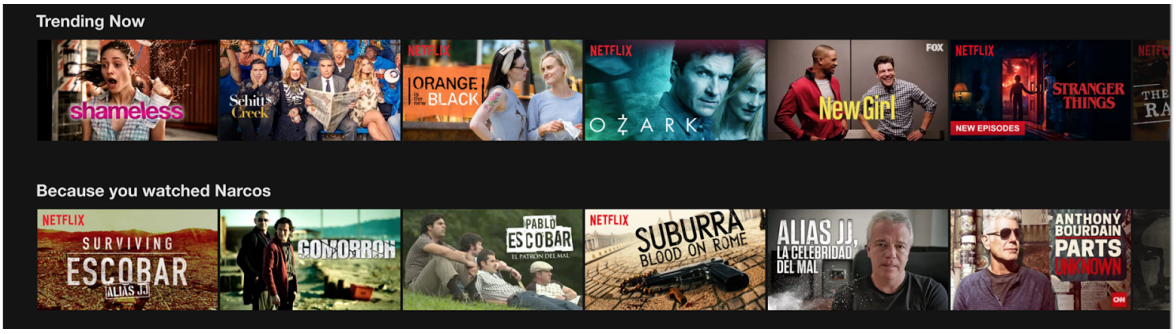

A simple movie recommender

Have you wondered how large companies gives you recommendations on the web? How Youtube always seems to give you the perfect recommendations based on your prior history? This post will give you a better insight of how this technique works and guide you into building a recommendation system of your own.

Background

Nowadays, there are almost recommendation systems for anything on the web. It can be anything from Netflix giving you recommendation on what series to watch next or Amazon giving you recommendations on what items you should purchase. But how can these companies know these things? It all boils down to the power of Machine Learning.

Machine Learning is getting more and more popular in today's society and is a huge help for companies to get relevant data from relevant user by analysing their history. Machine learning can be categorized into many different fields, but for a recommendation system there are two techniques that are most relevant; **content based** and **collaborative filtering**.

Content Based and Collaborative Filtering

The two types of data filtering algorithms usually used in recommendation systems are content based and collaborative filtering. They are similar in what they are trying to achieve, but works in different ways.

Collaborative filtering works by looking at how other user have interacted or rated with a specific item. For example, if User A and User B both likes movies from Director 1, and User A finds out that Director 2 creates great movies as well. Then it's very likely that User B also will like Director 2, and so Director 2 gets recommended to User B.

Content Based filtering on the other hand, works in a manner that find similarities in the items instead. For example, if a user likes an action movie, it is more likely that the user would like to get recommendations of other action movies than of a drama movie.

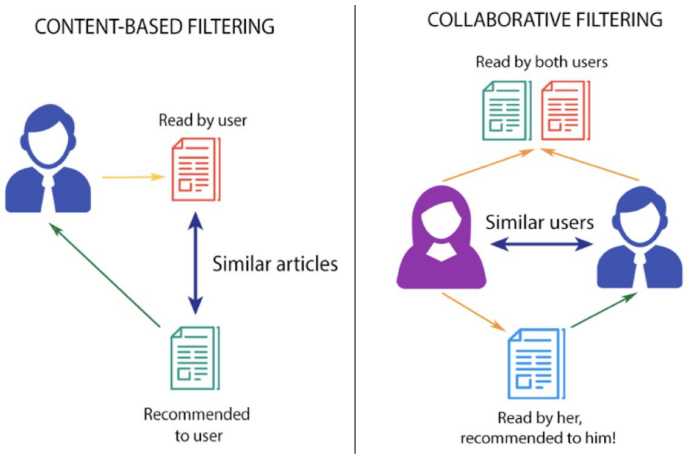

Figure 1, *A flow chart over content and collaborative filtering*

Implementing a recommendation system using Python and scikit-learn

The type of recommendation system this blogpost will cover is a content-based filtering system. So how do you find similarities between two movies and how can you do this in an efficient way? The simple answer is that you compare genres, directors and other feature of the movies, and the simplest way of doing this is by using *cosine similarity*.

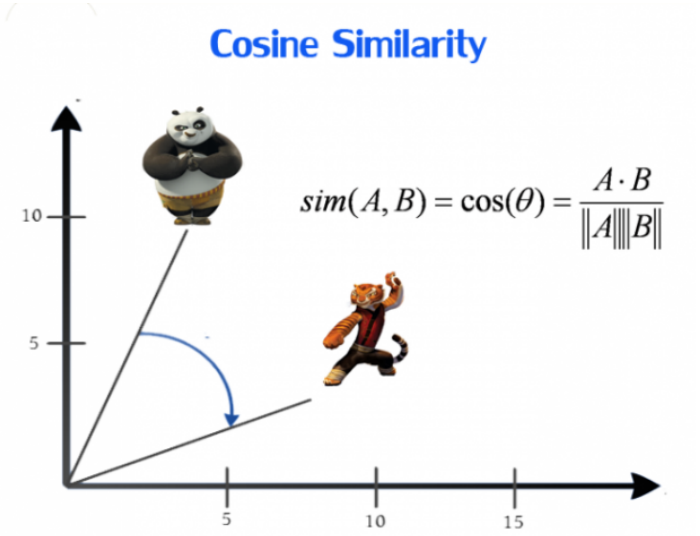

Figure 2: *Difference between a Kung Fu Panda and a Kung Fu Tiger*

Cosine similarity is a measure of similarity between two non-zero vectors in a inner-product space. To find the similarity between these two vectors you calculate the angle between them and get a measure of how similar they are. The reason why this method is often used, is because you get a *score*(angle) from each comparison that can later be used to see how similar each movie is compared to the other movies in the data set.

To be able to use cosine similarity you first need to prepare your dataset of movies. The way we will prepare the data, is to combine all the movie features that we think are of interest, in one long string. You might ask yourself why we are doing this and the reason is that we will compare this string with all the other strings that represent a movie using cosine similarity.

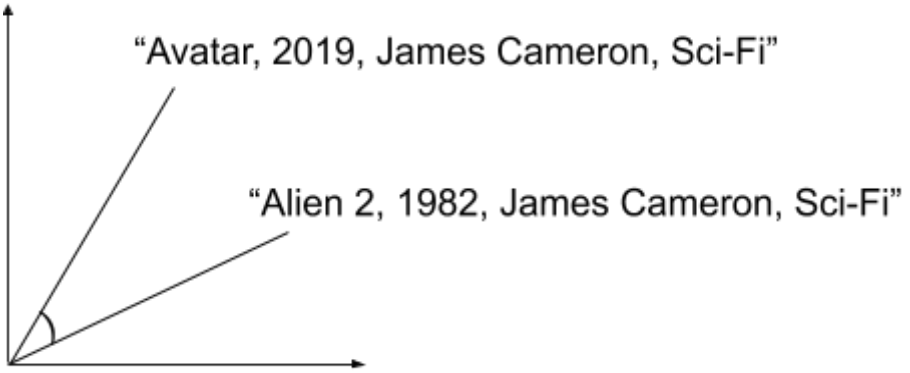

Figure 3, Cosine Similarity example

```
Note ____________________________________________________________
Figure 3 is showing a 2D graph of the angle between the two movies. However,
in reality this ought to be a multidimensional graph. The reason for this, is
because all of the features should be represented with a dimension of its own.
```

After all the features have been added together, the words in each string is being counted, and placed in a matrix. This is done by the help of the package from the python library *Sklearn Feature Extraction* called “*Count Vectorizer*”. This means when the above movies are being run through the algorithms, the *count_matrix* would look something like:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity

movies = ["Avatar James Cameron 2019 Sci-fi", "Alien 2 James Cameron 1968
Sci-fi"]
cv = CountVectorizer()
count_matrix = cv.fit_transform(movies)
print(count_matrix)

Console ____________________________________________________________
count_matrix = [[0 1 0 1 1 1 1 1]
                [1 0 1 0 1 1 1 1]]
```

Where the console output can be interpreted like this:

Table 1, The occurences of each feature of both movies.

words:	*Alien 2*	*Avatar*	*1968*	*2019*	*James*	*Cameron*	*Sci-Fi*	*Sci-Fi*
Avatar	0	1	0	1	1	1	1	1
Alien 2	1	0	1	0	1	1	1	1

As we can see, both movies are quite similar. They both have James Cameron as the director and Sci-Fi as the genre. Now with the help of the *count_matrix* we will be able to calculate the cosine similarity with the function from the *Sklearn Metrics Pairwise* library. As we can see in *count_matrix,* there are *6* unique words, and *2* of them can be found in both movies.

So how similar is Alien 2 to Avatar? With the help of some linear algebra (using *dot product*), it is possible to calculate the cosine of the angle, i.e the cosine similarity.

$$sim(Alien2, Avatar) = cos(\theta) = \frac{\vec{x}\vec{y}}{\|\vec{x}\| \|\vec{y}\|}$$

Where both vector **x** and **y** are the features of both movies, the same as the rows in Table 1. Calculating the dot product for both the movies, we get the cosine similarity for the movies towards each other, as shown below:

```
cosine_sim_matrix = cosine_similarity(count_matrix)
print(cosine_sim_matrix)

Console ____________________________________________________________________
cosine_sim_matrix = [[1.         0.66666667]
                     [0.66666667 1.        ]]
```

where the diagonal is how similar each movie are to themselves and the other slots are how similar movie each are towards each other, i.e the movies are similar with a score of **~0.67**.

That's it! With those simple 3 steps it is possible to get quite accurate results. Now we just need data to test this method on!

Dataset

The dataset that have been used in this similarity system is "The Movies Dataset" from *Kaggle* this dataset contains 45 000 different movies. This dataset contains features such as: *director, cast, genres, keywords, posters, budget, revenue, release dates, languages* and *production companies*. The dataset is divided up between different csv files and many of the movies included are not of interest in this system. Therefore the data needs to be cleaned before it can be used in the recommendation process. To be able to clean the data a python library called Pandas was used. *Pandas* provides easy to use data structures that are great to use when handling large datasets. The datastruce *DataFrame()* was used in this project because it structures the data like a csv document and can be easily mutable. Example on how you read a csv file to a *DataFrame()* object:

```
import pandas as pd

df = pd.DataFrame()
df = pd.read_csv("./movie_dataset_final.csv", low_memory=False, index_col=0)
```

After the data have imported, is time to clean and manipulate the data. In this project we mutated the data by creating a new *DataFrame()* containing the features we were interested in. Here is an example of how to get the genre column from the input dataset and adding it to the output dataset:

```
df_output = pd.DataFrame()
df_output['genre'] = df['genre']
```

The data in this column was structured in a .json format which is not readable by default in Python. One way of parsing that data is to use the library and function ast.literal_eval(). This way it is possible to access the data as a Python dictionary data structure. With this data, we made a column from that called "top_cast" which contained the 3 first actors/actresses. Here is an example of that using the *DataFrame().apply()* member function:

```
def getTopCast(row):
    cast = ''
    counter = 0
    for info in ast.literal_eval(row["cast"]):
        cast = cast + " " + info['name']
        counter = counter + 1
        if(counter == 3):
            return cast

    return ""
```

```
df_output["top_cast"] = df_credit.apply(getTopCast, axis=1)
```

Choosing the important features

When the dataset have been trimmed from all unnecessary data points, it is time to decide what features the algorithm will use to find similar movies. The accuracy of the recommendations comes down to what features you choose to compare, therefore it is important to choose them with a bit of thought.

The things you should consider when choosing these parameters, is do you want to find movies that have *similar cast, same director, similar budget* or *movies with similar plots*? Maybe all of these features are interesting for you, but it could be good to play around with your settings to see what results you get from choosing different features to compare. For this example *the top billed cast*, *the main production company*, *keywords* from the movie plot, *the genres* and *the director* was chosen.

As explained earlier, before the cosine similarity is used, all the features will be combined into one string and then compared. To combine the chosen features and creating the appropriate *count_matrix* the following method was used:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity

def combine_features(row):
    try:
        return row["top_cast"] + " " + row["director"] + " " + row["keywords"]
              + " " + row["genres"] + " " + row["prod_comp"]
    except ValueError as e:
        pass

df = pd.read_csv("cleaned_movie_dataset.csv")
features = ["top_cast", "director", "keywords", "genres", "prod_comp"]

#Filling the NaN-data with an empty string to make it parsable
for feature in features:
    df[feature] = df[feature].fillna('')

#Combining the features to one string and add them to a new column
df["combined_features"] = df.apply(combine_features, axis=1)

cv = CountVectorizer()
# Using the new column to create the count_matrix (occurences of words)
count_matrix = cv.fit_transform(df["combined_features"])
```

Getting the recommendations

We have reached the final stretch and what everyone have been waiting for, it is time to retrieve those recommended movies. Frankly, the biggest lift has already been done when the cosine similarity matrix was created. Now it is just a matter of choosing a movie to get recommendations from and returning the corresponding row in the matrix, sorted by the cosine similarity value in a descending order.

An example of this could be the Table 2. Avatar and Alien have been used again, with two new movies added. One similar, and one not alike at all (Grown Ups).

Table 2, *a cosine similarity matrix example*

	Avatar	Alien	Grown Ups	Rogue One: A Star Wars Story
Avatar	1	0.67	0.02	0.72
Alien	0.67	1	0.03	0.62
Grown Ups	0.02	0.03	1	0.04
Rogue One: A Star Wars Story	0.72	0.62	0.04	1

If we would want to find similar movies to Avatar from this small dataframe, the result of the sorted row would be:

1. Avatar
2. Rouge one: A Star Wars Story
3. Alien
4. Grown Ups

Final code for the recommendation system

Now, with a little wider understanding of how the cosine similarity matrix looks and works, we can look at the final code for the recommendation system. A few extra functions have been added to retrieve the title of the movie from its index in the dataframe, and also vice verse. And with this as the base of the system, adding a server and a frontend to run the recommendation system on, is a piece of cake.

```
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity

df = pd.read_csv("./movie_dataset_final_scrubbed.csv")

#Helper function to retrieve the title of the movie from its index
def get_title_from_index(index):
    return df["original_title"].iloc[index]

#Helper function to retrieve the index of the movie from the title
def get_index_from_title(title):
    return df[df["original_title"] == title].index[0]
```

```
def combine_features(row):
    try:
        return row["top_cast"] + " " + row["director"] + " " + row["keywords"]
               + " " + row["genres"] + " " + row["prod_comp"]
    except ValueError as e:
        pass

features = ["top_cast", "director", "keywords", "genres", "prod_comp"]
for feature in features:
    df[feature] = df[feature].fillna('')

df["combined_features"] = df.apply(combine_features,axis=1)
cv = CountVectorizer()
count_matrix = cv.fit_transform(df["combined_features"])
cosine_sim = cosine_similarity(count_matrix)

movie_user_likes = "Avatar"
#Getting the index of the movie in the Data Frame
movie_index = get_index_from_title(movie_user_likes)
similar_movies =  list(enumerate(cosine_sim[movie_index]))
#Sorting the movies in descending similarity order
sorted_similar_movies = sorted(similar_movies,
                               key=lambda x:x[1],reverse=True)[1:]
for movie in sorted_similar_movies:
    print(get_title_from_index(movie[0]))

Console ____________________________________________________________________
      Aliens, Rogue One: A Star Wars Story, Alien³, Alien, Star Trek Into
      Darkness …
```

Creating a web application with Flask and React.js

The recommendation system is working as it should at this moment, however, it is not very user friendly. One way of solving this issue is to have the recommendation system run on a backend server and then have a frontend application calling on the backend to run the calculations.

Since it's a bit out of the scope of what this post is about, it will only be shortly described how you can create a simple REST-api using Flask, and then get the recommendations on the frontend.

Create a Flask REST-api

Since the movie recommendation system is written in Python, it makes a lot of sense to use Flask as a backend framework since it's written in Python as well. It is also quick and easy to setup and create a REST-api. With just a few imports, the basic structure can be created, and then with a simple http call from the frontend, all the movie recommendations can be retrieved:

```
from flask import Flask, request, make_response
import requests, json
from src.movie_recomendation import getRecommendations

app = Flask(__name__, static_url_path="")
# route from front end to be called src: “get_recomendations”.
@app.route('/get_recomendations, , methods=['POST']
def getRecommendation():
    # The title of the liked movie is retrieved from the frontend.
    liked_movie = request.json['title']
    # Call the movie recommendation function
    movieListArray = getRecommendations(liked_movie)
    response = app.response_class(
        response=json.dumps(movieListArray),
        status=200,
        mimetype='application/json'
    )
    return response

if __name__ == '__main__':
    app.run(debug=True)
```

Create a React Frontend application

React is currently one of the most popular frontend *Javascript* frameworks out there. It is easy to setup and have a large community of users. It is also easily compatible with the REST-api created in Flask through http calls. React was set up using a multitude of components that represented different parts of the application. React makes it possible to easily create dynamic websites with a lot of structure.

Here is a screenshot from the web application:

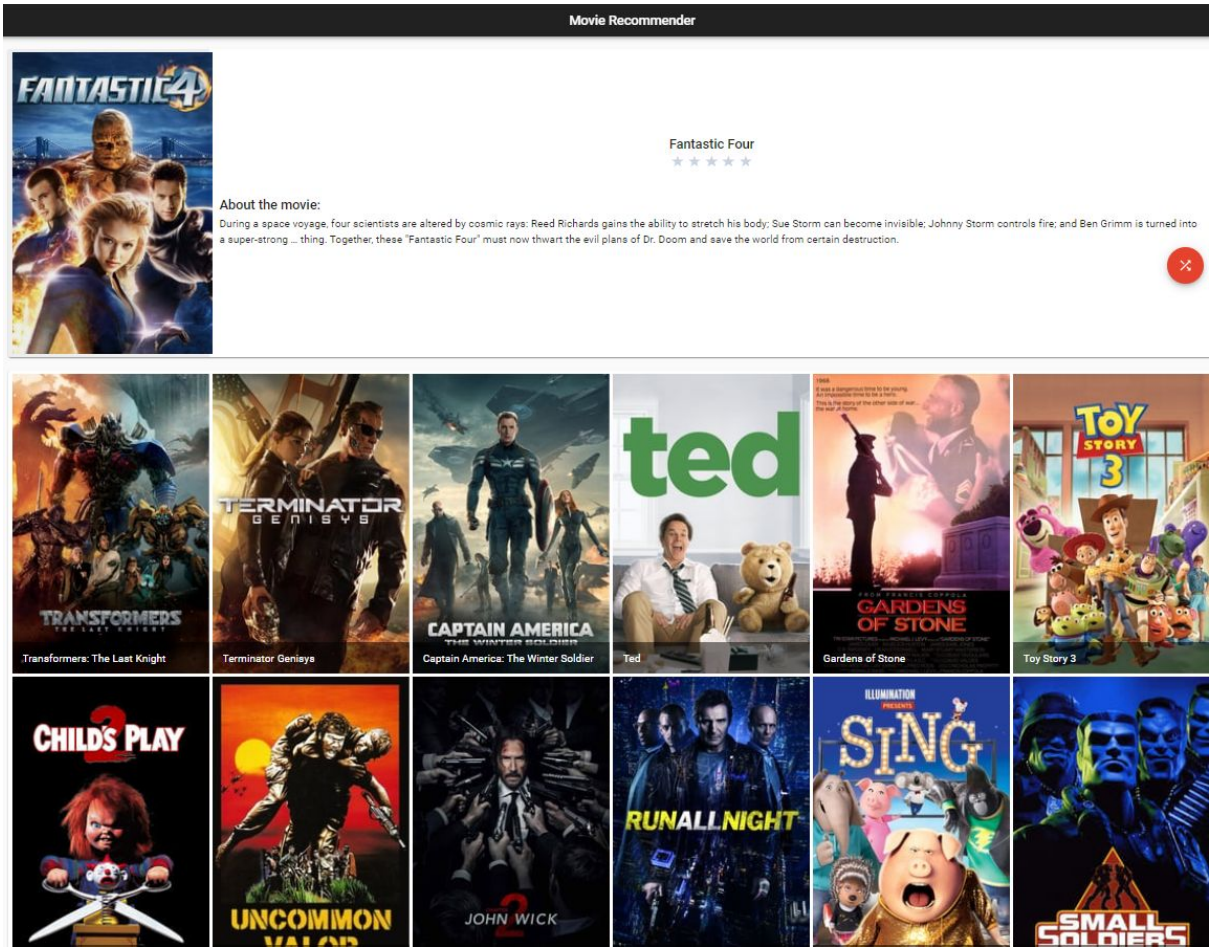

Figure 4, *the recommendation system in action with react.js*

As can be seen from this image you now have a clear visual representation of the movies and a way to rate them. This is much more intuitive than doing this from the command line. We know that this is outside of the scope of this post. But in order to be able to see the possibilities of this kind of recommender system, we thought building an application that used it made sense. Because this is the kind of applications that will use recommender systems in practise.

Here you can see how the data is collected from the REST-api through an http call to the backend:

```
getMovieToRate = () => {
    axios.get("http://127.0.0.1:5000/movies").then(res =>
      this.setState(prevState => ({
        movieToRank: {
          ...prevState.movieToRank,
          original_title: res.data.original_title,
          poster_path: res.data.poster_path,
          overview: res.data.overview
        }
      }))
    );
  };
```

Final thoughts

Now we have a functioning movie recommendation application using a content based filtering approach. But when using this system you will be able to see its flaws. It recommends good movies based on the features chosen, but as soon as you start using it you compare it to recommender systems Netflix and similar services. What you realise in this comparison is that the system does not feel very personal, that it does not know you, which Netflix seems to do. But in order to get a personalised recommender system you need to also bring in collaborative filtering as this gives that feeling.

We hope that this introduction to recommendation systems sparked your interest and that you now want to build an application on your own.

This post/project was based on the following post: Building a Movie Recommendation Engine in Python using Scikit-Learn

By: Viktor Sandberg and Jacob Nyman