Lab 1: Getting Started with Recommender Systems

There is no lab for this module.

Lab 2: Manipulating Data with the Pandas Library

In the next few portions of the course, we are going to get our hands dirty by building the various kinds of recommender systems. However, before we do so, it is important that we know how to handle, manipulate, and analyze data efficiently in Python.

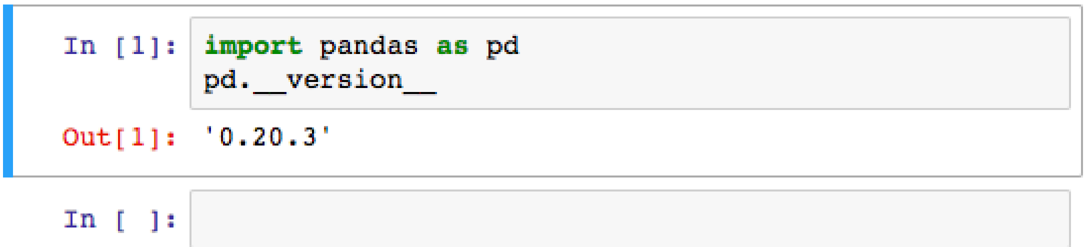
In this lab, we're going to get ourselves acquainted with the pandas library, which aims to overcome the aforementioned limitations, making data analysis in Python extremely efficient and user-friendly. We'll also introduce ourselves to the *Movies Dataset* that we're going to use to build our recommenders as well as use pandas to extract some interesting facts and narrate the history of movies using data.

Setting up the environment

Open Lab environment URL and open Jupyter lab or notebook and write following code:

```
import pandas as pd
pd.__version__
```

To execute the code in this cell, press *Shift* + *Enter*. If all goes well, you should see a new output cell, which prints the version of the pandas library:



The Pandas library

As a first step toward working with pandas, let's import our movies data into our Jupyter Notebook. To do this, we need the path to where our dataset is located. This can be a URL on the internet or your local computer. We highly recommend downloading the data to your local computer and accessing it from a local path instead of from a web URL.

NOTE: All datasets used in the labs are available in lab environment and GitHub repository:

`https://github.com/fenago/recommendation-systems-python/tree/main/data`

Go to the following URL to view the required CSV file: [https://www.kaggle.com/rounakbanik/the-movies-dataset/downloads/movies_metadata.csv/7.](https://www.kaggle.com/rounakbanik/the-movies-dataset/downloads/movies_metadata.csv/7)

Now, let's witness some pandas magic. In the Jupyter Notebook you ran in the previous section, go to the second cell and type the following code:

```
#Read the CSV File into df
df = pd.read_csv('../data/movies_metadata.csv')
```

```
#We will find out what the following code does a little later!
df.head()
```

Et voila! You should be able to see a table-like structure with five rows, each row representing a movie. You can also see that the table has 24 columns, although the columns were truncated to fit in the display.

What is this structure though? Let's find out by running the familiar [type] command:

```
#Output the type of df
type(df)
```

You should get an output stating that df is a [pandas.core.frame.DataFrame]. In other words, our code has read the CSV file into a pandas DataFrame object. But what are DataFrames? Let's find that out in the next section.

The Pandas DataFrame

As we saw in the previous section, the [df.head()] code outputted a table-like structure. In essence, the DataFrame is just that: a two-dimensional data structure with columns of different data types. You can think of it as an SQL Table. Of course, just being a table of rows and columns isn't what makes the DataFrame special. The DataFrame gives us access to a wide variety of functionality, some of which we're going to explore in this section.

Each row in our DataFrame represents a movie. But how many movies are there? We can find this out by running the following code:

```
#Output the shape of df
df.shape

OUTPUT:
(45466, 24)
```

The result gives us the number of rows and columns present in df. We can see that we have data on 45,466 movies.

We also see that we have 24 columns. Each column represents a feature or a piece of metadata about the movie. When we ran [df.head()], we saw that most of the columns were truncated to fit in the display. To view all the columns (henceforth, called features) we have, we can run the following:

```
#Output the columns of df
df.columns

OUTPUT:

Index(['adult', 'belongs_to_collection', 'budget', 'genres', 'homepage', 'id',
       'imdb_id', 'original_language', 'original_title', 'overview',
       'popularity', 'poster_path', 'production_companies',
       'production_countries', 'release_date', 'revenue', 'runtime',
       'spoken_languages', 'status', 'tagline', 'title', 'video',
       'vote_average', 'vote_count'],
      dtype='object')
```

We see that we have a lot of information on these movies, including their title, budget, genres, release date, and revenue.

Next, let's find out how to access a particular movie (or row). The first way to do this is by using the [.iloc] method. This allows us to select rows based on the numeric position, starting from zero. For example, if we wanted to access the second movie in the DataFrame, we'd run:

```
#Select the second movie in df
second = df.iloc[1]
second
```

The output will give you information about the movie on each of its 24 features. We see that the title of the movie is *Jumanji* and that it was released on December 15th, 1995, among other things.

A cell will always print the output of the last line of code. Therefore, we don't need to explicitly write it within a [print] function.

The second way to do it is by accessing the DataFrame index. Since we didn't explicitly set an index while reading the CSV file, pandas defaulted it to zero-based indexing. We can change the index of df quite easily. Let's change the index to the title of the movie and try to access [Jumanji] using this index:

```
#Change the index to the title
df = df.set_index('title')

#Access the movie with title 'Jumanji'
jum = df.loc['Jumanji']
jum
```

You should see an output identical to the previous cell. Let's revert back to our zero-based numeric index:

```
#Revert back to the previous zero-based indexing
df = df.reset_index()
```

It is also possible to create a new, smaller DataFrame with fewer columns. Let's create a new DataFrame that only has the following features: [title], [release_date], [budget], [revenue], [runtime], and [genres]:

```
#Create a smaller dataframe with a subset of all features
small_df = df[['title', 'release_date', 'budget', 'revenue', 'runtime', 'genres']]

#Output only the first 5 rows of small_df
small_df.head()
```

You should see a table with five movies and only the features that we've mentioned. The [.head()] method simply displays the first five rows of the DataFrame. You can display as many rows as you want by passing it as an argument into [.head()]*:*

```
#Display the first 15 rows
small_df.head(15)
```

Next, let's check out the data types of our various features:

```
#Get information of the data types of each feature
small_df.info()

OUTPUT:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45466 entries, 0 to 45465
Data columns (total 6 columns):
title 45460 non-null object
```

```
release_date 45379 non-null object
budget 45466 non-null object
revenue 45460 non-null float64
runtime 45203 non-null float64
genres 45466 non-null object
dtypes: float64(2), object(4)
memory usage: 2.1+ MB
```

A curious observation here is that pandas correctly deciphers [revenue] and [runtime] as float data, but assigns the generic object data type to [budget].

However, pandas allows us to manually convert the data type of a feature. Let's try to convert the [budget] feature to [float]:

```
#Convert budget to float
df['budget'] = df['budget'].astype('float')

OUTPUT:
...
...
ValueError: could not convert string to float: '/zaSf5OG7V8X8gqFvly88zDdRm46.jpg'
```

Running this cell throws [ValueError]. It is easy to guess that one of the budget fields had a ['/zaSf...'] string as its value, and pandas was not able to convert this into a floating number.

To solve this problem, we will use the [apply()] method. This will allow us to apply a function to every field in a particular column and convert it into the return value. We are going to convert every number field in [budget] to float and, if that fails, convert it to [NaN]:

```
#Import the numpy library
import numpy as np

#Function to convert to float manually
def to_float(x):
    try:
        x = float(x)
    except:
        x = np.nan
    return x

#Apply the to_float function to all values in the budget column
small_df['budget'] = small_df['budget'].apply(to_float)

#Try converting to float using pandas astype
small_df['budget'] = small_df['budget'].astype('float')

#Get the data types for all features
small_df.info()
```

This time around, there are no errors thrown. Also, we notice that the [budget] feature is now of the [float64] type.

Now, let's try to define a new feature, called [year], that represents the year of release. The recommended way to do this would be by using the [datetime] functionality that pandas gives us:

```
#Convert release_date into pandas datetime format
small_df['release_date'] = pd.to_datetime(small_df['release_date'], errors='coerce')

#Extract year from the datetime
small_df['year'] = small_df['release_date'].apply(lambda x: str(x).split('-')[0] if x
!= np.nan else np.nan)

#Display the DataFrame with the new 'year' feature
small_df.head()
```

What are the oldest movies available in this dataset? To answer this question, we can sort the DataFrame based on the year of release:

```
#Sort DataFrame based on release year
small_df = small_df.sort_values('year')

small_df.head()
```

We see that we have movies from as early as the 1870s, with *Passage of Venus* being the oldest movie on record. Next, let's find out the most successful movies of all time. To do this, we'll use the [sort_values()] method once again, but with an additional [ascending=False] parameter to sort [DataFrame] in descending order:

```
#Sort Movies based on revenue (in descending order)
small_df = small_df.sort_values('revenue', ascending=False)

small_df.head()
```

From our results, we observe that *Avatar* is the most successful movie of all time, with a revenue of over $2.78 billion.

Let's say we wanted to create a new DataFrame of movies that satisfied a certain condition. For instance, we only want movies that earned more than $1 billion. Pandas makes this possible using its Boolean Indexing feature. Let's see this in action:

```
#Select only those movies which earned more than 1 billion
new = small_df[small_df['revenue'] > 1e9]
new
```

It is also possible to apply multiple conditions. For instance, let's say we only wanted movies that earned more than $1 billion, but where the outlay less than $150 million, we'd do it as follows:

```
#Select only those movies which earned more than 1 billion and spent less than 150
million

new2 = small_df[(small_df['revenue'] > 1e9) & (small_df['budget'] < 1.5e8)]
new2
```

Only four movies make it into this list.

There is, of course, much more to what you can do with DataFrames (such as handling missing data), but we'll stop our exploration with it for now. Let's move on to another data structure we have unknowingly used extensively in this section: the Pandas Series.

The Pandas Series

When we accessed the Jumanji movie using [.loc] and [.iloc], the data structures returned to us were Pandas Series objects. You may have also noticed that we were accessing entire columns using [df[column_name]]. This, too, was a Pandas Series object:

```
type(small_df['year'])

OUTPUT:
pandas.core.series.Series
```

The Pandas Series is a one-dimensional labelled array capable of holding data of any type. You may think of it as a Python list on steroids. When we were using the [.apply()] and [.astype()] methods in the previous section, we were actually using them on these Series objects.

Therefore, like the DataFrame, the Series object comes with its own group of extremely useful methods that make data analysis a breeze.

First, let's check out the shortest- and longest-running movies of all time. We will do this by accessing the [runtime] column of the DataFrame as a Series object and applying its methods on it:

```
#Get the runtime Series object
runtime = small_df['runtime']

#Print the longest runtime of any movie
print(runtime.max())

#Print the shortest runtime of any movie
print(runtime.min())
```

We see that the longest movie is more than 1,256 minutes in length and the shortest is 0! Of course, such strange results demand a deeper inspection of the data but we shall skip that, for now.

It is also possible to calculate the mean and median of the Series in this way. Let's do so for the movie budgets:

```
#Get the budget Series object
budget = small_df['budget']

#Print the mean budget of the movies
print(budget.mean())

#Print the median budget of the movies
print(budget.median())
```

The average budget of a movie is $4.2 million and the median budget is 0! This suggests that at least half the movies in our dataset have no budget at all! Like in the previous case, such strange results demand closer inspection. In this case, it is highly likely that a zero budget indicates that the data is not available.

What is the revenue that the 90th-percentile movie generated? We can discover this using the [quantile] function:

```
#Get the revenue Series object
revenue = small_df['revenue']
```

```
#Revenue generated by the 90th percentile movie
revenue.quantile(0.90)
```

We get a result of $8.26 million. What this means is that only 10% of the movies in our dataset earned more than $8.26 million in revenue.

Finally, let's find out the number of movies released each year. We do this using the [value_counts()] method on the [year] series:

```
#Get number of movies released each year
small_df['year'].value_counts()
```

We have the highest number of movies released in 2014. There are also six years in our dataset (including 2020) that have only one movie on record.

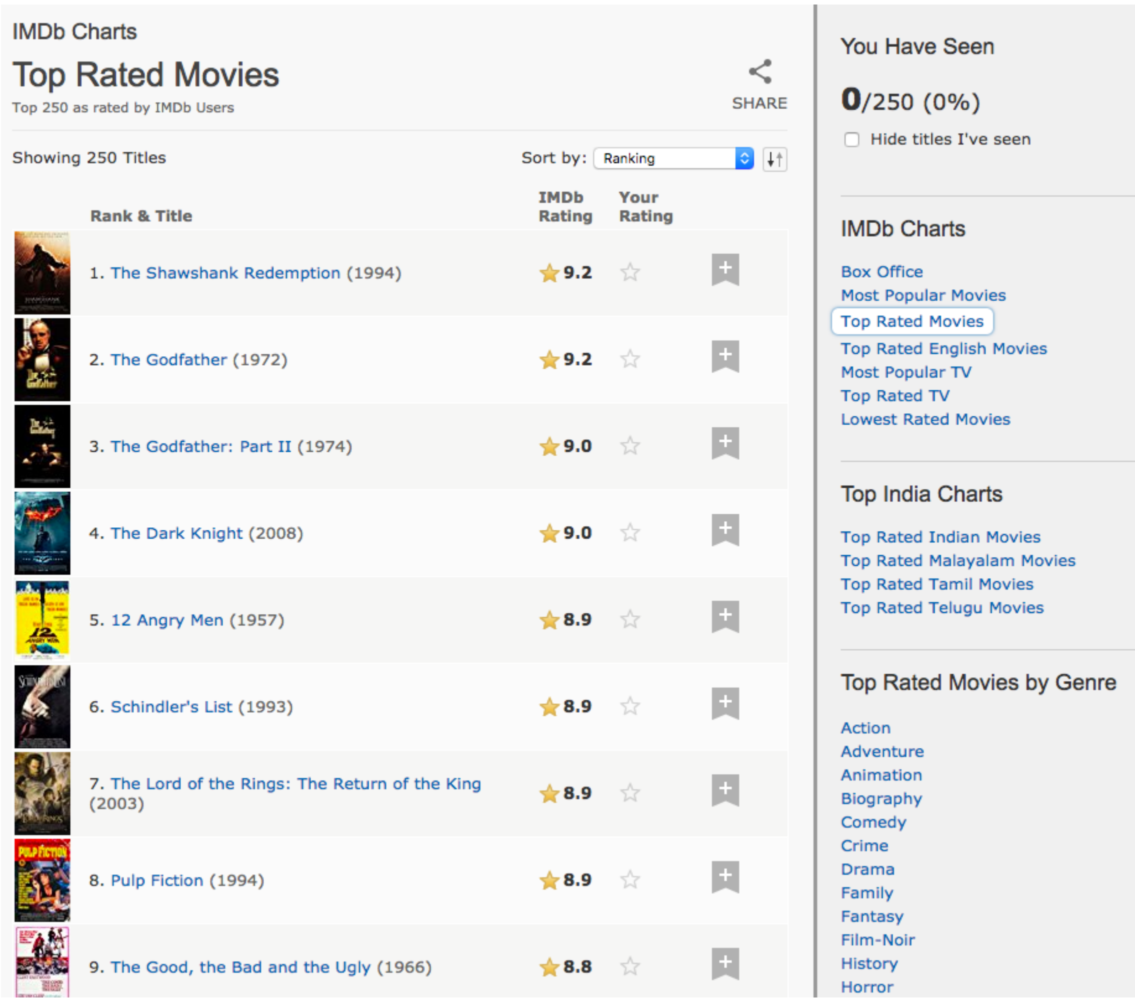
You may rename the notebook as [Lab2] by clicking on [Untitled] and then close it. For the next lab, we will create a new notebook.

Summary

In this lab, we gained an understanding of the limitations of using vanilla Python and its built-in data structures. We acquainted ourselves with the Pandas library and learned how it overcomes the aforementioned difficulties by giving us access to extremely powerful and easy-to-use data structures. We then explored the two main data structures, Series and DataFrame, by analyzing our movies-metadata dataset.

Lab 3: Building an IMDB Top 250 Clone with Pandas

The **Internet Movie Database** (**IMDB**) maintains a chart called the IMDB Top 250, which is a ranking of the top 250 movies according to a certain scoring metric. All the movies in this list are non-documentary, theatrical releases with a runtime of at least 45 minutes and over 250,000 ratings:



This chart can be considered the simplest of recommenders. It doesn't take into consideration the tastes of a particular user, nor does it try to deduce similarities between different movies. It simply calculates a score for every movie based on a predefined metric and outputs a sorted list of movies based on that score.

In this lab, we will be covering the following:

- Building a clone of the IMDB Top 250 chart (henceforth referred to as the simple recommender).
- Taking the functionalities of the chart one step further and building a knowledge-based recommender. This model takes user preferences with regards to genre, timeframe, runtime, language, and so on, and recommends movies that satisfy all conditions.

The simple recommender

The first step in building our simple recommender is setting up our workspace. Let's create a new directory named [Lab3]. Create a Jupyter Notebook in this directory named [Simple Recommender] and open it in the browser.

Let's now load the dataset we used in the previous lab into our notebook.

In case you have not downloaded it already, the dataset is available at https://www.kaggle.com/rounakbanik/the-movies-dataset/downloads/movies_metadata.csv/7.

```
import pandas as pd
import numpy as np

#Load the dataset into a pandas dataframe
df = pd.read_csv('../data/movies_')

#Display the first five movies in the dataframe
df.head()
```

Upon running the cell, you should see a familiar table-like structure output in the notebook.

Building the simple recommender is fairly straightforward. The steps are as follows:

1. Choose a metric (or score) to rate the movies on
2. Decide on the prerequisites for the movie to be featured on the chart
3. Calculate the score for every movie that satisfies the conditions
4. Output the list of movies in decreasing order of their scores

The metric

The metric is the numeric quantity based on which we rank movies. A movie is considered to be better than another movie if it has a higher metric score than the other movie. It is very important that we have a robust and a reliable metric to build our chart upon to ensure a good quality of recommendations.

The prerequisties

The variable **m** in the IMDb weighted rating formula sets a threshold for movie popularity, ensuring that only films with a certain number of votes are considered for the rankings. It is defined as the number of votes of the 80th percentile movie in the dataset, meaning that only movies with more votes than 80% of other films qualify.

The choice of **m** can be adjusted to control the balance between popularity and score quality. A higher value of **m** makes the rankings more selective, focusing on well-known films, while a lower value allows more movies to be considered, including lesser-known titles. Experimenting with different values helps optimize the recommender's performance.

Let us now calculate the value of *m*:

```
#Calculate the number of votes garnered by the 80th percentile movie
m = df['vote_count'].quantile(0.80)
m

OUTPUT:
50.0
```

We can see that only 20% of the movies have gained more than 50 votes. Therefore, our value of *m* is [50].

Another prerequisite that we want in place is the runtime. We will only consider movies that are greater than [45 minutes] and less than [300 minutes] in length. Let us define a new DataFrame, [q_movies], which will hold all the movies that qualify to appear in the chart:

```
#Only consider movies longer than 45 minutes and shorter than 300 minutes
q_movies = df[(df['runtime'] >= 45) & (df['runtime'] <= 300)]

#Only consider movies that have garnered more than m votes
q_movies = q_movies[q_movies['vote_count'] >= m]

#Inspect the number of movies that made the cut
q_movies.shape

OUTPUT:
(8963, 24)
```

We see that from our dataset of 45,000 movies approximately 9,000 movies (or 20%) made the cut.

Calculating the score

The final value that we need to discover before we calculate our scores is *C*, the mean rating for all the movies in the dataset:

```
# Calculate C
C = df['vote_average'].mean()
C

OUTPUT:
5.6182072151341851
```

We can see that the average rating of a movie is approximately 5.6/10. It seems that IMDB happens to be particularly strict with their ratings. Now that we have the value of *C*, we can go about calculating our score for each movie.

First, let us define a function that computes the rating for a movie, given its features and the values of *m* and *C*:

```
# Function to compute the IMDB weighted rating for each movie
def weighted_rating(x, m=m, C=C):
    v = x['vote_count']
    R = x['vote_average']
    # Compute the weighted score
    return (v/(v+m) * R) + (m/(m+v) * C)
```

Next, we will use the familiar [apply] function on our [q_movies] DataFrame to construct a new feature score. Since the calculation is done for every row, we will set the axis to [1] to denote row-wise operation:

```
# Compute the score using the weighted_rating function defined above
q_movies['score'] = q_movies.apply(weighted_rating, axis=1)
```

Sorting and output

There is just one step left. We now need to sort our DataFrame on the basis of the score we just computed and output the list of top movies:

	title	vote_count	vote_average	score	runtime
10309	Dilwale Dulhania Le Jayenge	661.0	9.1	8.855148	190.0
314	The Shawshank Redemption	8358.0	8.5	8.482863	142.0
834	The Godfather	6024.0	8.5	8.476278	175.0
40251	Your Name.	1030.0	8.5	8.366584	106.0
12481	The Dark Knight	12269.0	8.3	8.289115	152.0
2843	Fight Club	9678.0	8.3	8.286216	139.0
292	Pulp Fiction	8670.0	8.3	8.284623	154.0
522	Schindler's List	4436.0	8.3	8.270109	195.0
23673	Whiplash	4376.0	8.3	8.269704	105.0
5481	Spirited Away	3968.0	8.3	8.266628	125.0
2211	Life Is Beautiful	3643.0	8.3	8.263691	116.0
1178	The Godfather: Part II	3418.0	8.3	8.261335	200.0
1152	One Flew Over the Cuckoo's Nest	3001.0	8.3	8.256051	133.0
1176	Psycho	2405.0	8.3	8.245381	109.0
351	Forrest Gump	8147.0	8.2	8.184252	142.0
1184	Once Upon a Time in America	1104.0	8.3	8.183804	229.0
1154	The Empire Strikes Back	5998.0	8.2	8.178656	124.0
18465	The Intouchables	5410.0	8.2	8.176357	112.0
289	Leon: The Professional	4293.0	8.2	8.170276	110.0
3030	The Green Mile	4166.0	8.2	8.169381	189.0
1170	GoodFellas	3211.0	8.2	8.160414	145.0
2216	American History X	3120.0	8.2	8.159278	119.0
1161	12 Angry Men	2130.0	8.2	8.140785	96.0
9698	Howl's Moving Castle	2049.0	8.2	8.138499	119.0
2884	Princess Mononoke	2041.0	8.2	8.138264	134.0

And voila! You have just built your very first recommender. Congratulations!

We can see that the Bollywood film *Dilwale Dulhania Le Jayenge* figures at the top of the list. We can also see that it has a noticeably smaller number of votes than the other Top 25 movies. This strongly suggests that we should probably explore a higher value of *m*. This is left as an exercise for the reader; experiment with different values of *m* and observe how the movies in the chart change.

The knowledge-based recommender

In this section, we are going to go ahead and build a knowledge-based recommender on top of our IMDB Top 250 clone. This will be a simple function that will perform the following tasks:

1. Ask the user for the genres of movies he/she is looking for
2. Ask the user for the duration
3. Ask the user for the timeline of the movies recommended
4. Using the information collected, recommend movies to the user that have a high weighted rating (according to the IMDB formula) and that satisfy the preceding conditions

The data that we have has information on the duration, genres, and timelines, but it isn't currently in a form that is directly usable. In other words, our data needs to be wrangled before it can be put to use to build this recommender.

In our [Lab3] folder, let's create a new Jupyter Notebook named [Knowledge Recommender]. This notebook will contain all the code that we write as part of this section.

As usual, let us load our packages and the data into our notebook. Let's also take a look at the features that we have and decide on the ones that will be useful for this task:

```
import pandas as pd
import numpy as np

df = pd.read_csv('../data/movies_metadata.csv')

#Print all the features (or columns) of the DataFrame
df.columns

OUTPUT:
Index(['adult', 'belongs_to_collection', 'budget', 'genres', 'homepage', 'id',
       'imdb_id', 'original_language', 'original_title', 'overview',
       'popularity', 'poster_path', 'production_companies',
       'production_countries', 'release_date', 'revenue', 'runtime',
       'spoken_languages', 'status', 'tagline', 'title', 'video',
       'vote_average', 'vote_count'],
      dtype='object')
```

From our output, it is quite clear which features we do and do not require. Now, let's reduce our DataFrame to only contain features that we need for our model:

```
#Only keep those features that we require
df = df[['title','genres', 'release_date', 'runtime', 'vote_average', 'vote_count']]

df.head()
```

Next, let us extract the year of release from our [release_date] feature:

```
#Convert release_date into pandas datetime format
df['release_date'] = pd.to_datetime(df['release_date'], errors='coerce')

#Extract year from the datetime
df['year'] = df['release_date'].apply(lambda x: str(x).split('-')[0] if x != np.nan
else np.nan)
```

Our [year] feature is still an [object] and is riddled with [NaT] values, which are a type of null value used by Pandas. Let's convert these values to an integer, [0], and convert the datatype of the [year] feature into [int].

To do this, we will define a helper function, [convert_int], and apply it to the [year] feature:

```
#Helper function to convert NaT to 0 and all other years to integers.
def convert_int(x):
    try:
        return int(x)
    except:
        return 0

#Apply convert_int to the year feature
df['year'] = df['year'].apply(convert_int)
```

We do not require the [release_date] feature anymore. So let's go ahead and remove it:

```
#Drop the release_date column
df = df.drop('release_date', axis=1)

#Display the dataframe
df.head()
```

The [runtime] feature is already in a form that is usable. It doesn't require any additional wrangling. Let us now turn our attention to [genres].

Genres

Upon preliminary inspection, we can observe that the genres are in a format that looks like a JSON object (or a Python dictionary). Let us take a look at the [genres] object of one of our movies:

```
#Print genres of the first movie
df.iloc[0]['genres']

OUTPUT:
"[{'id': 16, 'name': 'Animation'}, {'id': 35, 'name': 'Comedy'}, {'id': 10751, 'name':
'Family'}]"
```

We can observe that the output is a stringified dictionary. In order for this feature to be usable, it is important that we convert this string into a native Python dictionary. Fortunately, Python gives us access to a function called [literal_eval] (available in the [ast] library) which does exactly that. [literal_eval] parses any string passed into it and converts it into its corresponding Python object:

```
#Import the literal_eval function from ast
from ast import literal_eval

#Define a stringified list and output its type
a = "[1,2,3]"
print(type(a))

#Apply literal_eval and output type
b = literal_eval(a)
print(type(b))

OUTPUT:
```

```
<class 'str'>
<class 'list'>
```

We now have all the tools required to convert the *genres* feature into the Python dictionary format.

Also, each dictionary represents a genre and has two keys: [id] and [name]. However, for this exercise (as well as all subsequent exercises), we only require the [name]. Therefore, we shall convert our list of dictionaries into a list of strings, where each string is a genre name:

```
#Convert all NaN into stringified empty lists
df['genres'] = df['genres'].fillna('[]')

#Apply literal_eval to convert to the list object
df['genres'] = df['genres'].apply(literal_eval)

#Convert list of dictionaries to a list of strings
df['genres'] = df['genres'].apply(lambda x: [i['name'] for i in x] if isinstance(x,
list) else [])

df.head()
```

Printing the head of the DataFrame should show you a new [genres] feature, which is a list of genre names. However, we're still not done yet. The last step is to [explode] the genres column. In other words, if a particular movie has multiple genres, we will create multiple copies of the movie, with each movie having one of the genres.

For example, if there is a movie called *Just Go With It* that has *romance* and *comedy* as its genres, we will [explode] this movie into two rows. One row will be *Just Go With It* as a *romance* movie. The other will be a *comedy* movie:

```
#Create a new feature by exploding genres
s = df.apply(lambda x: pd.Series(x['genres']),axis=1).stack().reset_index(level=1,
drop=True)

#Name the new feature as 'genre'
s.name = 'genre'

#Create a new dataframe gen_df which by dropping the old 'genres' feature and adding
the new 'genre'.
gen_df = df.drop('genres', axis=1).join(s)

#Print the head of the new gen_df
gen_df.head()
```

	title	runtime	vote_average	vote_count	year	genre
0	Toy Story	81.0	7.7	5415.0	1995	animation
0	Toy Story	81.0	7.7	5415.0	1995	comedy
0	Toy Story	81.0	7.7	5415.0	1995	family
1	Jumanji	104.0	6.9	2413.0	1995	adventure
1	Jumanji	104.0	6.9	2413.0	1995	fantasy

You should be able to see three *Toy Story* rows now; one each to represent *animation*, *family*, and *comedy*. This [gen_df] DataFrame is what we will use to build our knowledge-based recommender.

The build_chart function

We are finally in a position to write the function that will act as our recommender. We cannot use our computed values of *m* and *C* from earlier, as we will not be considering every movie just the ones that qualify. In other words, these are three main steps:

1. Get user input on their preferences
2. Extract all movies that match the conditions set by the user
3. Calculate the values of *m* and *C* for only these movies and proceed to build the chart as in the previous section

Therefore, the [build_chart] function will accept only two inputs: our [gen_df] DataFrame and the percentile used to calculate the value of *m*. By default, let's set this to 80%, or [0.8]:

```
def build_chart(gen_df, percentile=0.8):
    #Ask for preferred genres
    print("Input preferred genre")
    genre = input()

    #Ask for lower limit of duration
    print("Input shortest duration")
    low_time = int(input())

    #Ask for upper limit of duration
    print("Input longest duration")
    high_time = int(input())

    #Ask for lower limit of timeline
    print("Input earliest year")
    low_year = int(input())

    #Ask for upper limit of timeline
    print("Input latest year")
    high_year = int(input())

    #Define a new movies variable to store the preferred movies. Copy the contents of
gen_df to movies
    movies = gen_df.copy()

    #Filter based on the condition
    movies = movies[(movies['genre'] == genre) &
                    (movies['runtime'] >= low_time) &
                    (movies['runtime'] <= high_time) &
                    (movies['year'] >= low_year) &
                    (movies['year'] <= high_year)]

    #Compute the values of C and m for the filtered movies
    C = movies['vote_average'].mean()
    m = movies['vote_count'].quantile(percentile)
```

```
    #Only consider movies that have higher than m votes. Save this in a new dataframe
q_movies
    q_movies = movies.copy().loc[movies['vote_count'] >= m]

    #Calculate score using the IMDB formula
    q_movies['score'] = q_movies.apply(lambda x: (x['vote_count']/(x['vote_count']+m)
* x['vote_average'])
                                   + (m/(m+x['vote_count']) * C)
                                   ,axis=1)

    #Sort movies in descending order of their scores
    q_movies = q_movies.sort_values('score', ascending=False)

    return q_movies
```

Time to put our model into action!

We want recommendations for animated movies between 30 minutes and 2 hours in length, and released anywhere between 1990 and 2005. Let's see the results:

In [114]:
```
#Generate the chart for top animation movies and display top 5.
build_chart(gen_df).head()
```

```
Input preferred genre
animation
Input shortest duration
30
Input longest duration
120
Input earliest year
1990
Input latest year
2005
```

Out[114]:

	title	runtime	vote_average	vote_count	year	genre	score
9698	Howl's Moving Castle	119.0	8.2	2049.0	2004	animation	7.994823
359	The Lion King	89.0	8.0	5520.0	1994	animation	7.926672
0	Toy Story	81.0	7.7	5415.0	1995	animation	7.637500
6232	Finding Nemo	100.0	7.6	6292.0	2003	animation	7.549423
546	The Nightmare Before Christmas	76.0	7.6	2135.0	1993	animation	7.460500

We can see that the movies that it outputs satisfy all the conditions we passed in as input. Since we applied IMDB's metric, we can also observe that our movies are very highly rated and popular at the same time. The top 5 also includes *The Lion King*, which is my favorite animated movie of all time! I, for one, would be very happy with the results of this list.

Summary

In this lab, we built a simple recommender, which was a clone of the IMDB Top 250 chart. We then proceeded to build an improved knowledge-based recommender, which asked the user for their preferred genres, duration, and time. In the process of building these models, we also learned to perform some advanced data wrangling with the Pandas library.

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

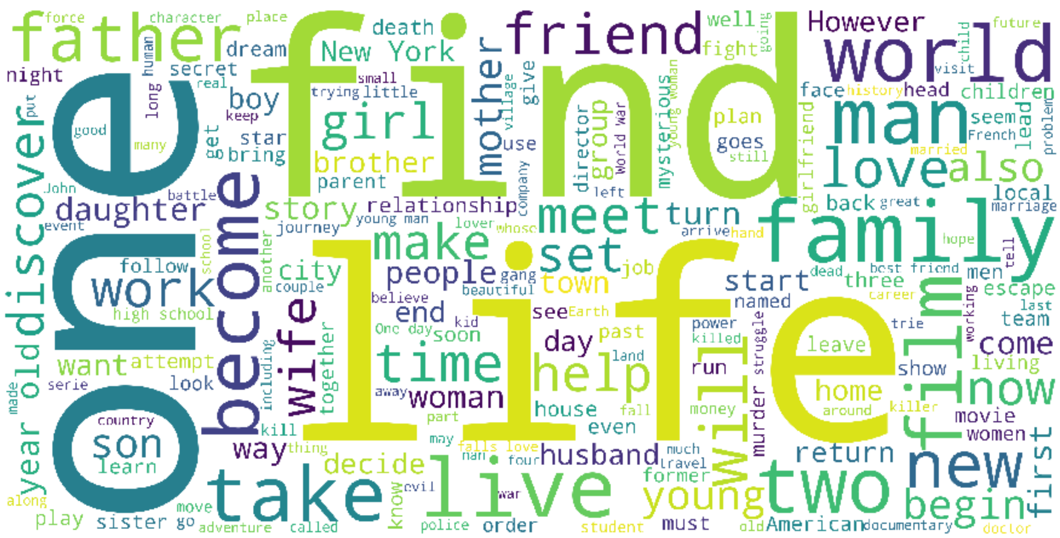
CountVectorizer converts documents into numerical vectors based on word frequency. For example, given three documents, we first create a vocabulary of unique words (ignoring common stop words). After eliminating words like "the" and "is," the vocabulary becomes: *like, little, lamb, love, mary, red,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

Lab 5: Getting Started with Data Mining Techniques

In 2003, Linden, Smith, and York of Amazon.com published a paper entitled Item-to-Item Collaborative Filtering, which explained how product recommendations at Amazon work. Since then, this class of algorithmg has gone on to dominate the industry standard for recommendations. Every website or app with a sizeable user base, be it Netflix, Amazon, or Facebook, makes use of some form of collaborative filters to suggest items (which may be movies, products, or friends):

We will be building powerful collaborative filters in the next lab. However, before we do that, it is important that we have a good grasp of the underlying techniques, principles, and algorithms that go into building collaborative filters.

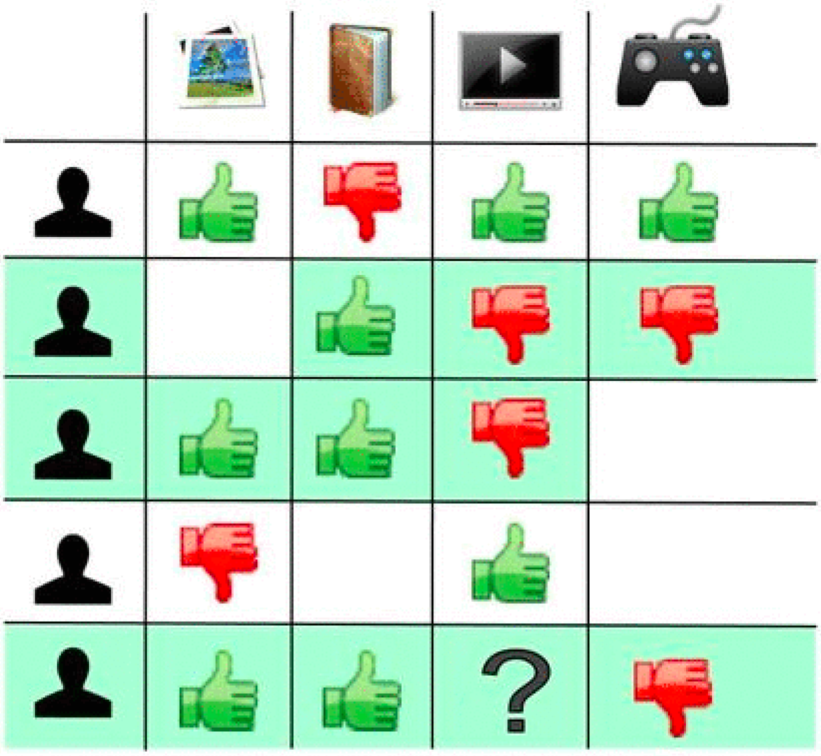
Therefore, in this lab, we will cover the following topics:

- Similarity measures
- Dimensionality reduction
- Supervised learning
- Clustering
- Evaluation methods and metrics

The topics covered in this lab merit an entire textbook. Since this is a hands-on recommendation engine tutorial, we will not be delving too deeply into the functioning of most of the algorithms. Nor will we code them up from scratch. What we will do is gain an understanding of how and when they work, their advantages and disadvantages, and their easy-to-use implementations using the scikit-learn library.

Problem statement

Collaborative filtering algorithms try to solve the prediction problem (as described in the Lab 1, *Getting Started with Recommender Systems*). In other words, we are given a matrix of i users and j items. The value in the ith row and the jth column (denoted by rij) denotes the rating given by user i to item j:



Matrix of i users and j items

Our job is to complete this matrix. In other words, we need to predict all the cells in the matrix that we have no data for. For example, in the preceding diagram, we are asked to predict whether user E will like the music player item. To accomplish this task, some ratings are available (such as User A liking the music player and video games) whereas others are not (for instance, we do not know whether Users C and D like video games).

Euclidean distance

Euclidean scores can take any value between 0 and infinity. The lower the Euclidean score (or distance), the more similar the two vectors are to each other. Let's now define a simple function using NumPy, which allows us to compute the Euclidean distance between two *n*-dimensional vectors using the aforementioned formula:

```
#Function to compute Euclidean Distance.
def euclidean(v1, v2):
```

```
    #Convert 1-D Python lists to numpy vectors
    v1 = np.array(v1)
    v2 = np.array(v2)

    #Compute vector which is the element wise square of the difference
    diff = np.power(np.array(v1)- np.array(v2), 2)

    #Perform summation of the elements of the above vector
    sigma_val = np.sum(diff)

    #Compute square root and return final Euclidean score
    euclid_score = np.sqrt(sigma_val)

    return euclid_score
```

Next, let's define three users who have rated five different movies:

```
#Define 3 users with ratings for 5 movies
u1 = [5,1,2,4,5]
u2 = [1,5,4,2,1]
u3 = [5,2,2,4,4]
```

From the ratings, we can see that users 1 and 2 have extremely different tastes, whereas the tastes of users 1 and 3 are largely similar. Let's see whether the Euclidean distance metric is able to capture this:

```
euclidean(u1, u2)

OUTPUT:
7.4833147735478827
```

The Euclidean distance between users 1 and 2 comes out to be approximately 7.48:

```
euclidean(u1, u3)

OUTPUT:
1.4142135623730951
```

Users 1 and 3 have a much smaller Euclidean score between them than users 1 and 2. Therefore, in this case, the Euclidean distance was able to satisfactorily capture the relationships between our users.

Pearson correlation

Consider two users, Alice and Bob, who have rated the same five movies. Alice is extremely stingy with her ratings and never gives more than a 4 to any movie. On the other hand, Bob is more liberal and never gives anything below a 2 when rating movies. Let's define the matrices representing Alice and Bob and compute their Euclidean distance:

```
alice = [1,1,3,2,4]
bob = [2,2,4,3,5]

euclidean(alice, bob)
```

```
OUTPUT:
2.23606797749979
```

We get a Euclidean distance of about 2.23. However, on closer inspection, we see that Bob always gives a rating that is one higher than Alice. Therefore, we can say that Alice and Bob's ratings are extremely correlated. In other words, if we know Alice's rating for a movie, we can compute Bob's rating for the same movie with high accuracy (in this case, by just adding 1).

Consider another user, Eve, who has the polar opposite tastes to Alice:

```
eve = [5,5,3,4,2]

euclidean(eve, alice)

OUTPUT:
6.324555320336759
```

A high Euclidean distance of 6.32 between Alice and Eve suggests they're very dissimilar. However, their ratings always add up to 6, indicating a strong negative correlation, where one person's rating can predict the other's.

Euclidean distances focus on magnitude, making it hard to assess similarity or dissimilarity accurately. The **Pearson correlation** addresses this by scoring similarity between -1 and 1. A score of -1 indicates strong negative correlation (like Alice and Eve), 1 indicates strong positive correlation (like Alice and Bob), and 0 means no correlation.

The SciPy package gives us access to a function that computes the Pearson Similarity Scores:

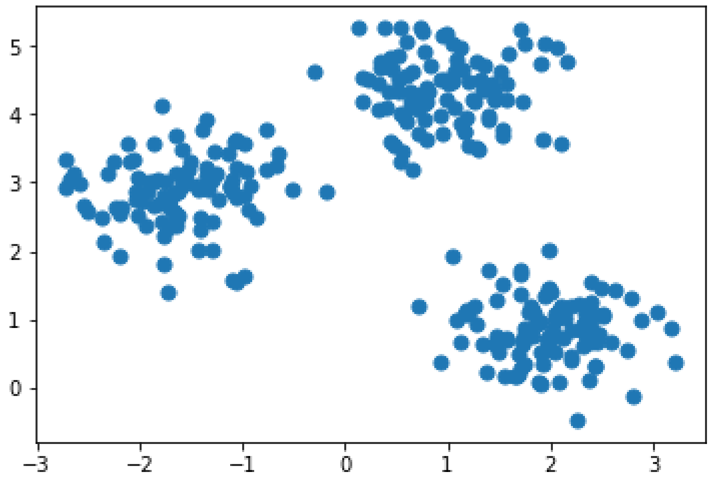
```
from scipy.stats import pearsonr

pearsonr(alice, bob)

OUTPUT:
(1.0, 0.0)
pearsonr(alice, eve)

OUTPUT:
(-1.0, 0.0)
```

The first element of our list output is the Pearson score. We see that Alice and Bob have the highest possible similarity score, whereas Alice and Eve have the lowest possible score.

Can you guess the similarity score for Bob and Eve?

Clustering

Clustering is one of the most popular techniques used in collaborative-filtering algorithms. It is a type of unsupervised learning that groups data points into different classes in such a way that data points belonging to a particular class are more similar to each other than data points belonging to different classes:

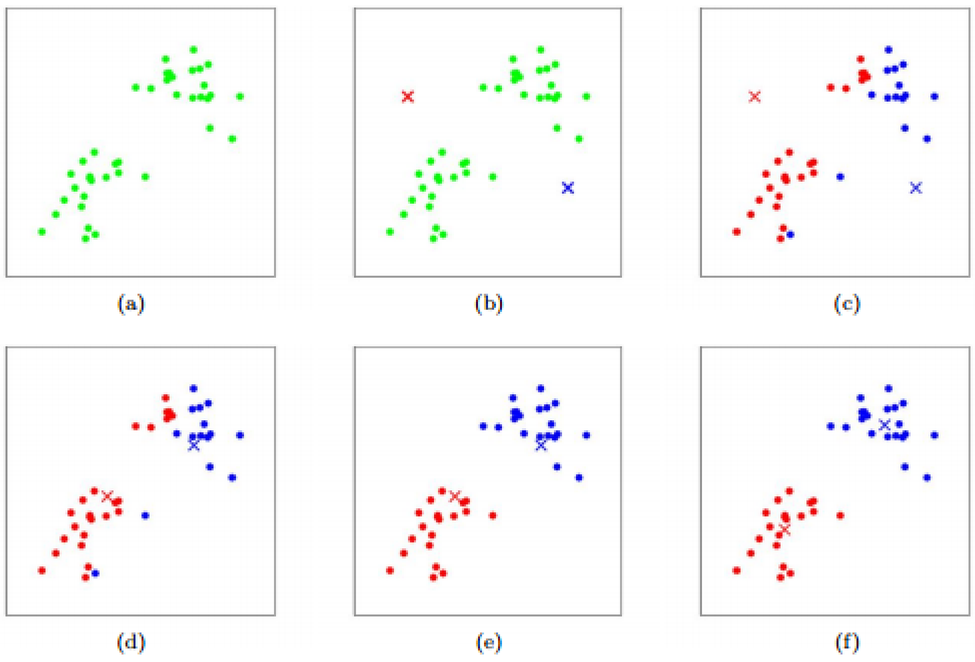


k-means clustering

The k-means algorithm is one of the simplest yet most popular machine learning algorithms. It takes in the data points and the number of clusters (k) as input.

Next, it randomly plots k different points on the plane (called centroids). After the k centroids are randomly plotted, the following two steps are repeatedly performed until there is no further change in the set of k centroids:

- Assignment of points to the centroids: Every data point is assigned to the centroid that is the closest to it. The collection of data points assigned to a particular centroid is called a cluster. Therefore, the assignment of points to k centroids results in the formation of k clusters.
- Reassignment of centroids: In the next step, the centroid of every cluster is recomputed to be the center of the cluster (or the average of all the points in the cluster). All the data points are then reassigned to the new centroids:



(a) (b) (c) (d) (e) (f)

The preceding screenshot shows a visualization of the steps involved in a k-means clustering algorithm, with the number of assigned clusters as two.

We will not be implementing the k-means algorithm from scratch. Instead, we will use its implementation provided by scikit-learn. As a first step, let's access the data points as plotted in the beginning of this section:

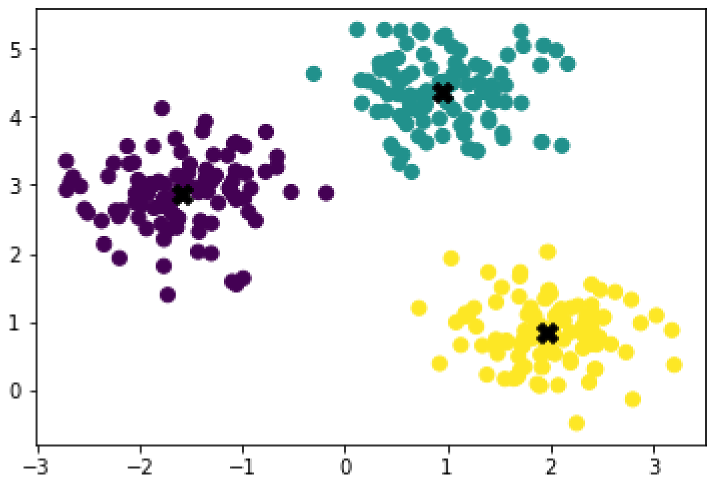
```
#Import the function that enables us to plot clusters
from sklearn.datasets.samples_generator import make_blobs

#Get points such that they form 3 visually separable clusters
X, y = make_blobs(n_samples=300, centers=3,
                       cluster_std=0.50, random_state=0)


#Plot the points on a scatterplot
plt.scatter(X[:, 0], X[:, 1], s=50)
```

One of the most important steps while using the k-means algorithm is determining the number of clusters. In this case, it can be clearly seen from the plot (and the code) that we've plotted the points in such a way that they form three clearly separable clusters. Let's now apply the k-means algorithm via scikit-learn and assess its performance:

```
#Import the K-Means Class
from sklearn.cluster import KMeans

#Initializr the K-Means object. Set number of clusters to 3,
#centroid initialization as 'random' and maximum iterations to 10
kmeans = KMeans(n_clusters=3, init='random', max_iter=10)

#Compute the K-Means clustering
kmeans.fit(X)
```

```
#Predict the classes for every point
y_pred = kmeans.predict(X)

#Plot the data points again but with different colors for different classes
plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=50)

#Get the list of the final centroids
centroids = kmeans.cluster_centers_

#Plot the centroids onto the same scatterplot.
plt.scatter(centroids[:, 0], centroids[:, 1], c='black', s=100, marker='X')
```

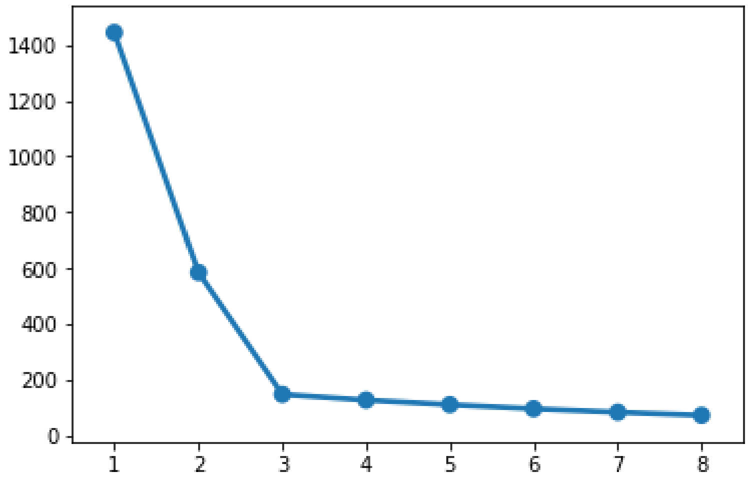
We see that the algorithm proves to be extremely successful in identifying the three clusters. The three final centroids are also marked with an X on the plot:



Choosing k

Scikit-learn's implementation of k-means automatically computes the value of sum-of-squares when it is computing the clusters. Let's now visualize the Elbow plot for our data and determine the best value of K:

```
#List that will hold the sum of square values for different cluster sizes
ss = []

#We will compute SS for cluster sizes between 1 and 8.
for i in range(1,9):

    #Initialize the KMeans object and call the fit method to compute clusters
    kmeans = KMeans(n_clusters=i, random_state=0, max_iter=10, init='random').fit(X)

    #Append the value of SS for a particular iteration into the ss list
    ss.append(kmeans.inertia_)

#Plot the Elbow Plot of SS v/s K
sns.pointplot(x=[j for j in range(1,9)], y=ss)
```



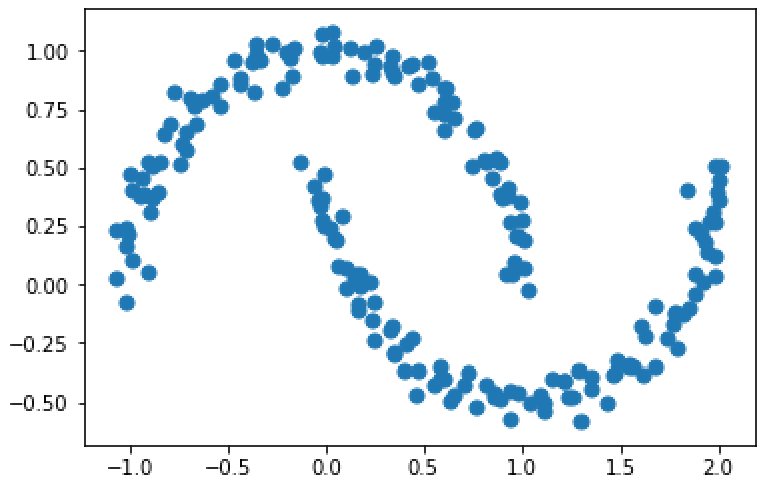
From the plot, it is clear that the Elbow is at K=3. From what we visualized earlier, we know that this is indeed the optimum number of clusters for this data.

Other clustering algorithms

The k-means algorithm, although very powerful, is not ideal for every use case.

To illustrate, let's construct a plot with two half moons. Like the preceding blobs, scikit-learn gives us a convenient function to plot half-moon clusters:

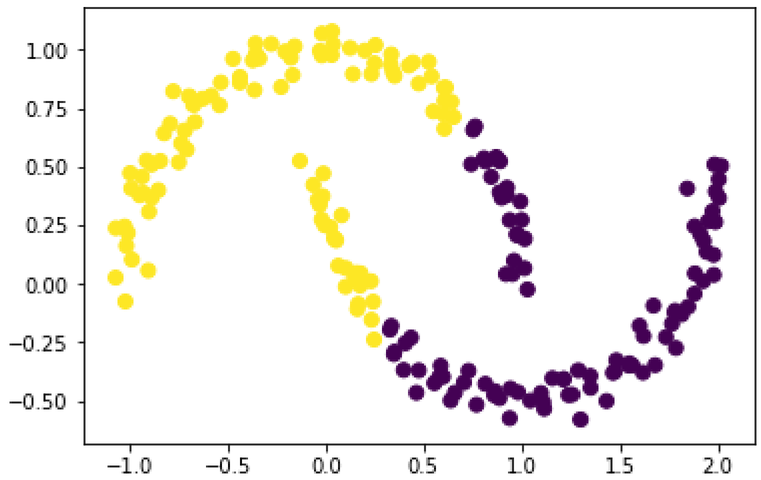
```
#Import the half moon function from scikit-learn
from sklearn.datasets import make_moons

#Get access to points using the make_moons function
X_m, y_m = make_moons(200, noise=.05, random_state=0)

#Plot the two half moon clusters
plt.scatter(X_m[:, 0], X_m[:, 1], s=50)
```



Will the k-means algorithm be able to figure out the two half moons correctly? Let's find out:

```
#Initialize K-Means Object with K=2 (for two half moons) and fit it to our data
kmm = KMeans(n_clusters=2, init='random', max_iter=10)
kmm.fit(X_m)

#Predict the classes for the data points
y_m_pred = kmm.predict(X_m)

#Plot the colored clusters as identified by K-Means
plt.scatter(X_m[:, 0], X_m[:, 1], c=y_m_pred, s=50)
```

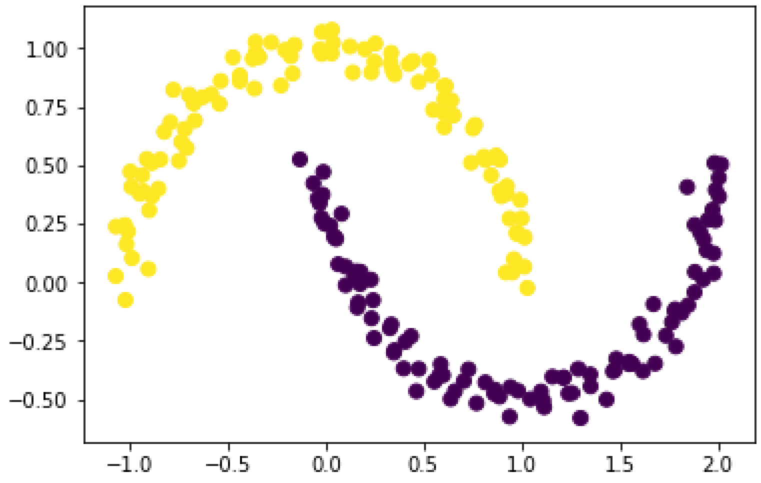
Let's now visualize what k-means thinks the two clusters that exist for this set of data points are:



We see that the k-means algorithm doesn't do a very good job of identifying the correct clusters. For clusters such as these half moons, another algorithm, called spectral clustering, with nearest-neighbor, affinity performs much better.

We will not go into the workings of spectral clustering. Instead, we will use its scikit-learn implementation and assess its performance directly:

```
#Import Spectral Clustering from scikit-learn
from sklearn.cluster import SpectralClustering

#Define the Spectral Clustering Model
model = SpectralClustering(n_clusters=2, affinity='nearest_neighbors')

#Fit and predict the labels
y_m_sc = model.fit_predict(X_m)

#Plot the colored clusters as identified by Spectral Clustering
plt.scatter(X_m[:, 0], X_m[:, 1], c=y_m_sc, s=50)
```



We see that spectral clustering does a very good job of identifying the half-moon clusters.

We have seen that different clustering algorithms are appropriate in different cases. The same applies to cases of collaborative filters. For instance, the surprise package, which we will visit in the next lab, has an implementation of a collaborative filter that makes use of yet another clustering algorithm, called co-clustering. We will wrap up our discussion of clustering and move on to another important data mining technique: dimensionality reduction.

Dimensionality reduction

Most machine learning algorithms tend to perform poorly as the number of dimensions in the data increases. This phenomenon is often known as the curse of dimensionality. Therefore, it is a good idea to reduce the number of features available in the data, while retaining the maximum amount of information possible. There are two ways to achieve this:

- Feature selection
- Feature extraction

In this section, we will take a look at an important feature-extraction method: **Principal component analysis** (or **PCA**).

Principal component analysis

Understanding the PCA algorithm requires linear algebraic concepts that are beyond the scope of this course. Instead, we will use the black box implementation of PCA that [scikit-learn] gives us and consider a use case with the well-known Iris dataset.

The first step is to load the Iris dataset from the UCI machine learning repository into a pandas DataFrame:

```
# Load the Iris dataset into Pandas DataFrame
iris = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data",
                 names=
['sepal_length','sepal_width','petal_length','petal_width','class'])
```

```
#Display the head of the dataframe
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

The PCA algorithm is extremely sensitive to scale. Therefore, we are going to scale all the features in such a way that they have a mean of 0 and a variance of 1:

```
#Import Standard Scaler from scikit-learn
from sklearn.preprocessing import StandardScaler

#Separate the features and the class
X = iris.drop('class', axis=1)
y = iris['class']

# Scale the features of X
X = pd.DataFrame(StandardScaler().fit_transform(X),
                 columns =
['sepal_length','sepal_width','petal_length','petal_width'])

X.head()
```

	sepal_length	sepal_width	petal_length	petal_width
0	-0.900681	1.032057	-1.341272	-1.312977
1	-1.143017	-0.124958	-1.341272	-1.312977
2	-1.385353	0.337848	-1.398138	-1.312977
3	-1.506521	0.106445	-1.284407	-1.312977
4	-1.021849	1.263460	-1.341272	-1.312977

We're now in a good place to apply the PCA algorithm. Let's transform our data into the two-dimensional space:

```
#Import PCA
from sklearn.decomposition import PCA

#Intialize a PCA object to transform into the 2D Space.
pca = PCA(n_components=2)
```

```
#Apply PCA
pca_iris = pca.fit_transform(X)
pca_iris = pd.DataFrame(data = pca_iris, columns = ['PC1', 'PC2'])

pca_iris.head()
```

	PC1	PC2
0	-2.264542	0.505704
1	-2.086426	-0.655405
2	-2.367950	-0.318477
3	-2.304197	-0.575368
4	-2.388777	0.674767

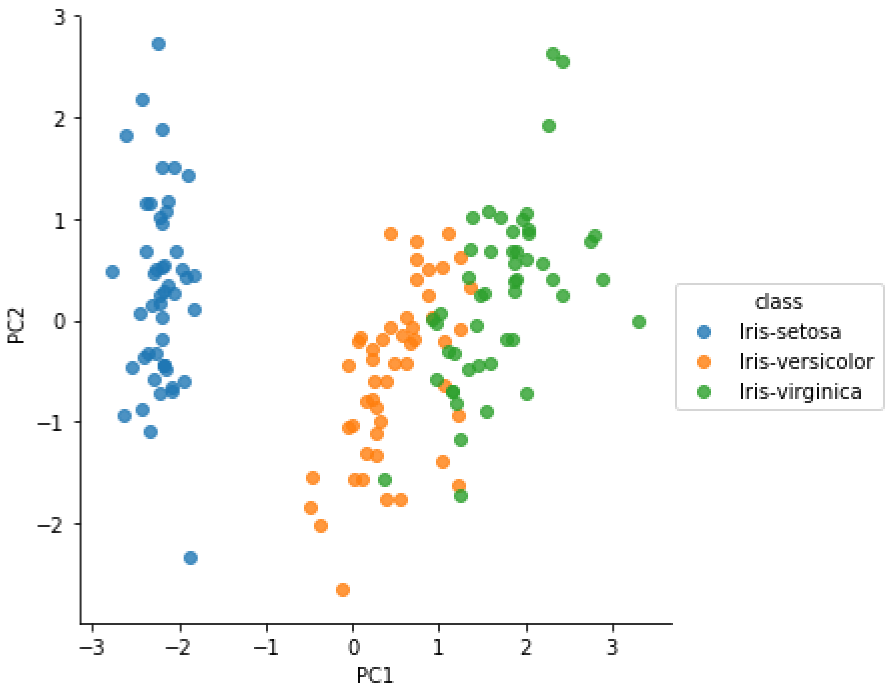
The [scikit-Learn]'s PCA implementation also gives us information about the ratio of variance contained by each principal component:

```
pca.explained_variance_ratio

OUTPUT:
array([ 0.72770452, 0.23030523])
```

We see that the first principal component holds about 72.8% of the information, whereas the second principal component holds about 23.3%. In total, 95.8% of the information is retained, whereas 4.2% of the information is lost in removing two dimensions.

Finally, let's visualize our data points by class in the new 2D plane:

```
#Concatenate the class variable
pca_iris = pd.concat([pca_iris, y], axis = 1)

#Display the scatterplot
sns.lmplot(x='PC1', y='PC2', data=pca_iris, hue='class', fit_reg=False)
```



Other dimensionality reduction techniques

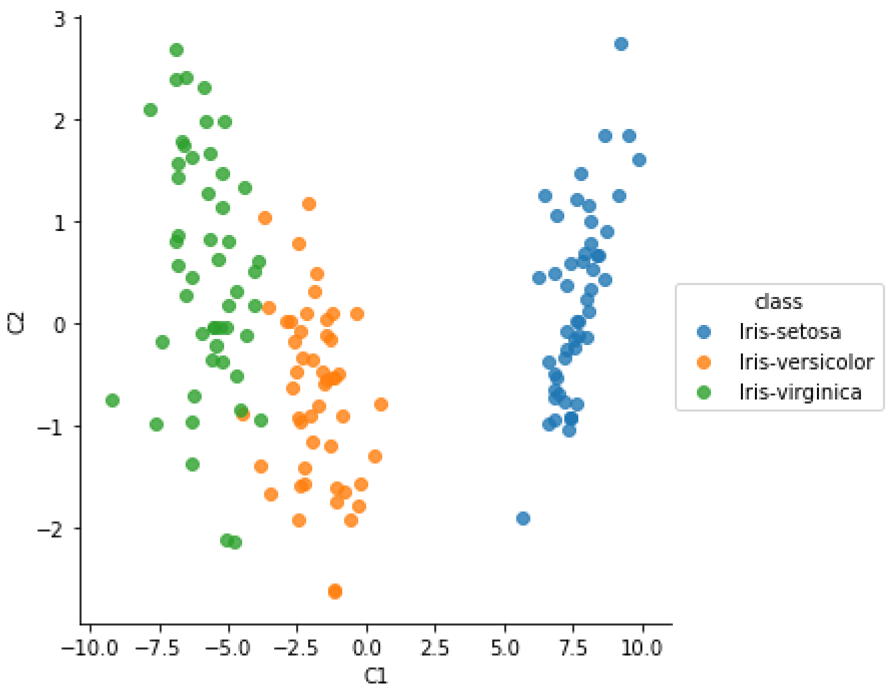
Linear-discriminant analysis

Like PCA, linear-discriminant analysis is a linear transformation method that aims to transform *m*-dimensional data into an *n*-dimensional output space.

However, unlike PCA, which tries to retain the maximum information, LDA aims to identify a set of *n* features that result in the maximum separation (or discrimination) of classes. Since LDA requires labeled data in order to determine its components, it is a type of supervised learning algorithm.

Let's now apply the LDA algorithm to the Iris dataset:

```
#Import LDA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

#Define the LDA Object to have two components
lda = LinearDiscriminantAnalysis(n_components = 2)

#Apply LDA
lda_iris = lda.fit_transform(X, y)
lda_iris = pd.DataFrame(data = lda_iris, columns = ['C1', 'C2'])

#Concatenate the class variable
lda_iris = pd.concat([lda_iris, y], axis = 1)

#Display the scatterplot
sns.lmplot(x='C1', y='C2', data=lda_iris, hue='class', fit_reg=False)
```



We see that the classes are much more separable than in PCA.

Supervised learning

Supervised learning is a class of machine learning algorithm that takes in a series of vectors and their corresponding output (a continuous value or a class) as input, and produces an inferred function that can be used to map new examples.

Boosting

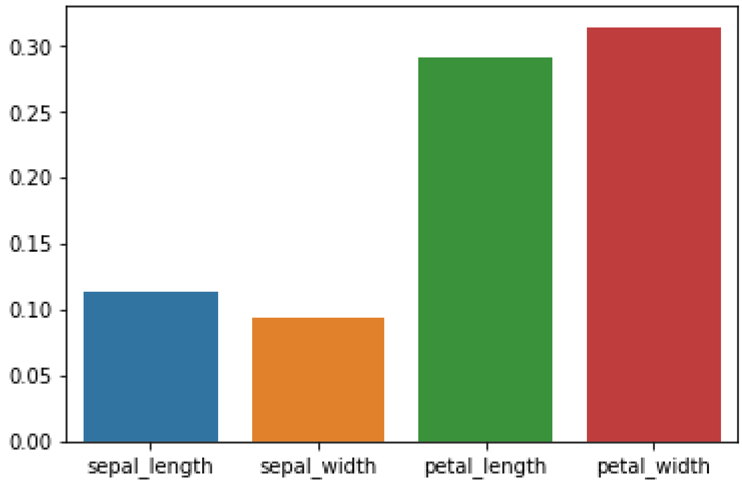
The [scikit-learn] gives us access to implementations of all the algorithms described in this section. The usage of every algorithm is almost the same. As an illustration, let's apply gradient boosting to classify the Iris dataset:

```
#Divide the dataset into the feature dataframe and the target class series.
X, y = iris.drop('class', axis=1), iris['class']

#Split the data into training and test datasets.
#We will train on 75% of the data and assess our performance on 25% of the data

#Import the splitting function
from sklearn.model_selection import train_test_split

#Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=0)

#Import the Gradient Boosting Classifier
from sklearn.ensemble import GradientBoostingClassifier
```

```
#Apply Gradient Boosting to the training data
gbc = GradientBoostingClassifier()
gbc.fit(X_train, y_train)

#Compute the accuracy on the test set
gbc.score(X_test, y_test)

OUTPUT:
0.97368421052631582
```

We see that the classifier achieves a [97.3]% accuracy on the unseen test data. Like random forests, gradient boosting machines are able to gauge the predictive power of each feature. Let's plot the feature importances of the Iris dataset:

```
#Display a bar plot of feature importances
sns.barplot(x= ['sepal_length','sepal_width','petal_length','petal_width'],
y=gbc.feature_importances_)
```



Summary

1. We explored clustering techniques to segment users and dimensionality reduction to improve algorithm performance.
2. Supervised learning algorithms were covered as a foundation for building collaborative filters.
3. The lab provided an overview of evaluation metrics for assessing collaborative filters.
4. For deeper learning, the *Python Machine Learning* course by Sebastian Thrun offers more detailed coverage of these topics.

Lab 6: Building Collaborative Filters

In the previous lab, we mathematically defined the collaborative filtering problem and gained an understanding of various data mining techniques that we assumed would be useful in solving this problem.

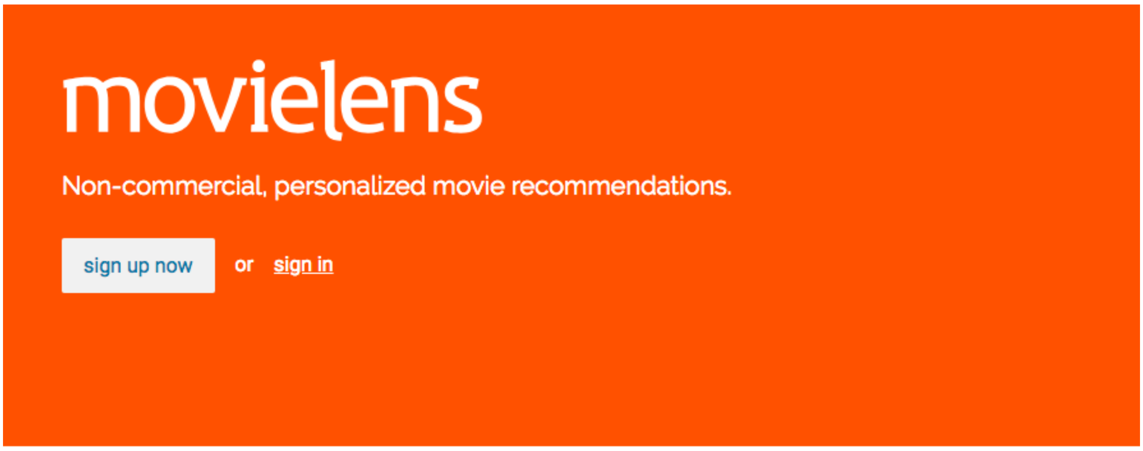
The time has finally come for us to put our skills to the test. In the first section, we will construct a well-defined framework that will allow us to build and test our collaborative filtering models effortlessly. This framework will consist of the data, the evaluation metric, and a corresponding function to compute that metric for a given model.

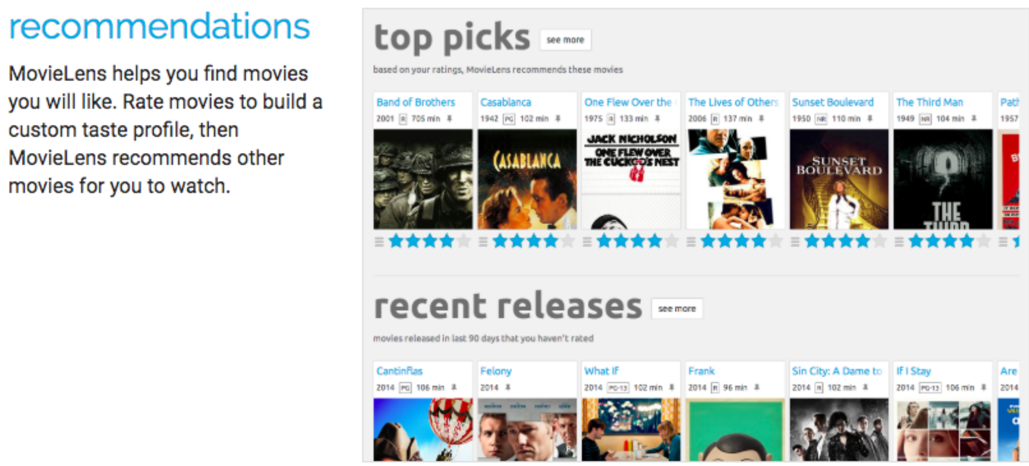
The framework

Just like the knowledge-based and content-based recommenders, we will build our collaborative filtering models in the context of movies. Since collaborative filtering demands data on user behavior, we will be using a different dataset known as MovieLens.

The MovieLens dataset

The MovieLens dataset is made publicly available by GroupLens Research, a computer science lab at the University of Minnesota. It is one of the most popular benchmark datasets used to test the potency of various collaborative filtering models and is usually available in most recommender libraries and packages:





MovieLens gives us user ratings on a variety of movies and is available in various sizes. The full version consists of more than 26,000,000 ratings applied to 45,000 movies by 270,000 users. However, for the sake of fast computation, we will be using the much smaller 100,000 dataset, which contains 100,000 ratings applied by 1,000 users to 1,700 movies.

Viewing the dataset

Without any further ado, let's go ahead and view the 100,000 dataset. The dataset available on the official GroupLens site does not provide us with user demographic information anymore. Therefore, we will use a legacy dataset made available on Kaggle by Prajit Datta.

View the MovieLens 100,000 dataset at https://www.kaggle.com/prajitdatta/movielens-100k-dataset/data.

Unzip the folder and rename it [movielens]. Next, move this folder into the [data] folder. The MovieLens dataset should contain around 23 files. However, the only files we are interested in are [u.data], [u.user], and [u.item]. Let's explore these files in the next section.

Exploring the data

As mentioned in the previous section, we are only interested in three files in the [movielens] folder: [u.data], [u.user], and [u.item]. Although these files are not in CSV format, the code required to load them into a Pandas DataFrame is almost identical.

Let's start with [u.user]:

```
#Load the u.user file into a dataframe
u_cols = ['user_id', 'age', 'sex', 'occupation', 'zip_code']

users = pd.read_csv('../data/movielens/u.user', sep='|', names=u_cols,
 encoding='latin-1')

users.head()
```

Here is its output:

	user_id	age	sex	occupation	zip_code
0	1	24	M	technician	85711
1	2	53	F	other	94043
2	3	23	M	writer	32067
3	4	24	M	technician	43537
4	5	33	F	other	15213

We see that the [u.user] file contains demographic information about our users, such as their [age], [sex], [occupation], and [zip_code].

Next, let's take a look at the [u.item] file, which gives us information about the movies that have been rated by our users:

```
#Load the u.items file into a dataframe
i_cols = ['movie_id', 'title' ,'release date','video release date', 'IMDb URL',
'unknown', 'Action', 'Adventure',
 'Animation', 'Children\'s', 'Comedy', 'Crime', 'Documentary', 'Drama', 'Fantasy',
 'Film-Noir', 'Horror', 'Musical', 'Mystery', 'Romance', 'Sci-Fi', 'Thriller', 'War',
'Western']

movies = pd.read_csv('../data/movielens/u.item', sep='|', names=i_cols,
encoding='latin-1')

movies.head()
```

Here is its output:

	movie id	movie title	release date	video release date	IMDb URL	unknown	Action	Adventure	Animation	Children's	...	Fantasy	Film-Noir	Horror	Musical	Myst
0	1	Toy Story (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?Toy%20Story%2...	0	0	0	1	1	...	0	0	0	0	
1	2	GoldenEye (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?GoldenEye%20(...	0	1	1	0	0	...	0	0	0	0	
2	3	Four Rooms (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?Four%20Rooms%...	0	0	0	0	0	...	0	0	0	0	
3	4	Get Shorty (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?Get%20Shorty%...	0	1	0	0	0	...	0	0	0	0	
4	5	Copycat (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?Copycat%20(1995)	0	0	0	0	0	...	0	0	0	0	

5 rows × 24 columns

We see that this file gives us information regarding the movie's title, [release date], [IMDb URL], and its genre(s). Since we are focused on building only collaborative filters in this lab, we do not require any of this information, apart from the movie title and its corresponding ID:

```
#Remove all information except Movie ID and title
movies = movies[['movie_id', 'title']]
```

Lastly, let's import the [u.data] file into our notebook. This is arguably the most important file as it contains all the ratings that every user has given to a movie. It is from this file that we will construct our ratings matrix:

```
#Load the u.data file into a dataframe
r_cols = ['user_id', 'movie_id', 'rating', 'timestamp']

ratings = pd.read_csv('../data/movielens/u.data', sep='\t', names=r_cols,
 encoding='latin-1')

ratings.head()
```

Here is its output:

	user_id	movie_id	rating	timestamp
0	196	242	3	881250949
1	186	302	3	891717742
2	22	377	1	878887116
3	244	51	2	880606923
4	166	346	1	886397596

We see that every row in our new [ratings] DataFrame denotes a rating given by a user to a particular movie at a particular time. However, for the purposes of the exercises in this lab, we are not really worried about the time at which the ratings were given. Therefore, we will just go ahead and drop it:

```
#Drop the timestamp column
ratings = ratings.drop('timestamp', axis=1)
```

Training and test data

The [ratings] DataFrame contains user ratings for movies that range from 1 to 5. Therefore, we can model this problem as an instance of supervised learning where we need to predict the rating, given a user and a movie. Although the ratings can take on only five discrete values, we will model this as a regression problem.

Consider a case where the true rating given by a user to a movie is 5. A classification model will not distinguish between the predicted ratings of 1 and 4. It will treat both as misclassified. However, a regression model will penalize the former more than the latter, which is the behavior we want.

As we saw in *Lab 5*, one of the first steps towards building a supervised learning model is to construct the test and training sets. The model will learn using the training dataset and its potency will be judged using the testing dataset.

Let's now split our ratings dataset in such a way that 75% of a user's ratings is in the training dataset and 25% is in the testing dataset. We will do this using a slightly hacky way: we will assume that the [user_id] field is the target variable (or [y]) and that our [ratings] DataFrame consists of the predictor variables (or [X]). We will then pass these two variables into scikit-learn's [train_test_split] function and [stratify] it along *y*. This ensures that the proportion of each class is the same in both the training and testing datasets:

```
#Import the train_test_split function
from sklearn.model_selection import train_test_split

#Assign X as the original ratings dataframe and y as the user_id column of ratings.
X = ratings.copy()
y = ratings['user_id']

#Split into training and test datasets, stratified along user_id
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25,
stratify=y, random_state=42)
```

Evaluation

We know from *Lab 5* that the RMSE, or root mean squared error, is the most commonly used performance metric for regressors. We will be using the RMSE to assess our modeling performance too. [scikit-learn] already gives us an implementation of the mean squared error. So, all that we have to do is define a function that returns the square root of the value returned by [mean_squared_error]*:*

```
#Import the mean_squared_error function
from sklearn.metrics import mean_squared_error

#Function that computes the root mean squared error (or RMSE)
def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))
```

Next, let's define our baseline collaborative filter model. All our **collaborative filter** (or **CF**) models will take in a [user_id] and [movie_id] as input and output a floating point number between 1 and 5. We define our baseline model in such a way that it returns [3] regardless of [user_id] or [movie_id]*:*

```
#Define the baseline model to always return 3.
def baseline(user_id, movie_id):
    return 3.0
```

To test the potency of our model, we compute the RMSE obtained by that particular model for all user-movie pairs in the test dataset:

```
#Function to compute the RMSE score obtained on the testing set by a model
def score(cf_model):

    #Construct a list of user-movie tuples from the testing dataset
    id_pairs = zip(X_test['user_id'], X_test['movie_id'])

    #Predict the rating for every user-movie tuple
    y_pred = np.array([cf_model(user, movie) for (user, movie) in id_pairs])

    #Extract the actual ratings given by the users in the test data
    y_true = np.array(X_test['rating'])

    #Return the final RMSE score
    return rmse(y_true, y_pred)
```

We're all set. Let's now compute the RMSE obtained by our baseline model:

```
score(baseline)

OUTPUT:
1.2470926188539486
```

We obtain a score of [1.247]. For the models that we build in the subsequent sections, we will try to obtain an RMSE that is less than that obtained for the baseline.

User-based collaborative filtering

In Lab 1, *Getting Started with Recommender Systems*, we learned what user-based collaborative filters do: they find users similar to a particular user and then recommend products that those users have liked to the first user.

In this section, we will implement this idea in code. We will build filters of increasing complexity and gauge their performance using the framework we constructed in the previous section.

To aid us in this process, let's first build a ratings matrix where each row represents a user and each column represents a movie. Therefore, the value in the i^th^ row and j^th^ column will denote the rating given by user [i] to movie [j]. As usual, pandas gives us a very useful function, called [pivot_table], to construct this matrix from our [ratings] DataFrame:

```
#Build the ratings matrix using pivot_table function
r_matrix = X_train.pivot_table(values='rating', index='user_id', columns='movie_id')

r_matrix.head()
```

Here is its output:

movie_id	1	2	3	4	5	6	7	8	9	10	...	1669	1670	1671	1673	1674	1675	1676	1679	1681	1682
user_id																					
1	5.0	3.0	4.0	3.0	3.0	5.0	4.0	1.0	5.0	3.0	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2.0	...	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	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5	NaN	3.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

5 rows × 1647 columns

We now have a new [r_matrix] DataFrame, where each row is a user and each column is a movie. Also, notice that most values in the DataFrame are unspecified. This gives us a picture of how sparse our matrix is.

Mean

Let's first build one of the simplest collaborative filters possible. This simply takes in [user_id] and [movie_id] and outputs the mean rating for the movie by all the users who have rated it. No distinction is made between the users. In other words, the rating of each user is assigned equal weight.

It is possible that some movies are available only in the test set and not the training set (and consequentially, not in our ratings matrix). In such cases, we will just default to a rating of [3.0], like the baseline model:

```
#User Based Collaborative Filter using Mean Ratings
def cf_user_mean(user_id, movie_id):

    #Check if movie_id exists in r_matrix
    if movie_id in r_matrix:
        #Compute the mean of all the ratings given to the movie
        mean_rating = r_matrix[movie_id].mean()

    else:
        #Default to a rating of 3.0 in the absence of any information
        mean_rating = 3.0

    return mean_rating

#Compute RMSE for the Mean model
score(cf_user_mean)
```

```
OUTPUT:
1.0234701463131335
```

We see that the score obtained for this model is lower and therefore better than the baseline.

Weighted mean

In the previous model, we assigned equal weights to all the users. However, it makes intuitive sense to give more preference to those users whose ratings are similar to the user in question than the other users whose ratings are not.

Therefore, let's alter our previous model by introducing a weight coefficient. This coefficient will be one of the similarity metrics that we computed in the previous lab. Mathematically, it is represented as follows:

$$r_{u,m} = \frac{\sum_{u',u' \neq u} sim(u,u') \cdot r_{u',m}}{\sum_{u',u' \neq u} |sim(u,u')|}$$

In this formula, $r_{u,m}$ represents the rating given by user *u* to movie *m*.

For the sake of this exercise, we will use the cosine score as our similarity function (or sim). Recall how we constructed a movie cosine similarity matrix while building our content-based engine. We will be building a very similar cosine similarity matrix for our users in this section.

However, scikit-learn's [cosine_similarity] function does not work with [NaN] values. Therefore, we will convert all missing values to zero in order to compute our cosine similarity matrix:

```
#Create a dummy ratings matrix with all null values imputed to 0
r_matrix_dummy = r_matrix.copy().fillna(0)

# Import cosine_score
from sklearn.metrics.pairwise import cosine_similarity

#Compute the cosine similarity matrix using the dummy ratings matrix
cosine_sim = cosine_similarity(r_matrix_dummy, r_matrix_dummy)

#Convert into pandas dataframe
cosine_sim = pd.DataFrame(cosine_sim, index=r_matrix.index, columns=r_matrix.index)

cosine_sim.head(10)
```

Here is its output:

user_id	1	2	3	4	5	6	7	8	9	10	...	934	935	936	937
user_id															
1	1.000000	0.099097	0.107680	0.034279	0.152789	0.086705	0.078864	0.068940	0.092399	0.098726	...	0.259636	0.289092	0.318824	0.149105
2	0.099097	1.000000	0.252131	0.026893	0.062539	0.039767	0.089474	0.078162	0.037670	0.031866	...	0.019031	0.065417	0.055373	0.086503
3	0.107680	0.252131	1.000000	0.000000	0.045543	0.078812	0.095354	0.059498	0.053879	0.074209	...	0.050703	0.056561	0.107294	0.098892
4	0.034279	0.026893	0.000000	1.000000	0.202843	0.299619	0.163724	0.038474	0.153021	0.290192	...	0.048524	0.048312	0.022202	0.091910
5	0.152789	0.062539	0.045543	0.202843	1.000000	0.375963	0.131795	0.110944	0.400758	0.181573	...	0.080312	0.162988	0.182856	0.114262
6	0.086705	0.039767	0.078812	0.299619	0.375963	1.000000	0.211282	0.107795	0.328923	0.253871	...	0.074170	0.094619	0.084235	0.115620
7	0.078864	0.089474	0.095354	0.163724	0.131795	0.211282	1.000000	0.037040	0.183375	0.126203	...	0.066843	0.058766	0.068759	0.087159
8	0.068940	0.078162	0.059498	0.038474	0.110944	0.107795	0.037040	1.000000	0.155435	0.032419	...	0.000000	0.101710	0.034568	0.045002
9	0.092399	0.037670	0.053879	0.153021	0.400758	0.328923	0.183375	0.155435	1.000000	0.164532	...	0.049310	0.153506	0.065471	0.060088
10	0.098726	0.031866	0.074209	0.290192	0.181573	0.253871	0.126203	0.032419	0.164532	1.000000	...	0.074822	0.092575	0.098653	0.136230

With the user cosine similarity matrix in hand, we are now in a position to efficiently calculate the weighted mean scores for this model. However, implementing this model in code is a little more nuanced than its simpler mean counterpart. This is because we need to only consider those cosine similarity scores that have a corresponding, non-null rating. In other words, we need to avoid all users that have not rated movie *m:*

```
#User Based Collaborative Filter using Weighted Mean Ratings
def cf_user_wmean(user_id, movie_id):

    #Check if movie_id exists in r_matrix
    if movie_id in r_matrix:

        #Get the similarity scores for the user in question with every other user
        sim_scores = cosine_sim[user_id]

        #Get the user ratings for the movie in question
        m_ratings = r_matrix[movie_id]

        #Extract the indices containing NaN in the m_ratings series
        idx = m_ratings[m_ratings.isnull()].index

        #Drop the NaN values from the m_ratings Series
        m_ratings = m_ratings.dropna()

        #Drop the corresponding cosine scores from the sim_scores series
        sim_scores = sim_scores.drop(idx)

        #Compute the final weighted mean
        wmean_rating = np.dot(sim_scores, m_ratings)/ sim_scores.sum()

    else:
 #Default to a rating of 3.0 in the absence of any information
 wmean_rating = 3.0

 return wmean_rating


score(cf_user_wmean)

OUTPUT:
1.0174483808407588
```

Since we are dealing with positive ratings, the cosine similarity score will always be positive. Therefore, we do not need to explicitly add in a modulus function while computing the normalizing factor (the denominator of the equation that ensures the final rating is scaled back to between 1 and 5).

However, if you're working with a similarity metric that can be negative in this scenario (for instance, the Pearson correlation score), it is important that we factor in the modulus.

Running this code takes significantly more time than the previous model. However, we achieve a (very small) improvement in our RMSE score.

User demographics

Let's now build a gender demographic filter. All this filter does is identify the gender of a user, compute the (weighted) mean rating of a movie by that particular gender, and return that as the predicted value.

Our [ratings] DataFrame does not contain the users' demographics. We will import that information from the [users] DataFrame by merging them into one (using pandas, as usual). Readers familiar with SQL can see that this is extremely similar to the JOIN functionality:

```
#Merge the original users dataframe with the training set
merged_df = pd.merge(X_train, users)

merged_df.head()
```

Here is its output:

	user_id	movie_id	rating	age	sex	occupation	zip_code
0	889	684	2	24	M	technician	78704
1	889	279	2	24	M	technician	78704
2	889	29	3	24	M	technician	78704
3	889	190	3	24	M	technician	78704
4	889	232	3	24	M	technician	78704

Next, we need to compute the [mean] rating of each movie by gender. Pandas makes this possible with the [groupby] method:

```
#Compute the mean rating of every movie by gender
gender_mean = merged_df[['movie_id', 'sex', 'rating']].groupby(['movie_id', 'sex'])
['rating'].mean()
```

We are now in a position to define a function that identifies the gender of the user, extracts the average rating given to the movie in question by that particular gender, and return that value as output:

```
#Set the index of the users dataframe to the user_id
users = users.set_index('user_id')

#Gender Based Collaborative Filter using Mean Ratings
def cf_gender(user_id, movie_id):

    #Check if movie_id exists in r_matrix (or training set)
    if movie_id in r_matrix:
        #Identify the gender of the user
        gender = users.loc[user_id]['sex']

        #Check if the gender has rated the movie
        if gender in gender_mean[movie_id]:

            #Compute the mean rating given by that gender to the movie
            gender_rating = gender_mean[movie_id][gender]
```

```
        else:
            gender_rating = 3.0

    else:
        #Default to a rating of 3.0 in the absence of any information
        gender_rating = 3.0

    return gender_rating

score(cf_gender)

OUTPUT:
1.0330308800874282
```

We see that this model actually performs worse than the standard mean ratings collaborative filter. This indicates that a user's gender isn't the strongest indicator of their taste in movies.

Let's try building one more demographic filter, but this time using both gender and occupation:

```
#Compute the mean rating by gender and occupation
gen_occ_mean = merged_df[['sex', 'rating', 'movie_id', 'occupation']].pivot_table(
    values='rating', index='movie_id', columns=['occupation', 'sex'], aggfunc='mean')

gen_occ_mean.head()
```

We see that the [pivot_table] method gives us the required DataFrame. However, this could have been done using [groupby] too. [pivot_table] is simply a more compact, easier-to-use interface for the [groupby] method:

```
#Gender and Occupation Based Collaborative Filter using Mean Ratings
def cf_gen_occ(user_id, movie_id):

    #Check if movie_id exists in gen_occ_mean
    if movie_id in gen_occ_mean.index:

        #Identify the user
        user = users.loc[user_id]

        #Identify the gender and occupation
        gender = user['sex']
        occ = user['occupation']

        #Check if the occupation has rated the movie
        if occ in gen_occ_mean.loc[movie_id]:

            #Check if the gender has rated the movie
            if gender in gen_occ_mean.loc[movie_id][occ]:

                #Extract the required rating
                rating = gen_occ_mean.loc[movie_id][occ][gender]

                #Default to 3.0 if the rating is null
                if np.isnan(rating):
                    rating = 3.0
```

```
                return rating

    #Return the default rating
    return 3.0

score(cf_gen_occ)

OUTPUT:
1.1391976012043645
```

We see that this model performs the worst out of all the filters we've built so far, beating only the baseline. This strongly suggests that tinkering with user demographic data may not be the best way to go forward with the data that we are currently using. However, you are encouraged to try different permutations and combinations of user demographics to see what performs best. You are also encouraged to try other techniques of improving the model, such as using a weighted mean for the [aggfunc] of the [pivot_table] and experimenting with different (perhaps more informed) default ratings.

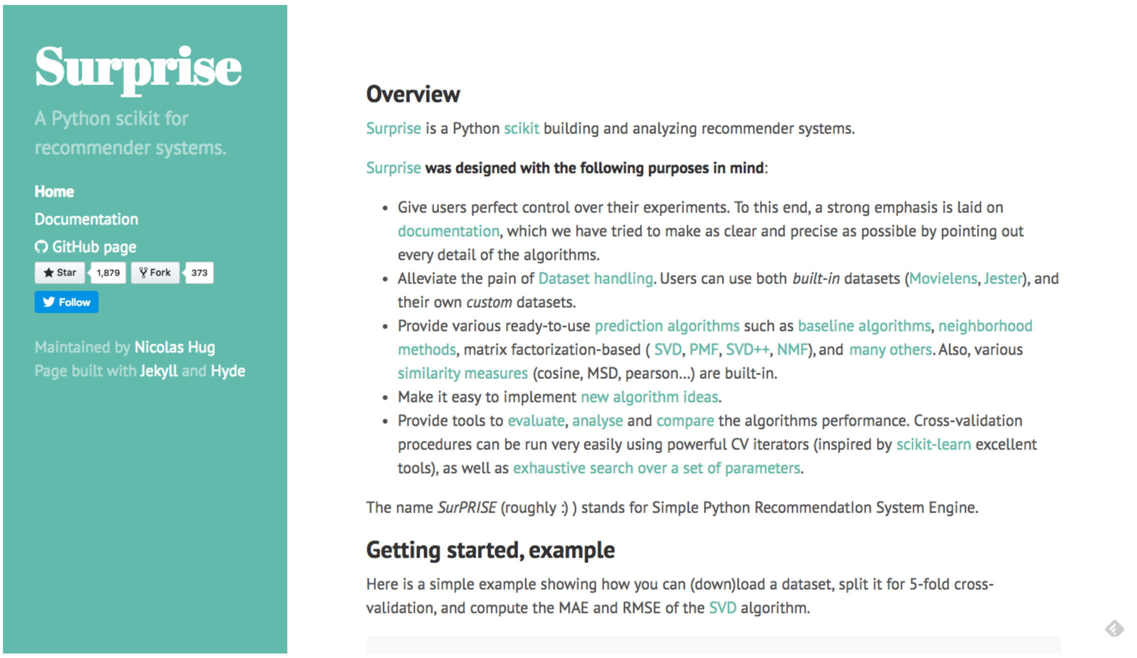
Clustering

In this section, we will use k-means' sister algorithm, kNN, to build our clustering-based collaborative filter. In a nutshell, given an user, *u*, and a movie, *m*, these are the steps involved:

1. Find the k-nearest neighbors of *u* who have rated movie *m*
2. Output the average rating of the *k* users for the movie *m*

That's it. This extremely simply algorithm happens to be one of the most popularly used.

Just like kNN, we will not be implementing the kNN-based collaborative filter from scratch. Instead, we will use an extremely popular and robust library called [surprise]:



Let's now build and evaluate our kNN-based collaborative filter. Although *surprise* has the MovieLens datasets available within the library, we will still use the external data we have in order to get a feel for using the library with alien datasets:

```
#Import the required classes and methods from the surprise library
from surprise import Reader, Dataset, KNNBasic, evaluate

#Define a Reader object
#The Reader object helps in parsing the file or dataframe containing ratings
reader = Reader()

#Create the dataset to be used for building the filter
data = Dataset.load_from_df(ratings, reader)

#Define the algorithm object; in this case kNN
knn = KNNBasic()

#Evaluate the performance in terms of RMSE
evaluate(knn, data, measures=['RMSE'])
```

Here is its output:

```
Evaluating RMSE of algorithm KNNBasic.

------------
Fold 1
Computing the msd similarity matrix...
Done computing similarity matrix.
RMSE: 0.9776
------------
Fold 2
Computing the msd similarity matrix...
Done computing similarity matrix.
RMSE: 0.9789
------------
Fold 3
Computing the msd similarity matrix...
Done computing similarity matrix.
RMSE: 0.9695
------------
Fold 4
Computing the msd similarity matrix...
Done computing similarity matrix.
RMSE: 0.9810
------------
Fold 5
Computing the msd similarity matrix...
Done computing similarity matrix.
RMSE: 0.9849
------------
------------
Mean RMSE: 0.9784
------------
------------
```

The output indicates that the filter is making use of a technique known as fivefold [cross-validation]. In a nutshell, this means that [surprise] divides the data into five equal parts. It then uses four parts as the training data and tests it on the fifth part. This is done five times, in such a way that every part plays the role of the test data once.

We see that the RMSE obtained by this model is [0.9784]. This is, by far, the best result we have achieved.

Let's now take a tour of some other model-based approaches to collaborative filtering and implement a few of them using the *surprise* library.

Let's now turn our attention to perhaps the most famous recommendation algorithm of all time: singular-value decomposition.

Singular-value decomposition

Let's now implement the SVD filter using the [surprise] package:

```
#Import SVD
from surprise import SVD
```

```
#Define the SVD algorithm object
svd = SVD()

#Evaluate the performance in terms of RMSE
evaluate(svd, data, measures=['RMSE'])
```

Here is its output:

```
Evaluating RMSE of algorithm SVD.

------------
Fold 1
RMSE: 0.9371
------------
Fold 2
RMSE: 0.9417
------------
Fold 3
RMSE: 0.9289
------------
Fold 4
RMSE: 0.9379
------------
Fold 5
RMSE: 0.9379
------------
------------
Mean RMSE: 0.9367
------------
------------
```

The SVD filter outperforms all other filters, with an RMSE score of [0.9367].

Summary

1. We built and explored various user-based and item-based collaborative filters.
2. We introduced model-based approaches using machine learning, including clustering with kNN and supervised learning algorithms for rating predictions.
3. We gained an understanding of singular-value decomposition (SVD) and implemented it using the *surprise* library.
4. In the next lab, we'll learn how to deploy our models to the web for public use.

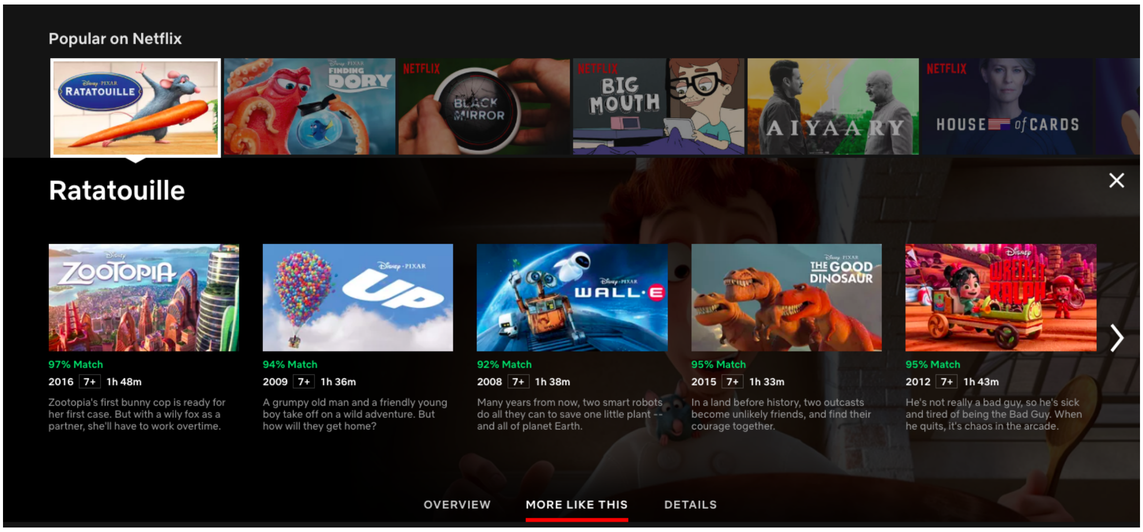
Lab 7: Hybrid Recommenders

In this final lab, we will discuss recommender systems in the context of practicality and industrial use. Until now, we have learned about various types of recommender, including knowledge, content, and collaborative filtering-based engines. However, when used in practice, each recommender usually suffers from one shortcoming or another.

Introduction

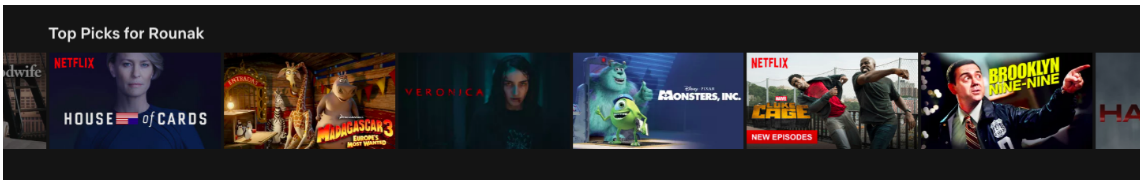
As already mentioned a couple of times, hybrid recommenders are extremely powerful, robust systems that combine various simpler models to give us predictions. There is no single way in which a hybrid model could do this; some hybrids predict using content and collaborative filtering techniques separately to produce results. Some others introduce content-based techniques into collaborative filters and vice versa.

Netflix is a very good example of a hybrid recommender. Netflix employs content-based techniques when it shows you similar movies to a movie you're watching (the [MORE LIKE THIS] section), as shown in the following screenshot:



Here, we can see that while watching *Ratatouille,* Netflix recommends movies to me that are very similar to *Ratatouille*. All the top five recommended movies are all animated and produced by Disney Pixar.

However, animated movies are not the only genre I watch on Netflix. I also like watching drama and comedy. Netflix has a separate row of recommendations for me entitled [Top Picks for Rounak], where it uses collaborative filtering to identify users similar to me and recommend movies that they have liked, but that I haven't watched:



In this way, Netflix employs both content- and collaborative-based techniques separately to produce results that are extremely satisfactory.

Case study -- Building a hybrid model

In this section, let's build a content-based model that incorporates some collaborative filtering techniques into it.

Imagine that you have built a website like Netflix. Every time a user watches a movie, you want to display a list of recommendations in the side pane (like YouTube). At first glance, a content-based recommender seems appropriate for this task. This is because, if the person is currently watching something they find interesting, they will be more inclined to watch something similar to it.

Let's say our user is watching *The Dark Knight.* Since this is a Batman movie, our content-based recommender is likely to recommend other Batman (or superhero) movies regardless of quality. This may not always lead to the best recommendations. For instance, most people who like *The Dark Knight* do not rate *Batman and Robin* very highly, although they feature the same lead character. Therefore, we will introduce a collaborative filter here that predicts the ratings of the movies recommended by our content-based model and return the top few movies with the highest predictions.

In other words, the workflow of our hybrid model will be as follows:

1. Take in a movie title and user as input
2. Use a content-based model to compute the 25 most similar movies
3. Compute the predicted ratings that the user might give these 25 movies using a collaborative filter
4. Return the top 10 movies with the highest predicted rating

We will be using different datasets for this task. Go ahead and download the datasets from the following links.

NOTE: All datasets used in the lab are available in lab environment and GitHub repository:

`https://github.com/fenago/recommendation-systems-python/tree/main/data`

With these files in hand, let's proceed to build our model. The first step is to compute the [cosine_sim] matrix for our movies. In addition, we also need to map every movie to the indices in the [cosine_sim] matrix. We've already learned how to do this in Lab 3, *Building an IMDB Top 250 Clone with Pandas*. Computing this matrix and the mapping, therefore, is left as an exercise for the reader.

Next, let's build a collaborative filtering model. We will use the SVD model from the last lab for this purpose:

```
#Build the SVD based Collaborative filter
from surprise import SVD, Reader, Dataset

reader = Reader()
ratings = pd.read_csv('../data/ratings_small.csv')
data = Dataset.load_from_df(ratings[['userId', 'movieId', 'rating']], reader)
data.split(n_folds=5)
svd = SVD()
trainset = data.build_full_trainset()
svd.train(trainset)
```

Next, let's load the [movie_ids.cs][v] file into a DataFrame and construct two mappings: one that returns the movie title for a given movie ID, and the other vice versa:

```
#Build title to ID and ID to title mappings
id_map = pd.read_csv('../data/movie_ids.csv')
id_to_title = id_map.set_index('id')
title_to_id = id_map.set_index('title')
```

Now, let's import the metadata for our movies so that our recommender can display useful information, such as the IMDB rating and the year of release. This information can be extracted from the main [movies_metadata.csv]file, and is again left as an exercise for the reader.

We're finally in a position to build the hybrid recommender function according to the workflow described previously:

```
def hybrid(userId, title):
    #Extract the cosine_sim index of the movie
    idx = cosine_sim_map[cosine_sim_map == title].index[0]

    #Extract the TMDB ID of the movie
    tmdbId = title_to_id.loc[title]['id']

    #Extract the movie ID internally assigned by the dataset
    movie_id = title_to_id.loc[title]['movieId']

    #Extract the similarity scores and their corresponding index for every movie from
the cosine_sim matrix
    sim_scores = list(enumerate(cosine_sim[str(int(idx))]))

    #Sort the (index, score) tuples in decreasing order of similarity scores
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)

    #Select the top 25 tuples, excluding the first
    #(as it is the similarity score of the movie with itself)
    sim_scores = sim_scores[1:26]

    #Store the cosine_sim indices of the top 25 movies in a list
    movie_indices = [i[0] for i in sim_scores]

    #Extract the metadata of the aforementioned movies
    movies = smd.iloc[movie_indices][['title', 'vote_count', 'vote_average', 'year',
'id']]

    #Compute the predicted ratings using the SVD filter
    movies['est'] = movies['id'].apply(lambda x: svd.predict(userId,
id_to_title.loc[x]['movieId']).est)

    #Sort the movies in decreasing order of predicted rating
    movies = movies.sort_values('est', ascending=False)

    #Return the top 10 movies as recommendations
    return movies.head(10)
```

Let's put our hybrid model to the test. Let's imagine that users with the IDs 1 and 2 are both watching the movie *Avatar:*

```
hybrid(1, 'Avatar')
```

	title	vote_count	vote_average	year	id	est
1011	The Terminator	4208.0	7.4	1984	218	3.140748
974	Aliens	3282.0	7.7	1986	679	3.126947
8401	Star Trek Into Darkness	4479.0	7.4	2013	54138	3.079551
7705	Alice in Wonderland	8.0	5.4	1933	25694	3.054995
3060	Sinbad and the Eye of the Tiger	39.0	6.3	1977	11940	3.028386
8658	X-Men: Days of Future Past	6155.0	7.5	2014	127585	2.997411
2014	Fantastic Planet	140.0	7.6	1973	16306	2.957614
522	Terminator 2: Judgment Day	4274.0	7.7	1991	280	2.914548
1621	Darby O'Gill and the Little People	35.0	6.7	1959	18887	2.844940
1668	Return from Witch Mountain	38.0	5.6	1978	14822	2.804012

```
hybrid(2, 'Avatar')
```

	title	vote_count	vote_average	year	id	est
522	Terminator 2: Judgment Day	4274.0	7.7	1991	280	3.943639
2834	Predator	2129.0	7.3	1987	106	3.866272
8401	Star Trek Into Darkness	4479.0	7.4	2013	54138	3.858491
1011	The Terminator	4208.0	7.4	1984	218	3.856029
7705	Alice in Wonderland	8.0	5.4	1933	25694	3.701565
922	The Abyss	822.0	7.1	1989	2756	3.676465
974	Aliens	3282.0	7.7	1986	679	3.672303
1621	Darby O'Gill and the Little People	35.0	6.7	1959	18887	3.628234
1668	Return from Witch Mountain	38.0	5.6	1978	14822	3.614118
2014	Fantastic Planet	140.0	7.6	1973	16306	3.602051

We can see that although both users are currently watching *Avatar,* the recommendations differ in the content as well as the order. This is influenced by the collaborative filter. However, all the movies listed are similar to *Avatar.* This is because of the content-based filtering carried out by the model.

Summary

With this, we come to the end of this lab, as well as the main part of the course. In this course, we learned the following:

1. We introduced recommender systems, defined the problem, and discussed different types of recommendation engines.
2. We learned data wrangling with pandas, focusing on Series and DataFrames.
3. We built an IMDB Top 250 clone and enhanced it into a knowledge-based recommender using genre, duration, and release year.
4. We explored content-based recommenders using plot lines and metadata, applying vectorizers and cosine similarity.
5. We covered data mining techniques, including clustering (k-means), PCA, supervised learning, and evaluation metrics.
6. We experimented with collaborative filtering models and used the *surprise* library to simplify implementation.
7. We built a hybrid recommender combining collaborative filtering and content-based approaches for personalized recommendations.