Gideon Blinick

Capstone Project 2 Final Report: Movie Recommender

Problem statement

In this project, I created 3 different movie recommendation models that recommend movies to a user. The models take as input a movie, as well as the number of similar movies the user wants to see. So the use case is for a user to enter a movie they like and get back the n most similar movies where n is chosen by the user. 'Most similar' will be defined in the modelling section, and indeed, each of the 3 recommenders uses a different notion or type of similarity

The recommenders find the most similar movies based on plot text information. This text can be plot synopses, plot outlines, plot summaries, or plot keywords. I use keywords in this project, but it is simple to use the other types of text information because the natural language processing steps are identical.

Client and Use

My client is anyone who enjoys watching movies and wants to watch more movies that they enjoy. The use case for my engines is for a client to think of a movie that they have enjoyed in the past and to enter that movie in one of my engines. The engine will return the n movies most similar to the given movie. If the client hasn't seen the most similar movie, they now have a movie to watch that they will hopefully enjoy. If they have seen the movie, they can continue down the list of similar movies until they find a movie that they haven't seen.

The goal is to be able to provide movie-watchers with a convenient and fast way of finding content they love.

Dataset Description

Dataset creation notebook available here.

My dataset was custom created from the IMDb website using 2 different sources. The first source was IMDb's public dataset's repository.

This repository contains seven different tsv files of thousands of movie and TV episode information. All 7 files were downloaded and unzipped into the project's data folder and each was individually examined for useful information. Since the goal of this project is to use plot text data as the primary means by which to recommend movies, it was disappointing to find that none of those files contained text data. Still, 3 of the public tables looked interesting, including the title_basics, title_crew, and title_ratings tables. These tables had numeric features and a genre feature that would be nice to incorporate alongside text.

The title_basics table contained the most useful information I wanted including movie ID, movie name, release year, runtime, and genre. The title_ratings table was useful for its information about movie ratings and the number of votes each movie had received, both features that would be nice in a model. The title_crew table was initially included for the information it had about directors and writers. In theory, adding these features to my dataset would improve the accuracy of my models. However, after some consideration, I decided to remove these features because they would dramatically increase the dimensionality of my feature space and would likely lead to poorer recommendation performance. That

is, since there are so many directors and writers out there, I would have needed many, many columns in my dataset because one-hot-encoding is the appropriate way for handling such categorical data.

The other tables were of no use to me because they contained information about people (name.basics, title.principals) , TV episodes (title.episode), and titles (title_akas), all features I didn't need or want in my dataset.

To get the plot information I needed, I used the wonderful IMDbPY API.

IMDbPY made pulling plot data from IMDb easy. All I needed to do was call the API 10,000 times to get whatever plot information I wanted. I called it 10,000 times because I wanted the dataset to contain a sufficient number of movies with which to recommend other movies, and 10,000 was a nice, round number for this. Furthermore, to make my engine more relevant, I only called the API for the 10,000 most popular movies, where popularity was determined by the number of votes a movie received (i.e. how many times the movie had been rated). This information was available from the title_ratings table.

I called the API a few times because there were multiple types of plot information that I wanted to retrieve: basic plot descriptions (usually a sentence or two), plot outlines (usually a paragraph), plot keywords (a list of words describing a movie), and plot synopses (the longest plot descriptions that describe a movie scene by scene).

After making the API calls, the data was stored in CSV files so the API didn't need to be used again. Below is the code used for retrieving the plot outlines information and saving it to a CSV. The same code worked, with minor adjustments, for obtaining the other plot information.

In [14]:
```
plot_outlines= {}

for movie_index in tqdm(movies_index):
    sleep(1.5)
    movie = ia.get_movie(movie_index[2:])
    try:
        plot_outlines[movie_index] = movie['plot outline']
    except:
        plot_outlines[movie_index] = ''

## Convert out dictionary to a Dataframe and raname our column to 'plot'

plot_outlines = pd.DataFrame.from_dict(plot_outlines, orient='index')
plot_outlines.rename(columns={0:'plot'}, inplace=True)

## Save the plots to a CSV
plot_outlines.to_csv(path_or_buf='plot_outlines.csv')
```

```
100%|██████████| 10000/10000 [8:35:19<00:00,  2.93s/it]
```

Once I had finished all the API calls, I was able to join all the information together into one dataframe using the pandas' join function. This was possible since each of the constituent dataframes had movie identifier as a column and I set that column as the index in order to join them.

Exploratory Data Analysis (EDA)

EDA notebook available here.

First, I looked at the average length by number of characters for each of the text columns: plot, plot outline, keywords, and synopsis:

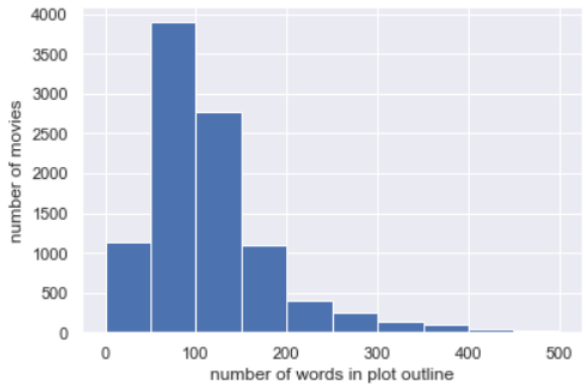
```
for col in text_cols.columns:
    print("%s: %.3f" %(col,np.mean(text_cols[col].str.len())) )
```

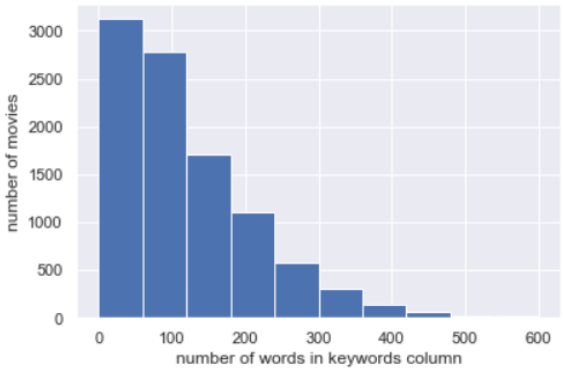
```
plot: 177.802
plot outlines: 586.338
keywords: 1585.197
synopsis: 7080.901
```

I found that synopsis is by far the longest on average, followed by keywords, plot outline, and plot. This matched my expectations since, as described above, synopsis contains paragraphs describing the movie scene by scene, keywords contains many keywords describing the movie, plot outline is generally around a paragraph of description about the movie, and plot is generally a sentence or two of description.

Next, I looked at the distributions of the number of words for just plot outlines and keywords. Keywords is a great text column for modelling purposes because it contains words that are expressly used to categorize and classify a movie. Plot outline is a nice column to use because it is not too long or too short, unlike plot and synopsis.

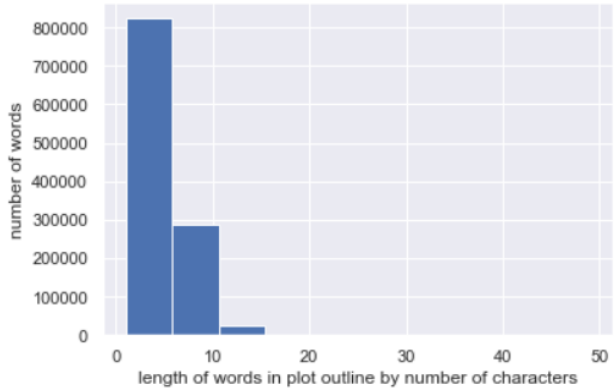
Plotting the number of words for plot outlines and keywords, I obtained the following charts:

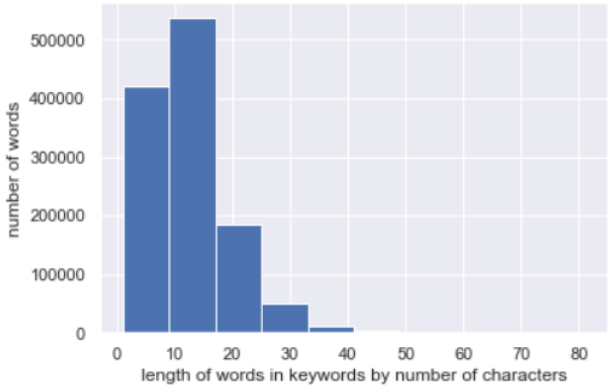




When looking at the summary statistics for these distributions, I found that the mean number of words for keywords is only slightly greater than the number of words for plot outline (~122 vs. ~115). Indeed the distribution for keywords (the right chart) doesn't make it seem like there are more keywords per movie than words in the plot outline. This is misleading however, since most keywords are compound words (e.g. "dc-comics"). This is why the number of characters for keywords is far greater than for plot outlines. Additionally, the words in keywords are more sophisticated than the words in plot outlines (because there are no stop words among the keywords and because the words are more descriptive).

When I plotted the lengths of words for plot outlines and keywords, I obtained the following distributions.





So we see that the keywords are longer than the words in the plot outlines.

Finally, I found the top 5 words across plot outlines after tokenizing, lowercasing, only taking alpha words, removing stop words and lemmatizing using the nltk and genism libraries. The words were “life, one, find, get, and new”. When doing the same for the keywords, I found the top words to be “murder, death, blood, husband-wife-relationship, and violence”. So I confirmed that the top keywords are more complicated and descriptive than the most common words from plot outlines, which are more commonly used words. This holds in general for the keywords and the words in plot outlines.

Modelling

In this part of the project, I created 3 different movie recommenders. The recommenders work based only on the text features of the movies, specifically the keywords used to describe the movies. It is a fairly simple matter to have the models use text from the plot outlines column or from the plot and synopsis columns, but using these other sources of data should result in inferior performance given how rich the text data is from the keywords columns. Richness here refers to both the quantity and quality of the data. That is, there are many keywords - more keywords per movie, on average, than words used in plot outline and plot; and the keywords are unique since they don't contain stop words, unlike the synopsis column.

There was effectively no preprocessing to do on the text before feeding it into the models for the following reasons:

1) Tokenization: Normally, we need to take a block of text and create tokens out of it. Since my keywords column contains comma separated words or groups of words, it was effectively tokenized for me and all I needed to do was split on the commas to get the text.
2) Lowercasing: All the keywords were already lowercased.
3) Taking Only Alpha Words: Ordinarily, I would want to use the .isalpha() string method to avoid taking punctuation and the like. In this case, there was no punctuation to avoid and using .isalpha() would have additionally caused me to lose most of the words since most of the keywords were actually 2 or 3 words grouped together with dashes ("-") .
4) Removing stop words: None of the keywords were, logically enough, stop words.
5) Lemmatizing: This is something that I might have done that could have made a lot of sense. Still, I decided not to lemmatize after seeing how the meanings of some descriptor words could be changed. For example, "woods" which indicates a forest, becomes "wood" the material. Or "avengers" becomes "avenger". In both cases, the first words have a meaning that is more than just the plural of the second words.

1st Model: Tf-idf with Cosine Similarity

The first model I created was a Tf-idf model that works with Gensim Similarity.

Tf-idf stands for *term frequency-inverse document frequency* and is a way of assigning importance to words in a document based on how often they appear in that document and in the other documents in a corpus (here corpus just means 'collection of documents'). Tf-idf allows you to get a sense of what a document is about by seeing what words are most important in it.

The first term in the calculation of word importance is *tf* or term frequency. This is quite simply the number of times a word appears in a document divided by the number of words in that document: $tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$. In this equation, $tf_{i,j}$ stands for the term frequency of word *i* in document *j*, $n_{i,j}$ represents the number of times word *i* appears in document *j*, and the summation in the dominator is over *k*, the number of unique words in document *j*. So the denominator represents the number of words in the document. The higher the *tf* number is for a particular word, the greater the importance of that word.

The second term in the calculation of word importance is *idf* or inverse-document frequency. This is a little trickier to understand. The idea here is that if a word appears in a document but it also appears in many other documents, then it probably does not say very much about the document. Classic examples

of such words are stop-words like “and” and “the” which are very common words but would not tell you much about the contents of a document. We would like such words to be down-weighted in importance. Conversely, if a word appears in a document and nowhere else, there is reason to believe that it is very important to the document by virtue of its rarity. We should think that this word would tell us a lot about what the document is about.

To calculate *idf*, we take the logarithm of the total number of documents divided by the number of documents that a given word appears in: $idf_w = \log(\frac{N}{df_t})$. In the equation, N is the total number of documents in a corpus and df_t is the number of documents that a particular word appears in. df_t can be at most *N*, in which case idf_w becomes 0. Conversely, the smaller df_t is, the greater idf_w becomes.

Putting it all together we get an equation for word importance for every word *i* in every document *j*:

$$w_{i,j} = tf_{i,j} * \log(\frac{N}{df_i})$$

You can learn more about tf-idf here and here.

To create this model, I began with what I had after dataset creation, EDA, and preprocessing, which is a list of lists where the inner lists are lists of keywords for every movie in my dataset. I created a dictionary mapping every unique keyword to a unique id using a Gensim Dictionary object. This was useful because by using that dictionary object, I could create a bag-of-words for every keyword list I had using the Gensim Dictionary object .doc2bow() method. Gensim calls this resulting list of bags-of-words a ‘corpus’. Note that this is different from how I used corpus earlier, where it just meant ‘collection of documents’. Now it means, collection of bags-of-words.

This corpus is critical to creating a tf-idf model because it contains all the info I needed to come up with word importances: specific word counts per doc, total word counts per doc, total number of docs, and number of docs where any particular word appears in.

Creating the actual tf-idf model was as simple as passing my Gensim corpus to a TfidfModel class imported from Gensim. That gave me the weights for every word in every document.

So now I had a tf-idf model but was only halfway to my goal of being able to recommend movies when given a movie. The reason for this is because I still needed some way of determining how similar two movies are.

The most common way of computing similarity between text documents is to use cosine similarity. Cosine similarity calculates similarity by measuring the cosine of the angle between 2 vectors. This is calculated as

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

Figure 1: Cosine Similarity (image taken from Wikipedia)

Unlike Jaccard similarity (later explained) which only requires lists of words, cosine similarity requires numerical vectors on which to operate. Fortunately for me, I had such vectors: the document representations in my tf-idf model. Each document there is represented by word importances. So now I could find the similarity between 2 documents by finding the similarity between each document's word importance vectors.

In practice, the cosine similarity between each movie's tf-idf representation and every other movie's tf-idf representation can be calculated using the MatrixSimilarity function from the Gensim similarities module. By feeding it my tf-idf corpus, finding the most similar movies to a given movie became a simple task. All I need to do when given a movie is perform the following steps:

1. Retrieve the movie's keywords;
2. Convert those keywords into a bag-of-words model;
3. Convert the bag-of-words model into a tf-idf representation of those words using my corpus to calculate the idf term;
4. Find the n most similar movies to the movie's representation using MatrixSimilarity which uses cosine similarity between the given movie and every other movie

Note that the model can be used to find the most similar movies for any given set of keywords. Its most natural use is the specific case where the given set of keywords used is the set of keywords used to describe a particular movie that someone likes, but it can be used for any set of keywords.

The implementation of the entire tf-idf model with cosine similarity is as follows:

```
import csv
with open('processed_docs.csv', 'r') as f:
    reader = csv.reader(f)
    processed_docs = list(reader)
processed_docs = processed_docs[0::2] # get rid of empty lists

dictionary = Dictionary(processed_docs) # create a dictionary of words from our keywords

corpus = [dictionary.doc2bow(doc) for doc in processed_docs] #create corpus where the corpus is a bag of words for each docum

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

import gensim
from gensim.similarities import Similarity
from gensim.similarities import MatrixSimilarity

# Create the similarity data structure. This is the most important part where we get the similarities between the movies.
sims = MatrixSimilarity(tfidf[corpus], num_features=len(dictionary))

def movie_recommendation(movie_title, number_of_hits=5):
    movie = movies.loc[movies.primaryTitle==movie_title] # get the movie row
    keywords = movie['keywords'].iloc[0].split(',') #get the keywords as a Series (movie['keywords']),
    # get just the keywords string ([0]), and then convert to a list of keywords (.split(',') )
    query_doc = keywords #set the query_doc to the list of keywords

    query_doc_bow = dictionary.doc2bow(query_doc) # get a bag of words from the query_doc
    query_doc_tfidf = tfidf[query_doc_bow] #convert the regular bag of words model to a tf-idf model where we have tuples
    # of the movie ID and it's tf-idf value for the movie

    similarity_array = sims[query_doc_tfidf] # get the array of similarity values between our movie and every other movie.
    #So the length is the number of movies we have. To do this, we pass our list of tf-idf tuples to sims.

    similarity_series = pd.Series(similarity_array.tolist(), index=movies.primaryTitle.values) #Convert to a Series
    top_hits = similarity_series.sort_values(ascending=False)[1:number_of_hits+1]
    #get the top matching results, i.e. most similar movies; start from index 1 because every movie is most similar to itself

    #print the words with the highest tf-idf values for the provided movie:
    sorted_tfidf_weights = sorted(tfidf[corpus[movie.index.values.tolist()[0]]], key=lambda w: w[1], reverse=True)
    print('The top 5 words associated with this movie by tf-idf are: ')
    for term_id, weight in sorted_tfidf_weights[:5]:
        print(" '%s' with a tf-idf score of %.3f" %(dictionary.get(term_id), weight))

    # Print the top matching movies
    print("Our top %s most similar movies for movie %s are:" %(number_of_hits, movie_title))
    for idx, (movie,score) in enumerate(zip(top_hits.index, top_hits)):
        print("%d %s with a similarity score of %.3f" %(idx+1, movie, score))
```

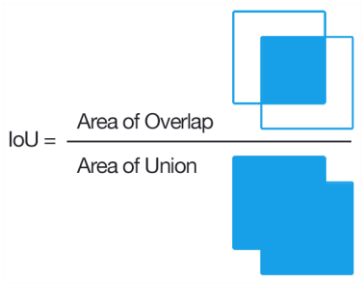
2nd Model: Jaccard Similarity

My 2nd model works on Jaccard Similarity between the keywords for any 2 movies. Jaccard similarity is defined as the intersection of 2 sets divided by the union of those sets. So this model finds the similarity between 2 movies by dividing the number of common keywords between 2 movies by the number of unique keywords in the union of the movies' keywords. Mathematically, this is given as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}.$$

Figure 2: Jaccard Similarity (image taken from Wikipedia)

And pictorially:



So for a given movie, I can calculate Jaccard Similarity with every other movie in my dataset and return the most similar movies, as defined by Jaccard Similarity. Note that there is no need for document vectors here as there was for finding cosine similarity.

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

```
def get_jaccard_sim(str1, str2):
    a = set(str1.split(','))
    b = set(str2.split(','))
    c = a.intersection(b)
    return(float(len(c)) / (len(a) + len(b) - len(c)))
```

Creating a recommender model from this is correspondingly simple. I simply found the keyword list for the given inputted movie, computed the Jaccard similarity between that list and every other movie keyword list and then ranked the movies by their similarities and returned 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, movie))
    jaccards = pd.Series(jaccards)
    jaccards_index = jaccards.nlargest(number_of_hits+1).index
    matches = movies.loc[jaccards_index]
    for match,score in zip(matches['primaryTitle'][1:],jaccards[jaccards_index][1:]) :
        print(match,score )
```

3rd Model: Cosine Similarity Between Word Counts

My last model is based on cosine similarity. It is like the first model then, in that both use cosine similarity but it is different in that it just uses word counts not *tf-idf*. This alternate implementation is performed with the CountVectorizer class in scikit-learn which converts a collection of text documents (i.e. keyword lists in this case) into a matrix of token counts.

Computing cosine similarity between any 2 word vectors is achieved easily using the cosine_similarity function from the scikit-learn metrics.pairwise submodule.

To use it effectively, I first needed to compute a matrix containing word counts for every keywords list in my dataset using CountVectorizer. The word counts for each movie are effectively word vectors for every movie. This was achieved with the following code:

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

def get_vectors(text):
    vectorizer = CountVectorizer(text)
    X = vectorizer.fit_transform(text).toarray()
    return(X)
```

```
vectors = get_vectors(movies.keywords.tolist())
```

Then, when given a movie, all I 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 was achieved with the my cosine_recommender function:

```
def cosine_recommender(movie_title, number_of_hits=5):
    movie_index = movies[movies.primaryTitle == movie_title].index.values[0]

    cosines = []
    for i in range(len(vectors)):
        vector_list = [vectors[movie_index], vectors[i]]
        cosines.append(cosine_similarity(vector_list)[0,1])

    cosines = pd.Series(cosines)
    index = cosines.nlargest(number_of_hits+1).index

    matches = movies.loc[index]
    for match,score in zip(matches['primaryTitle'][1:],cosines[index][1:]):
        print(match,score )
```

Discussion and Next Steps

Here are the results of calling my 3 recommenders on the same movie, Marvel Studios' 2012 "The Avengers". The results show the recommended movies and their similarities.

```
1 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
```

```
1 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
```

```
1 movie_recommendation('The Avengers')
```

```
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
```

We see that all 3 recommenders return similar movies and that these movies seem to be good matches. Anecdotally, when I use the recommenders with other movies that I am familiar with, the matches are good fits as well. But therein lies the first and most obvious issues with the models as they currently exist: they have no ready, obvious evaluation metrics. This was something I was aware of prior to working on this project since recommendation problems are neither supervised nor unsupervised problems but rather information retrieval problems. And as a result, performance metrics are harder to come by. So the next step for me to take would be to come up with an evaluation metric.

The 2nd step I would take is deploying the recommender using Flask. This could also help with the 'lack-of-evaluation-metric' problem as getting feedback from users about whether the recommendations were good or bad would provide an easy way for me to score my models.

The 3rd improvement I would make is incorporating other features into the model, namely numeric features (year, rating, number of reviews, and runtime of the movies) and the movie genre. In doing so, I could either create a separate recommender system that uses those features and then combines its recommendation with my current systems to produce a hybrid recommendation or I could combine my numeric features with my text ones to form new features.

A 4th step I would take is better EDA on my text data. This would include clustering the movies to better understand them and create a basic classifier where for a given movie I would return as recommendations the top 10 movies by rating from the given movies' cluster. This basic model would serve as a benchmark by which to measure other models. Clustering movies based on text alone would

have the additional benefit of converting text to a numerical feature: a single cluster number. It would then be easy to use text alongside the other numeric features I mentioned since it itself would be numeric.

A final improvement I would make is incorporating additional similarity metrics into my models and creating hybrid recommenders.

So to recap, my 5 next steps are:

1. Come up with good evaluation metrics
2. Deploy recommenders using Flask
3. Incorporate numeric and genre features from IMDb
4. Better EDA and perform clustering
5. Use additional measures for similarity and create hybrid models

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.

For inspiration for the use of Jaccard and cosine similarity recommenders, I am similarly indebted to Sanket Gupta for his tutorial Overview of Text Similarity Metrics in Python.