Movie Recommendation System

by Pratiek Sonare

1. Overview

What is a recommendation system?

An algorithm that helps a platform's users find content / information pertaining to their relevance. Basically, it is the same algorithm that helps us discover new movies as per our choice, display targeted advertisements according to our online behaviour. Recommendation Systems are a powerful tools for any platforms - since they help in user engagement and retention. A platform that helps recommend their content to the users finds a greater user engagement than others.

Different types of recommendation systems

There exists different types of recommendation systems. The one that differentiates elements based on their content is called a content-based recommendation system. The recommendation system that relies on user's history is called a collaborative-based. A combination of the two is called a hybrid approach, which is what some companies like Netflix, YouTube have tried using for their own respective platforms.

More complex recommendation systems also exist which combine and improve upon the techniques mentioned above with deep learning (RNNs) - for example: Session-based RS

```
flowchart TD
    id1(Recommendation Systems) --> id2(Content-based)
    id1(Recommendation Systems) --> id3(Collaborative)
    id2 & id3 <-..-> id4(Hybrid)
```

2. Project Objectives

This project implements a content-based movie recommendation system using **cosine similarity** and **vectorization** techniques. It is designed to recommend movies to users based on their preferences or the characteristics of a selected movie.

- Develop a system to recommend movies based on textual metadata such as genre, director, cast, etc. available in the *tmdb-movie-metadata/tmdb_5000_movies* dataset.
- The following vectorization techniques have been leveraged:
    1. Count Vectorization
    2. TF-IDF Vectorization

We'll also look at -

Word2vec, an NLP technique that uses a machine learning algorithm to convert words into multi-dimensional vectors

- Sources:

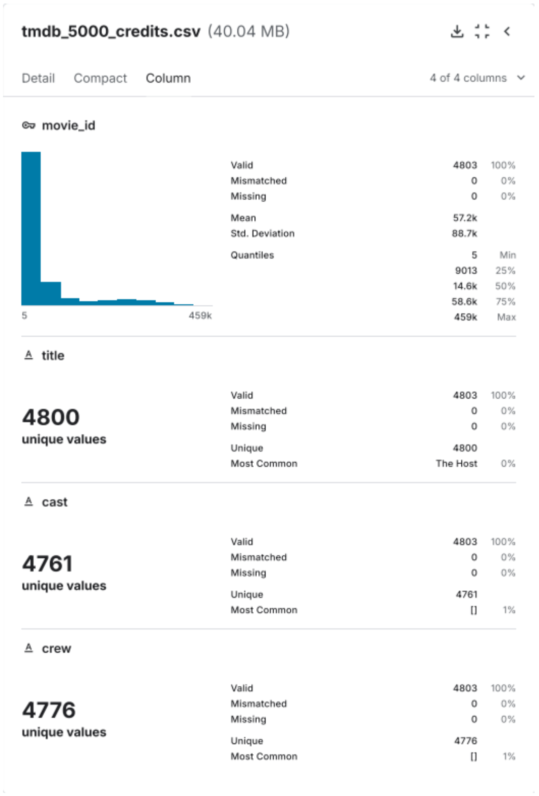
  1) https://www.geeksforgeeks.org/python-word-embedding-using-word2vec/

  2) https://arxiv.org/pdf/1301.3781

3. Data Source

The dataset used is the *tmdb-movie-metadata/tmdb_5000_movies*, with data:

> We only use the important features with data type - **string** for vectorization (data loss!)
>
> - movie_id - identification number according to the tmbd website
> - title - movie's name
> - cast - movie's cast
> - crew - list of entire crew involved in the movie



tmdb_5000_credits.csv (40.04 MB)

movie_id

Valid	4803	100%
Mismatched	0	0%
Missing	0	0%
Mean	57.2k	
Std. Deviation	88.7k	
Quantiles	5	Min
	9013	25%
	14.6k	50%
	58.6k	75%
	459k	Max

title

4800 unique values

Valid	4803	100%
Mismatched	0	0%
Missing	0	0%
Unique	4800	
Most Common	The Host	0%

cast

4761 unique values

Valid	4803	100%
Mismatched	0	0%
Missing	0	0%
Unique	4761	
Most Common	[]	1%

crew

4776 unique values

Valid	4803	100%
Mismatched	0	0%
Missing	0	0%
Unique	4776	
Most Common	[]	1%

4. Tools and Libraries

The following tools and Python libraries were used:

- **Python**: Programming language for data processing and implementation.
- **Pandas**: For data manipulation and preprocessing.
- **Numpy**: For numerical computations.
- **Scikit-learn**: For implementing count vectorization and cosine similarity.

5. Methodology

1. **Data Preprocessing**:
    - Filtering columns of importance that are of data type - **string**
    - Cleaning the dataset by handling missing values.
    - Converting each column into string-like format (without brackets [ ]).

```
def convert(text): #to filter movie "name" from genres
    L = []
    for i in ast.literal_eval(text):
        L.append(i['name'])
    return L

def convert3(text): #to filter lead cast (top 3) from
    L = []
    counter = 0
    for i in ast.literal_eval(text):
        if counter < 3:
            L.append(i['name'])
        counter+=1
    return L

def fetch_director(text): #to filter movie's director'
    L = []
    for i in ast.literal_eval(text):
        if i['job'] == 'Director':
            L.append(i['name'])
    return L



movies['genres'] = movies['genres'].apply(convert)
movies['keywords'] = movies['keywords'].apply(convert)
movies['cast'] = movies['cast'].apply(convert3)
movies['crew'] = movies['crew'].apply(fetch_director)
```

- Eliminating any white-space between two subsequent words to maintain format
- Or else individual words of a proper noun would be counted as a separate word. For ex: “The Rise of Batman” should not be considered as four different words - “The”, “Rise”, “of”, “Batman”

```
def collapse(L):
    L1 = []
    for i in L:
        L1.append(i.replace(" ", ""))
    return L1

movies['cast'] = movies['cast'].apply(collapse)
movies['genres'] = movies['genres'].apply(collapse)
movies['keywords'] = movies['keywords'].apply(collapse
movies['crew'] = movies['crew'].apply(collapse)
```

- Combining all filtered features into a single text-based feature called **tags**.

```
movies['overview'] = movies['overview'].apply(lambda x
movies['tags'] = movies['overview'] + movies['genres']

db = movies.drop(columns=['overview', 'genres', 'keywo
db['tags'] = db['tags'].apply(lambda x: " ".join(x))
db.head()
```

2. **Feature Extraction**:
    - Utilizing **count vectorization/ TF-DIF vectorization** to convert text data into numerical vectors.

```
from sklearn.feature_extraction.text import CountVecto

cv = CountVectorizer(max_features=5000, stop_words='en
cv_matrix = cv.fit_transform(db['tags']).toarray()

######   OR   #######

from sklearn.feature_extraction.text import TfidfVecto
```

```
tfidf_vectorizer = TfidfVectorizer(max_features=5000, 
tfidf_matrix = tfidf_vectorizer.fit_transform(movies['
```

3. **Similarity Calculation**:
    - Applying **cosine similarity** to determine the similarity between movies based on their feature vectors.

```
from sklearn.metrics.pairwise import cosine_similarity
similarity = cosine_similarity(tfidf_matrix)

######   OR   #######

from sklearn.metrics.pairwise import cosine_similarity
similarity = cosine_similarity(cv_matrix)
```

4. **Recommendation**:
    - Building a function that takes a movie title as input and recommends the top-N most similar movies.

```
def recommend(movie):
    index = db[db['title'] == movie].index[0]
    distances = sorted(list(enumerate(similarity[index
    for i in distances[1:6]:
        print(db.iloc[i[0]].title)
```

Inference

▼ (click to know more)

Input

```
recommend('Avatar')
```

Output

```


# Count Vectorization
Titan A.E.
Small Soldiers
Ender's Game
Aliens vs Predator: Requiem
Independence Day

#TF-IDF Vectorization
Falcon Rising
Battle: Los Angeles
Apollo 18
Star Trek Into Darkness
Titan A.E.
```

Hence, we find some difference in the movies recommended by both these techniques. Let’s try again for a different movie.

Input

```
recommend('Gandhi')
```

Output

```
# Count Vectorization
Gandhi, My Father
The Wind That Shakes the Barley
A Passage to India
Guiana 1838
Ramanujan

#TF-IDF Vectorization
Gandhi, My Father
The Wind That Shakes the Barley
A Passage to India
Catch a Fire
Wah-Wah
```

Here, while the top 3 recommendations are the same, the rest are different. **Why?**

Let us find out how Vectorization and Cosine Similarity Techniques work!

▼ (click to find out)

CountVectorization

- A simple technique where the number of unique words are counted and represented as a sparse matrix.

- Each unique word is given an index number and the sparse matrix produced is the **embedding** of the document.
- Each column in this matrix, thus, represents a word - with the numerical values representing a specific vector.
- CountVectorization represents words as vectors based on how often they occur in each sentences. Words that have similar frequency patterns across the documents will have more similar vectors, regardless of their actual meanings or relationships.

Example:

- Document 1: "Cats love milk."
- Document 2: "Dogs love bones."

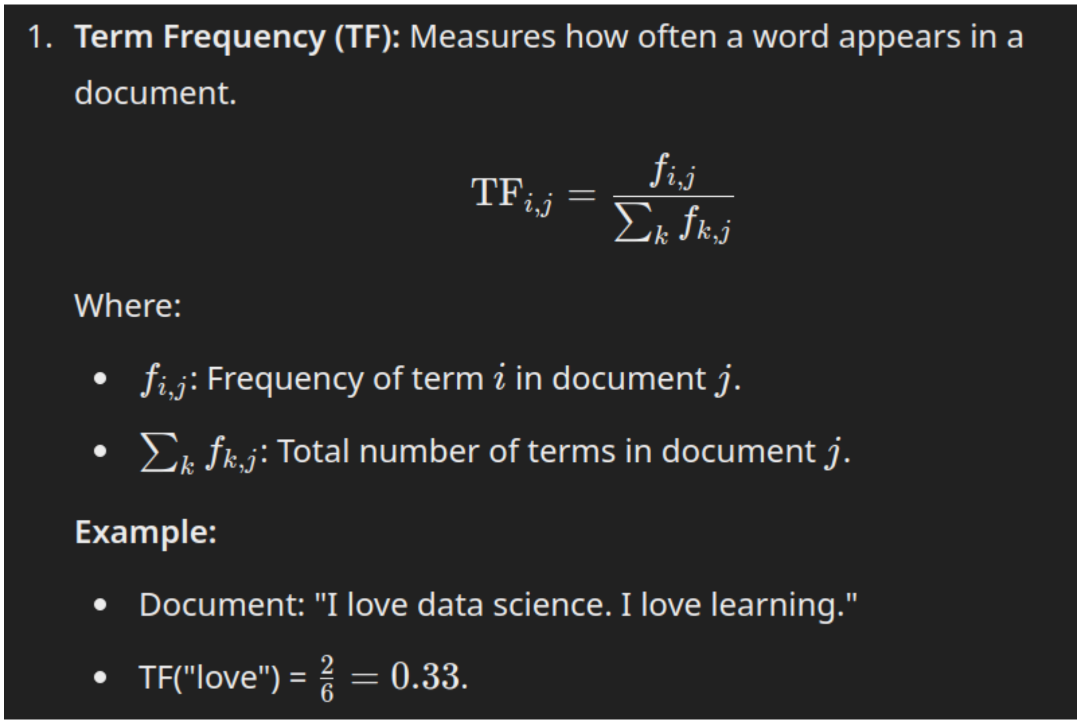
If we CountVectorize:

Word	Document 1 Count	Document 2 Count
cats	1	0
love	1	1
milk	1	0
dogs	0	1
bones	0	1

The word "love" has the same count in both documents, but its context differs completely. "Cats" and "milk" occur in the same document, but CountVectorization doesn’t indicate any relationship between them either.

TF-IDF Vectorization

Term Frequency - Inverse Document Frequency is a statistical technique used to represent the importance of a word in a document relative to its occurrence in the entire corpus. It helps highlight words that are **important to a specific document** while downweighting words that are too common across the entire dataset.



1. **Term Frequency (TF):** Measures how often a word appears in a document.

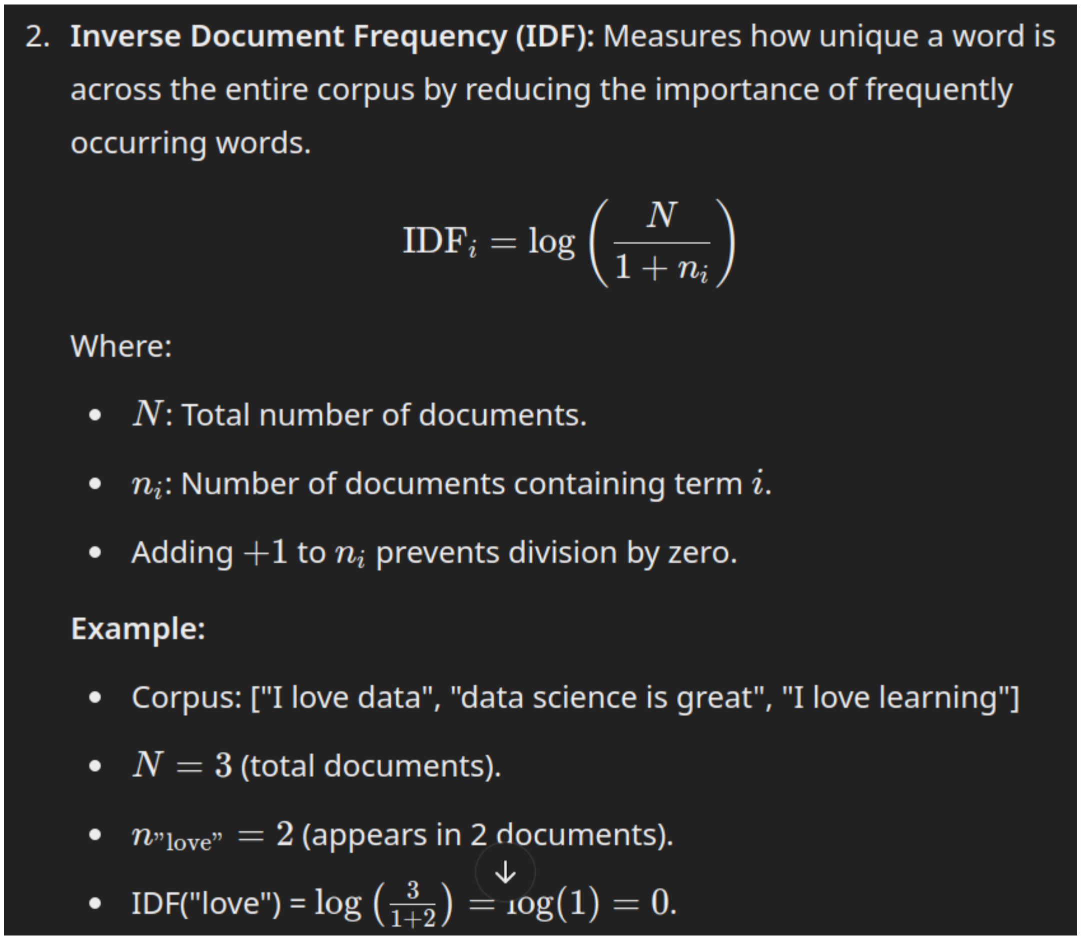
$$\mathrm{TF}_{i,j} = \frac{f_{i,j}}{\sum_k f_{k,j}}$$

Where:

- $f_{i,j}$: Frequency of term i in document j.
- $\sum_k f_{k,j}$: Total number of terms in document j.

Example:

- Document: "I love data science. I love learning."
- TF("love") = $\frac{2}{6} = 0.33.$



2. **Inverse Document Frequency (IDF):** Measures how unique a word is across the entire corpus by reducing the importance of frequently occurring words.

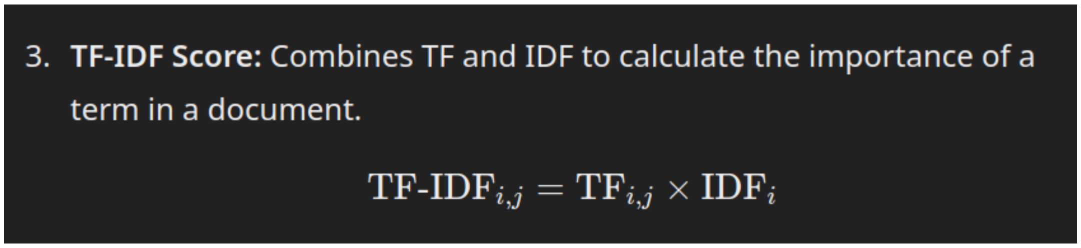
$$\text{IDF}_i = \log\left(\frac{N}{1+n_i}\right)$$

Where:

- N: Total number of documents.
- n_i: Number of documents containing term i.
- Adding $+1$ to n_i prevents division by zero.

Example:

- Corpus: ["I love data", "data science is great", "I love learning"]
- $N = 3$ (total documents).
- $n_{\text{"love"}} = 2$ (appears in 2 documents).
- IDF("love") = $\log\left(\frac{3}{1+2}\right) = \log(1) = 0$.



3. **TF-IDF Score:** Combines TF and IDF to calculate the importance of a term in a document.

$$\text{TF-IDF}_{i,j} = \text{TF}_{i,j} \times \text{IDF}_i$$

Example:

Corpus:

1. Document 1: "I love data science."
2. Document 2: "Data science is amazing."
3. Document 3: "I love learning data."

Vocabulary: {I, love, data, science, is, amazing, learning}

Step 1: Calculate Term Frequency (TF):

Term	TF (Doc 1)	TF (Doc 2)	TF (Doc 3)
I	1/4	0	1/4
love	1/4	0	1/4
data	1/4	1/4	1/4
science	1/4	1/4	0
is	0	1/4	0
amazing	0	1/4	0
learning	0	0	1/4

Step 2: Calculate Inverse Document Frequency (IDF):

Term	Document Frequency (nin_ini)	IDF (log(N/(1+ni))\log(N/(1+n_i))log(N/(1+ni)))
I	2	log(3/3)=0\log(3/3) = 0log(3/3)=0
love	2	log(3/3)=0\log(3/3) = 0log(3/3)=0
data	3	log(3/4)=−0.12\log(3/4) = -0.12log(3/4)=−0.12
science	2	log(3/3)=0\log(3/3) = 0log(3/3)=0
is	1	log(3/2)=0.18\log(3/2) = 0.18log(3/2)=0.18
amazing	1	log(3/2)=0.18\log(3/2) = 0.18log(3/2)=0.18
learning	1	log(3/2)=0.18\log(3/2) = 0.18log(3/2)=0.18

Step 3: Compute TF-IDF:

Term	TF-IDF (Doc 1)	TF-IDF (Doc 2)	TF-IDF (Doc 3)

I	0	0	0
love	0	0	0
data	-0.03	-0.03	-0.03
science	0	0	0
is	0	0.045	0
amazing	0	0.045	0
learning	0	0	0.045

- This way - unique and important words get higher weights and are highlighted. Repetitive words are penalised by reduced importance.
- This TF-IDF Score of each word is then used as a vector for vectorization of each word.

Cosine Similarity

Cosine is the angle between two vectors. The lower is the angle, more related / similar are the vectors (i.e movies)

$$\text{Cosine Similarity} = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \|\vec{B}\|}$$

8. Drawbacks

- We consider only string data type features. There is a huge data loss since, we do not consider numerical data (here) for recommending movies. One can utilize deep learning systems to find similarity between two movies based on alpha and numerical features.
- No semantic relationship evaluated here. Semantic relationship / meanings of the words with each other aren't evaluated here leading to somewhat accurate but un-intelligent recommendations. We only take frequency similarity of words into consideration for differentiating between movies.