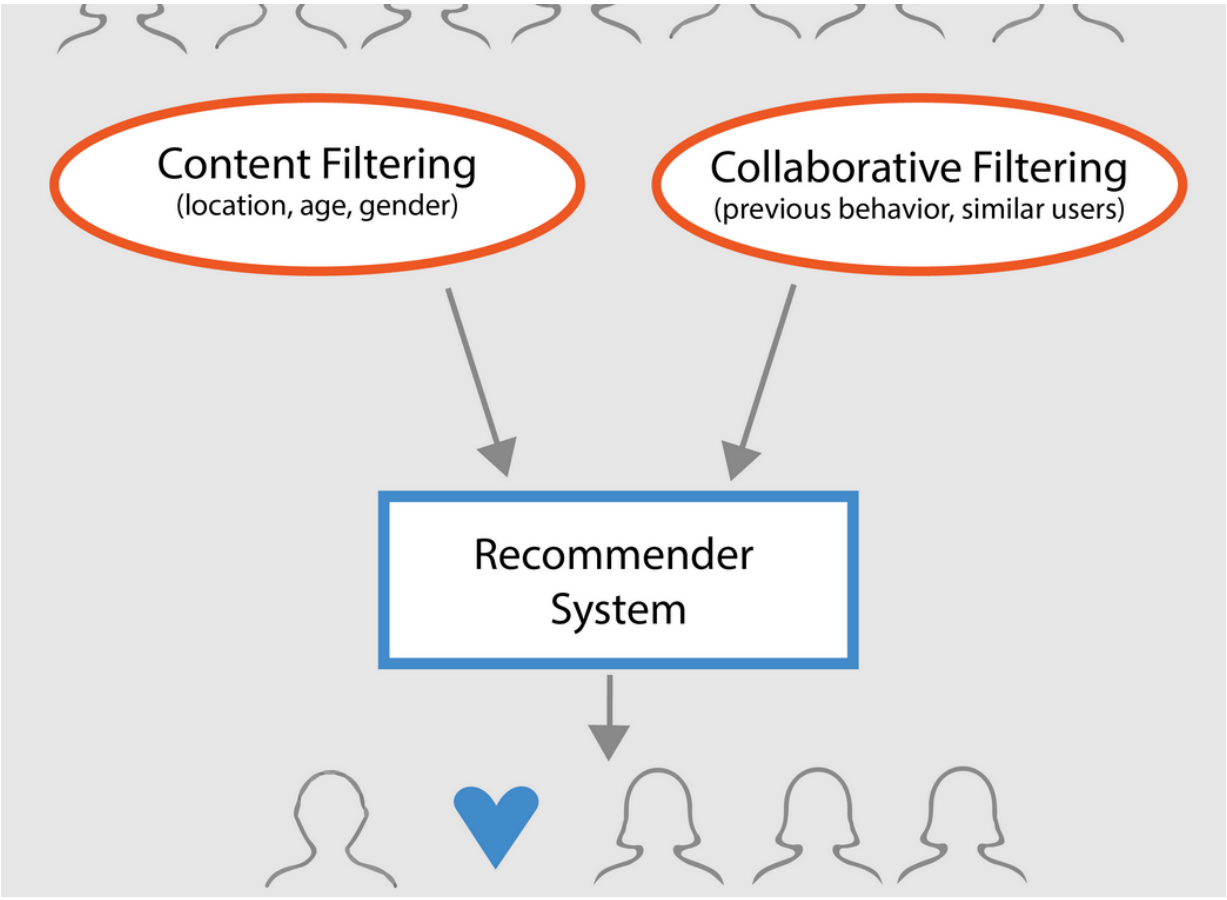
Aditya Sharma
May 29th, 2020

PYTHON

Beginner Tutorial: Recommender Systems in Python

Build your recommendation engine with the help of Python, from basic models to content-based and collaborative filtering recommender systems.



Source

The purpose of this tutorial is not to make you an expert in building recommender system models. Instead, the motive is to get you started by giving you an overview of the type of recommender systems that exist and how you can build one by yo

In this tutorial, you will learn how to build a basic model of simple and content-based recommender systems. While these models will be nowhere close to the industry standard in terms of complexity, quality, or accuracy, it will help you to get started with building more complex models that produce even better results.

Recommender systems are among the most popular applications of data science today. They are used to predict the "rating" or "preference" that a user would give to an item. Almost every major tech company has applied them in some form. Amazon uses it to suggest products to customers, YouTube uses it to decide which video to play next on autoplay, and Facebook uses it to recommend pages to like and people to follow.

Netflix even offered a million dollars in 2009 to anyone who could improve its system by 10%.

There are also popular recommender systems for domains like restaurants, movies, and online dating. Recommender systems have also been developed to explore research articles and experts, collaborators, and financial services. YouTube uses the recommendation system at a large scale to suggest you videos based on your history. For example, if you watch a lot of educational videos, it would suggest those types of videos.

But what are these recommender systems?

Broadly, recommender systems can be classified into 3 types:

- **Simple recommenders:** offer generalized recommendations to every user, based on movie popularity and/or genre. The basic idea behind this system is that movies that are more popular and critically acclaimed will have a higher probability of being liked by the average audience. An example could be IMDB Top 250.
- **Content-based recommenders:** suggest similar items based on a particular item. This system uses item metadata, such as genre, director, description, actors, etc. for movies, to make these recommendations. The general idea behind these recommender systems is that if a person likes a particular item, he or she will also like an item that is similar to it. And to recommend that, it will make use of the user's past item metadata. A good example could be YouTube, where based on your history, it suggests you new videos that you could potentially watch.
- **Collaborative filtering engines:** these systems are widely used, and they try to predict the rating or preference that a user would give an item-based on past ratings and preferences of other users. Collaborative filters do not require item metadata like its content-based counterparts.

Simple Recommenders

simplified clone of IMDB Top 250 Movies using metadata collected from IMDB.

The following are the steps involved:

- Decide on the metric or score to rate movies on.
- Calculate the score for every movie.
- Sort the movies based on the score and output the top results.

About the Dataset

The dataset files contain metadata for all 45,000 movies listed in the `Full MovieLens Dataset`. The dataset consists of movies released on or before July 2017. This dataset captures feature points like cast, crew, plot keywords, budget, revenue, posters, release dates, languages, production companies, countries, TMDB vote counts, and vote averages.

These feature points could be potentially used to train your machine learning models for content and collaborative filtering.

This dataset consists of the following files:

- *movies_metadata.csv*: This file contains information on ~45,000 movies featured in the Full MovieLens dataset. Features include posters, backdrops, budget, genre, revenue, release dates, languages, production countries, and companies.
- *keywords.csv*: Contains the movie plot keywords for our MovieLens movies. Available in the form of a stringified JSON Object.
- *credits.csv*: Consists of Cast and Crew Information for all the movies. Available in the form of a stringified JSON Object.
- *links.csv*: This file contains the TMDB and IMDB IDs of all the movies featured in the Full MovieLens dataset.

- *ratings_small.csv*: The subset of 100,000 ratings from 700 users on 9,000 movies.

The Full MovieLens Dataset comprises of 26 million ratings and 750,000 tag applications, from 270,000 users on all the 45,000 movies in this dataset. It can be accessed from the official GroupLens website.

Note: The subset dataset used in today's tutorial can be downloaded from here.

To load your dataset, you would be using the `pandas` DataFrame library. `pandas` library is mainly used for data manipulation and analysis. It represents your data in a row-column format. Pandas library is backed by the `NumPy` array for the implementation of pandas data objects. `pandas` offer off the shelf data structures and operations for manipulating numerical tables, time-series, imagery, and natural language processing datasets. Basically, `pandas` is useful for those datasets which can be easily represented in a tabular fashion.

Before you perform any of the above steps, let's load your movies metadata dataset into a `pandas` DataFrame:

```
# Import Pandas
import pandas as pd

# Load Movies Metadata
metadata = pd.read_csv('movies_metadata.csv', low_memory=False)

# Print the first three rows
metadata.head(3)
```

	adult	belongs_to_collection	budget	genres	h

0	False	{'id': 10194, 'name': 'Toy Story Collection', ...	30000000	[{'id': 16, 'name': 'Animation'}, {'id': 35, '...	http://toystory.disney.c story
1	False	NaN	65000000	[{'id': 12, 'name': 'Adventure'}, {'id': 14, '...	NaN
2	False	{'id': 119050, 'name': 'Grumpy Old Men Collect...	0	[{'id': 10749, 'name': 'Romance'}, {'id': 35, ...	NaN

3 rows × 24 columns

One of the most basic metrics you can think of is the ranking to decide which top 250 movies are based on their respective ratings.

However, using a rating as a metric has a few caveats:

- For one, it does not take into consideration the popularity of a movie. Therefore, a movie with a rating of 9 from 10 voters will be considered 'better' than a movie with a rating of 8.9 from 10,000 voters.

  For example, imagine you want to order Chinese food, you have a couple of options, one restaurant has a 5-star rating by only 5 people while the other restaurant has 4.5 ratings by 1000 people. Which restaurant would you prefer? The second one, right?

operational for a year.

- On a related note, this metric will also tend to favor movies with a smaller number of voters with skewed and/or extremely high ratings. As the number of voters increases, the rating of a movie regularizes and approaches towards a value that is reflective of the movie's quality and gives the user a much better idea as to which movie he/she should choose. While it is difficult to discern the quality of a movie with extremely few voters, you might have to consider external sources to conclude.

Taking these shortcomings into consideration, you must come up with a weighted rating that takes into account the average rating and the number of votes it has accumulated. Such a system will make sure that a movie with a 9 rating from 100,000 voters gets a (far) higher score than a movie with the same rating but a mere few hundred voters.

Since you are trying to build a clone of IMDB's Top 250, let's use its weighted rating formula as a metric/score. Mathematically, it is represented as follows:

$$Weighted Rating(\mathbf{WR}) = \left(\frac{\mathbf{v}}{\mathbf{v}+\mathbf{m}} \cdot \mathbf{R}\right) + \left(\frac{\mathbf{m}}{\mathbf{v}+\mathbf{m}} \cdot \mathbf{C}\right)$$

In the above equation,

- v is the number of votes for the movie;
- m is the minimum votes required to be listed in the chart;
- R is the average rating of the movie;
- C is the mean vote across the whole report.

You already have the values to `v (vote_count)` and `R (vote_average)` for each movie in the dataset. It is also possible to directly calculate C from this data.

Determining an appropriate value for `m` is a hyperparameter that you can choose accordingly since there is no right value for `m`. You can consider it as a preliminary negative

In this tutorial, you will use cutoff `m` as the 90th percentile. In other words, for a movie to be featured in the charts, it must have more votes than at least 90% of the movies on the list. (On the other hand, if you had chosen the 75th percentile, you would have considered the top 25% of the movies in terms of the number of votes garnered. As percentile decreases, the number of movies considered will increase).

As a first step, let's calculate the value of C, the mean rating across all movies using the pandas `.mean()` function:

```
# Calculate mean of vote average column
C = metadata['vote_average'].mean()
print(C)
```

```
5.618207215133889
```

From the above output, you can observe that the average rating of a movie on IMDB is around 5.6 on a scale of 10.

Next, let's calculate the number of votes, m, received by a movie in the 90th percentile. The `pandas` library makes this task extremely trivial using the `.quantile()` method of pandas:

```
# Calculate the minimum number of votes required to be in the chart, m
m = metadata['vote_count'].quantile(0.90)
print(m)
```

```
160.0
```

Since now you have the `m` you can simply use a greater than equal to condition to filter out movies having greater than equal to 160 vote counts:

q_movies DataFrame will not affect the original metadata data frame.

```
# Filter out all qualified movies into a new DataFrame
q_movies = metadata.copy().loc[metadata['vote_count'] >= m]
q_movies.shape
```

```
(4555, 24)
```

```
metadata.shape
```

```
(45466, 24)
```

From the above output, it is clear that there are around 10% movies with vote count more than 160 and qualify to be on this list.

Next and the most important step is to calculate the weighted rating for each qualified movie. To do this, you will:

- Define a function, `weighted_rating()` ;
- Since you already have calculated `m` and `C` you will simply pass them as an argument to the function;
- Then you will select the `vote_count` (v) and `vote_average` (R) column from the `q_movies` data frame;
- Finally, you will compute the weighted average and return the result.

You will define a new feature `score` , of which you'll calculate the value by applying this function to your DataFrame of qualified movies:

```
def weighted_rating(x, m=m, C=C):
    v = x['vote_count']
    R = x['vote_average']
    # Calculation based on the IMDB formula
    return (v/(v+m) * R) + (m/(m+v) * C)


# Define a new feature 'score' and calculate its value with `weighted_rating()`
q_movies['score'] = q_movies.apply(weighted_rating, axis=1)
```

Finally, let's sort the DataFrame in descending order based on the `score` feature column and output the title, vote count, vote average, and weighted rating (score) of the top 20 movies.

```
#Sort movies based on score calculated above
q_movies = q_movies.sort_values('score', ascending=False)

#Print the top 15 movies
q_movies[['title', 'vote_count', 'vote_average', 'score']].head(20)
```

	title	vote_count	vote_average	score
314	The Shawshank Redemption	8358.0	8.5	8.445869
834	The Godfather	6024.0	8.5	8.425439
10309	Dilwale Dulhania Le Jayenge	661.0	9.1	8.421453
12481	The Dark Knight	12269.0	8.3	8.265477
2843	Fight Club	9678.0	8.3	8.256385
292	Pulp Fiction	8670.0	8.3	8.251406
522	Schindler's List	4436.0	8.3	8.206639

23673	Whiplash	4376.0	8.3	8.205404
5481	Spirited Away	3968.0	8.3	8.196055
2211	Life Is Beautiful	3643.0	8.3	8.187171
1178	The Godfather: Part II	3418.0	8.3	8.180076
1152	One Flew Over the Cuckoo's Nest	3001.0	8.3	8.164256
351	Forrest Gump	8147.0	8.2	8.150272
1154	The Empire Strikes Back	5998.0	8.2	8.132919
1176	Psycho	2405.0	8.3	8.132715
18465	The Intouchables	5410.0	8.2	8.125837
40251	Your Name.	1030.0	8.5	8.112532
289	Leon: The Professional	4293.0	8.2	8.107234
3030	The Green Mile	4166.0	8.2	8.104511
1170	GoodFellas	3211.0	8.2	8.077459

Well, from the above output, you can see that the `simple recommender` did a great job!

Since the chart has a lot of movies in common with the IMDB Top 250 chart: for example, your top two movies, "Shawshank Redemption" and "The Godfather", are the same as IMDB and we all know they are indeed amazing movies, in fact, all top 20 movies do deserve to be in that list, isn't it?

Content-Based Recommender

Plot Description Based Recommender

similarity scores for all movies based on their plot descriptions and recommend movies based on that similarity score threshold.

The plot description is available to you as the `overview` feature in your `metadata` dataset. Let's inspect the plots of a few movies:

```
#Print plot overviews of the first 5 movies.
metadata['overview'].head()
```

```
0    Led by Woody, Andy's toys live happily in his ...
1    When siblings Judy and Peter discover an encha...
2    A family wedding reignites the ancient feud be...
3    Cheated on, mistreated and stepped on, the wom...
4    Just when George Banks has recovered from his ...
Name: overview, dtype: object
```

The problem at hand is a Natural Language Processing problem. Hence you need to extract some kind of features from the above text data before you can compute the similarity and/or dissimilarity between them. To put it simply, it is not possible to compute the similarity between any two overviews in their raw forms. To do this, you need to compute the word vectors of each overview or document, as it will be called from now on.

As the name suggests, word vectors are vectorized representation of words in a document. The vectors carry a semantic meaning with it. For example, man & king will have vector representations close to each other while man & woman would have representation far from each other.

You will compute `Term Frequency-Inverse Document Frequency` (TF-IDF) vectors for each document. This will give you a matrix where each column represents a word in the overview vocabulary (all the words that appear in at least one document), and each column represents a movie, as before.

importance of words that frequently occur in plot overviews and, therefore, their significance in computing the final similarity score.

Fortunately, scikit-learn gives you a built-in `TfIdfVectorizer` class that produces the TF-IDF matrix in a couple of lines.

- Import the Tfidf module using scikit-learn;
- Remove stop words like 'the', 'an', etc. since they do not give any useful information about the topic;
- Replace not-a-number values with a blank string;
- Finally, construct the TF-IDF matrix on the data.

```
#Import TfIdfVectorizer from scikit-learn
from sklearn.feature_extraction.text import TfidfVectorizer

#Define a TF-IDF Vectorizer Object. Remove all english stop words such as 'the', 'a'
tfidf = TfidfVectorizer(stop_words='english')

#Replace NaN with an empty string
metadata['overview'] = metadata['overview'].fillna('')

#Construct the required TF-IDF matrix by fitting and transforming the data
tfidf_matrix = tfidf.fit_transform(metadata['overview'])

#Output the shape of tfidf_matrix
tfidf_matrix.shape
```

```
(45466, 75827)
```

```
#Array mapping from feature integer indices to feature name.
tfidf.get_feature_names()[5000:5010]
```

```
['avaiis',
 'avaks',
 'avalanche',
 'avalanches',
 'avallone',
 'avalon',
 'avant',
 'avanthika',
 'avanti',
 'avaracious']
```

From the above output, you observe that 75,827 different vocabularies or words in your dataset have 45,000 movies.

With this matrix in hand, you can now compute a similarity score. There are several similarity metrics that you can use for this, such as the manhattan, euclidean, the Pearson, and the cosine similarity scores. Again, there is no right answer to which score is the best. Different scores work well in different scenarios, and it is often a good idea to experiment with different metrics and observe the results.

You will be using the `cosine similarity` to calculate a numeric quantity that denotes the similarity between two movies. You use the cosine similarity score since it is independent of magnitude and is relatively easy and fast to calculate (especially when used in conjunction with TF-IDF scores, which will be explained later). Mathematically, it is defined as follows:

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}^\intercal}{\|\mathbf{x}\| \cdot \|\mathbf{y}\|} = \frac{\sum_{i=1}^{n} \mathbf{x}_i \cdot \mathbf{y}^\intercal{}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{x}_i)^2} \sqrt{\sum_{i=1}^{n} (\mathbf{y}_i)^2}}$$

Since you have used the TF-IDF vectorizer, calculating the dot product between each vector will directly give you the cosine similarity score. Therefore, you will use `sklearn's linear_kernel()` instead of `cosine_similarities()` since it is faster.

1x45466 column vector where each column will be a similarity score with each movie.

```
# Import linear_kernel
from sklearn.metrics.pairwise import linear_kernel

# Compute the cosine similarity matrix
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
```

```
cosine_sim.shape
```

```
(45466, 45466)
```

```
cosine_sim[1]
```

```
array([0.01504121, 1.        , 0.04681953, ..., 0.        , 0.02198641,
       0.00929411])
```

You're going to define a function that takes in a movie title as an input and outputs a list of the 10 most similar movies. Firstly, for this, you need a reverse mapping of movie titles and DataFrame indices. In other words, you need a mechanism to identify the index of a movie in your `metadata` DataFrame, given its title.

```
#Construct a reverse map of indices and movie titles
indices = pd.Series(metadata.index, index=metadata['title']).drop_duplicates()
```

```
indices[:10]
```

```
title
Toy Story                      0
Jumanji                        1
```

```
Father of the Bride Part II    4
Heat                           5
Sabrina                        6
Tom and Huck                   7
Sudden Death                   8
GoldenEye                      9
dtype: int64
```

You are now in good shape to define your recommendation function. These are the following steps you'll follow:

- Get the index of the movie given its title.
- Get the list of cosine similarity scores for that particular movie with all movies. Convert it into a list of tuples where the first element is its position, and the second is the similarity score.
- Sort the aforementioned list of tuples based on the similarity scores; that is, the second element.
- Get the top 10 elements of this list. Ignore the first element as it refers to self (the movie most similar to a particular movie is the movie itself).
- Return the titles corresponding to the indices of the top elements.

```
# Function that takes in movie title as input and outputs most similar movies
def get_recommendations(title, cosine_sim=cosine_sim):
    # Get the index of the movie that matches the title
    idx = indices[title]

    # Get the pairwsie similarity scores of all movies with that movie
    sim_scores = list(enumerate(cosine_sim[idx]))

    # Sort the movies based on the similarity scores
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
```

```
    sim_scores = sim_scores[1:11]

    # Get the movie indices
    movie_indices = [i[0] for i in sim_scores]

    # Return the top 10 most similar movies
    return metadata['title'].iloc[movie_indices]


get_recommendations('The Dark Knight Rises')
```

```
12481                                     The Dark Knight
150                                        Batman Forever
1328                                       Batman Returns
15511                          Batman: Under the Red Hood
585                                                Batman
21194    Batman Unmasked: The Psychology of the Dark Kn...
9230                   Batman Beyond: Return of the Joker
18035                                     Batman: Year One
19792             Batman: The Dark Knight Returns, Part 1
3095                           Batman: Mask of the Phantasm
Name: title, dtype: object
```

```
get_recommendations('The Godfather')
```

```
1178               The Godfather: Part II
44030    The Godfather Trilogy: 1972-1990
1914              The Godfather: Part III
23126                          Blood Ties
11297                    Household Saints
34717                   Start Liquidation
10821                            Election
38030            A Mother Should Be Loved
17729                   Short Sharp Shock
```

You see that, while your system has done a decent job of finding movies with similar plot descriptions, the quality of recommendations is not that great. "The Dark Knight Rises" returns all Batman movies while it is more likely that the people who liked that movie are more inclined to enjoy other Christopher Nolan movies. This is something that cannot be captured by your present system.

Credits, Genres, and Keywords Based Recommender

The quality of your recommender would be increased with the usage of better metadata and by capturing more of the finer details. That is precisely what you are going to do in this section. You will build a recommender system based on the following metadata: the 3 top actors, the director, related genres, and the movie plot keywords.

The keywords, cast, and crew data are not available in your current dataset, so the first step would be to load and merge them into your main DataFrame `metadata`.

```
# Load keywords and credits
credits = pd.read_csv('credits.csv')
keywords = pd.read_csv('keywords.csv')

# Remove rows with bad IDs.
metadata = metadata.drop([19730, 29503, 35587])

# Convert IDs to int. Required for merging
keywords['id'] = keywords['id'].astype('int')
credits['id'] = credits['id'].astype('int')
metadata['id'] = metadata['id'].astype('int')

# Merge keywords and credits into your main metadata dataframe
metadata = metadata.merge(credits, on='id')
metadata = metadata.merge(keywords, on='id')
```

```
metadata.head(2)
```

	adult	belongs_to_collection	budget	genres	h
0	False	{'id': 10194, 'name': 'Toy Story Collection', ...	30000000	[{'id': 16, 'name': 'Animation'}, {'id': 35, '...	http://toystory.disney.c story
1	False	NaN	65000000	[{'id': 12, 'name': 'Adventure'}, {'id': 14, '...	NaN

2 rows × 27 columns

From your new features, cast, crew, and keywords, you need to extract the three most important actors, the director and the keywords associated with that movie.

But first things first, your data is present in the form of "stringified" lists. You need to convert them into a way that is usable for you.

```
# Parse the stringified features into their corresponding python objects
from ast import literal_eval

features = ['cast', 'crew', 'keywords', 'genres']
for feature in features:
    metadata[feature] = metadata[feature].apply(literal_eval)
```

Next, you write functions that will help you to extract the required information from each feature.

```
# Import Numpy
import numpy as np
```

Get the director's name from the crew feature. If the director is not listed, return `NaN`

```
def get_director(x):
    for i in x:
        if i['job'] == 'Director':
            return i['name']
    return np.nan
```

Next, you will write a function that will return the top 3 elements or the entire list, whichever is more. Here the list refers to the `cast`, `keywords`, and `genres`.

```
def get_list(x):
    if isinstance(x, list):
        names = [i['name'] for i in x]
        #Check if more than 3 elements exist. If yes, return only first three. If no, return e
        if len(names) > 3:
            names = names[:3]
        return names

    #Return empty list in case of missing/malformed data
    return []
```

```
# Define new director, cast, genres and keywords features that are in a suitable form.
metadata['director'] = metadata['crew'].apply(get_director)

features = ['cast', 'keywords', 'genres']
for feature in features:
    metadata[feature] = metadata[feature].apply(get_list)
```

```
# Print the new features of the first 3 films
metadata[['title', 'cast', 'director', 'keywords', 'genres']].head(3)
```

	title	cast	director	keywords	genres
0	Toy Story	[Tom Hanks, Tim Allen, Don Rickles]	John Lasseter	[jealousy, toy, boy]	[Animation, Comedy, Family]
1	Jumanji	[Robin Williams, Jonathan Hyde, Kirsten Dunst]	Joe Johnston	[board game, disappearance, based on children'...	[Adventure, Fantasy, Family]
2	Grumpier Old Men	[Walter Matthau, Jack Lemmon, Ann-Margret]	Howard Deutch	[fishing, best friend, duringcreditsstinger]	[Romance, Comedy]

The next step would be to convert the names and keyword instances into lowercase and strip all the spaces between them.

Removing the spaces between words is an important preprocessing step. It is done so that your vectorizer doesn't count the Johnny of "Johnny Depp" and "Johnny Galecki" as the same. After this processing step, the aforementioned actors will be represented as "johnnydepp" and "johnnygalecki" and will be distinct to your vectorizer.

Another good example where the model might output the same vector representation is "bread jam" and "traffic jam". Hence, it is better to strip off any space that is present.

The below function will exactly do that for you:

```
# Function to convert all strings to lower case and strip names of spaces
def clean_data(x):
    if isinstance(x, list):
        return [str.lower(i.replace(" ", "")) for i in x]
    else:
        #Check if director exists. If not, return empty string
```

```
        else:
            return ''


# Apply clean_data function to your features.
features = ['cast', 'keywords', 'director', 'genres']

for feature in features:
    metadata[feature] = metadata[feature].apply(clean_data)
```

You are now in a position to create your "metadata soup", which is a string that contains all the metadata that you want to feed to your vectorizer (namely actors, director and keywords).

The `create_soup` function will simply join all the required columns by a space. This is the final preprocessing step, and the output of this function will be fed into the word vector model.

```
def create_soup(x):
    return ' '.join(x['keywords']) + ' ' + ' '.join(x['cast']) + ' ' + x['director'] + ' ' + '
```

```
# Create a new soup feature
metadata['soup'] = metadata.apply(create_soup, axis=1)


metadata[['soup']].head(2)
```

	soup
0	jealousy toy boy tomhanks timallen donrickles ...
1	boardgame disappearance basedonchildren'sbook ...

`CountVectorizer()` instead of `TF-IDF`. This is because you do not want to down-weight the actor/director's presence if he or she has acted or directed in relatively more movies. It doesn't make much intuitive sense to down-weight them in this context.

The major difference between `CountVectorizer()` and `TF-IDF` is the inverse document frequency (IDF) component which is present in later and not in the former.

```
# Import CountVectorizer and create the count matrix
from sklearn.feature_extraction.text import CountVectorizer

count = CountVectorizer(stop_words='english')
count_matrix = count.fit_transform(metadata['soup'])


count_matrix.shape


(46628, 73881)
```

From the above output, you can see that there are 73,881 vocabularies in the metadata that you fed to it.

Next, you will use the `cosine_similarity` to measure the distance between the embeddings.

```
# Compute the Cosine Similarity matrix based on the count_matrix
from sklearn.metrics.pairwise import cosine_similarity

cosine_sim2 = cosine_similarity(count_matrix, count_matrix)


# Reset index of your main DataFrame and construct reverse mapping as before
metadata = metadata.reset_index()
indices = pd.Series(metadata.index, index=metadata['title'])
```

```
get_recommendations('The Dark Knight Rises', cosine_sim2)
```

```
12589      The Dark Knight
10210        Batman Begins
9311                Shiner
9874       Amongst Friends
7772              Mitchell
516      Romeo Is Bleeding
11463         The Prestige
24090            Quicksand
25038             Deadfall
41063                 Sara
Name: title, dtype: object
```

```
get_recommendations('The Godfather', cosine_sim2)
```

```
1934            The Godfather: Part III
1199             The Godfather: Part II
15609                   The Rain People
18940                        Last Exit
34488                             Rege
35802            Manuscripts Don't Burn
35803            Manuscripts Don't Burn
8001     The Night of the Following Day
18261                 The Son of No One
28683            In the Name of the Law
Name: title, dtype: object
```

Great! You see that your recommender has been successful in capturing more information due to more metadata and has given you better recommendations. There are, of course, numerous ways of experimenting with this system to improve recommendations.

- Introduce a popularity filter: this recommender would take the 30 most similar movies, calculate the weighted ratings (using the IMDB formula from above), sort movies based on this rating, and return the top 10 movies.
- Other crew members: other crew member names, such as screenwriters and producers, could also be included.
- The increasing weight of the director: to give more weight to the director, he or she could be mentioned multiple times in the soup to increase the similarity scores of movies with the same director.

Collaborative Filtering with Python

In this tutorial, you have learned how to build your very own Simple and Content-Based Movie Recommender Systems. There is also another extremely popular type of recommender known as collaborative filters.

Collaborative filters can further be classified into two types:

- `User-based Filtering` : these systems recommend products to a user that similar users have liked. For example, let's say Alice and Bob have a similar interest in books (that is, they largely like and dislike the same books). Now, let's say a new book has been launched into the market, and Alice has read and loved it. It is, therefore, highly likely that Bob will like it too, and therefore, the system recommends this book to Bob.
- `Item-based Filtering` : these systems are extremely similar to the content recommendation engine that you built. These systems identify similar items based on how people have rated it in the past. For example, if Alice, Bob, and Eve have given 5 stars to *The Lord of the Rings* and *The Hobbit*, the system identifies the items as similar. Therefore, if someone buys *The Lord of the Rings*, the system also recommends *The Hobbit* to him or her.

An example of collaborative filtering based on a rating system:




You will not be building these systems in this tutorial, but you are already familiar with most of the ideas required to do so. A good place to start with collaborative filters is by examining the MovieLens dataset, which can be found here.

Conclusion

Congratulations on finishing this tutorial!

You have successfully gone through our tutorial that taught you all about recommender systems in Python. You learned how to build simple and content-based recommenders.

One good exercise for you all would be to implement collaborative filtering in Python using the subset of MovieLens dataset that you used to build simple and content-based recommenders.

If you are just getting started in Python and would like to learn more, take DataCamp's Introduction to Data Science in Python course.