# A Primer on Recommendation Systems

# PyconES 2016

## About me (Manuel Garrido)

- Freelance Data Scientist
- Portugal-->Spain-->/United States
- consultant-->analyst-->data scientist
- excel-->R-->python-->spark

## Contact


- manugarri.com (http://manugarri.com)
- @manugarri
- hola@manugarri.com
- manuel@tribeclick.com / hiring@tribeclick.com


In [2]:
```
%load_ext watermark
%watermark
```

```
2016-10-06T10:48:20+02:00

CPython 3.5.1
IPython 4.2.0

compiler   : GCC 4.4.7 20120313 (Red Hat 4.4.7-1)
system     : Linux
release    : 4.4.0-38-generic
machine    : x86_64
processor  : x86_64
CPU cores  : 8
interpreter: 64bit
```

# About the talk

- Entry level
- Focused on examples

### If you meet the requirements...

```
pandas>=0.17
numpy
scipy
```

### ...then clone the talk's repo and run a notebook

```
git clone git@github.com:manugarri/pydata_2016_talk_recommenders.git
cd pydata_2016_talk_recommenders
gunzip assets/reddit.db.gz
jupyter notebook
```

# 1. What are recommendation systems?

**Recommender systems** or **recommendation systems** (sometimes replacing "**system**" with a synonym such as platform or engine) are a subclass of information filtering **system** that seek to predict the 'rating' or 'preference' that a user would give to an item.

Recommender system - Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/**Recommender_system**


## Foundations for recommending items to users:

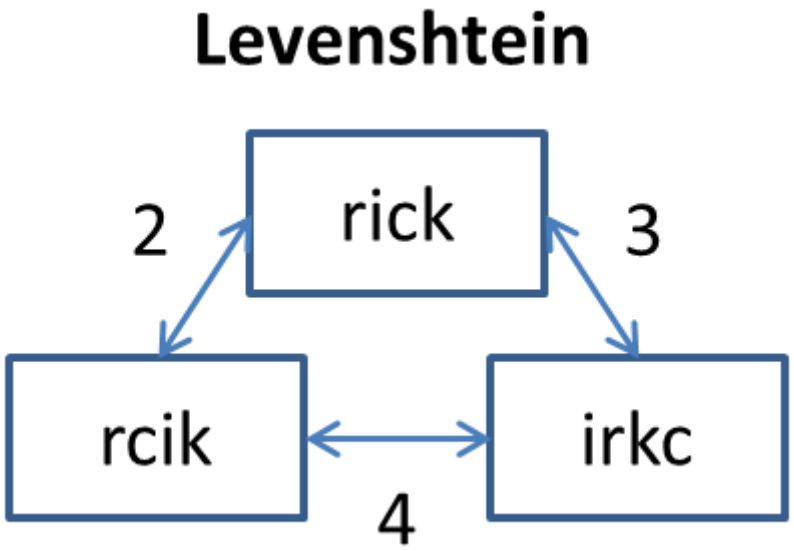
- **Item Information**
  - Item characteristics
- **User Information**
  - User characteristics.
  - User preferences.
  - Users relationships
  - User's previous interaction with our platform
- **Platform Information**
  - Business goals
  - Availability

How do we recommend items? To do so we need information, any source of information.

One clear source of information is the item information , that is, the intrinsic features of the items (is the shoe blue, is it a sneaker, and so on). Another source is the information about the users, who they are, what do they like, which items do they like, and so on Another source of information, is the platform itself. Things like the business goals or stock availability have some saying in what are we recommending to our users.

# A short note on Similarity

- **Edit distance (Lehvenstein & variations)**: used to measure similarity between two strings. It takes into account how many changes would need to be applied to a word to become another.



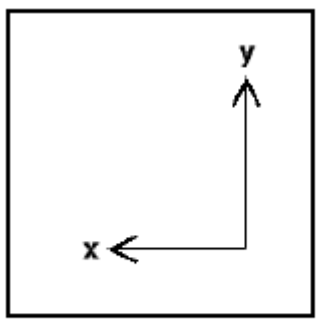

- **Jaccard Index**: Useful when computing similarity among sets (one to many relations)

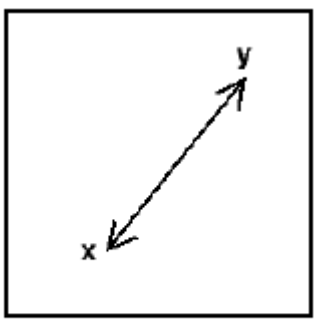
$$J(A,B) = \frac{|A \bigcap B|}{|A \bigcup B|}$$

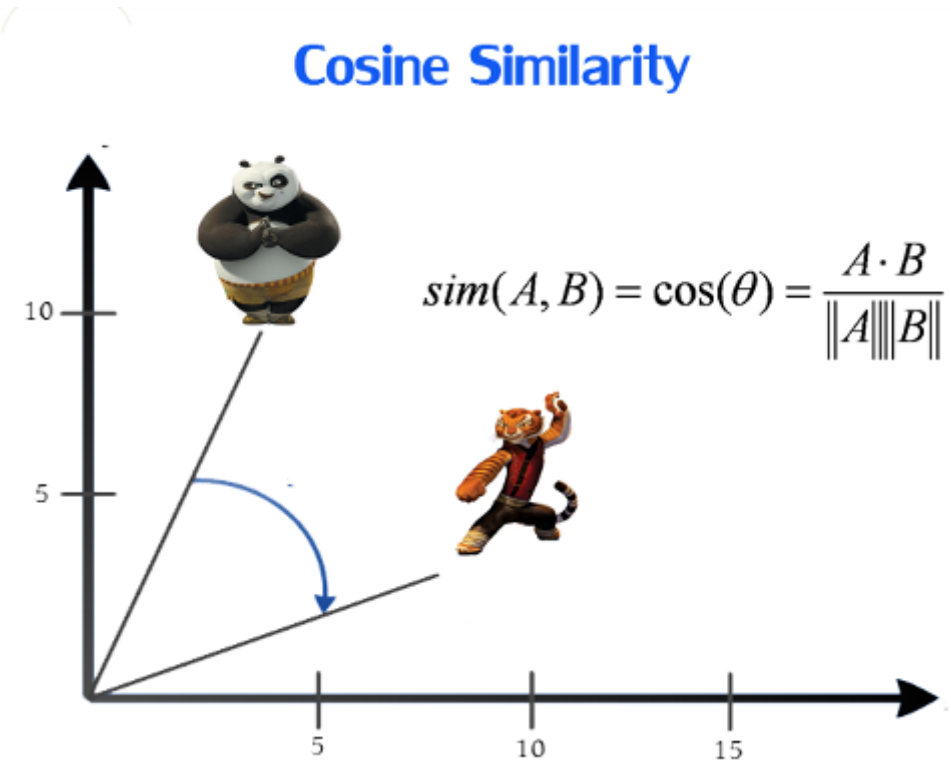
One concept that is key to recommendation systems is the concept of similarity. These systems always recommend based on similarities between entities. But one important thing to remember is that similarity can be measure differently depending on what you are measuring.

For example, a good way to measure similarity between strings is to use the Lehvenstein distance, which measures how many insertions, deletions or substitutions do you need to apply to string A to become sttring B.

Another metric would be the Jaccard index, that measures similarities between sets. For example, you could compute Jaccards similarity on two groups of friends by dividing how many friends are in both groups by the number of friends on any of the two groups.




- **and many more!**

Then you have other measures that are more popular and general purpose.

The Manhattan Distance is the distance between two points measured along axes at right angles The Euclidean distance or Euclidean metric is the "ordinary" (i.e. straight-line) distance between two points Cosine similarity measures the cosine of the angle between them.

## Recommendation Systems Approaches

- **Content Filtering**
- **Collaborative Filtering**
- **Hybrid Systems**
- **Other (Demographic / Social Recommendations)**

Here are the main approaches to recommendation systems. Content Filtering, Collaborative filtering, Hybrid systems, and then other (because you always have to have an other in a list, isnt it).

**Demographic recommendations**

- Use demographic information from the user to recommend items relevant to the user's stratum

**Social recommendations**

- Users sharing with other users

Regarding other recommender systems, one way to provide recommendations to your users, if you are lucky to have demographic information about them, is to stratify them based on this demographic information and recommend based on those strata.

Then we have a very simple way to recommend items to users, which happens any time we click the "Share" button anywhere online.

## Content Filtering

In a Content filtering recommendation system, items are mapped into a feature space, and recommendations depend on item characteristics.

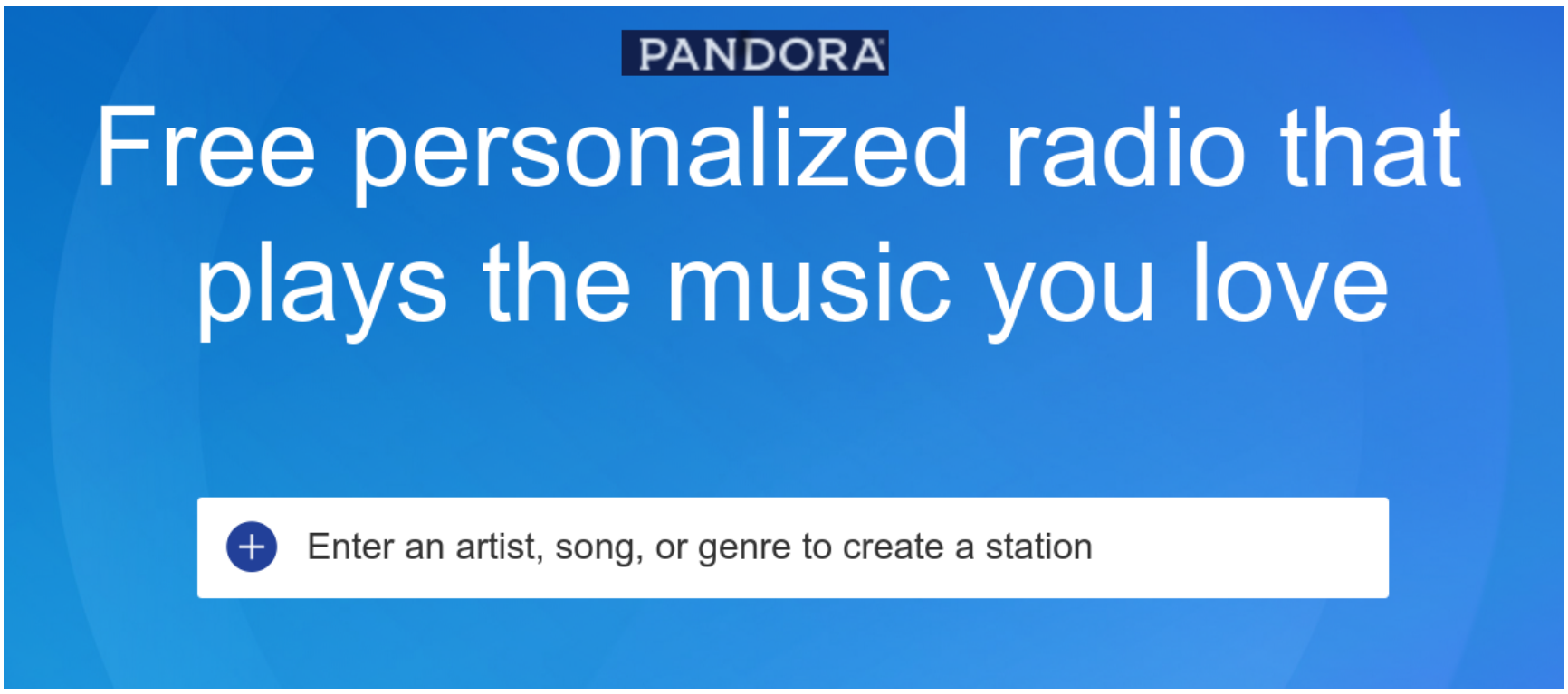
- That means that all the feature information we use is derived only from the items.
- That does not mean that we don't rely on any user information, just that information is used only on the recommendation step, and not at computing time.

We start with content filtering. Which is not a very popular approach nowadays but can still be useful depending on the circumstances.

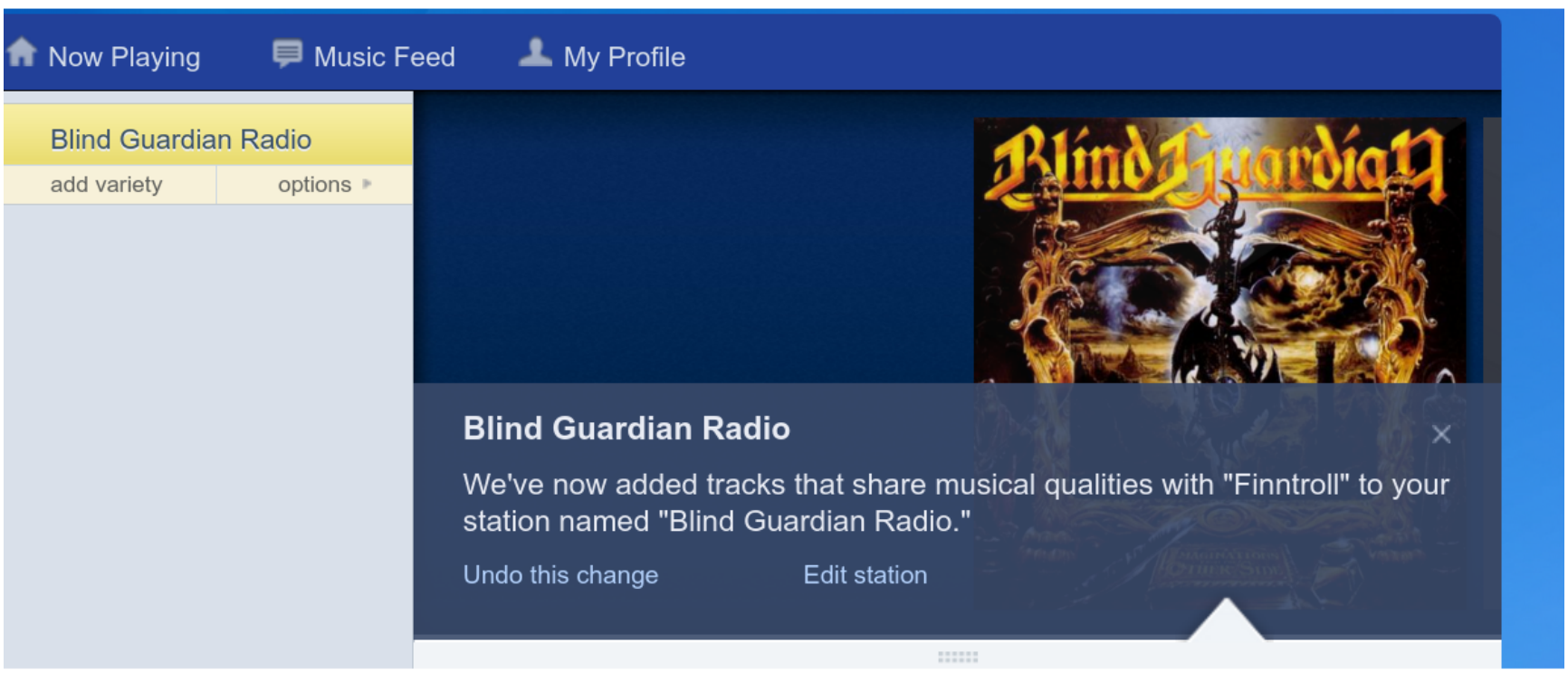
On content filtering, the features used to provide recommendations are derived from the items and only from the items.

That does not mean that we dont get any user's input, just that that input is required at recommendation time, not at computing time.

## Example: Content Filtering




One example of Content filtering would be Pandora, pandora is an online radio that is quite popular in the US. When you join you can select whicn songs, artists or genres you like. Then Pandora uses those songs /artists you selected to to create a personalized radio station using similar music.




Pandora allows us to add as many songs as we want to our station, each one improving the recommendations Pandora provides us.

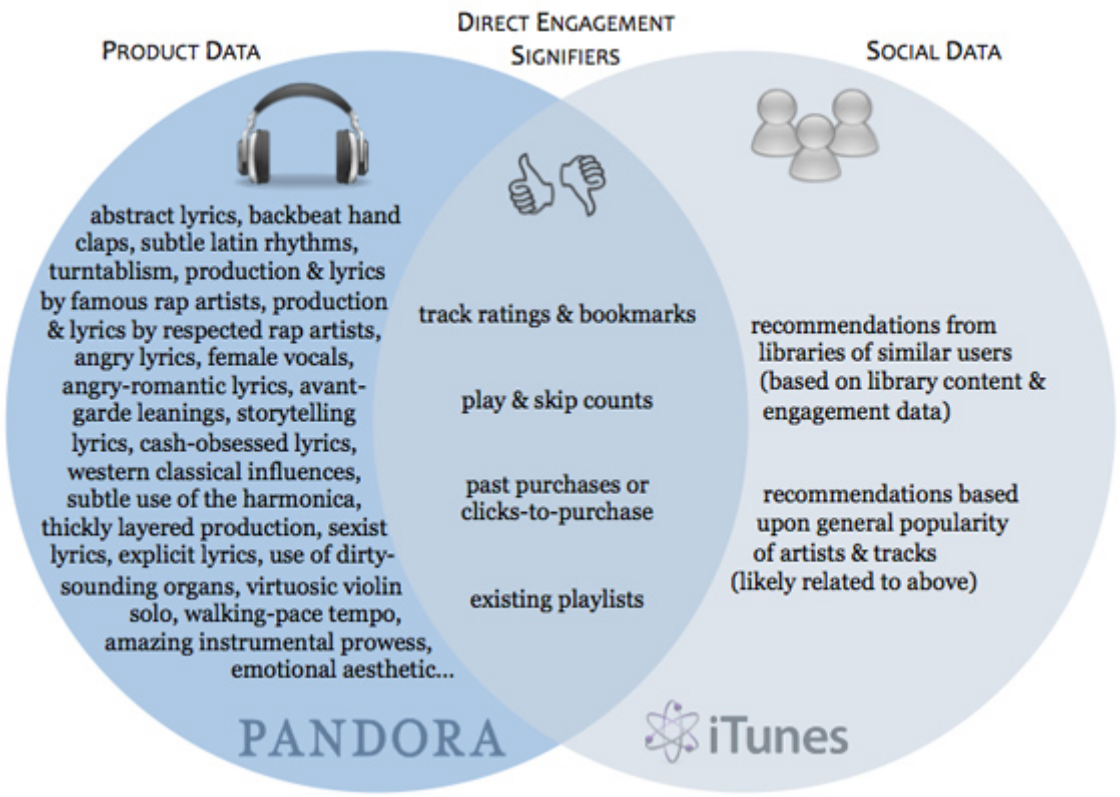
### About The Music Genome Project®

We believe that each individual has a unique relationship with music – no one else has tastes exactly like yours. So delivering a great radio experience to each and every listener requires an incredibly broad and deep understanding of music. That's why Pandora is based on the Music Genome Project, the most sophisticated taxonomy of musical information ever collected. It represents over ten years of analysis by our trained team of musicologists, and spans everything from this past Tuesday's new releases all the way back to the Renaissance and Classical music.

Each song in the Music Genome Project is analyzed using up to 450 distinct musical characteristics by a trained music analyst. These attributes capture not only the musical identity of a song, but also the many significant qualities that are relevant to understanding the musical preferences of listeners. The typical music analyst working on the Music Genome Project has a four-year degree in music theory, composition or performance, has passed through a selective screening process and has completed intensive training in the Music Genome's rigorous and precise methodology. To qualify for the work, analysts must have a firm grounding in music theory, including familiarity with a wide range of styles and sounds.

https://www.pandora.com/about/mgp (https://www.pandora.com/about/mgp)

Pandora's Recommendation system is based off the Music Genome Project a project that clains to be the most in depth taxonomy of music. Pandora has a team of musical analysts analyzing 10,000 songs every month, mapping them to a set of 450 distinct musical features. They use these features to power their recommendations.




Here on the left you can see some of those features, which are very very specific

# Content Filtering

## Pros

- Recomending to a new user is easy. We just get user's input and we can recommend right away.
- Domain knowledge can be applied

## Cons

- Need to map each item into the feature space. That means that any time a new item gets added, someone has to manually categorize that item (unless using unsupervised clustering methods).
- Recommendations are limited in scope. This means items can't be categorized in new features. Who can say 450 features are all the features necessary to map music?

The benefits of Content filtering is that, once we have the items mapped, we can recommend to any user right away. Another one is that domain knowledge can be applied, for example, Pandora's team of experts can specify the features required on music better than anyone.

One big con of Content filtering systems is that each item needs to be mapped. For example, if there is a new song with a uniquely new characteristic, Pandora's musical analysts would have to first, add that characteristic's features, then, map EVERY SINGLE song into that new characteristic.

Another con CF systems have is that the items can only be categorized by the predefined features.

## Practical Example: Content Filtering Movie recommendation Engine

movielens

Precisionrecall.png In this example we will build a recommendation system that recommends movies to users based on Movie features

We will use the Movielens (http://grouplens.org/datasets/movielens/) dataset. A free dataset containing millions of movie ratings by users.

In [3]:
```
import pandas as pd
import numpy as np

movies_df = pd.read_table('../data/movies.dat', header=None, sep='::',
                          names=['movie_id', 'movie_title', 'movie_genre'],
engine='python')
movies_df = pd.concat([movies_df, movies_df.movie_genre.str.get_dummies(sep='|')],
axis=1)
movie_categories = movies_df.columns[3:].tolist()
movies_df.head()
```

Out[3]:

| | movie_id | movie_title | movie_genre | Action | Adventure | Animation | Ch |
|---|---|---|---|---|---|---|---|
| **0** | 1 | Toy Story (1995) | Animation\|Children's\|Comedy | 0 | 0 | 1 | 1 |
| **1** | 2 | Jumanji (1995) | Adventure\|Children's\|Fantasy | 0 | 1 | 0 | 1 |
| **2** | 3 | Grumpier Old Men (1995) | Comedy\|Romance | 0 | 0 | 0 | 0 |
| **3** | 4 | Waiting to Exhale (1995) | Comedy\|Drama | 0 | 0 | 0 | 0 |
| **4** | 5 | Father of the Bride Part II (1995) | Comedy | 0 | 0 | 0 | 0 |

5 rows × 21 columns

In [4]:
```
user_movies = [
    'Die Hard: With a Vengeance (1995)',
    'Die Hard (1988)',
    'Braveheart (1995)',
    'Star Wars: Episode IV - A New Hope (1977)',
    'Star Wars: Episode VI - Return of the Jedi (1983)',
    'Indiana Jones and the Last Crusade (1989)',
    'Toy Story (1995)',
    'Aladdin (1992)',
    'Lion King, The (1994)',
]
```

In [5]:
```
def get_user_preferences(user_movies, verbose=False):
    user_features = movies_df[movies_df.movie_title.isin(user_movies)].ix[:,3:].T
    user_features = user_features.mean(axis=1).reset_index()
    if verbose:
        print(user_features)
    return user_features.ix[:,1].tolist()

user_preferences_list = get_user_preferences(user_movies, verbose=True)
```

| | index | 0 |
|---|---|---|
| 0 | Action | 0.666667 |
| 1 | Adventure | 0.333333 |
| 2 | Animation | 0.333333 |
| 3 | Children's | 0.333333 |
| 4 | Comedy | 0.222222 |
| 5 | Crime | 0.000000 |
| 6 | Documentary | 0.000000 |
| 7 | Drama | 0.111111 |
| 8 | Fantasy | 0.111111 |
| 9 | Film-Noir | 0.000000 |
| 10 | Horror | 0.000000 |
| 11 | Musical | 0.222222 |
| 12 | Mystery | 0.000000 |
| 13 | Romance | 0.111111 |
| 14 | Sci-Fi | 0.222222 |
| 15 | Thriller | 0.222222 |
| 16 | War | 0.222222 |
| 17 | Western | 0.000000 |

In [6]:
```
def get_predicted_movie_score(movie_name, user_preferences):
    movie_slice = movies_df[movies_df.movie_title==movie_name].iloc[0]
    movie_features = movie_slice[movie_categories]
    return np.dot(movie_features, user_preferences_list)
```

In [7]:
```
#Action +Sci-Fi + Thriller
get_predicted_movie_score('Terminator 2: Judgment Day (1991)', user_preferences_lis
t)
```

Out[7]: 1.1111111111111112

In [8]:
```
#Action + Drama
get_predicted_movie_score('Rocky (1976)', user_preferences_list)
```

Out[8]: 0.77777777777777768

In [9]:
```
#Animation + Musical
get_predicted_movie_score('Prince of Egypt, The (1998)', user_preferences_list)
```

Out[9]: 0.55555555555555558

In [10]:
```
#Horror + Thriller
get_predicted_movie_score('Scream (1996)', user_preferences_list)
```

Out[10]: 0.22222222222222221

In [11]:
```
def get_movie_recommendations(user_preferences, n_recommendations):
    movies_df['score'] = movies_df.movie_title.apply(get_predicted_movie_score, arg
s=([user_preferences]))
    movies_df.sort_values(by=['score'], ascending=False, inplace=True)
    del movies_df['score']
    return movies_df[['movie_title','movie_genre']].head(n_recommendations)

user_movie_recomendations = get_movie_recommendations(user_preferences_list, 10)
user_movie_recomendations
```

Out[11]:

| | movie_title | movie_genre |
|---|---|---|
| **1187** | Transformers: The Movie, The (1986) | Action\|Animation\|Children's\|Sci-Fi\|Thriller\|War |
| **554** | Pagemaster, The (1994) | Action\|Adventure\|Animation\|Children's\|Fantasy |
| **2253** | Soldier (1998) | Action\|Adventure\|Sci-Fi\|Thriller\|War |
| **1192** | Star Wars: Episode VI - Return of the Jedi (1983) | Action\|Adventure\|Romance\|Sci-Fi\|War |
| **1178** | Star Wars: Episode V - The Empire Strikes Back... | Action\|Adventure\|Drama\|Sci-Fi\|War |
| **606** | Heavy Metal (1981) | Action\|Adventure\|Animation\|Horror\|Sci-Fi |
| **1972** | Condorman (1981) | Action\|Adventure\|Children's\|Comedy |
| **2651** | Inspector Gadget (1999) | Action\|Adventure\|Children's\|Comedy |
| **542** | Super Mario Bros. (1993) | Action\|Adventure\|Children's\|Sci-Fi |
| **1197** | Army of Darkness (1993) | Action\|Adventure\|Comedy\|Horror\|Sci-Fi |

**We can even ask the user to explicitly state their opinion**

In [12]:
```
from collections import OrderedDict


user_preferences = OrderedDict(zip(movie_categories, [3]*len(movie_categories)))


user_preferences['Action'] = 5
user_preferences['Adventure'] = 5
user_preferences['Fantasy'] = 5
user_preferences['Sci-Fi'] = 5

user_preferences['Crime'] = 2
user_preferences['Horror'] = 2
user_preferences['Thriller'] = 2

user_preferences['Musical'] = 1
user_preferences['Romance'] = 1
user_preferences['Western'] =1
user_preferences['Animation'] = 1
user_preferences["Children's"] = 1
user_preferences['Documentary'] = 1
user_preferences['Drama'] = 1
user_preferences['Film-Noir'] = 1

user_preferences_list = list(user_preferences.values())
```

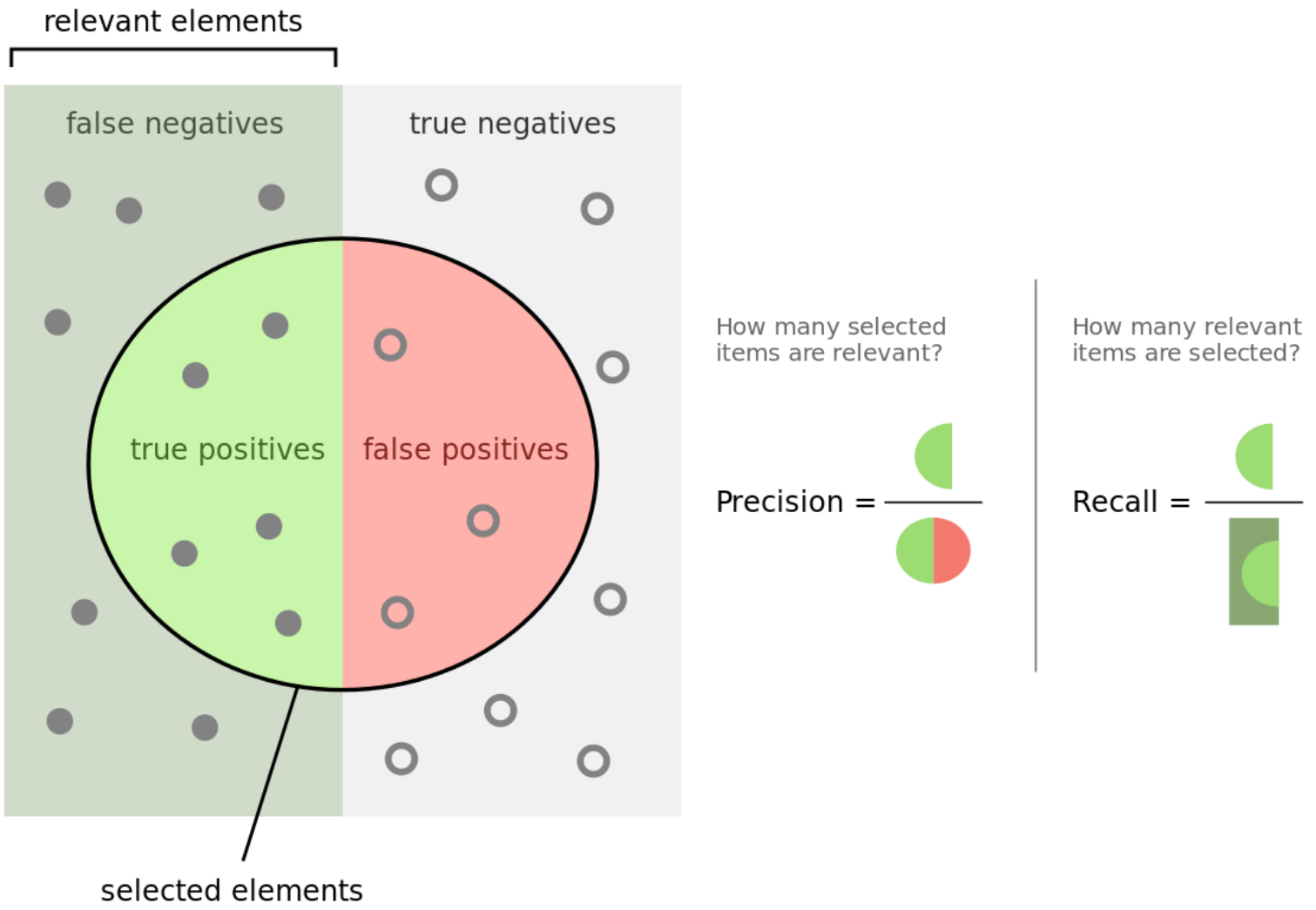
In [13]:
```
get_movie_recommendations(user_preferences_list, 10)
```

Out[13]:

| | movie_title | movie_genre |
|---|---|---|
| **257** | Star Wars: Episode IV - A New Hope (1977) | Action\|Adventure\|Fantasy\|Sci-Fi |
| **2253** | Soldier (1998) | Action\|Adventure\|Sci-Fi\|Thriller\|War |
| **2559** | Star Wars: Episode I - The Phantom Menace (1999) | Action\|Adventure\|Fantasy\|Sci-Fi |
| **2036** | Tron (1982) | Action\|Adventure\|Fantasy\|Sci-Fi |
| **1197** | Army of Darkness (1993) | Action\|Adventure\|Comedy\|Horror\|Sci-Fi |
| **1985** | Honey, I Shrunk the Kids (1989) | Adventure\|Children's\|Comedy\|Fantasy\|Sci-Fi |
| **1192** | Star Wars: Episode VI - Return of the Jedi (1983) | Action\|Adventure\|Romance\|Sci-Fi\|War |
| **1178** | Star Wars: Episode V - The Empire Strikes Back... | Action\|Adventure\|Drama\|Sci-Fi\|War |
| **1630** | Starship Troopers (1997) | Action\|Adventure\|Sci-Fi\|War |
| **1539** | Men in Black (1997) | Action\|Adventure\|Comedy\|Sci-Fi |

## Recommendation Systems Evaluation

- **Recommendation as classification (recommend some good items)**




*others include (Area Under the Curve (AUC) , F1-score)*

- **Recommendation as ranking ( prediction of user ratings)**

$$\mathrm{RMSE} = \sqrt{\frac{\sum_{i=1}^{n} \left(P_i - O_i\right)^2}{n}}$$

*another evaluation metrics would be NDCG*

- **Recommendation as utility optimization**

*Here (http://jmlr.csail.mit.edu/papers/volume10/gunawardana09a/gunawardana09a.pdf) is an extremely interesting paper on Recsys evaluation*

In [14]:
```
def calculate_precision_recall(user_movies, user_recommendations):
    true_positive = len([m for m in user_recommendations if m in user_movies])
    false_positive = len(user_recommendations) - true_positive
    false_negatives = len([m for m in user_movies if m not in
user_recommendations])
    precision = true_positive / (true_positive + false_positive)
    recall = true_positive / (true_positive + false_negatives)
    if not (precision or recall):
        return 0,0,0
    f1_score = 2 * (precision*recall) / (precision + recall)
    return precision, recall, f1_score
```

In [15]:
```
user_movies
```

Out[15]:
```
['Die Hard: With a Vengeance (1995)',
 'Die Hard (1988)',
 'Braveheart (1995)',
 'Star Wars: Episode IV - A New Hope (1977)',
 'Star Wars: Episode VI - Return of the Jedi (1983)',
 'Indiana Jones and the Last Crusade (1989)',
 'Toy Story (1995)',
 'Aladdin (1992)',
 'Lion King, The (1994)']
```

In [16]:
```
user_movie_recomendations_list = user_movie_recomendations.movie_title.tolist()
user_movie_recomendations_list
```

Out[16]:
```
['Transformers: The Movie, The (1986)',
 'Pagemaster, The (1994)',
 'Soldier (1998)',
 'Star Wars: Episode VI - Return of the Jedi (1983)',
 'Star Wars: Episode V - The Empire Strikes Back (1980)',
 'Heavy Metal (1981)',
 'Condorman (1981)',
 'Inspector Gadget (1999)',
 'Super Mario Bros. (1993)',
 'Army of Darkness (1993)']
```

In [17]:
```
calculate_precision_recall(user_movies, user_movie_recomendations_list)
```

Out[17]:
```
(0.1, 0.1111111111111111, 0.10526315789473685)
```

# Collaborative Filtering (Breese et al., 1998) (https://www.microsoft.com/en-us/research/publication/empirical-analysis-of-predictive-algorithms-for-collaborative-filtering/)

In collaborative filtering, item features are derived from individual user behaviors or attitudes.

The assumption is that users get value from recommendations based on other users with similar tastes. Users who agreed on preferred items in the past will tend to agree in the future too

Unleases a potentially much bigger dataset to base our recommendations on.

## 2 Basic approaches:

- Item based collaborative filtering (Item-Item). Measure similarity between items
- User based collaborative filtering (Item-User). Measure similarity between users

# Example: Item-Item Collaborative Filtering


```
For each item in product catalog, I1
  For each customer C who purchased I1
    For each item I2 purchased by
      customer C
        Record that a customer purchased I1
         and I2
  For each item I2
    Compute the similarity between I1 and I2
```

On earlier versions of Amazon, recommendations where provided based on user purchases. How Amazon (http://www.cin.ufpe.br/~idal/rs/Amazon-Recommendations.pdf) described their algorithm is above. They used cosine similarity to compute the similarity between items.

## Example: User-Item Collaborative Filtering


One obvious example of User item collaborative filtering is Facebook 'Friends you may know' even though the implementation is graph based, is using users that are similar to you in terms of the friends you share to recommend new friends to you based on your friends' friends

## Collaborative filtering

### Pros

- Capable of recommending new items without having to manually map them on the feature space.
- Capable to find recommendations based on hidden features that an expert wouldn't be able to find (for example, combination of genres or actors).
- Domain agnostic.

**Cons**

- Cold Start. We need lots of user interaction data to be able to start building our features
- Every user interaction affects the features.

# Exercise: Item-Item Collaborative filtering on Reddit.

I am going to show you how to build a recommendation engine for Reddit!

A live example of it is located at http://findasub.manugarri.com (http://findasub.manugarri.com)

The goal can be written down as "given a user's set of subreddits, recommend new subredits that are similar".

In this system, the features of each item (subreddit) will be the similarities of this subreddit with the others

```
manuel@manuel-P35V2 pycones_2016_talk $ sqlite3 assets/reddit.db
SQLite version 3.9.2 2015-11-02 18:31:45
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE redditors (redditor varchar(30), sub varchar(1000));
CREATE TABLE similarity (sub1 varchar(30), sub2 varchar(1000), similarity float);
sqlite>
```

In [25]:
```
import sqlite3
conn = sqlite3.connect('../data/reddit.db')

print(pd.io.sql.read_sql('select count(*) from redditors;',conn))

pd.io.sql.read_sql('select redditor, sub from redditors limit 10;',conn)
```

```
   count(*)
0   1756328
```

Out[25]:

| | redditor | sub |
|---|---|---|
| **0** | bananashirt178 | memes |
| **1** | tweed08 | memes |
| **2** | tweed08 | Portal |
| **3** | KTM950Guy | beerporn |
| **4** | braidedbutthair | beerporn |
| **5** | ipathological | beerporn |
| **6** | justinthreee | memes |
| **7** | jake420 | shittyadvice |
| **8** | jake420 | memes |
| **9** | jake420 | findareddit |

We have 1.8 million rows on the redditors table, each one is a comment of a redditor to a subreddit.

In [26]:
```
#https://github.com/manugarri/Reddit-Recommendation-Engine/blob/master/seed/similarity.py

#In production, be always mindful of Bobby Tables
def compute_sim(sub1, sub2):
    users_union = conn.execute('''SELECT COUNT(DISTINCT(redditor)) from redditors
                                    WHERE sub ="{0}" OR sub ="{1}";'''.format(sub1, sub2)).fetchone()
    users_intersect = conn.execute('''SELECT COUNT(DISTINCT(redditor)) from redditors
                                    WHERE sub ="{0}" and redditor in (
                                    SELECT DISTINCT(redditor) FROM redditors WHERE sub="{1}")'''.format(sub1, sub2)).fetchone()
    users_intersect = int(users_intersect[0])
    users_union = int(users_union[0])
    if users_intersect:
        return users_intersect * 1.0 / users_union
    else:
        return 0.0
```

So here we see how we are going to measure the similarity between subs. We will use the Jaccard similarity index that I explained before.

So given a sample of reddit comments, we can estimate how similar two subreddits are by counting how many of the redditors that wrote messages on BOTH subreddits (intersection) divided by the number of redditors that wrote on either one of them,

This step takes quite some time to run

| | sub1 | sub2 | similarity |
|---|---|---|---|

In [27]:
```
compute_sim('aviation','flying')
```

Out[27]: 0.10175922731976543

In [28]:
```
import json
from itertools import combinations

with open('../data/subs.json' ) as f:
    subreddits = json.load(f)[:5]

for sim1, sim2 in combinations(subreddits, 2):
    if sim1!=sim2:
        print(sim1,sim2,compute_sim(sim1, sim2))
```

```
30ROCK 3DS 0.00051440329218107
30ROCK 3Dprinting 0.0006369426751592356
30ROCK 3amjokes 0.0006451612903225806
30ROCK 49ers 0.0006439150032195 75
3DS 3Dprinting 0.0013764624913971094
3DS 3amjokes 0.0010391409767925182
3DS 49ers 0.0017313019390581717
3Dprinting 3amjokes 0.001193792280143255
3Dprinting 49ers 0.0007945967421533572
3amjokes 49ers 0.00040032025620496394
```

In [29]:
```
pd.io.sql.read_sql('select sub1, sub2, similarity from similarity order by similari
ty desc limit 15;',conn)
```

Out[29]:

| | sub1 | sub2 | similarity |
|---|---|---|---|
| **0** | iOSthemes | jailbreak | 0.142021 |
| **1** | asktransgender | transgender | 0.112782 |
| **2** | aviation | flying | 0.101759 |
| **3** | ukpolitics | unitedkingdom | 0.101731 |
| **4** | Liberal | progressive | 0.099843 |
| **5** | keto | ketorecipes | 0.097407 |
| **6** | Pokemongiveaway | pokemontrades | 0.089431 |
| **7** | ImaginaryCharacters | ImaginaryMonsters | 0.088480 |
| **8** | frugalmalefashion | malefashionadvice | 0.084672 |
| **9** | GiftofGames | RandomActsOfGaming | 0.081259 |
| **10** | Entrepreneur | startups | 0.080465 |
| **11** | CrusaderKings | paradoxplaza | 0.080354 |
| **12** | hiphopheads | HipHopImages | 0.078541 |
| **13** | medicalschool | medicine | 0.076310 |
| **14** | keto | xxketo | 0.074339 |

In [30]:
```
def get_redditor_subs(redditor):
    records = conn.execute('''select sub from redditors where redditor = "{}"'''.fo
rmat(redditor)).fetchall()
    return [r[0] for r in records]

def get_sub_recommendations(redditor, n_recommendations=20):
    redditor_subs = get_redditor_subs(redditor)
    print(redditor_subs)

    query_1 = "select sub1 as sub, sum(similarity) as similarity from similarity wh
ere sub2 in ({})\
                group by sub;".format(', '.join(["'" + s + "'" for s in redditor_su
bs]))
    all_sub_scores_1 = pd.io.sql.read_sql(query_1, conn)

    query_2 = "select sub2 as sub, sum(similarity) as similarity from similarity wh
ere sub1 in ({})\
                group by sub;".format(', '.join(["'" + s + "'" for s in redditor_su
bs]))
    all_sub_scores_2 = pd.io.sql.read_sql(query_2, conn)

    sub_scores = pd.concat([all_sub_scores_1, all_sub_scores_2])
    sub_scores = sub_scores.groupby('sub').sum().reset_index()
    sub_scores.sort_values(by='similarity', ascending=False, inplace=True)

    sub_scores = sub_scores[-sub_scores['sub'].isin(redditor_subs)]
    return sub_scores.head(n_recommendations)
```

In [31]:
```
get_sub_recommendations('NoeticIntelligence')
```

```
['IAmA', 'atheism', 'askscience', 'conspiracy', 'scifi', 'photography', 'changemyv
iew', 'worldpolitics', 'Entrepreneur', 'compsci', 'dogs', 'webdev', 'google', 'Mil
itary', 'Paranormal', 'ragecomics', 'AskSocialScience', 'NeutralPolitics', 'privac
y', 'PoliticalDiscussion', 'software', 'conspiratard', 'islam', 'printSF', 'NorthK
oreaNews', 'Ask_Politics', 'moderatepolitics', 'commandline', 'hackers']
```

Out[31]:

| | sub | similarity |
|---|---|---|
| **699** | environment | 0.185935 |
| **567** | business | 0.170226 |
| **1062** | politics | 0.165885 |
| **341** | Libertarian | 0.155474 |
| **1113** | restorethefourth | 0.150343 |
| **1075** | progressive | 0.147675 |
| **206** | Foodforthought | 0.146236 |
| **340** | Liberal | 0.134288 |
| **1344** | worldnews | 0.133243 |
| **1184** | space | 0.132084 |
| **1199** | startups | 0.131689 |
| **1013** | offbeat | 0.131457 |
| **29** | AskHistorians | 0.131250 |
| **1020** | opensource | 0.128743 |
| **21** | AnythingGoesNews | 0.128490 |
| **1234** | technology | 0.128183 |
| **778** | geek | 0.127947 |
| **1010** | occupywallstreet | 0.127887 |
| **1176** | socialism | 0.126388 |
| **1003** | nottheonion | 0.118827 |

# Exercise: Item-Item Based Collaborative filtering in the MovieLens dataset

On this example we are going to build a new version of our Movie Recommendation. This time though, we will use correlation between ratings that users give as features.

In [32]:
```
ratings_df = pd.read_table('../data/ratings.dat', header=None, sep='::',
                           names=['user_id', 'movie_id', 'rating', 'timestamp'], eng
ine='python')

#we dont care about the time the rating was given
del ratings_df['timestamp']

#replace movie_id with movie_title for legibility
ratings_df = pd.merge(ratings_df, movies_df, on='movie_id')[['user_id', 'movie_titl
e', 'movie_id','rating', 'movie_genre']]

ratings_df.head()
```

Out[32]:

| | user_id | movie_title | movie_id | rating | movie_genre |
|---|---|---|---|---|---|
| **0** | 1 | One Flew Over the Cuckoo's Nest (1975) | 1193 | 5 | Drama |
| **1** | 2 | One Flew Over the Cuckoo's Nest (1975) | 1193 | 5 | Drama |
| **2** | 12 | One Flew Over the Cuckoo's Nest (1975) | 1193 | 4 | Drama |
| **3** | 15 | One Flew Over the Cuckoo's Nest (1975) | 1193 | 4 | Drama |
| **4** | 17 | One Flew Over the Cuckoo's Nest (1975) | 1193 | 5 | Drama |

In [33]:
```
ratings_mtx_df = ratings_df.pivot_table(values='rating', index='user_id',
columns='movie_title')

ratings_mtx_df = ratings_mtx_df.apply(lambda x: x.fillna(x.mean()),axis=0)
ratings_mtx_df = ratings_mtx_df.apply(lambda x: x - x.mean(),axis=1)

movie_index = ratings_mtx_df.columns

ratings_mtx_df.head()
```

Out[33]:

| movie_title | $1,000,000 Duck (1971) | 'Night Mother (1986) | 'Til There Was You (1997) | 'burbs, The (1989) | ...And Justice for All (1979) | 1-900 (1994) | 10 Things I Hate About You (1999) | 1( D (1 |
|---|---|---|---|---|---|---|---|---|
| **user_id** | | | | | | | | |
| **1** | -0.215674 | 0.128728 | -0.550393 | -0.331810 | 0.470867 | -0.742701 | 0.180156 | 0. |
| **2** | -0.210064 | 0.134337 | -0.544784 | -0.326200 | 0.476476 | -0.737091 | 0.185766 | 0. |
| **3** | -0.213476 | 0.130926 | -0.548195 | -0.329612 | 0.473065 | -0.740503 | 0.182354 | 0. |
| **4** | -0.212675 | 0.131726 | -0.547394 | -0.328811 | 0.473866 | -0.739702 | 0.183155 | 0. |
| **5** | -0.182296 | 0.162105 | -0.517016 | -0.298432 | 0.504244 | -0.709323 | 0.213534 | 0. |

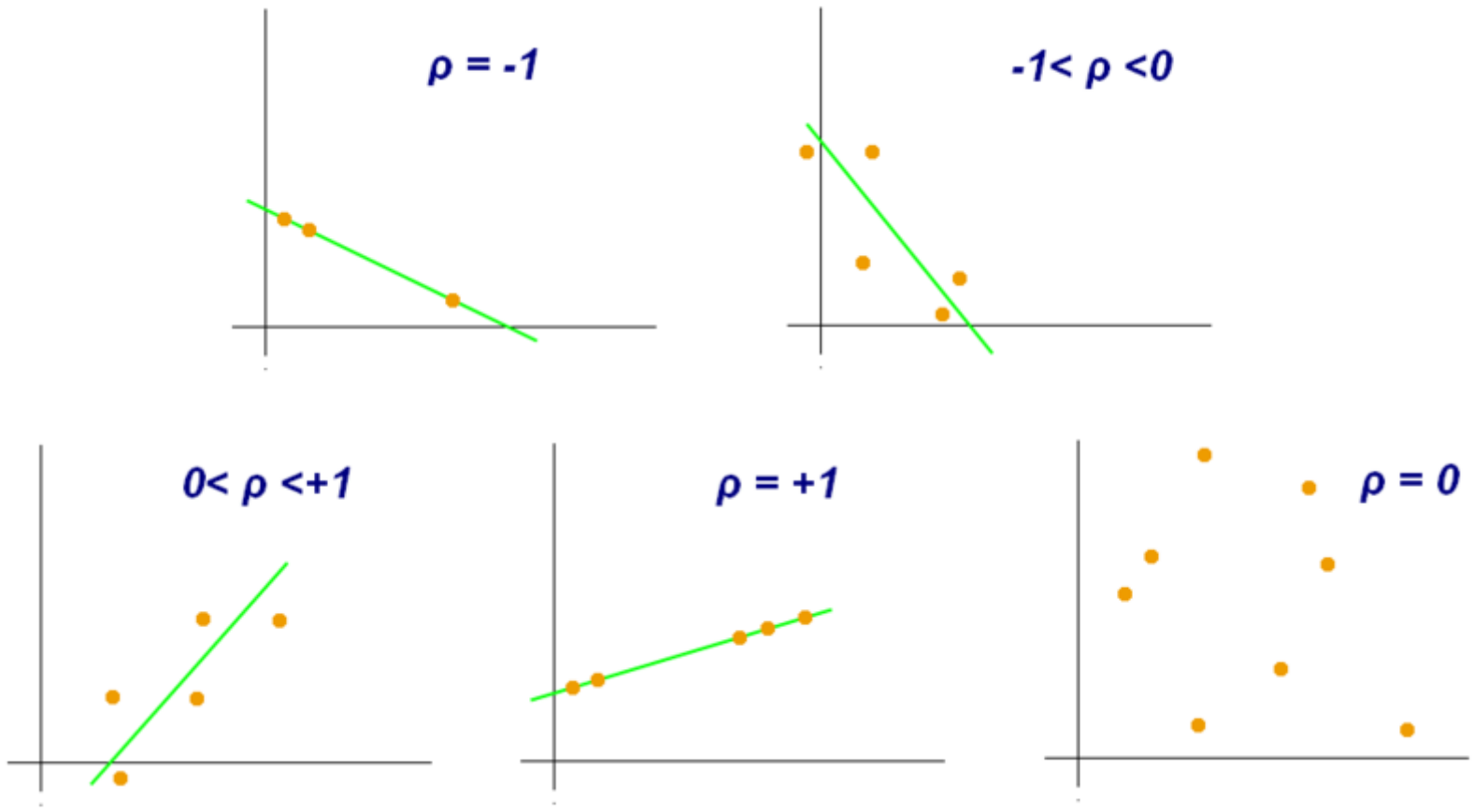
5 rows × 3706 columns

Since the user rating matrix is very sparse, we normalize it by

1. Filling non rated user-movie interactions with the users' average rating,
2. Deduct the movie bias by removing the movie's global average rating from each rating

Now that we have a normalized user-ratings matrix, we can compute the relationships between movies by calculating the correlation between movies and ratings.

We will use **Pearson product-moment correlation coefficient (PPMC)** to compute the similarities between movies based off the relation of the ratings users give to them.




In [34]:
```
corr_matrix = np.corrcoef(ratings_mtx_df.T)
corr_matrix.shape
```

Out[34]: (3706, 3706)

To compute the correlation we transpose the matrix so the movies are the rows. The result of calling the corrcoef method is a matrix with the movies as columns and rows and on each intersection the PMCC between those movies.

In [35]:
```
def get_movie_correlations(movie_title):
    '''Returns correlation vector for a movie'''
    movie_idx = list(movie_index).index(movie_title)
    return corr_matrix[movie_idx]
```

```
def get_similar_movies(movie_title, threshold=0.2):
    movie_correlations_array =  get_movie_correlations(movie_title)
    return movie_index[movie_correlations_array>threshold]
```

In [36]:
```
get_similar_movies('Star Wars: Episode IV - A New Hope (1977)')
```

Out[36]:
```
Index(['Raiders of the Lost Ark (1981)',
       'Star Wars: Episode IV - A New Hope (1977)',
       'Star Wars: Episode V - The Empire Strikes Back (1980)',
       'Star Wars: Episode VI - Return of the Jedi (1983)'],
      dtype='object', name='movie_title')
```

In [37]:
```
get_similar_movies('Die Hard (1988)')
```

Out[37]:
```
Index(['Die Hard (1988)', 'Die Hard 2 (1990)',
       'Die Hard: With a Vengeance (1995)', 'Lethal Weapon (1987)',
       'Lethal Weapon 2 (1989)', 'Terminator 2: Judgment Day (1991)',
       'Terminator, The (1984)'],
      dtype='object', name='movie_title')
```

In [38]:
```
def get_movie_recommendations(user_movies):
    '''given a set of movies, it returns all the movies sorted by their correlation
 with the user'''
    movie_similarities = np.zeros(corr_matrix.shape[0])
    for movie_id in user_movies:
        movie_similarities = movie_similarities + get_movie_correlations(movie_id)
    similarities_df = pd.DataFrame({
        'movie_title': movie_index,
        'sum_similarity': movie_similarities
        })
    similarities_df = similarities_df[~(similarities_df.movie_title.isin(user_movie
s))]
    similarities_df = similarities_df.sort_values(by=['sum_similarity'],
ascending=False)
    return similarities_df
```

How do we get the recommendations for a user,

we get the list of movies the user has reviewed, we create an empty array of lenght the number of movies Then for each movie of the user's movies, we sum the correlations of that movie to the array we just created. Then we sort the resulting similarities by descending order.

In [39]:
```
sample_user = 21
sample_user_movies =
ratings_df[ratings_df.user_id==sample_user].movie_title.tolist()
sample_user_movies
```

Out[39]:
```
["Bug's Life, A (1998)",
 'Bambi (1942)',
 'Antz (1998)',
 'Aladdin (1992)',
 'Toy Story (1995)',
 'Star Wars: Episode VI - Return of the Jedi (1983)',
 'Who Framed Roger Rabbit? (1988)',
 'South Park: Bigger, Longer and Uncut (1999)',
 'Akira (1988)',
 'Pinocchio (1940)',
 'Mad Max Beyond Thunderdome (1985)',
 'Titan A.E. (2000)',
 "Devil's Advocate, The (1997)",
 'Prince of Egypt, The (1998)',
 'Wild Wild West (1999)',
 'Iron Giant, The (1999)',
 'Brady Bunch Movie, The (1995)',
 'Princess Mononoke, The (Mononoke Hime) (1997)',
 'Little Nemo: Adventures in Slumberland (1992)',
 'Messenger: The Story of Joan of Arc, The (1999)',
 'Stop! Or My Mom Will Shoot (1992)',
 'House Party 2 (1991)']
```

In [40]:
```
recommendations = get_movie_recommendations(sample_user_movies)

recommendations.movie_title.head(20)
```

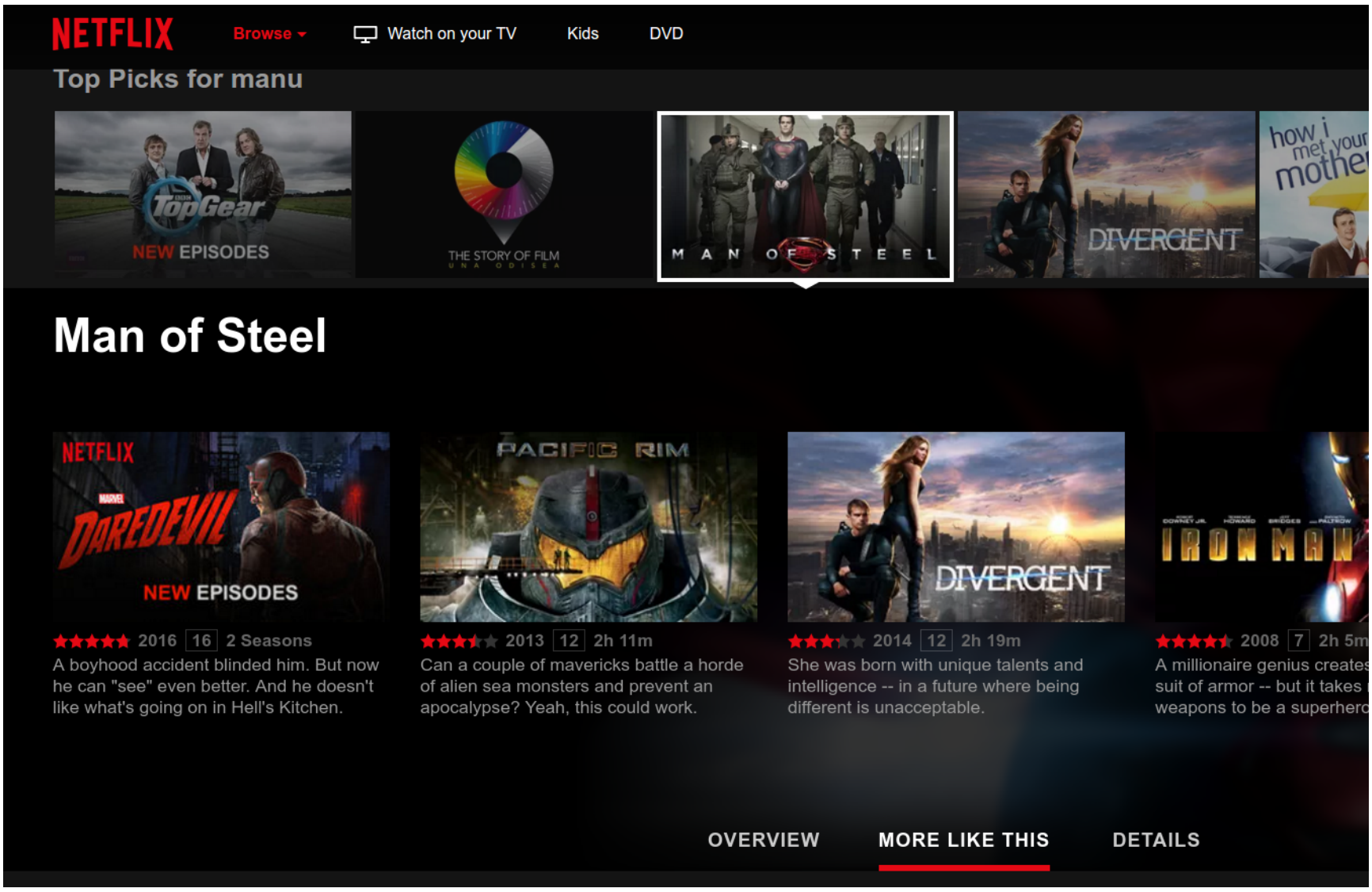
Out[40]:
```
3055    Snow White and the Seven Dwarfs (1937)
1865                 Lady and the Tramp (1955)
679                         Cinderella (1950)
1002                              Dumbo (1941)
1939                    Lion King, The (1994)
324                Beauty and the Beast (1991)
7                       101 Dalmatians (1961)
2770           Rescuers Down Under, The (1990)
1948                Little Mermaid, The (1989)
3412                        Toy Story 2 (1999)
2808                         Robin Hood (1973)
3026                    Sleeping Beauty (1959)
1739           James and the Giant Peach (1996)
3275                              Tarzan (1999)
97                  Alice in Wonderland (1951)
1781                    Jungle Book, The (1967)
1611         Hunchback of Notre Dame, The (1996)
2771                       Rescuers, The (1977)
2432                    Oliver & Company (1988)
1111                             Fantasia (1940)
Name: movie_title, dtype: object
```

## Hybrid Systems

Hybrid Systems are a way to limit the problems that Content Filtering and Collaborative Filtering.

- First, we use Content Filtering (so we get around the cold start problem)
- Once we have enough user data, we can use Collaborative Filtering.

## Example of Hybrid Systems: Netflix




Netflix is a good example of the use of hybrid recommender systems. They make recommendations by comparing the watching and searching habits of similar users (i.e. collaborative filtering) as well as by offering movies that share characteristics with films that a user has rated highly (content-based filtering).

## Libraries in python

- Crab (http://muricoca.github.io/crab/)
- PyFM (https://github.com/coreylynch/pyFM)
- python-recsys (https://github.com/ocelma/python-recsys)
- LightFM (https://github.com/lyst/lightfm)

## Big Data Recommendation Systems Frameworks

- Apache Spark MLLib Recommendation module (http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html)
- Apache Mahout (https://mahout.apache.org/)
- Oryx2 (http://oryx.io/index.html)
- LensKit (lenskit.org/)
- MyMediaLite (http://www.mymedialite.net/)

Benchmark on AM, LK and MML (http://www.slideshare.net/alansaid/comparative-recommende system-evaluation-benchmarking-recommendation-frameworks)

## Its 2016, so TensorFlow

- SVD implementation in TensorFlow (https://github.com/songgc/TF-recomm)
- Google's Play App recommendation Wide and Deep Network Paper (https://arxiv.org/pdf/1606.07792v1.pdf) (or a much more accesible tutorial (https://www.tensorflow.org/versions/r0.11/tutorials/wide_and_deep/index.html))

- Google's Paper on Deep Neural Networks on Youtube Recommendations (https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45530. (or a simpler article on the subject (https://www.infoq.com/news/2016/09/How-YouTube-Recommendation-Works))


Thanks for your time! . That is the best dog in the world, who happens to be my dog.

## Bonus Exercise!: Collaborative filtering by Singular Value Decomposition

## What is Singular Value Decomposition (SVD)?

Consider a matrix $A$ with $n$ rows and $d$ features.

The **singular value decomposition** of $A$ is given by:

$$A = U \Sigma V^T$$

$$(n \times d) \quad (n \times n) \quad (n \times d) \quad (d \times d)$$

st. $U, V$ are **orthogonal** matrices and $\Sigma$ is a **diagonal** matrix.

$$\rightarrow UU^T = I_n, \; VV^T = I_d \qquad \rightarrow \Sigma_{ij} = 0 \;\; (i \neq j)$$

SVD (Singular Value Decomposition) is a matrix factorization method, which basically means that is a method to decompose a matrix into a product of matrices.

| | user_id | movie_id | rating | movie_genre |
|---|---|---|---|---|

Given A, m×n real matrix with m>n, then A can be written using a so-called singular value decomposition of the form;

$$A = U\Sigma V^T$$

With:

- U being the left singular vectors
- Σ being a diagonal matrix containing the eigenvalues (singular values) of the original matrix A
- V^T being the right singular vectors

Full SVD is a computationally intensive process. However, we can select the the k higher eigenvalues and get a truncated version of A.

$$A_k \approx U_k \Sigma_k V_k^T$$

That holds similar information about the structure of A, but now the matrix is much less sparse as the original.

In [41]:
```
movies_df = pd.read_table('../data/movies.dat',  header=None, sep='::',
                          names=['movie_id', 'movie_title', 'movie_genre'],
engine='python')
movies_df.head()
```

Out[41]:

| | movie_id | movie_title | movie_genre |
|---|---|---|---|
| **0** | 1 | Toy Story (1995) | Animation\|Children's\|Comedy |
| **1** | 2 | Jumanji (1995) | Adventure\|Children's\|Fantasy |
| **2** | 3 | Grumpier Old Men (1995) | Comedy\|Romance |
| **3** | 4 | Waiting to Exhale (1995) | Comedy\|Drama |
| **4** | 5 | Father of the Bride Part II (1995) | Comedy |

In [42]:
```
ratings_df = pd.read_table('../data/ratings.dat', header=None, sep='::',
                           names=['user_id', 'movie_id', 'rating', 'timestamp'], en
gine='python')
del ratings_df['timestamp']
ratings_df.head()
```

Out[42]:

| | user_id | movie_id | rating |
|---|---|---|---|
| **0** | 1 | 1193 | 5 |
| **1** | 1 | 661 | 3 |
| **2** | 1 | 914 | 3 |
| **3** | 1 | 3408 | 4 |
| **4** | 1 | 2355 | 5 |

In [43]:
```
users = sorted(ratings_df.user_id.unique())
user_indices = {}

for i in range(len(users)):
    user_indices[users[i]] = i
```

Here we load a dictionary for users and another for movies, with the keys being the ids and the values being the range from 0 to the number of users or movies

In [44]:
```
ratings_mtx_df = ratings_df.pivot_table(values='rating', index='user_id',
columns='movie_id')
movie_index = ratings_mtx_df.columns
```

In [45]:
```
ratings_mtx_df.head()
```

Out[45]:

| movie_id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ... | 3943 | 3944 | 3945 | 394 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| user_id | | | | | | | | | | | | | | | |
| 1 | 5.0 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | ... | NaN | NaN | NaN | NaN |
| 2 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | ... | NaN | NaN | NaN | NaN |
| 3 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | ... | NaN | NaN | NaN | NaN |
| 4 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | ... | NaN | NaN | NaN | NaN |
| 5 | NaN | NaN | NaN | NaN | NaN | 2.0 | NaN | NaN | NaN | NaN | ... | NaN | NaN | NaN | NaN |

5 rows × 3706 columns

In [46]:
```
'''
http://web.eecs.umich.edu/~cscott/past_courses/eecs545f11/projects/AsendorfMcgaffin
PressSchwartz.pdf

Mixture Mean Algorithm for data imputation

0.452μuser(j) + 0.548μmovie(i)
'''

user_avgs = ratings_mtx_df.mean(axis=1)
movie_avgs = ratings_mtx_df.mean(axis=0)

user_alpha = 0.452
movie_alpha = 0.548

user_fill_values = user_alpha * user_avgs
movie_fill_values = movie_alpha * movie_avgs

for movie_id in ratings_mtx_df.columns:
    ratings_mtx_df.loc[:,movie_id][ratings_mtx_df.loc[:,movie_id].isnull()] = movie
_fill_values[movie_id] + user_fill_values
```

In [47]:
```
ratings_mtx_df.head()
```

Out[47]:

| movie_id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| user_id | | | | | | | | |
| 1 | 5.000000 | 3.647508 | 3.546455 | 3.389001 | 3.540986 | 4.018823 | 3.762226 | 3.54534 |
| 2 | 3.950828 | 3.432582 | 3.331528 | 3.174074 | 3.326059 | 3.803897 | 3.547300 | 3.33041 |
| 3 | 4.036158 | 3.517912 | 3.416858 | 3.259404 | 3.411389 | 3.889227 | 3.632630 | 3.41574 |
| 4 | 4.166567 | 3.648321 | 3.547267 | 3.389813 | 3.541798 | 4.019636 | 3.763038 | 3.54615 |
| 5 | 3.694674 | 3.176427 | 3.075374 | 2.917920 | 3.069905 | 2.000000 | 3.291145 | 3.07426 |

5 rows × 3706 columns

In [48]:
```
ratings_mtx = ratings_mtx_df.as_matrix()
ratings_mtx.shape
```

Out[48]: (6040, 3706)

We normalize the values by substracting the mean

In [49]:
```
ratings_mtx -= ratings_mtx.mean(axis=0)
```

In [50]:
```
from scipy.sparse.linalg import svds

u,s, vt = svds(ratings_mtx,  k = 100)
s_diag_matrix = np.zeros((s.shape[0], s.shape[0]))

for i in range(s.shape[0]):
    s_diag_matrix[i,i] = s[i]

X_lr = np.dot(np.dot(u, s_diag_matrix), vt)
```

Now we have a matrix that has the same rank as the original one but that is much less sparse

In [51]:
```
X_lr
```

Out[51]:
```
array([[ 0.52805675,  0.24178712,  0.32304414, ...,  0.22931869,
         0.22138418,  0.20595553],
       [ 0.01958462, -0.09468096,  0.09954601, ..., -0.00704268,
         0.00855895, -0.04017098],
       [-0.14653931,  0.06395302,  0.15608795, ...,  0.07448842,
         0.08648682,  0.03406283],
       ...,
       [-0.01194653,  0.18973783, -0.05391132, ...,  0.04176359,
         0.04398215,  0.07126904],
       [ 0.13725914,  0.00369558,  0.09266569, ...,  0.07563042,
         0.07424781,  0.11557894],
       [-0.33178927, -0.28589298, -0.12613732, ..., -0.03126921,
        -0.01449678, -0.07422521]])
```

In [52]:
```
X_lr.shape
```

Out[52]: (6040, 3706)

In [53]:
```
sample_user = 21
user_ratings = ratings_df[ratings_df.user_id==sample_user].merge(
                movies_df, on='movie_id'
               ).sort_values(by='rating',ascending=False)
user_ratings_movies = user_ratings.movie_id.tolist()
user_ratings
```

Out[53]:

| | user_id | movie_id | rating | movie_title | movie_genre |
|---|---|---|---|---|---|
| **1** | 21 | 2700 | 5 | South Park: Bigger, Longer and Uncut (1999) | Animation\|Comedy |
| **17** | 21 | 1210 | 5 | Star Wars: Episode VI - Return of the Jedi (1983) | Action\|Adventure\|Romance\|Sci-Fi\|War |
| **15** | 21 | 3745 | 5 | Titan A.E. (2000) | Adventure\|Animation\|Sci-Fi |
| **5** | 21 | 3000 | 5 | Princess Mononoke, The (Mononoke Hime) (1997) | Action\|Adventure\|Animation |
| **11** | 21 | 3704 | 4 | Mad Max Beyond Thunderdome (1985) | Action\|Sci-Fi |
| **9** | 21 | 2800 | 4 | Little Nemo: Adventures in Slumberland (1992) | Animation\|Children's |
| **19** | 21 | 3268 | 3 | Stop! Or My Mom Will Shoot (1992) | Action\|Comedy |
| **14** | 21 | 2294 | 3 | Antz (1998) | Animation\|Children's |
| **13** | 21 | 2355 | 3 | Bug's Life, A (1998) | Animation\|Children's\|Comedy |
| **12** | 21 | 2761 | 3 | Iron Giant, The (1999) | Animation\|Children's |
| **0** | 21 | 2987 | 3 | Who Framed Roger Rabbit? (1988) | Adventure\|Animation\|Film-Noir |
| **8** | 21 | 1274 | 3 | Akira (1988) | Adventure\|Animation\|Sci-Fi\|Thriller |
| **6** | 21 | 1 | 3 | Toy Story (1995) | Animation\|Children's\|Comedy |
| **4** | 21 | 588 | 3 | Aladdin (1992) | Animation\|Children's\|Comedy\|Musical |
| **3** | 21 | 585 | 3 | Brady Bunch Movie, The (1995) | Comedy |
| **2** | 21 | 2701 | 3 | Wild Wild West (1999) | Action\|Sci-Fi\|Western |
| **10** | 21 | 3053 | 1 | Messenger: The Story of Joan of Arc, The (1999) | Drama\|War |
| **7** | 21 | 596 | 1 | Pinocchio (1940) | Animation\|Children's |
| **16** | 21 | 1645 | 1 | Devil's Advocate, The (1997) | Crime\|Horror\|Mystery\|Thriller |
| **18** | 21 | 2018 | 1 | Bambi (1942) | Animation\|Children's |

| | user_id | movie_id | rating | movie_title | movie_genre |
|---|---|---|---|---|---|
| **20** | 21 | 2394 | 1 | Prince of Egypt, The (1998) | Animation\|Musical |
| **21** | 21 | 3774 | 1 | House Party 2 (1991) | Comedy |

So we see this user has a strong preference for animation , comedy and action

In [54]:
```
user_predicted_scores_df = pd.DataFrame({
        'movie_id': movie_index,
        'normalized_score': ratings_mtx[user_indices[sample_user],:],
        'pred_score': X_lr[user_indices[sample_user],:]})
user_predicted_scores_df = user_predicted_scores_df[-user_predicted_scores_df.movie
_id.isin(user_ratings_movies)]
user_predicted_scores_df.sort_values(by=['pred_score'], ascending=False).merge(movi
es_df, on='movie_id').head(20)
```

Out[54]:

| | movie_id | normalized_score | pred_score | movie_title | movie_genre |
|---|---|---|---|---|---|
| **0** | 2699 | -0.304906 | -0.157461 | Arachnophobia (1990) | Action\|Comedy\|Sci-Fi\|Thrille |
| **1** | 1196 | -0.491275 | -0.178976 | Star Wars: Episode V - The Empire Strikes Back... | Action\|Adventure\|Drama\|Sc Fi\|War |
| **2** | 1405 | -0.344705 | -0.191407 | Beavis and Butt-head Do America (1996) | Animation\|Comedy |
| **3** | 2012 | -0.334497 | -0.192460 | Back to the Future Part III (1990) | Comedy\|Sci-Fi\|Western |
| **4** | 2628 | -0.322285 | -0.203008 | Star Wars: Episode I - The Phantom Menace (1999) | Action\|Adventure\|Fantasy\|S |
| **5** | 2657 | -0.331750 | -0.204444 | Rocky Horror Picture Show, The (1975) | Comedy\|Horror\|Musical\|Sci- |
| **6** | 597 | -0.364036 | -0.212922 | Pretty Woman (1990) | Comedy\|Romance |
| **7** | 780 | -0.349633 | -0.233169 | Independence Day (ID4) (1996) | Action\|Sci-Fi\|War |
| **8** | 546 | -0.320233 | -0.236172 | Super Mario Bros. (1993) | Action\|Adventure\|Children's Fi |
| **9** | 849 | -0.322850 | -0.241353 | Escape from L.A. (1996) | Action\|Adventure\|Sci-Fi\|Thr |
| **10** | 1917 | -0.331129 | -0.243973 | Armageddon (1998) | Action\|Adventure\|Sci-Fi\|Thr |
| **11** | 3033 | -0.343274 | -0.245560 | Spaceballs (1987) | Comedy\|Sci-Fi |
| **12** | 442 | -0.335771 | -0.247444 | Demolition Man (1993) | Action\|Sci-Fi |
| **13** | 2427 | -0.348146 | -0.251078 | Thin Red Line, The (1998) | Action\|Drama\|War |

| | movie_id | normalized_score | pred_score | movie_title | movie_genre |
|---|---|---|---|---|---|
| **14** | 737 | -0.327698 | -0.253248 | Barb Wire (1996) | Action\|Sci-Fi |
| **15** | 1080 | -0.383726 | -0.255048 | Monty Python's Life of Brian (1979) | Comedy |
| **16** | 173 | -0.311637 | -0.256414 | Judge Dredd (1995) | Action\|Adventure\|Sci-Fi |
| **17** | 2022 | -0.358277 | -0.257692 | Last Temptation of Christ, The (1988) | Drama |
| **18** | 1882 | -0.318121 | -0.258350 | Godzilla (1998) | Action\|Sci-Fi |
| **19** | 1320 | -0.327142 | -0.263105 | Alien🔒 (1992) | Action\|Horror\|Sci-Fi\|Thriller |

# Thanks again!

In [ ]: