Lab 4: Building Content-Based Recommenders

In this lab, we are going to build two types of content-based recommender:

- **Plot description-based recommender:** This model compares the descriptions and taglines of different movies, and provides recommendations that have the most similar plot descriptions.
- **Metadata-based recommender:** This model takes a host of features, such as genres, keywords, cast, and crew, into consideration and provides recommendations that are the most similar with respect to the aforementioned features.

Exporting the clean DataFrame

In the previous lab, we performed a series of data wrangling and cleaning processes on our metadata in order to convert it into a form that was more usable. To avoid having to perform these steps again, let's save this cleaned DataFrame into a CSV file. As always, doing this with pandas happens to be extremely easy.

In the knowledge recommender notebook from Lab 4, enter the following code in the last cell:

```
#Convert the cleaned (non-exploded) dataframe df into a CSV file and save it in the
data folder
#Set parameter index to False as the index of the DataFrame has no inherent meaning.
df.to_csv('../data/metadata_clean.csv', index=False)
```

Your [data] folder should now contain a new file, [metadata_clean.csv].

Let's create a new folder, [Lab 4], and open a new Jupyter Notebook within this folder. Let's now import our new file into this Notebook:

```
import random
import pandas as pd
import numpy as np

#Import data from the clean file
df = pd.read_csv('../data/metadata_clean.csv')

#Print the head of the cleaned DataFrame
df.head()
```

The cell should output a DataFrame that is already clean and in the desired form.

Document vectors

To quantify the similarity between documents, we represent them as vectors, where each document is a series of *n* numbers, with *n* being the size of the combined vocabulary. The values of these vectors depend on the *vectorizer* used, such as **CountVectorizer** or **TF-IDFVectorizer**, which convert text into numerical representations for similarity calculations.

CountVectorizer

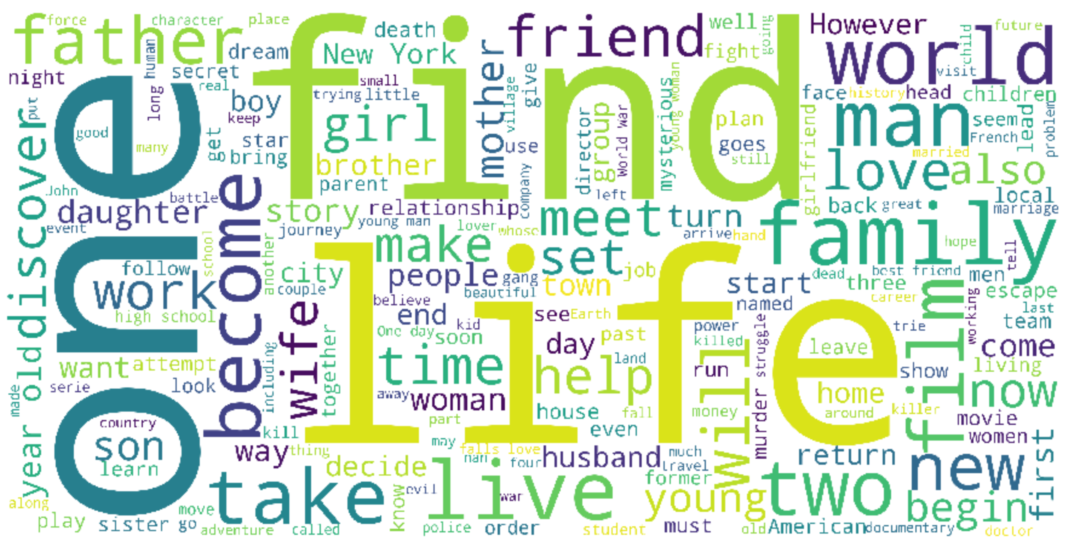
CountVectorizer converts documents into numerical vectors based on word frequency. For example, given three documents, we first create a vocabulary of unique words (ignoring common stop words). After eliminating words like "the" and "is," the vocabulary becomes: *like, little, lamb, love, mary, red, rose, sun, star*.

Each document is then represented as a vector where each dimension counts the occurrences of these words. For example:

- **A**: (0, 0, 0, 0, 0, 0, 0, 1, 1)
- **B**: (1, 0, 0, 1, 0, 2, 1, 0, 0)
- **C**: (0, 1, 1, 0, 1, 0, 0, 0, 0)

These vectors allow us to compare document similarities based on word frequency.

TF-IDFVectorizer



We use **TF-IDFVectorizer** to assign weights to words based on their importance, improving the representation of plot descriptions. It also speeds up the calculation of **cosine similarity** between documents by emphasizing key terms.

Plot description-based recommender

Our plot description-based recommender will take in a movie title as an argument and output a list of movies that are most similar based on their plots. These are the steps we are going to perform in building this model:

1. Obtain the data required to build the model
2. Create TF-IDF vectors for the plot description (or overview) of every movie
3. Compute the pairwise cosine similarity score of every movie
4. Write the recommender function that takes in a movie title as an argument and outputs movies most similar to it based on the plot

Preparing the data

In its present form, the DataFrame, although clean, does not contain the features that are required to build the plot description-based recommender. Fortunately, these requisite features are available in the original metadata file.

All we have to do is import them and add them to our DataFrame:

```
#Import the original file
orig_df = pd.read_csv('../data/movies_metadata.csv', low_memory=False)

#Add the useful features into the cleaned dataframe
df['overview'], df['id'] = orig_df['overview'], orig_df['id']

df.head()
```

The DataFrame should now contain two new features: [overview] and [id]. We will use [overview] in building this model and [id] for building the next.

The [overview] feature consists of strings and, ideally, we should clean them up by removing all punctuation and converting all the words to lowercase. However, as we will see shortly, all this will be done for us automatically by [scikit-learn], the library we're going to use heavily in building the models in this lab.

Creating the TF-IDF matrix

The next step is to create a DataFrame where each row represents the TF-IDF vector of the [overview] feature of the corresponding movie in our main DataFrame. To do this, we will use the [scikit-learn] library, which gives us access to a TfidfVectorizer object to perform this process effortlessly:

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

#Define a TF-IDF Vectorizer Object. Remove all english stopwords
tfidf = TfidfVectorizer(stop_words='english')

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

#Construct the required TF-IDF matrix by applying the fit_transform method on the
overview feature
tfidf_matrix = tfidf.fit_transform(df['overview'])

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

Output

```
OUTPUT:
(45466, 75827)
```

We see that the vectorizer has created a 75,827-dimensional vector for the overview of every movie.

Computing the cosine similarity score

Like TF-IDFVectorizer, [scikit-learn] also has functionality for computing the aforementioned similarity matrix. Calculating the cosine similarity is, however, a computationally expensive process. Fortunately, since our movie plots are represented as TF-IDF vectors, their magnitude is always 1. Hence, we do not need to calculate the denominator in the cosine similarity formula as it will always be 1. Our work is now reduced to computing the much simpler and computationally cheaper dot product (a functionality that is also provided by [scikit-learn]):

```
# Import linear_kernel to compute the dot product
from sklearn.metrics.pairwise import linear_kernel

sample_indices = random.sample(range(tfidf_matrix.shape[0]), 15000)
tfidf_matrix_sampled = tfidf_matrix[sample_indices]

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

Although we're computing the cheaper dot product, the process will still take a few minutes to complete. With the similarity scores of every movie with every other movie, we are now in a very good position to write our final recommender function.

Building the recommender function

The final step is to create our recommender function. However, before we do that, let's create a reverse mapping of movie titles and their respective indices. In other words, let's create a pandas series with the index as the movie title and the value as the corresponding index in the main DataFrame:

```
#Construct a reverse mapping of indices and movie titles, and drop duplicate titles,
if any
indices = pd.Series(df.index, index=df['title']).drop_duplicates()
```

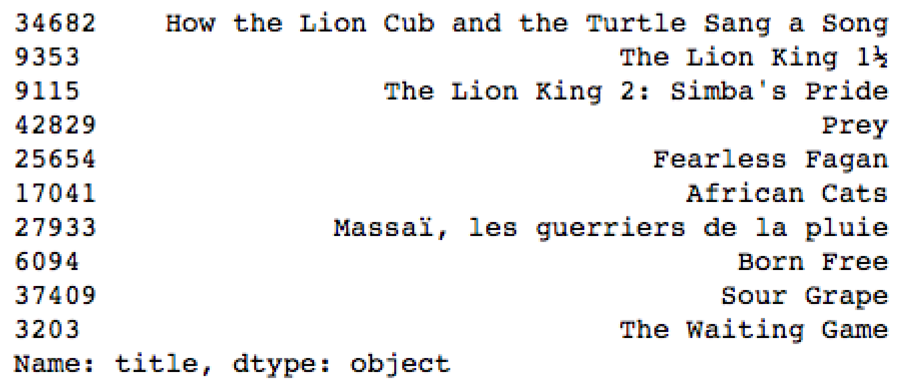
We will perform the following steps in building the recommender function:

1. Declare the title of the movie as an argument.
2. Obtain the index of the movie from the [indices] reverse mapping.
3. Get the list of cosine similarity scores for that particular movie with all movies using [cosine_sim]. Convert this into a list of tuples where the first element is the position and the second is the similarity score.
4. Sort this list of tuples on the basis of the cosine similarity scores.
5. Get the top 10 elements of this list. Ignore the first element as it refers to the similarity score with itself (the movie most similar to a particular movie is obviously the movie itself).
6. Return the titles corresponding to the indices of the top 10 elements, excluding the first:

```
# Function that takes in movie title as input and gives recommendations
def content_recommender(title, cosine_sim=cosine_sim, df=df, indices=indices):
    # Obtain the index of the movie that matches the title
    idx = indices[title]

    # Get the pairwsie similarity scores of all movies with that movie
    # And convert it into a list of tuples as described above
    sim_scores = list(enumerate(cosine_sim[idx]))

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

    # Get the scores of the 10 most similar movies. Ignore the first movie.
    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 df['title'].iloc[movie_indices]
```

Congratulations! You've built your very first content-based recommender. Now it is time to see our recommender in action! Let's ask it for recommendations of movies similar to [The Lion King]*:*

```
#Get recommendations for The Lion King
content_recommender('The Lion King')
```



```
34682    How the Lion Cub and the Turtle Sang a Song
9353                                 The Lion King 1½
9115                   The Lion King 2: Simba's Pride
42829                                            Prey
25654                                  Fearless Fagan
17041                                    African Cats
27933               Massaï, les guerriers de la pluie
6094                                        Born Free
37409                                      Sour Grape
3203                                 The Waiting Game
Name: title, dtype: object
```

Our current recommender suggests mostly *The Lion King* sequels and lion-themed movies, but it misses broader preferences like Disney or animated films. To improve, we'll build a new system using advanced metadata (genres, cast, crew, keywords) to better capture individual tastes in directors, actors, and sub-genres.

Metadata-based recommender

We will largely follow the same steps as the plot description-based recommender to build our metadata-based model. The main difference, of course, is in the type of data we use to build the model.

Preparing the data

To build this model, we will be using the following metdata:

- The genre of the movie.
- The director of the movie. This person is part of the crew.
- The movie's three major stars. They are part of the cast.
- Sub-genres or keywords.

With the exception of genres, our DataFrames (both original and cleaned) do not contain the data that we require. Therefore, for this exercise, we will need two additional files: [credits.csv], which contains information on the cast and crew of the movies, and [keywords.csv], which contains information on the sub-genres.

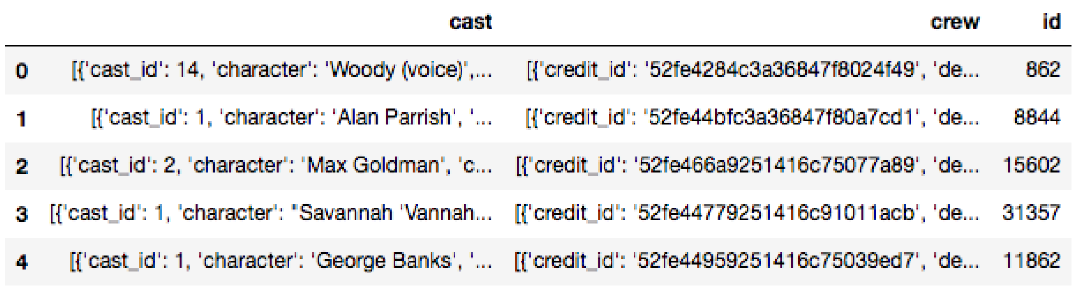
You can view the necessary files from the following URL: https://www.kaggle.com/rounakbanik/the-movies-dataset/data.

Place both files in your [data] folder. We need to perform a good amount of wrangling before the data is converted into a form that is usable. Let's begin!

The keywords and credits datasets

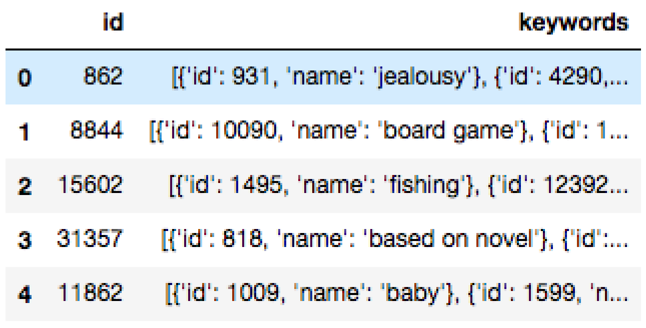
Let's start by loading our new data into the existing Jupyter Notebook:

```
# Load the keywords and credits files
cred_df = pd.read_csv('../data/credits.csv')
key_df = pd.read_csv('../data/keywords.csv')

#Print the head of the credit dataframe
cred_df.head()
```



	cast	crew	id
0	[{'cast_id': 14, 'character': 'Woody (voice)',...	[{'credit_id': '52fe4284c3a36847f8024f49', 'de...	862
1	[{'cast_id': 1, 'character': 'Alan Parrish', '...	[{'credit_id': '52fe44bfc3a36847f80a7cd1', 'de...	8844
2	[{'cast_id': 2, 'character': 'Max Goldman', 'c...	[{'credit_id': '52fe466a9251416c75077a89', 'de...	15602
3	[{'cast_id': 1, 'character': "Savannah 'Vannah...	[{'credit_id': '52fe44779251416c91011acb', 'de...	31357
4	[{'cast_id': 1, 'character': 'George Banks', '...	[{'credit_id': '52fe44959251416c75039ed7', 'de...	11862

```
#Print the head of the keywords dataframe
key_df.head()
```



	id	keywords
0	862	[{'id': 931, 'name': 'jealousy'}, {'id': 4290,...
1	8844	[{'id': 10090, 'name': 'board game'}, {'id': 1...
2	15602	[{'id': 1495, 'name': 'fishing'}, {'id': 12392...
3	31357	[{'id': 818, 'name': 'based on novel'}, {'id':...
4	11862	[{'id': 1009, 'name': 'baby'}, {'id': 1599, 'n...

We can see that the cast, crew, and the keywords are in the familiar [list of dictionaries] form. Just like [genres], we have to reduce them to a string or a list of strings.

Before we do this, however, we will join the three DataFrames so that all our features are in a single DataFrame. Joining pandas DataFrames is identical to joining tables in SQL. The key we're going to use to join the DataFrames is the [id] feature. However, in order to use this, we first need to explicitly convert is listed as an ID. This is clearly bad data. Therefore, we should fin into an integer. We already know how to do this:

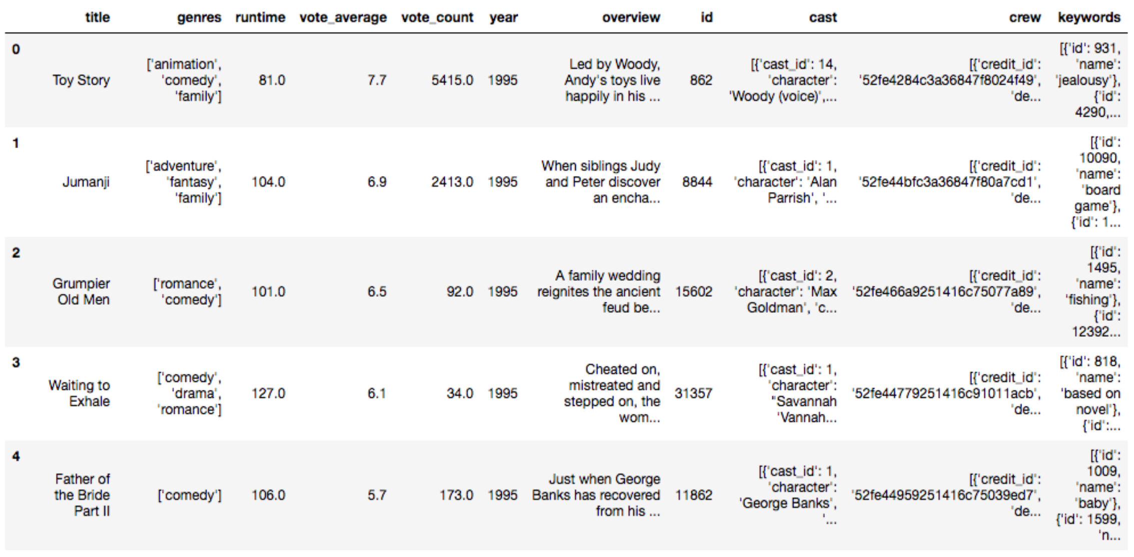
```
#Convert the IDs of df into int
df['id'] = df['id'].astype('int')
```

Running the preceding code results in a [ValueError]. On closer inspection, we see that *1997-08-20* is listed as an ID. This is clearly bad data. Therefore, we should find all the rows with bad IDs and remove them in order for the code execution to be successful:

```
# Function to convert all non-integer IDs to NaN
def clean_ids(x):
    try:
        return int(x)
    except:
        return np.nan

#Clean the ids of df
df['id'] = df['id'].apply(clean_ids)

#Filter all rows that have a null ID
df = df[df['id'].notnull()]
```

We are now in a good position to convert the IDs of all three DataFrames into integers and merge them into a single DataFrame:

```
# Convert IDs into integer
df['id'] = df['id'].astype('int')
key_df['id'] = key_df['id'].astype('int')
cred_df['id'] = cred_df['id'].astype('int')

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

#Display the head of the merged df
df.head()
```



	title	genres	runtime	vote_average	vote_count	year	overview	id	cast	crew	keywords
0	Toy Story	['animation', 'comedy', 'family']	81.0	7.7	5415.0	1995	Led by Woody, Andy's toys live happily in his ...	862	[{'cast_id': 14, 'character': 'Woody (voice)',...	[{'credit_id': '52fe4284c3a36847f8024f49', 'de...	[{'id': 931, 'name': 'jealousy'}, {'id': 4290,...
1	Jumanji	['adventure', 'fantasy', 'family']	104.0	6.9	2413.0	1995	When siblings Judy and Peter discover an encha...	8844	[{'cast_id': 1, 'character': 'Alan Parrish', '...	[{'credit_id': '52fe44bfc3a36847f80a7cd1', 'de...	[{'id': 10090, 'name': 'board game'}, {'id': 1...
2	Grumpier Old Men	['romance', 'comedy']	101.0	6.5	92.0	1995	A family wedding reignites the ancient feud be...	15602	[{'cast_id': 2, 'character': 'Max Goldman', 'c...	[{'credit_id': '52fe466a9251416c75077a89', 'de...	[{'id': 1495, 'name': 'fishing'}, {'id': 12392...
3	Waiting to Exhale	['comedy', 'drama', 'romance']	127.0	6.1	34.0	1995	Cheated on, mistreated and stepped on, the wom...	31357	[{'cast_id': 1, 'character': "Savannah 'Vannah...	[{'credit_id': '52fe44779251416c91011acb', 'de...	[{'id': 818, 'name': 'based on novel'}, {'id':...
4	Father of the Bride Part II	['comedy']	106.0	5.7	173.0	1995	Just when George Banks has recovered from his ...	11862	[{'cast_id': 1, 'character': 'George Banks', '...	[{'credit_id': '52fe44959251416c75039ed7', 'de...	[{'id': 1009, 'name': 'baby'}, {'id': 1599, 'n...

Wrangling keywords, cast, and crew

Now that we have all the desired features in a single DataFrame, let's convert them into a form that is more usable. More specifically, these are the transformations we will be looking to perform:

- Convert [keywords] into a list of strings where each string is a keyword (similar to genres). We will include only the top three keywords. Therefore, this list can have a maximum of three elements.
- Convert [cast] into a list of strings where each string is a star. Like [keywords], we will only include the top three stars in our cast.
- Convert [crew] into [director]. In other words, we will extract only the director of the movie and ignore all other crew members.

The first step is to convert these stringified objects into native Python objects:

```
# Convert the stringified objects into the native python objects
from ast import literal_eval

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

Next, let's extract the director from our [crew] list. To do this, we will first examine the structure of the dictionary in the [crew] list:

```
#Print the first cast member of the first movie in df
df.iloc[0]['crew'][0]
```

Output

```
OUTPUT:
{'credit_id': '52fe4284c3a36847f8024f49',
 'department': 'Directing',
 'gender': 2,
 'id': 7879,
 'job': 'Director',
 'name': 'John Lasseter',
 'profile_path': '/7EdqiNbr4FRjIhKHyPPdFfEEEFG.jpg'}
```

We see that this dictionary consists of [job] and [name] keys. Since we're only interested in the director, we will loop through all the crew members in a particular list and extract the [name] when the [job] is [Director]. Let's write a function that does this:

```
# Extract the director's name. If director is not listed, return NaN
def get_director(x):
    for crew_member in x:
        if crew_member['job'] == 'Director':
            return crew_member['name']
    return np.nan
```

Now that we have the [get_director] function, we can define the new [director] feature:

```
#Define the new director feature
df['director'] = df['crew'].apply(get_director)
```

```
#Print the directors of the first five movies
df['director'].head()
```

Output

```
OUTPUT:
0 John Lasseter
1 Joe Johnston
2 Howard Deutch
3 Forest Whitaker
4 Charles Shyer
Name: director, dtype: object
```

Both [keywords] and [cast] are dictionary lists as well. And, in both cases, we need to extract the top three [name] attributes of each list. Therefore, we can write a single function to wrangle both these features. Also, just like [keywords] and [cast], we will only consider the top three genres for every movie:

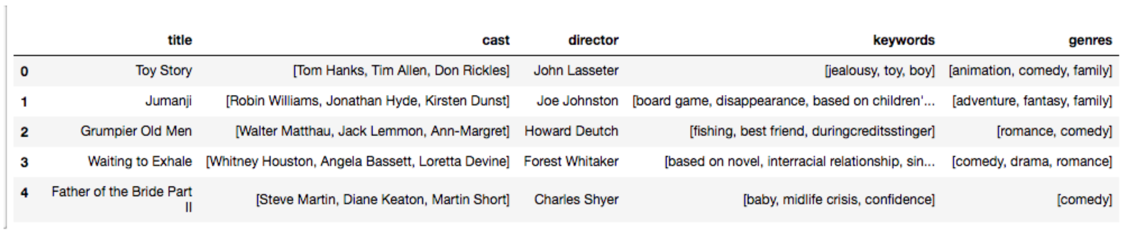
```
# Returns the list top 3 elements or entire list; whichever is more.
def generate_list(x):
    if isinstance(x, list):
        names = [ele['name'] for ele in x]
        #Check if more than 3 elements exist. If yes, return only first three.
        #If no, return entire list.
        if len(names) > 3:
            names = names[:3]
        return names

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

We will use this function to wrangle our [cast] and [keywords] features. We will also only consider the first three [genres] listed:

```
#Apply the generate_list function to cast and keywords
df['cast'] = df['cast'].apply(generate_list)
df['keywords'] = df['keywords'].apply(generate_list)

#Only consider a maximum of 3 genres
df['genres'] = df['genres'].apply(lambda x: x[:3])
```

Let's now take a look at a sample of our wrangled data:

```
# Print the new features of the first 5 movies along with title
df[['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]
3	Waiting to Exhale	[Whitney Houston, Angela Bassett, Loretta Devine]	Forest Whitaker	[based on novel, interracial relationship, sin...	[comedy, drama, romance]
4	Father of the Bride Part II	[Steve Martin, Diane Keaton, Martin Short]	Charles Shyer	[baby, midlife crisis, confidence]	[comedy]

In the subsequent steps, we are going to use a vectorizer to build document vectors. If two actors had the same first name (say, Ryan Reynolds and Ryan Gosling), the vectorizer will treat both Ryans as the same, although they are clearly different entities. This will impact the quality of the recommendations we receive. If a person likes Ryan Reynolds' movies, it doesn't imply that they like movies by all Ryans.

Therefore, the last step is to strip the spaces between keywords, and actor and director names, and convert them all into lowercase. Therefore, the two Ryans in the preceding example will become *ryangosling* and *ryanreynolds*, and our vectorizer will now be able to distinguish between them:

```
# Function to sanitize data to prevent ambiguity.
# Removes spaces and converts to lowercase
def sanitize(x):
    if isinstance(x, list):
        #Strip spaces and convert to lowercase
        return [str.lower(i.replace(" ", "")) for i in x]
    else:
        #Check if director exists. If not, return empty string
        if isinstance(x, str):
            return str.lower(x.replace(" ", ""))
        else:
            return ''
#Apply the generate_list function to cast, keywords, director and genres
for feature in ['cast', 'director', 'genres', 'keywords']:
    df[feature] = df[feature].apply(sanitize)
```

Creating the metadata soup

In the plot description-based recommender, we worked with a single *overview* feature, which was a body of text. Therefore, we were able to apply our vectorizer directly.

However, this is not the case with our metadata-based recommender. We have four features to work with, of which three are lists and one is a string. What we need to do is create a [soup] that contains the actors, director, keywords, and genres. This way, we can feed this soup into our vectorizer and perform similar follow-up steps to before:

```
#Function that creates a soup out of the desired metadata
def create_soup(x):
    return ' '.join(x['keywords']) + ' ' + ' '.join(x['cast']) + ' ' + x['director'] +
' ' + ' '.join(x['genres'])
```

With this function in hand, we create the [soup] feature:

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

Let's now take a look at one of the [soup] values. It should be a string containing words that represent genres, cast, and keywords:

```
#Display the soup of the first movie
df.iloc[0]['soup']
```

Output

```
OUTPUT:
'jealousy toy boy tomhanks timallen donrickles johnlasseter animation comedy family'
```

With the [soup] created, we are now in a good position to create our document vectors, compute similarity scores, and build the metadata-based recommender function.

Generating the recommendations

The next steps are almost identical to the corresponding steps from the previous section.

Instead of using TF-IDFVectorizer, we will be using CountVectorizer. This is because using TF-IDFVectorizer will accord less weight to actors and directors who have acted and directed in a relatively larger number of movies.

This is not desirable, as we do not want to penalize artists for directing or appearing in more movies:

```
#Define a new CountVectorizer object and create vectors for the soup
count = CountVectorizer(stop_words='english')
count_matrix = count.fit_transform(df['soup'])
```

Unfortunately, using CountVectorizer means that we are forced to use the more computationally expensive [cosine_similarity] function to compute our scores:

```
#Import cosine_similarity function
from sklearn.metrics.pairwise import cosine_similarity

sample_indices = random.sample(range(count_matrix.shape[0]), 15000)
count_matrix_sampled = count_matrix[sample_indices]

#Compute the cosine similarity score (equivalent to dot product for tf-idf vectors)
cosine_sim2 = cosine_similarity(count_matrix_sampled, count_matrix_sampled)
```

Since we dropped a few movies with bad indices, we need to construct our reverse mapping again. Let's do that as the next step:

```
# Reset index of your df and construct reverse mapping again
df = df.reset_index()
indices2 = pd.Series(df.index, index=df['title'])
```

With the new reverse mapping constructed and the similarity scores computed, we can reuse the [content_recommender] function defined in the previous section by passing in [cosine_sim2] as an argument. Let's now try out our new model by asking recommendations for the same movie, [The Lion King]*:*

```
content_recommender('The Lion King', cosine_sim2, df, indices2)
```

```
29607                                          Cheburashka
40904                   VeggieTales: Josh and the Big Wall
40913    VeggieTales: Minnesota Cuke and the Search for...
27768                                 The Little Matchgirl
15209              Spiderman: The Ultimate Villain Showdown
16613                            Cirque du Soleil: Varekai
24654                                  The Seventh Brother
29198                                       Superstar Goofy
30244                                               My Love
31179                Pokémon: Arceus and the Jewel of Life
Name: title, dtype: object
```

The recommendations given in this case are vastly different to the ones that our plot description-based recommender gave. We see that it has been able to capture more information than just lions. Most of the movies in the list are animated and feature anthropomorphic characters.

Personally, I found the *Pokemon: Arceus and the Jewel of Life* recommendation especially interesting. Both this movie and *The Lion King* feature cartoon anthropomorphic characters who return after a few years to exact revenge on those who had wronged them.

Suggestions for improvements

The content-based recommenders we've built in this lab are, of course, nowhere near the powerful models used in the industry. There is still plenty of scope for improvement. In this section, I will suggest a few ideas for upgrading the recommenders that you've already built:

- **Experiment with the number of keywords, genres, and cast**: In the model that we built, we considered at most three keywords, genres, and actors for our movies. This was, however, an arbitrary decision. It is a good idea to experiment with the number of these features in order to be considered for the metadata soup.
- **Come up with more well-defined sub-genres**: Our model only considered the first three keywords that appeared in the keywords list. There was, however, no justification for doing so. In fact, it is entirely possible that certain keywords appeared in only one movie (thus rendering them useless). A much more potent technique would be to define, as with the genres, a definite number of sub-genres and assign only these sub-genres to the movies.
- **Give more weight to the director**: Our model gave as much importance to the director as to the actors. However, you can argue that the character of a movie is determined more by the former. We can give more emphasis to the director by mentioning this individual multiple times in our soup instead of just once. Experiment with the number of repetitions of the director in the soup.
- **Consider other members of the crew**: The director isn't the only person that gives the movie its character. You can also consider adding other crew members, such as producers and screenwriters, to your soup.
- **Experiment with other metadata**: We only considered genres, keywords, and credits while building our metadata model. However, our dataset contains plenty of other features, such as production companies, countries, and languages. You may consider these data points, too, as they may be able to capture important information (such as if two movies are produced by *Pixar)*.
- **Introduce a popularity filter**: It is entirely possible that two movies have the same genres and sub-genres, but differ wildly in quality and popularity. In such cases, you may want to introduce a popularity filter that

considers the *n* most similar movies, computes a weighted rating, and displays the top five results. You have already learned how to do this in the previous lab.

Summary

We have come a long way in this lab. We first learned about document vectors and gained a brief introduction to the cosine similarity score. Next, we built a recommender that identified movies with similar plot descriptions. We then proceeded to build a more advanced model that leveraged the power of other metadata, such as genres, keywords, and credits. Finally, we discussed a few methods by which we could improve our existing system.

With this, we formally come to an end of our tour of content-based recommendation system. In the next labs, we will cover what is arguably the most popular recommendation model in the industry today: collaborative filtering.