, where:

- U is the user latent feature matrix ($N \times k$)
- V is the item latent feature matrix ($M \times k$)

For each iteration, the optimization alternates:

- Fix V, solve $U = \arg\min_U \|R - U \cdot V^\top\|^2 + \lambda\|U\|^2$
- Fix U, solve $V = \arg\min_V \|R - U \cdot V^\top\|^2 + \lambda\|V\|^2$

Non-negative Matrix Factorization

NMF is a matrix factorization approach where all matrix elements are constrained to be non-negative, making it particularly suitable for datasets where negative values lack interpretation (e.g., user preferences or counts). It factors R into two non-negative matrices W and H, ensuring interpretability and sparsity.

Given R, W ($N \times k$) user feature matrix, H ($k \times M$) item feature matrix, the objective is to minimize $\|R - W \cdot H\|^2$, subject to $W \geq 0$ and $H \geq 0$. This is solved iteratively using multiplicative updates. The sklearn package `decomposition` has an implementation of NMF.

Emerging Trends in Recommender Systems

- Federated learning for recommenders
- Privacy-preserving recommendations
- Zero-shot and few-shot learning for recommendations
- Personalization in recommender systems
- Large language models in recommendation systems

References

1. Falk, Kim. Practical recommender systems. Simon and Schuster, 2019.
2. Harper, F. Maxwell, and Joseph A. Konstan. "The movielens datasets: History and context." Acm transactions on interactive intelligent systems (tiis) 5.4 (2015): 1-19.
3. Su, Xiaoyuan, and Taghi M. Khoshgoftaar. "A survey of collaborative filtering techniques." Advances in artificial intelligence 2009 (2009).

Exercises

Exercise 1. Reduce the number of items by updating the minimum number of ratings and rerun the analysis.

Exercise 2. Why did the item filtering reveal a much lower rating than the user? How do you remedy this?

Exercise 3. Implement content-based filtering by generating similarities between movies based on their tags.

In [40]:
```
%%html
<style>
    table {margin-left: 0 !important;}
    p {font-family: verdana;}
    li {font-family: verdana;}
    div {font-size: 10pt;}
</style>
<!-- Display markdown tables left oriented in this notebook. -->
```
