Content

Recommendation Systems

What are Recommendation Systems?

Recommendation systems date back to the early 90's where they were first developed by researchers to suggest important documents to the readers based on the review ratings provided by other users.

It became mainstream after tech giants like Google, Amazon, Netflix, Spotify and others integrated it with their systems to recommend relevant content and products to the users. Especially as personalized recommendations gained traction so did recommendation system algorithms.

These systems are behind suggesting movies, ads, youtube videos, music to the users based on their interests or we can say based on their predicted interests.

Easier said than done, it's not easy for a computer to figure out what hooks a user to a particular video or movie - is it the lead actor, story line, timepass, curiosity etc etc.

Using maths as a tool, researchers have been able to predict user's interest based on explicit reviews or by collecting information from similar users or using metadata like watch time, clicks etc (implicit review).

In this case study, we analyze two of the most popular approaches used in recommendation systems.

1. Content-Based Filtering
2. Collaborative Filtering

But, before we get into the specifics of these approaches,let's analyze all the known options available to implement these systems.

Different Types of Recommendation Systems

For large size applications like youtube, netflix etc. the recommendation system cannot rely on one approach. As one might have noticed on the Netflix home page, there are different categories of suggestions - one based on view history, other based on popularity, another one based on similar interests etc. On the other hand on youtube we generally get more recommendations based on our view history. Both these systems use a hybrid method of recommendation systems plus neural networks to add additional context while predicting a new favorite.

For a far more simpler use case like a small business that sells a few products to a limited number of users, a simple collaborative filtering approach might work.

Just like any other technology in use today, depending on the specific use case, engineers will have to pick a suitable method.

A few different approaches that are available today are:

1. Content-Based Filtering: Recommend products that are similar to the one user liked/watched/purchased.
2. Collaborative Filtering: Recommend products that users with similar interest are watching / buying.
3. Hybrid Methods: Using both the approaches mentioned in #1 and #2.
4. Deep Learning-Based Methods: Neural Collaborative Filtering: Uses neural networks to model user-item interactions more complexly and accurately. eg.Autoencoders (This was first tried by Amazon and won them an award https://www.amazon.science/the-history-of-amazons-recommendation-algorithm)
5. Context-Aware Recommendations: Incorporates additional context (e.g., time, location, device) to enhance recommendations. For example, suggesting different types of restaurants based on the time of day.

Apart from these approaches there are also a few lesser known/discussed techniques like,

1. Rule-Based Filtering: Uses domain-specific rules and knowledge to make recommendations. For example, a travel recommendation system might suggest destinations based on the user's preferences and constraints.**This can also be integrated as an additional step on top of the filtering methods discussed earlier**
2. Association Rule Mining / Apriori Algorithm:Identifies frequent item sets and generates association rules to suggest items that are often bought together.

Content Based Filtering

Most basic and yet effective form of a recommendation system can be built using content based filtering. Base concept here is - suggest items that are similar to the items

user has previously engaged with. For example - If someone buys "The Alchemist" by Paulo Coelho, also suggest other books that falls under the same genre.

Implementing a content based filtering system involves two main steps - one is to analyze and make sense of the data and second is to find data that are similar to each other.

Making sense of the data is achieved using TF-IDF vectorization and similarity can be measured using cosine similarity, euclidean distance etc.

Let's take a look at each of these techniques one by one.

TF-IDF Vectorization

TF-IDF (Term Frequency-Inverse Document Frequency) calculates the frequency of a word in a given document and estimates its importance with respect to the given collection of documents.

Using this technique, we can create embeddings of a given text data.

However, its important to note that TF-IDF is a shallow, count-based method. It captures the importance of words in a document but doesn't capture the semantic relationships between words as effectively as more sophisticated methods like BERT, GloVe etc. So, this technique is useful for simpler models and tasks.

This statistics(TF-IDF) is calculated using below given formula - `TF-IDF(t,d,D)=TF(t,d)×IDF(t,D)`

1. **Term Frequency (TF):** TF(t,d)= Total number of terms in document d / Number of times term t appears in document d - (1)
2. **Inverse Document Frequency (IDF):** IDF(t,D)=log((Total number of documents in the corpus/Number of documents containin + 1) - (2)

TF-IDF score = product of (1) and (2)

Run the below code cell and verify the generated embeddings vector.

In [ ]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

# Example documents
documents = [
    "The cat sat on the mat.",
    "The dog sat on the log.",
    "The cat chased the mouse."
]

# Initialize the TfidfVectorizer
```

```
tfidf = TfidfVectorizer()

# Fit and transform the documents to get the TF-IDF matrix
tfidf_matrix = tfidf.fit_transform(documents)

# Convert the matrix to a pandas DataFrame for better readability
import pandas as pd
tfidf_df = pd.DataFrame(tfidf_matrix.toarray(), columns=tfidf.get_feature_na

print(tfidf_df)
```

```
        cat    chased       dog       log       mat     mouse        on  \
0  0.374207  0.000000  0.000000  0.000000  0.492038  0.000000  0.374207
1  0.000000  0.000000  0.468699  0.468699  0.000000  0.000000  0.356457
2  0.381519  0.501651  0.000000  0.000000  0.000000  0.501651  0.000000

        sat       the
0  0.374207  0.581211
1  0.356457  0.553642
2  0.000000  0.592567
```

Cosine Similarity

Once the embeddings are generated(TF-IDF Vectorization), we would need to figure out a way to measure the distance between any two embeddings in the vector space.

Let's see how this works.

Consider 2 sentences - inputs A and B,

1. A = Love you
2. B = Love

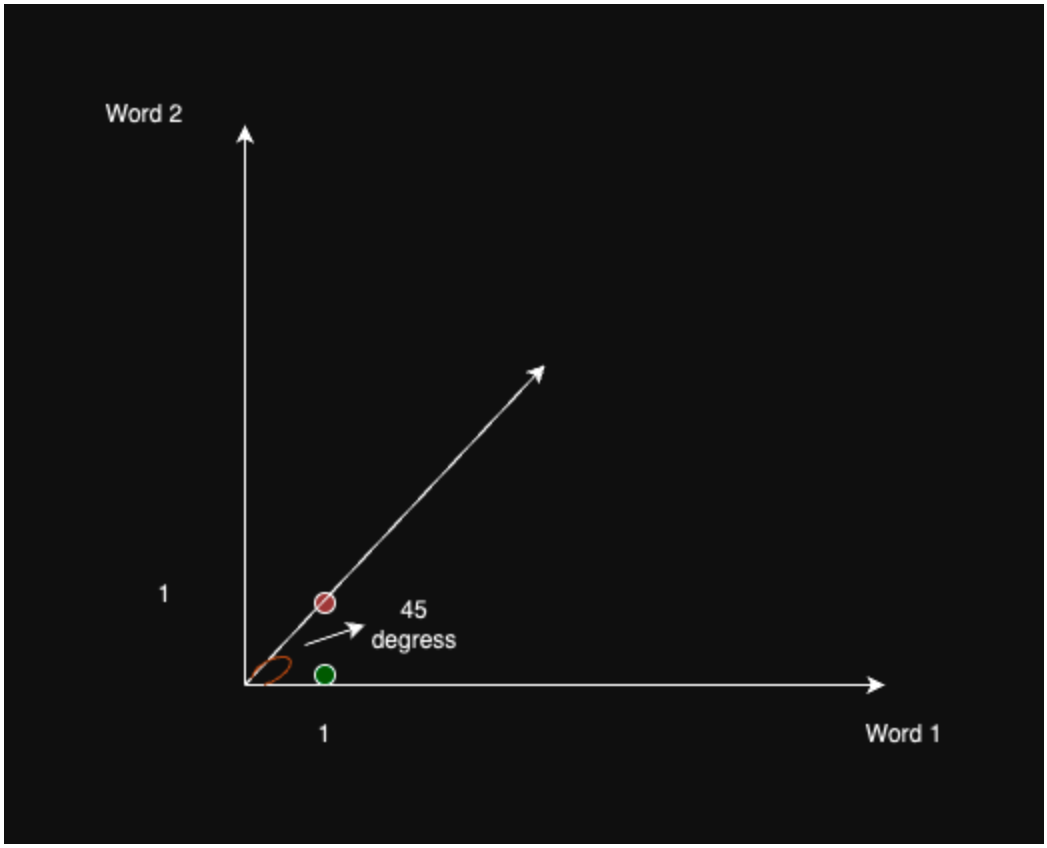
Now, we can represent these sentences in a matrix and translate to 0 and 1,

1. Collect all the words from the two sentences and each word becomes the column
2. Next the sentences themselves become the rows

Now, for each sentence,if the word mentioned in the column is present in the sentence, mark it as 1 else mark it 0. So, the above two examples can be added to a matrix as shown below,

```
   Love | You
A  1    1
B  1    0
```

Now, if we mark these values on a 2 dimensional graph (each dimension represents a word from the example, as our examples have maximum 2 words we represent the data in a 2 dimensonal graph), we would get a graph like this - red represents A and green represents B



As seen in the graph, we have converted sentences into embeddings and marked them in the graph. Our next goal is to figure out how similar these two marked embeddings are? We get that by finding the cosine of angle between the two points A and B. Value of cos(45 degree) is approximately 0.7071.

A similar approach is used to find the cosine similarity between two inputs(embeddings) in a high-dimensional space.

It's easy to find the angle on a graph(with a few dimensions) but in statistics we will use a formula to find this.

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

The above formula for cosine similarity can also be expressed in terms of a summation.

$$\frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Next, using the numerical values of A and B, let's try to find the cosine similarity between the two inputs using the cosine similarity formula - run the code cell below and compare the results with value of cos(45 degree) that we calculated earlier.

In [ ]:
```
import math
```

```
((1*1) + (1*0))/(math.sqrt(math.pow(1,2) + math.pow(1,2)) * math.sqrt(math.p
```

Out[ ]: 0.7071067811865475

Let's run the same logic using functions from numpy library.

numpy also provides a built-in function to effortlessly measure cosine similarity between two vectors but let's use that in the main example.

In [ ]:
```
import numpy as np

# Define two vectors
vector_a = np.array([1, 1])
vector_b = np.array([1, 0])

# Compute cosine similarity
np.dot(vector_a, vector_b) / (np.linalg.norm(vector_a) * np.linalg.norm(vect
```

Out[ ]: 0.7071067811865475

Now that we understand both vectorization and cosine similarity, let's try to implement content based filtering.

For demo purposes, we have created a small set of synthetic data that consists of movie id's and their corresponding genres. Firstly we would get the embeddings of genres, then using the same embeddings matrix we extract user_movies matrix by keeping the titles that user has already watched and then compare the two matrices to get the similarity score of each title.

To get the final output, we are only considering movies that have a similarity score of 1.0 but that's just for demo purposes, ideally anything closer to 1 should be good enough for recommendation.

In [ ]:
```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Example data
movies = pd.DataFrame({
            'title-id': ['61', '302', '31', '311'],
            'genres': ['Rom-Com', 'Action', 'Rom-Com', 'Action']
        })

#fetch movies watched by user from DB
user_movies = ['61']

tfidf = TfidfVectorizer(stop_words='english')
tfidf_matrix = tfidf.fit_transform(movies['genres'])
user_profile = tfidf_matrix[movies['title-id'].isin(user_movies)].mean(axis=

similarity_scores = cosine_similarity(np.asarray(user_profile), tfidf_matrix
```

```
movies['similarity'] = similarity_scores.flatten()
print(movies)
recommended_movies = movies[(~movies['title-id'].isin(user_movies)) & movies
print("Final Output")
print(recommended_movies[['title-id', 'similarity']])
```

```
[[0.         0.70710678 0.70710678]]
  title-id   genres  similarity
0       61  Rom-Com         1.0
1      302   Action         0.0
2       31  Rom-Com         1.0
3      311   Action         0.0
  title-id  similarity
2       31         1.0
```

Collaborative Filtering

As seen earlier, content based recommendations are efficient but have their limitations, for example, in a movie recommendation system it would not make sense to just recommend movies based on what the user has already watched. The system will have to suggest something different or surprise the user with something out of the routine every now and then. For these purposes, we need a technique other than content based recommendations.

This is where collaborative filtering becomes useful.

Two prominent types of collaborative filtering are:

1. User-User Collaborative Filtering: Finds similar users to the target user and recommends items that those users liked.
2. Item-Item Collaborative Filtering: Finds items similar users have bought and suggest them.

Another technique from statistics is used to make this type of filtering possible. Let's take a look at that before we get into the filtering itself.

Matrix Factorization:

The basic idea is to decompose a matrix into the product of two or more matrices, typically with lower dimensions.

What is Factorization?

Let's take a quadratic polynomial as an example,

$$(x^2 - 5x + 6).$$

The above given equation can be factored into the product of two linear factors as shown below

$$x^2 - 5x + 6 = (x - 2)(x - 3)$$

So, factorization is a way to extract the factors such that the factors can be multiplied together to obtain the original polynomial.

In recommendation systems, matrix factorization can be used to predict missing entries in a user-item interaction matrix by decomposing it into a user-feature matrix and an item-feature matrix. This approach is purely algebraic and does not involve neural networks.

For example, in the case of movie recommendation system, the data we initially work on to implement a collabotive filtering technique would be a review matrix. That is, user ids, movie ids and the ratings user gave to the movies.

It's easy to figure out that this matrix would be sparsely populated, that is, compared to the number of movies there are versus how many one would have watched and properly reviewed can only give us a rarely populated matrix. So, we use matrix factorization to extract values from the entries there are and then using those extracted factors predict the ratings of movies for which the initial data did not have entries.

Techniques like Singular Value Decomposition (SVD) and Alternating Least Squares (ALS) decompose the user-item interaction matrix to find latent factors that explain user preferences. In this case study, we will be using SVD.

Multiple Python based libraries are available to implement these techniques, the one we will be using here is Surprise library. Surprise is a Python scikit for building and analyzing recommender systems that deal with explicit rating data.

Now that we have established all the building blocks, let's implement a Recommendation system based on Collaborative filtering. We could generate our own synthetic data and import it to the program in many different ways but for the convenience of using readily available data, we will be using built-in movie rating dataset provided by Surprise library.

In [ ]:
```python
from surprise import SVD, Dataset

class RecommenderSystem:
    def __init__(self):
        self.model = None

    def train_model(self, data):
        # Define the SVD model
        self.model = SVD()

        # Train the model
        trainset = data.build_full_trainset()
        self.model.fit(trainset)
```

```
    def predict_rating(self, user_id, item_id):
        # Predict the rating for a given user and item
        prediction = self.model.predict(user_id, item_id)
        return prediction.est

# Load the dataset
data = Dataset.load_builtin('ml-100k')


# Instantiate the RecommenderSystem
recommender = RecommenderSystem()

# Train the model
recommender.train_model(data)

# Predict the rating for a user and an item
predicted_rating = recommender.predict_rating(user_id='1', item_id='61')
print(f"Predicted Rating: {predicted_rating}")
```

```
Predicted Rating: 4.351118802827308
```

In the above code,

```
predicted_rating = recommender.predict_rating(user_id='1',
item_id='61')
```

we pass userid, movieid that is available in the dataset and get the predictions.

Based on the business logic, this step can be either performed when a API call is made or can also be done by running a batch program in the background and store the predictions in database and cache the top recommendations for a quick access.

Summary

In summary, recommendation systems can be used to enhance customer engagement and when used effectively lays foundation for long-term customer loyalty.

This case study serves as an example of how data-driven decisions can lead to significant business impact. As you consider your own strategies, remember that with careful planning and execution one can create opportunities for growth and innovation.

In [ ]: