How To Create Simple Keyword-based Movie Recommender Models From Scratch

Gideon Blinick Follow
May 13 · 12 min read ★

Introduction

Have you ever tried to use a movie recommender? In theory, it is something useful that can help figure out what to watch next instead of browsing through Netflix for a few hours, but their results tend to be hit-or-miss. This is a problem that most people can relate to, so I decided to create a homemade recommender system myself and share it in this blog post. I will show you how to create 3 simple recommender models from scratch that accept a movie as input and return the "n" most similar movies as output, with "n" being provided by the user.

In general, recommender systems are either content-based or collaborative with the user's history and interests. I chose to create content-based models since they make predictions based on the specific input item (movie) and not based on the user.

Note that the recommendations for this blog are based entirely off of movie keywords. As you will see, it is fairly simple to use other text information, such as plot outlines or plot synopses, instead of movie keywords.

To limit the scope of this post, I did not include numeric features (e.g. runtime, year of release, rating, etc.) and genre for recommendations.

If you want to follow along or see the code for this post on GitHub, visit: https://github.com/gblinick/Movie-Recommender-with-NLP/blob/master/Keyword%20Movie%20Recommender.ipynb

Dataset Creation

The first step in any data science project is to acquire your dataset. Sometimes your dataset(s) will be given to you (like with Kaggle competitions). In our case, since this project is "from scratch" we must obtain the data ourselves.

The first place to look for data on movies is IMDb's public datasets repository: https://www.imdb.com/interfaces/. This repository contains 7 tables with all sorts of great movie data. Unfortunately, none of the tables contain movie keywords that we can use for recommendations. That does not mean that we cannot use any of the tables. Looking at what the tables have to offer, we notice that one of them, *title.ratings.tsv.gz* contains a column called "numVotes" representing the "number of votes the title has received". This is useful to us because we can use this column to get the top 10,000 movies by number of votes. Of course, it would be nice to be able to include every movie ever made in our model but that is not practical due to the extra time needed to compute similarity. Additionally, even if we did not have a practical computation drawback, I am not sure most people are interested in watching movies outside of the top 10,000. The top 10,000 movies should suffice to keep the vast majority of viewers happy.

We also need a second table, *title.basics.tsv.gz*, which provides the column TitleType, which tells us if the title is a movie (as opposed to a TV episode). Since we only want to include movies in our recommenders, we use this column to filter for movies.

Putting everything together, this is our workflow so far:

1. Import necessary libraries (pandas and numpy):

```
# Import libraries
import pandas as pd
import numpy as np
```

2. Import the 2 IMDb tables we need into pandas' DataFrames (in the image, the files are in a folder called 'data' which is parallel to our notebook):

```
# Import 2 tables from IMDb datasets that we need
title_basics = pd.read_csv('data/title.basics.tsv.gz',
title_ratings = pd.read_csv('data/title.ratings.tsv.gz'
```

3. Select only movies from the *title.basics* table and create a new DataFrame, 'movies':

```
# Select only movies from the title_basics table
movies = title_basics[(title_basics.titleType == 'movie
```

4. Create a common index for the tables (the IMDb unique identifier of each title in both tables is a great choice here) and then join the tables with a simple inner join:

```
# Set the indicies for our 2 tables to the IMDb key for
movies.set_index('tconst', inplace=True)
title_ratings.set_index('tconst', inplace=True)
# Joint the 2 tables by tconst, the IMDb key for all mo
```

Great! We now have a table called "movies_with_rating" that has all the IMDb movies with their ratings (and all the other information in the 2 tables we used). Let's take a look at the top row:

```
movies_with_rating.head(1)
```

tconst	titleType	primaryTitle	originalTitle	isAdult	startYear	endYear	runtimeMinutes	genres	averageRating	numVotes
tt0000009	movie	Miss Jerry	Miss Jerry	0	1894	\N	45	Romance	5.6	75

We see some interesting information about the first movie in the table. We don't need all this data for our future models, however. All we care about for this project is getting the top 10,000 movies by numVotes, and using movie keywords to find similarity between movies.

We can do that easily using the pandas' .sort_values function:

```
# Take only the top 10,000 movies, where we rank movies
top_10000_movies = movies_with_rating.sort_values(by='r
movies_index = top_10000_movies.index
```

```
top_10000_movies.shape
```

```
(10000, 10)
```

Perfect. We can now turn our attention to getting the plot information we want to use for our recommender models. We can get such plot information using the wonderful IMDbPY API: https://imdbpy.readthedocs.io/en/latest/index.html. You will need to install the API using pip or conda.

Before using the API, we need to import it and instantiate an IMDb movie object. We'll do that and also import a couple other libraries that make running for-loops with an API a nicer experience:

```
from tqdm import tqdm
from time import sleep
from imdb import IMDb

```

The syntax for pulling plot keywords from IMDb is as follows:

```
keywords_dict = {}
for movie_index in tqdm(movies_index):
    sleep(1)
    try:
        keywords_dict[movie_index] = ia.get_movie_keywo
    except:
```

In the above code, I first instantiate a dictionary to store the keywords I will get back from IMDb (in line 1). I will use the movie IDs as keys and the list of keywords for each movie as values.

In the for loop beginning on line 2, I loop through the indexes of the top 10,000 movies. On line 5, I use each movie index to get the

corresponding IMDb movie object. Note that on line 5 I subset the movie index from index 2 and on because the indexes start with 'tt' followed by a number and we only need the number. I then subset the movie object by 'plot outline' to get the required plot outline, and then store the resulting keywords list in my keywords dictionary as a value corresponding to the movie index key.

Note my use of the tqdm and sleep libraries in the code above. tqdm allows you to see where you are in the running of a for-loop. When you are running a for-loop 10,000 times (as we are), it's nice to know how far along you are.

The sleep library allows us to be courteous to IMDb. Given that we are asking the website to give us information 10,000 times, it is nice to space our requests to it so that we do not overwhelm the server.

After finishing the for-loop, we have a dictionary where the keys are movie IDs, and the values are keywords lists. It looks like this:

```
1 keywords_dict
```

```
{'tt0111161': ['wrongful-imprisonment',
  'escape-from-prison',
  'based-on-the-works-of-stephen-king',
  'prison',
  'voice-over-narration',
  'suicide-by-hanging',
  'prison-cell-search',
  'sexual-assault',
  'infidelity',
  'police-brutality',
```

We should convert this dictionary to a pandas DataFrame to take advantage of pandas' abilities. However, if we just do a simple *pd.DataFrame(keywords_dict),* pandas will complain that our dictionaries aren't the same length. So we need a solution to get us what we want, as illustrated in the following code:

```
1 keywords = pd.DataFrame(dict([ (k,pd.Series(v)) for k,v
2 keywords = keywords.apply(lambda x: ','.join(x.dropna()
3 keywords = pd.DataFrame(keywords)
4 keywords.rename(columns={0:'keywords'}, inplace=True)
```

```
# Save the plots to a CSV
keywords.to_csv(path_or_buf='Blog Post/keywords.csv')
```

In line 1, we create a new dictionary where we have the same keys but different values from our earlier dictionary. Instead of the values being keywords lists, they are now pandas' Series. We can convert this dictionary to a DataFrame and then join all the keywords (which are now their own columns) with a simple lambda function in line 10. After applying the lambda function we get back a Series, so we can convert this Series to a DataFrame, rename our columns, and save to a CSV.

We don't need to save the plot outlines to a CSV, but anytime you obtain data from a process that takes a long time, it is very smart to save it so that you do not have to re-run the process that got you the data.

To use our plot outlines data, we can load it from the keywords.csv file, rename the 2 columns that exist in the file (i.e. the key and value) and join it to our existing table:

```
# Load our keywords for the CSV
keywords = pd.read_csv('Blog Post/keywords.csv')
keywords.rename(columns={'Unnamed: 0':'tconst'}, inplac
keywords.set_index('tconst', inplace=True)
```

```
# Join to our table
movies = top_10000_movies.join(keywords, how='inner')
```

And that's it!

We now have a dataset of plot text information that we can use to calculate similarity between movies.

EDA

Before creating our models, let's do some quick EDA. Text data is not the easiest data to do exploratory data analysis with, so we'll keep the EDA short.

The first thing we can do is take a look at the distribution for the number of characters in each keyword list. Let's first get rid of movies

with no keywords:

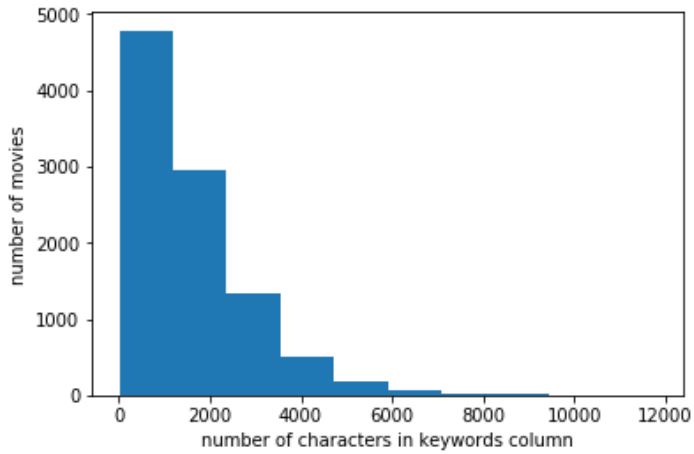
```
movies = movies[movies.keywords.notnull()]
```

We can now find the length (in characters) of each keyword list:

```
import matplotlib.pyplot as plt

number_of_keyword_chars = []
for keywords in movies['keywords'][:10000]:
    n_chars = len(keywords)
    number_of_keyword_chars.append(n_chars)

plt.hist(number_of_keyword_chars)
plt.xlabel('number of characters in keywords column')
plt.ylabel('number of movies')
plt.show()
```




```
for col in movies.columns:
    print("%s: %.3f" %(col,np.mean(movies[col].str.len())) )
```

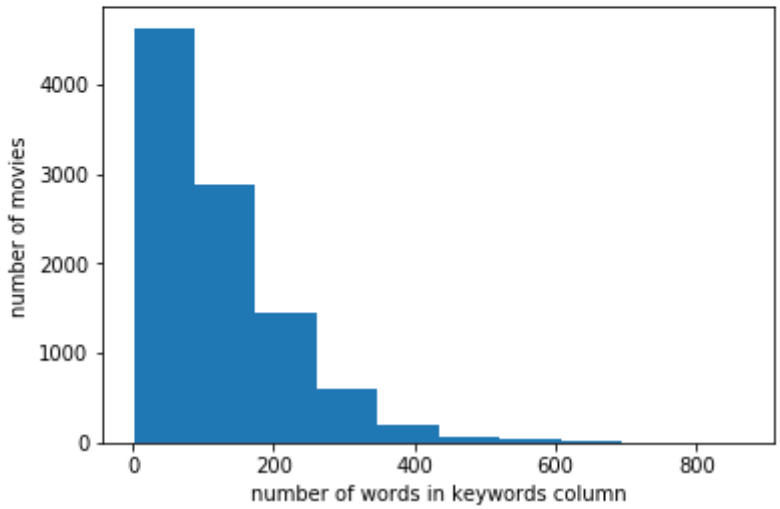
```
keywords: 1585.197
```

We see from the distribution and the average number of character per column that the distribution is quite right-tailed.

Let's see if we get a similar-looking distribution if we plot the distribution for the number of words in each keyword list:

```
number_of_keywords = []
for keywords in movies['keywords']:
    n_keywords = len(keywords.split(','))
    number_of_keywords.append(n_keywords)

plt.hist(number_of_keywords)
plt.xlabel('number of words in keywords column')
plt.ylabel('number of movies')
plt.show()
```



Yep, also quite right-tailed.

So we looked at 2 different measures for our keyword lists—number of characters of the list and number of words in the list—and they showed the same pattern. Which makes sense, as the measures are intuitively correlated. What the right-tails tell us is that most movies do not have a lot of keywords or characters but a small few—presumably the most successful movies—have a lot of keywords and characters.

Creating Models

We can now create our models. First, I would like to discuss a point about NLP and using text in general. Ordinarily, when using text, you do not have nice, easy-to-use keyword lists. This would have been the case for us if we had decided to use plot synopses instead of keyword lists (which would have happened if IMDb did not provide movie keywords). In that case, there are a number of standard, simple steps you can take in order to essentially create your own keyword lists. They are:

1. Lowercasing words

2. Removing punctuation

3. Removing stop words

4. Lemmatizing words

These can all be accomplished relatively simply using the nltk library.

Back to our models: We are going to create 3 different models, based both on different notions of similarity as well as different formulations of movie text 'vectors'. The first 2 models will use cosine similarity, and the last one will use Jaccard similarity. Within the first 2 models, the first will use tf-idf to create movie vectors, and the second will use simple word counts.

We start by selecting the columns we will need. Since we are creating keyword-based recommenders, we need the keywords column. Because we also want to input a movie title and get back other movies, we will need the 'primaryTitle' column. So we will select those 2 columns. We also reset the index because our recommending function later will use a movie's position in the DataFrame, not its IMDb ID, for recommendations:

```
1  movies.reset_index(inplace=True)
2  movies = movies[['primaryTitle', 'keywords']]
```

Next, we need to get a list of lists, where the inner lists are the lists of keywords for each movie. What we have in our keywords column currently for each movie is a string with all the keywords in it and separated by commas.

We can get such a list of lists by using the nltk library:

```
1  # Processing Keywords
2  keywords = movies['keywords'].tolist()
3
4  from nltk.tokenize import word_tokenize
```

In line 2, we get the aforementioned list of strings. In line 4, we import the *work_tokenize* module from the nltk library. In line 5, we have a list comprehension where we loop through every keyword string and for each one we get the individual words with *word_tokenize*. Note that *word_tokenize(keyword.lower())* returns a list of keywords, so we end up with a list of lists.

We are not quite done at this point since *word_tokenize* left us with many commas in our list of keyword lists. We can easily get rid of them by defining a custom function, *no_commas*, and applying it to every keyword list in our list of keyword lists:

```
def no_commas(doc):
    no_commas = [t for t in doc if t!=',']
    return(no_commas)

keywords = [no_commas(kw) for kw in keywords]
```

Nice. So we now have what we want in *processed_keywords*.

We can now create our first model, a tf-idf model that uses cosine similarity between word-vectors. I'm not going to explain how those work here. A quick Google search should return many helpful resources.

We start by creating a dictionary of words using gensim:

```
from gensim.corpora.dictionary import Dictionary
dictionary = Dictionary(processed_keywords) # create a
```

The dictionary here just contains every word in our processed keywords list matched with an ID.

Next, we create a genism corpus, where corpus here just means 'bag-of-words for each movie':

```
corpus = [dictionary.doc2bow(doc) for doc in processed_
#create corpus where the corpus is a bag of words for e
```

Next, we can convert these bags-of-words into tf-idf models using a genism tf-idf model:

```
from gensim.models.tfidfmodel import TfidfModel
tfidf = TfidfModel(corpus) #create tfidf model of the c
```

Finally, we create an index for our set of movie keywords that allows us to compute similarities between any given set of keywords and the keywords of every movie in our dataset:

```
from gensim.similarities import MatrixSimilarity
# Create the similarity data structure. This is the mos
sims = MatrixSimilarity(tfidf[corpus], num_features=ler
```

We are now at the point where we can write a function that accepts a movie, and returns the “n” most similar movies. The way our function works will be as follows:

1. Accept a movie from a user

2. Accept “n” from the user where “n” is how many movies the user wants to be returned

3. Retrieve the movie’s keywords

4. Convert the movie’s keywords into a bag-of-words

5. Convert the bag-of-words representation into a tf-idf representation

6. Use the tf-idf representation as a query doc to query our similarity measure object

7. Get back the similarity results for every movie in our set, sort the set by decreasing similarity, and return the “n” most similar movies along with their similarity measures. Do not return the most similar movie because every movie will be most similar to itself

In code:

```
def movie_recommendation(movie_title, number_of_hits=5
    movie = movies.loc[movies.primaryTitle==movie_titl
    keywords = movie['keywords'].iloc[0].split(',') #g
    # get just the keywords string ([0]), and then cor
    query_doc = keywords #set the query_doc to the lis

    query_doc_bow = dictionary.doc2bow(query_doc) # ge
    query_doc_tfidf = tfidf[query_doc_bow] #convert th
    # of the movie ID and it's tf-idf value for the mo

    similarity_array = sims[query_doc_tfidf] # get the
    #So the length is the number of movies we have. To

    similarity_series = pd.Series(similarity_array.tol
    top_hits = similarity_series.sort_values(ascending
    #get the top matching results, i.e. most similar m

    #print the words with the highest tf-idf values fo
```

Now let's test it out. Because the Avengers are so popular these days, let's give the original movie to our function:

```
movie_recommendation('The Avengers', 5)
```

```
The top 5 words associated with this movie by tf-idf are:
 'black-eye-patch' with a tf-idf score of 0.101
 'imax,3-dimensional' with a tf-idf score of 0.101
 'superhero-team,2010s' with a tf-idf score of 0.101
 'flying-fortress' with a tf-idf score of 0.093
 'marvel-comic' with a tf-idf score of 0.093
Our top 5 most similar movies for movie The Avengers are:
1 Avengers: Age of Ultron with a similarity score of 0.399
2 Avengers: Infinity War with a similarity score of 0.286
3 Iron Man 2 with a similarity score of 0.274
4 Captain America: Civil War with a similarity score of 0.251
5 Captain America: The Winter Soldier with a similarity score of 0.250
```

All seem to be good matches.

Note that this kind of recommender system isn't limited to just recommending movies based off of other movies. We can query our similarity measure object using any given set of keywords. It just so happens that most people will want to query it using movies they already know, but it is not limited to that.

Here's the more general function for recommending based on provided keywords:

```
def keywords_recommendation(keywords, number_of_hits):
    query_doc_bow = dictionary.doc2bow(keywords) # get
    query_doc_tfidf = tfidf[query_doc_bow] #convert th
    # of the movie ID and it's tf-idf value for the mo

    similarity_array = sims[query_doc_tfidf] # get the
    #So the length is the number of movies we have. To

    similarity_series = pd.Series(similarity_array.tol
    top_hits = similarity_series.sort_values(ascending
    # i.e. most similar movies
```

We can move on to our second recommender, which also uses cosine similarity to calculate the similarity between word vectors. It will differ from the first model by using simpler word counts instead of creating tf-idf word vectors. This alternate implementation is performed with the *CountVectorizer* class in scikit-learn, which converts a collection of text documents (in this case, keyword lists) into a matrix of token counts.

We will also compute cosine similarity a little differently for simplicity. Instead of using *MatrixSimilarity*, we will use the *cosine_similarity* function from the scikit-learn *metrics.pairwise* submodule.

We compute the word counts for each movie with the following code:

```
from sklearn.feature_extraction.text import CountVector
def get_vectors(text):
    vectorizer = CountVectorizer(text)
    X = vectorizer.fit_transform(text).toarray()
    return(X)
```

Then, when given a movie, all we need to do is compute the cosine similarity between that movie's word vector and every other word vector and return the most similar matches. This is achieved with our *cosine_recommender* function:

```
from sklearn.metrics.pairwise import cosine_similarity

def cosine_recommender(movie_title, number_of_hits=5):
    movie_index = movies[movies.primaryTitle == movie_

    cosines = []
    for i in range(len(vectors)):
        vector_list = [vectors[movie_index], vectors[i
        cosines.append(cosine_similarity(vector_list)[

    cosines = pd.Series(cosines)
```

Let's try it out with "The Avengers" as well:

```
cosine_recommender('The Avengers')
```

```
Avengers: Infinity War 0.8044695892203602
Avengers: Age of Ultron 0.7913277135173737
Captain America: Civil War 0.7509793475917405
Iron Man 2 0.7470155795225362
Justice League 0.7091790018343553
```

It works! Similar matches to our first model.

For our third model, we use Jaccard similarity instead of cosine similarity. As a result, we have no need for word vectors at all. We simply calculate similarity as the intersection of the set of keywords divided by the union of the set of keywords.

The code for computing Jaccard Similarity between any two lists of keywords is straightforward:

```
def get_jaccard_sim(str1, str2):
    a = set(str1.split(','))
    b = set(str2.split(','))
    c = a.intersection(b)
```

Creating a recommender model from this is correspondingly simple. We find the keyword list for the given inputted movie, compute the Jaccard similarity between that list and every other movie keyword list and then rank the movies by their similarities and return the top "n" results.

In code:

```
def jaccard_recommender(movie_title, number_of_hits=5)
    movie = movies[movies.primaryTitle==movie_title]
    keyword_string = movie.keywords.iloc[0]

    jaccards = []
    for movie in movies['keywords']:
        jaccards.append(get_jaccard_sim(keyword_string
    jaccards = pd.Series(jaccards)
    jaccards_index = jaccards.nlargest(number_of_hits+
```

Testing this with our favourite movie, we get back good results:

```
jaccard_recommender('The Avengers')
```

```
Avengers: Age of Ultron 0.27450980392156865
Avengers: Infinity War 0.2370266479663394
Captain America: The Winter Soldier 0.23141891891891891
Captain America: Civil War 0.21246458923512748
Thor: The Dark World 0.20722433460076045
```

Next Steps

There are a number of ways we can improve the models as they currently exist.

At the top of the list is deploying the models with Flask so people can use them. Usability is pretty important.

The next step would be to find some way of measuring performance for our models. This is tricky because recommender systems often have no obvious evaluation criteria. Still, we can come up with some (e.g. using user feedback to rank models), and there are good resources out there for doing so.

A third improvement to consider is using deep-learning methods on plot summaries with embeddings: https://tfhub.dev/google/universal-sentence-encoder/2. These methods should allow us to incorporate context into our recommendations, rather than simple individual words like our models currently use.

Lastly, the models could be improved by incorporating other non-text features such as genre or numeric features.

Check back in this space in a few weeks time. I might do another blog post where I implement these improvements.

Please let me know what you think of the project and if you have any recommendations for improvement. All constructive comments are much appreciated.

References

For inspiration for the use of the Gensim MatrixSimilarity class to compare documents, I used O'Rielly's wonderful tutorial "How do I compare document similarity using Python": https://www.oreilly.com/learning/how-do-i-compare-document-similarity-using-python

For inspiration for the use of Jaccard and cosine similarity recommenders, I am indebted to Sanket Gupta for his tutorial "Overview of Text Similarity Metrics in Python": https://towardsdatascience.com/overview-of-text-similarity-metrics-3397c4601f50