Gideon Blinick

Capstone Project 2 Milestone Report 2: Movie Recommender/Clusterer

Problem statement

This project seeks to solve a subproblem of the recommendation problem. The domain is movies and the recommender should work based not on collaborative filtering but on content-based filtering. More specifically, this project seeks to solve the specific problem of recommending the most similar movie to a given, provided movie.

Different measures of similarity will be tried to determine what 'most similar' means, and ultimately one will be chosen that works best.

The recommender will find the most similar movie based on a variety of features, most notably plot text information (plot synopses, keywords, outlines) and additional non-text information about the movies including numerical data (year of release, runtime, ratings, and number of votes) and categorical data (genres).

Client and Use

Our client is anyone who enjoys watching movies and wants to watch movies they enjoy. The ideal use for this engine is for a client to think of a movie they have enjoyed in the past and to enter that movie in our engine. The engine will return the movie most similar to their given movie. If the client hasn't seen the movie, they now have a movie to watch that they will hopefully enjoy. If they have seen the movie, the engine will also be able to output the next most similar movies after the top movie, and the user can browse that list until they find a movie that they haven't seen which they can then enjoy.

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:
https://github.com/gblinick/Movie-Recommender-Clusterer/blob/master/Dataset%20Creation.ipynb

Our dataset was custom created from the IMDb website using 2 different sources. The first source was IMDb's public dataset's repository: https://www.imdb.com/interfaces/

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 were used, including the title_basics, title_crew, and title_ratings tables.

The title_basics table contained the most useful information we 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 we wanted to include in our dataset. The title_crew table was initially included for the information it had about directors and writers. In theory, adding these features to our dataset would improve the accuracy of our model. However, after some consideration, it was decided remove these features out because they would dramatically increase the dimensionality of our feature space and would likely lead to poorer recommendation performance. That is, since there are many, many directors and writers out there, we

would have needed many, many columns in our dataset because one-hot-encoding is the appropriate way for handling such categorical data.

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

To get the plot information we needed, we used the wonderful IMDbPY API: https://imdbpy.sourceforge.io/

IMDbPY made pulling plot data from IMDb very easy. All that we needed to do was call the API 10,000 times to get whatever plot information we wanted. We called it 10,000 times because we wanted our 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 our engine more relevant we only called the API for the 10,000 most popular movies, where popularity was determined by the number of votes a movie received. This information was available from the title_ratings table.

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

After making the API calls, the data was stored in CSV filed so the API didn't need to be used again. Below is the code used for retrieving the plot outlines information and saving to a CSV. The same code works, 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 we had finished all the API calls, we were able to join all our information together into one Dataframe using the pandas' join function. This was possible since each of the constituent dataframes had the unique movie ID ('tconst') in them and we set that column as the index in order to join them.

Finally, we removed unnecessary columns including titleType (we were only interested in movies so we only retrieved movies and thus this column provided no information), originalTitle (we didn't care if it was the same or different from its current title), isAdult (none of the movies in our set of 10,000 most

popular movies was ‘Adult’), and endYear (the values for this column ended in ‘\N’ for all movies, so it was unhelpful).

Exploratory Data Analysis (EDA)

EDA notebook available here:
https://github.com/gblinick/Movie-Recommender-Clusterer/blob/master/EDA.ipynb

In our EDA, we examined the numeric features (startYear, runtimeMinutes, averageRating, and numVotes) and categorical feature (genres).

To begin, we examined the numeric features, first individually, then in pairs. We use the pandas’ describe to get some summary statistics of the features. The results are below:

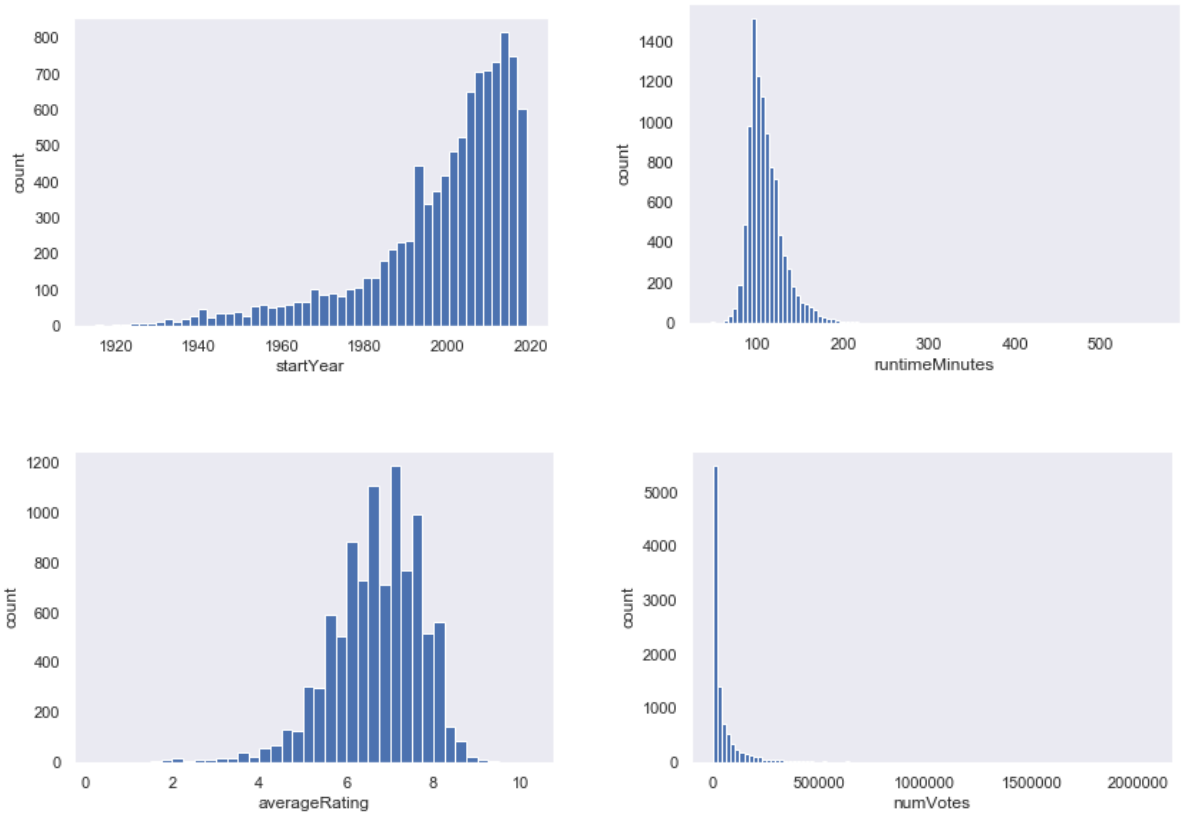
	startYear	runtimeMinutes	averageRating	numVotes
count	10000.00000	10000.000000	10000.000000	1.000000e+04
mean	1998.73320	108.832700	6.655770	6.658774e+04
std	18.01907	22.337511	1.043811	1.277254e+05
min	1915.00000	45.000000	1.300000	6.554000e+03
25%	1992.00000	94.000000	6.100000	1.103175e+04
50%	2004.00000	105.000000	6.800000	2.238150e+04
75%	2012.00000	118.000000	7.400000	6.455650e+04
max	2019.00000	566.000000	9.700000	2.057323e+06

Looking at the table, we made a couple of quick points. First, most movies in our dataset are from 2000 and on. This is seen from the fact that the median year is 2004 in the table. This result makes sense when considering that more movies are released each year and more recent movies are more likely to be voted on given that more people today watch movies than ever before (and in general this has increased with time).

Second, the average rating is about 6.7 stars and is close to the median of 6.8 stars.

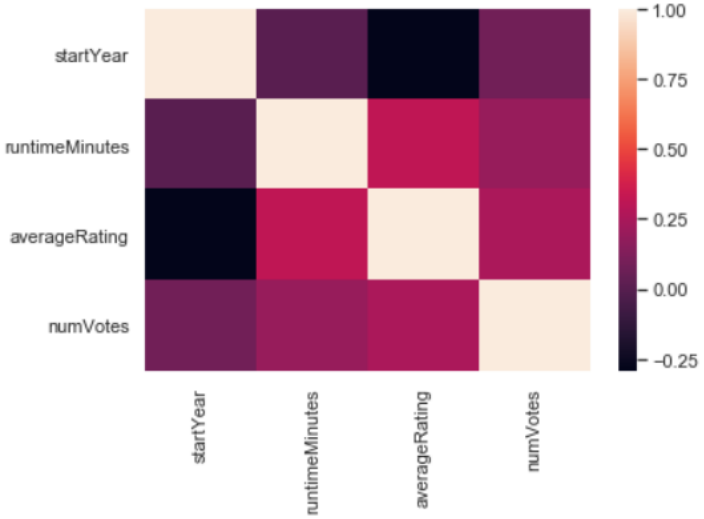
Third, we see that the mean number of votes for a movie is about 66.5 thousand. The standard deviation is almost double that though indicating most movies are below the mean, and a few are **way** above it. This can also be seen from the fact that the 75th percentile is smaller than the mean.

Next, we plot the distribution of each feature to get a more intuitive feel for them.



This histograms of the distributions of our numerical features confirm our earlier observations about those features. We move on to examining correlations between the features.

The following plot displays a heatmap of our variables:

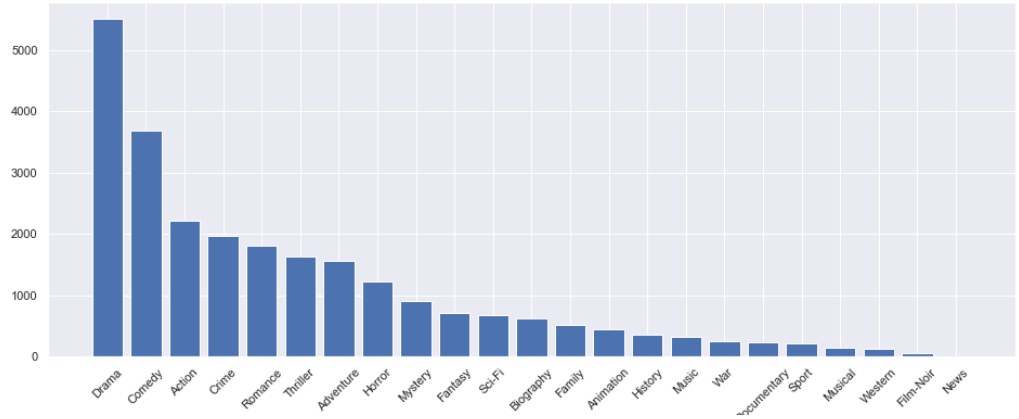


The corresponding correlations are:

Feature 1	Feature 2	
runtimeMinutes	averageRating	0.314285
averageRating	numVotes	0.254650
runtimeMinutes	numVotes	0.196031
startYear	numVotes	0.083236
	runtimeMinutes	0.007503
	averageRating	-0.292108

We see that longer movies tend to get better ratings, better rated movies are voted on more, and that more recent movies are likely to get lower ratings, among other things. These trends are explored in greater depth in the above linked notebook where we create scatter plots of the features and plot the line of best fit through each scatter plot.

We conclude by analyzing the genres column. This column contains genre tags or labels for a movie. Each movie can be assigned 1 to 3 genre tags. After preprocessing the column by one-hot-encoding the genres, we get the following breakdown of genres applied to movies:



So we see that Drama is the most common genre, then Comedy, then Action, and so on. It's important to note that the above chart counts the number of times each label was applied, and since most movies had more than one label applied, the number of genre labels will exceed the number of movies (10,000). Indeed, 6189 movies in our dataset had 3 labels applied, 2765 had 2 applied, and only 1046 were 1-label movies.

Finally, we looked at which genres are most correlated and anti-correlated with each other. That is, which genres are most likely to appear together, and which are most likely to not appear together.

For positive correlation, we obtained the following table:

Genre 1	Genre 2	Pearson Correlation Coefficient
Adventure	Animation	0.318
Action	Adventure	0.274
Biography	History	0.217

We see that Adventure and Animation, Action and Adventure, and Biography and History are genres that are fairly correlated with each other. These facts are unsurprising. We are using a benchmark of 0.2 to define what is significantly correlated.

On the negative correlation side, we obtained the following:

Genre 1	Genre 2	
Drama	Comedy	-0.245
	Action	-0.246
	Adventure	-0.259
	Horror	-0.229
Comedy	Thriller	-0.297
Action	Romance	-0.208

We see that Drama is negatively correlated with a bunch of other genres (Comedy, Action, Adventure and Horror). It seems like the Drama genre likes to 'hog' the movies it is tagged to, that is, it doesn't like to share tags with other movies. We also see that Comedy and Thriller and Action and Romance are negatively correlated, which is unsurprising since those genres can be described as opposites.

For text, we kept the analysis short.

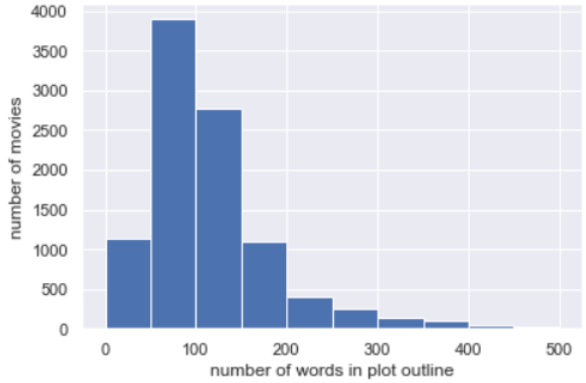
First, we looked at the average length for each of the text columns: plot, plot outline, keywords, and synopsis:

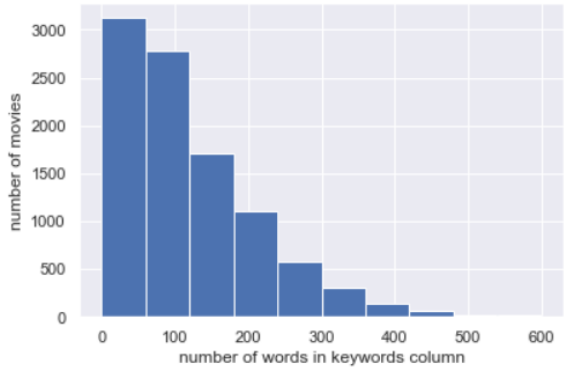
```
for col in text_cols.columns:
    print("%s: %.3f" %(col,np.mean(text_cols[col].str.len())) )
```

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

We found that synopsis is by far the longest on average, followed by keywords, plot outline, and plot. This matched our expectations since in creating the dataset we noticed that for each movie, synopsis contains paragraphs describing the movie scene by scene, keywords contains may 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.

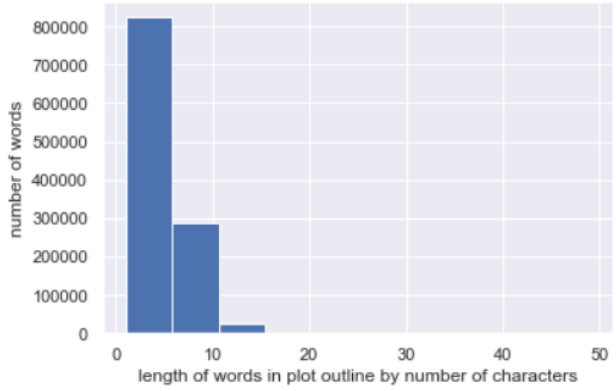
Next, we looked at the distributions of the number of words for both plot outlines and keywords. We chose to focus on these 2 text features because we decided that they were best for modelling purposes. Keywords is a great text column for modelling purposes because it contains words that are expressly used to categorize and classify a movie. Plot outline is a nice column to use because it is not too long or too short.

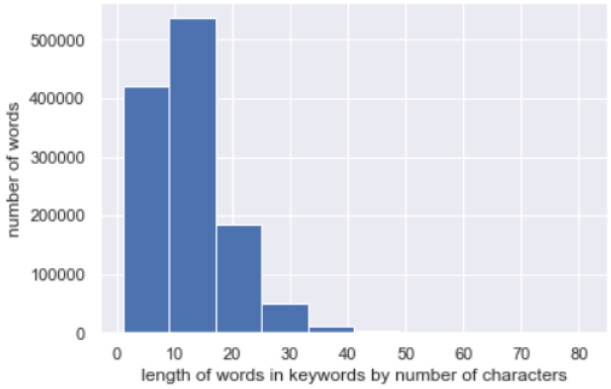




When looking at the summary statistics for these distributions, we found that the mean number of words for keywords is only slightly greater than the number of words for plot outline (~122 vs. ~115). Indeed the distribution for keywords (the right one) doesn't make it seem like there are more keywords per movie than words in the plot outline. This is misleading however, since most keywords tend to be 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 in plot outlines (because for one, there are no stop words among the keywords).

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





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

Finally, we found the top 5 words across plot outlines by 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, we found the top words to be "murder, death, blood, husband-wife-relationship, and violence". So we 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

At this stage in the project, we have created 3 different movie recommenders. The recommenders so far work only based 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 synopses 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.

It is our plan to extend the model to also use the numeric and categorical features available in our dataset, and that should be achieved by the end of the project.

The first model is a Tf-idf model that works with Gensim Similarity. There was effectively no preprocessing to do on the text before feeding it into the model for the following reasons:

1) Tokenization: Normally, we need to take a block of text and create tokens out of it. Since our keywords column contains comma separated words or groups of words, it was effectively tokenized for us and all that we needed to do was split on the commas to get our text.
2) Lowercasing: All the keywords were already lowercased.
3) Taking Only Alpha Words: Ordinarily, we would want to use the .isalpha() string method to avoid taking punctuation and the like. In our case, there was no punctuation to avoid and using .isalpha() would have additionally caused us to lose most of our words since most of the keywords were actually 2 or 3 words grouped together through dashes ("-") .
4) Removing stop words: None of the keywords were, logically enough, stop words.
5) Lemmatizing: This is something that we might have done that could have made a lot of sense. Still, it was 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.

So there was effectively no preprocessing to do.

To create the first model, the keywords were fed into a Gensim Dictionary to create a dictionary with words as keys and word ids as values. The dictionary was used to create a corpus by creating a bag-of-words for every list of movie keywords by using the Gensim .doc2bow() method. The corpus was in turn used to create a tf-idf corpus model. Tf-idf stands for "term frequency–inverse document frequency" and it measures the importance of a word to a document. It does this by assigning a weight to each word in a document based on how often a word appears in the document and how often it appears in other documents. If a word appears often in a document, its weight will go up. Conversely, if a word appears in many other documents, the weight of the word for the document we are examining will go down.

After creating the tf-idf corpus model, we created a structure to measure the similarity between every document in our corpus and a given set of words. This was achieved using the MatrixSimilarity() function from the genism.similarities submodule which uses cosine similarity. By feeding it our tf-idf corpus, finding the most similar movies to a given movie becomes a simple task. All we need to do when given a movie is retrieve its keywords, convert those keywords into a bag-of-words and then into a tf-idf representation of those words, and then find the most similar movies to that representation using MatrixSimilarity(). Note that the model can be used to find the most similar movies for any given set of keywords (that are in the corpus). Its most natural use is a specific case where the given set of keywords we use is the set of keywords used to describe a particular movie we like.

The code that implements the above logic 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))
```

We also created 2 other movie recommender models.

The first of these works based off of the Jaccard Similarity between the keywords for any 2 movies. Jaccard similarity is defined as the intersection of 2 sets divided by the union. 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. So for a given movie, we can

calculate Jaccard Similarity with every other movie in our dataset and return the most similar movies, as defined by Jaccard Similarity.

The code for computing Jaccard Similarity is simple to implement:

```
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:

```
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 )
```

Finally, our 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 term frequency (or tf) and not tf-idf. This alternate implementation is performed with the CountVectorizer class in scikit-learn.

Computing cosine similarity between any 2 word vectors is achieved with the following functions:

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

def get_cosine_sim(*strs):
    vectors = [t for t in get_vectors1(*strs)]
    return(cosine_similarity(vectors))

def get_vectors1(*strs):
    text = [t for t in strs]
    vectorizer = CountVectorizer(text)
    vectorizer.fit(text)
    return(vectorizer.transform(text).toarray())
```

To use these effectively, we first compute a matrix containing word counts for every keywords list in our dataset using CountVectorizer(). The word counts for each movie are effectively word vectors for every movie. 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 the following code.

```
15 def get_vectors2(text):
16     vectorizer = CountVectorizer(text)
17     X = vectorizer.fit_transform(text)
18     return(X.toarray())
```

```
1 vectors = get_vectors2(movies.keywords.tolist())
```

```
1  def cosine_recommender(movie_title, number_of_hits=5):
2      movie_index = movies[movies.primaryTitle == movie_title].index.values[0]
3
4      cosines = []
5      for i in range(len(vectors)):
6          vector_list = [vectors[movie_index], vectors[i]]
7          cosines.append(cosine_similarity(vector_list)[0,1])
8
9      cosines = pd.Series(cosines)
10     index = cosines.nlargest(number_of_hits+1).index
11
12     matches = movies.loc[index]
13     for match,score in zip(matches['primaryTitle'][1:],cosines[index][1:]):
14         print(match,score )
```

We conclude by showing the matches returned by calling our 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 are good matches. Further work will be done exploring the strengths of each model.