**Recommendation Systems**

**Abstraction:**

Recommendation systems play a crucial role in the field of video recommendation, and their scope continues to evolve as technology advances and user preferences change. Here are some key aspects of the scope of recommendation systems in video recommendation:

Personalization: Personalization is at the core of video recommendation systems. These systems use algorithms to analyze user behavior, such as what videos they watch, how long they watch them, and their interactions, to provide personalized video recommendations. The scope here is to refine and enhance personalization algorithms to better understand and cater to individual tastes and preferences.

Content Discovery: Video recommendation systems help users discover new and relevant content they might not have found otherwise. Expanding the scope means improving content discovery by considering factors like genre, language, release date, and user demographics.

Diverse Recommendations: Beyond simply suggesting videos similar to what a user has already watched, there is a growing emphasis on providing diverse recommendations. The scope includes ensuring that recommendations expose users to a variety of content, preventing filter bubbles, and promoting serendipitous discoveries.

User Engagement: Beyond just click-through rates, the scope includes optimizing for user engagement metrics, such as watch time, likes, shares, and comments. Recommendation systems aim to not just recommend videos that users click on but also content that keeps them engaged and satisfied.

**Languages, Libraries Used:**

Language : Python 3.11

Libraries:

Tensorflow 2.13.0

Sklearn 1.3.0

Pandas 2.1.0

Numpy 1.24.3

NLTK 3.8.1

VaderSentiment 3.3.2

**Dataset Provided:**

2 Files:

1. Video.csv contains

VideoID : unique IDs for Videos

ChannelName : Video’s parent channel

Title : Title provided by the user

Category : Categories provided by user in comma separated

Tags : Tags provided by user for the video

Description : String provided by the user

VideoDuration : Duration of video

Location : Location from which the video is uploaded

1. User.csv contains

UserID : unique IDs of users

VideoID : ID of the video watched by user

Liked : Binary Digit representing the state (0 for no interaction, 1 for liked)

Disliked : Binary Digit representing the state (0 for no interaction, 1 for Disliked)

WatchPercentage : Percentage of user spent time watching the video with respect to total video length

StarRating : explicit rating given by user to a specific video

Comment : Text comment given by user for a particular video (if applicable)

**Methodology:**

**A diagram of a diagram

Description automatically generated**

Knowledge- based Model :

A knowledge-based model for video recommendation is a system that relies on a structured knowledge base to make video recommendations to users. Unlike collaborative filtering or content-based recommendation systems, which primarily use user behavior or content features, a knowledge-based recommendation system leverages explicit knowledge about videos, users, and their preferences to make recommendations.

Content-Based Video Recommendation Using Co-Sine Similarity (Personalized User-Video Recommendation) :

Combine Metadata of a specific video into a single string column in a DataFrame using Python's pandas library. Create a bag of words (BoW) representation of the 'content' column in your DataFrame using scikit-learn's CountVectorizer.

The assigned variable (boW) will contain a sparse matrix where each row corresponds to a document (in this case, a video’s content), and each column represents a unique word in your corpus. The values in the matrix represent the count of each word's occurrence in the corresponding document.

Convert your bag of words (BoW) representation into a Term Frequency-Inverse Document Frequency (TF-IDF) representation using scikit-learn's TfidfTransformer.

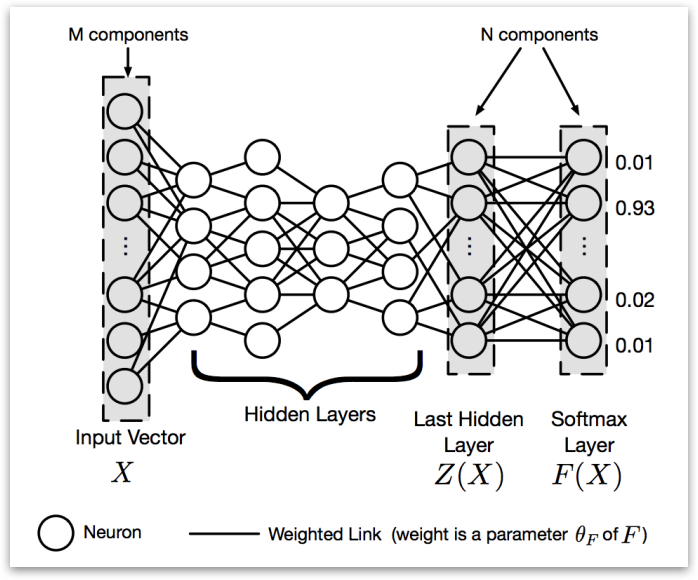
LSA/LSI is a dimensionality reduction technique commonly used in natural language processing to discover the underlying structure in text data.

The dimensionality of the data will be reduced to the specified number of components (n\_components), which is typically chosen based on the desired trade-off between dimensionality reduction and information retention.

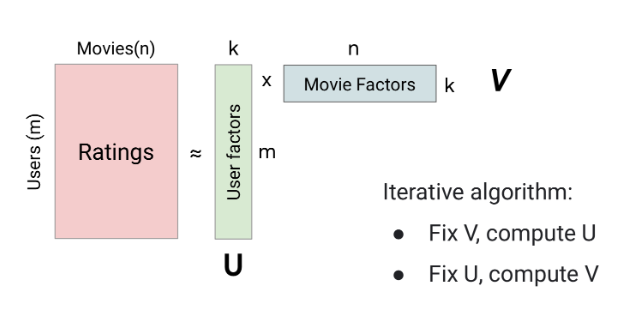
Find Videos similar to those seen by a user based on the content of the Videos. It accomplishes this by utilizing a bag-of-words (BoW) representation of video descriptions and applying cosine similarity to measure the similarity between videos. The code iterates through the list of videos seen by the user (seen\_videos) and, for each user-seen video, identifies its index in the DataFrame (df) containing information about all videos. Then, it calculates cosine similarity scores between the user's video and all other video in the dataset. The top 20 most similar video are selected and stored in a DataFrame (similar\_video\_df), where each row contains the title of a similar video and its corresponding similarity score. The code iterates through all user-seen video, aggregating similar videos for each of them. This process allows for personalized video recommendations based on the user's viewing history and the content of the videos.

Deep Learning Neural Network (User - item Collaborative filtering recommendation) :

One possible DNN model is softmax, which treats the problem as a multiclass prediction problem in which: The input is the user query. The output is a probability vector with size equal to the number of items in the corpus, representing the probability to interact with each item; for example, the probability to click on or watch a video.



Representation of deep learning neural network methodology.



.

Why Deep Neural Network?

Flexibility: DNNs are highly flexible and can capture complex patterns in user behavior and video content, making them suitable for recommendation systems. The softmax activation function in the output layer is commonly used for multiclass classification problems, allowing you to assign probabilities to different videos, which can be interpreted as a measure of user interest.

Personalization: DNNs can capture user-specific preferences and provide personalized recommendations based on historical interactions. They can learn intricate patterns in user behavior, such as the types of videos a user prefers, how often they watch, and when they watch.

Implicit Feedback: DNNs can handle implicit feedback, such as clicks, views, and dwell times, which is often the case in video recommendation systems. They can learn from user engagement data even when explicit ratings or reviews are not available.

Why cosine similarity?

User Similarity: Cosine similarity measures the cosine of the angle between two vectors, which represents the similarity between those vectors. In the context of collaborative filtering, these vectors typically represent users or items. When calculating cosine similarity between users, it helps identify users with similar preferences or behaviors. Users who have rated or interacted with similar videos will have higher cosine similarity scores, indicating a stronger similarity in their preferences.

Content-Based Recommendation: In content-based recommendation systems, cosine similarity is often used to measure the similarity between user profiles (representing their preferences) and video feature vectors (representing video content). This similarity can help recommend videos that are most similar to a user's expressed preferences or content they have interacted with in the past.

Handling Sparsity: Recommendation datasets are often sparse, meaning that users interact with only a small subset of available videos. Cosine similarity can work well in these scenarios because it focuses on the dimensions (features) where both users have expressed preferences or items have shared attributes. This makes it effective in finding meaningful similarities even when there are many missing values in the data.

Why knowledge-based model?

A knowledge-based model can be beneficial for addressing the "cold start" problem in video recommendation. The cold start problem occurs when a recommendation system needs to make personalized recommendations for new users who have little or no historical interaction data or for new videos with limited user engagement history. Here are reasons why knowledge-based models are well-suited for handling this challenge:

No Dependency on User History: Knowledge-based models do not rely on a user's historical interactions with videos to make recommendations. Instead, they leverage explicit knowledge about videos, such as metadata, genres, tags, and descriptions. This means they can provide relevant recommendations to new users who have not yet interacted with any content.

Content-Based Information: Knowledge-based models can use content-based features to make recommendations. For instance, they can recommend videos to new users based on the genre, actors, directors, or other attributes of the videos. This approach doesn't require historical user data and is particularly useful for new users who haven't provided any preferences yet.

**Working:**

**Base Function:**

* *R* be the rating given by the user (on a scale of, say, 1 to 5).
* *L* be the number of likes received on the video.
* *D* be the number of dislikes received on the video.
* *C* be a sentiment score based on comments (e.g., a positive sentiment could be represented as +1, negative sentiment as -1, and neutral as 0).
* *W* be the watch percentage (the fraction of the video watched by the user, ranging from 0 to 1).
* *wR*​, *wL*​, *wD*​, *wC*​, and *wW*​ be the weights assigned to each of these factors, respectively. These weights should add up to 1.

Now, you can calculate the user's likeness score (*S*) as follows:

*S*  = wR . R + wL . L + wD . D + wC . C + wW . W

1. *wR*​⋅*R*: This represents the contribution of the user's rating. The weight *wR*​ determines how much importance you want to give to the user's rating in the final score.
2. *wL*​⋅*L*: This represents the contribution of likes. Likes are generally seen as a positive indicator of user engagement, so they can contribute positively to the score.
3. *wD*​⋅*D*: This represents the contribution of dislikes. Dislikes are generally seen as a negative indicator, so they can decrease the score.
4. *wC*​⋅*C*: This represents the contribution of comments sentiment. Positive sentiment can increase the score, negative sentiment can decrease it, and neutral sentiment may have no impact.
5. *wW*​⋅*W*: This represents the contribution of watch percentage. The longer a user watches a video, the more likely it is that they enjoyed it, so this can positively influence the score.

By adjusting the weights *wR*​, *wL*​, *wD*​, *wC*​, and *wW*​, you can control the relative importance of each factor in determining the user's likeness score. For example, if you believe that user ratings are the most important, you can assign a higher weight to *wR*​ compared to the other factors.

Knowledge Based Model:

**recommendations\_genre**(location) that takes a location as input and provides video recommendations based on two criteria: average recommendation score and popularity within that location. Here's a step-by-step explanation of the functionality of the code:

Input Parameter:

The function takes a single parameter location, which presumably represents a geographical location or a specific category/genre of videos.

Print Location Information:

The code starts by printing a header indicating the location being processed.

Filtering Videos by Location:

It filters videos from the items\_dataset (a dataset containing video information) based on the provided location. This step is performed using items\_dataset[items\_dataset['Location'] == x], where x is the input location.

Merging Datasets:

It merges the filtered video dataset with another dataset called merged\_dataset using an inner join operation. The merging is based on a common column called 'VideoID'.

Calculating High-Rated Videos:

It calculates the average recommendation score for each video within the specified location and sorts the videos in descending order of their recommendation scores. These high-rated videos are stored in the high\_rated\_videos DataFrame.

Printing High-Rated Videos:

It prints the top 10 videos with the highest average recommendation scores for the given location. These are the videos that are recommended based on their high ratings.

Calculating Popular Videos:

It calculates the number of users who have watched each video within the specified location and stores this information in the popular\_videos\_inlocation DataFrame.

Printing Popular Videos:

It prints the top 10 videos with the highest number of users who have watched them in the given location. These are the videos that are recommended based on their popularity.

Overall, this code is designed to provide video recommendations for a specific location by considering both average recommendation scores and popularity. It's helpful for suggesting videos to users in that location based on their preferences and what other users in the same location have enjoyed.

**Cosine Similarity Function:**

1. **Combining Columns into a Single String:**
   * The code concatenates the values from multiple columns into a single string, which will be stored in a new column called 'content.'
   * The columns being concatenated are:
     + **ChannelName**: The name of the channel.
     + **Title**: The title of the video or content.
     + **Category**: The category of the content.
     + **MainCategories**: Main categories associated with the content.
     + **Description**: A description of the content.
2. **String Concatenation:**
   * For each row in the DataFrame **df**, the code concatenates the values from the specified columns into a single string.
   * The **+** operator is used to concatenate these values together.
   * The **.astype(str)** method is used to ensure that numerical or other non-string values are converted to strings before concatenation.
   * **.fillna(' ')** is used to handle cases where there might be missing (NaN) values in any of the columns. It replaces missing values with a space ' '.
3. **Assigning to the 'content' Column:**
   * After creating the combined string for each row, the code assigns these strings to a new column named 'content' in the DataFrame.
   * **df['content'] = df['content']** is essentially updating the 'content' column with the concatenated strings.

**Transforming Text Data into BoW:**

[**https://scikitlearn.org/stable/modules/generated/sklearn.feature\_extraction.text.CountVectorizer.html**](https://scikitlearn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html)

bow = vectorizer.fit\_transform(df['content']) performs the transformation of the text data. Here's how it works:

df['content']: This is the Series containing the text data (textual content) that you want to convert into a BoW representation. In this case, it appears to contain concatenated text from multiple columns.

vectorizer.fit\_transform(): This method of the CountVectorizer object takes the text data as input and performs two main operations:

Fit: It analyzes the text data to learn the vocabulary (unique words) present in the entire corpus of text.

Transform: It converts the text data into a matrix where each row represents a document (in this case, a row from the 'content' column), and each column represents a unique word from the vocabulary. The values in the matrix represent the count of how many times each word appears in each document.

**BoW Representation:**

After this code is executed, the variable bow contains the BoW representation of the text data. This variable is typically a sparse matrix (in the form of a SciPy sparse matrix) where each row corresponds to a document, and each column corresponds to a unique word. The values in the matrix represent the frequency of each word in each document.

This BoW representation is a common way to prepare text data for machine learning tasks, such as text classification, clustering, or sentiment analysis. It converts text data into a format that can be used by machine learning algorithms that require numerical input.

<https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html>

1. **Creating a TfidfTransformer Object:**
   * **tfidf\_transformer = TfidfTransformer()** creates an instance of the **TfidfTransformer** class. This object will be used to perform the TF-IDF transformation on the BoW data.
2. **Transforming BoW into TF-IDF:**
   * **tfidf = tfidf\_transformer.fit\_transform(bow)** performs the transformation of the BoW data into TF-IDF representation. Here's how it works:
   * **bow**: This variable likely contains the BoW representation of the text data, which was previously created using a **CountVectorizer**.
   * **tfidf\_transformer.fit\_transform()**: This method of the **TfidfTransformer** object takes the BoW data as input and calculates the TF-IDF values for each term (word) in each document (row). The TF-IDF transformation consists of two main steps:
     + **Fit**: In this step, the transformer analyzes the BoW data to compute the inverse document frequency (IDF) for each term in the vocabulary. IDF measures the importance of a term by considering how often it appears across all documents in the corpus. Terms that appear in many documents have lower IDF values, while terms that appear in fewer documents have higher IDF values.
     + **Transform**: After computing the IDF values, the transformer applies the TF-IDF formula to each term in each document. TF-IDF combines two factors:
       - Term Frequency (TF): Measures how often a term appears in a specific document.
       - Inverse Document Frequency (IDF): Measures the importance of a term across all documents.
   * The result of this transformation is a matrix, similar in structure to the original BoW matrix, where each value represents the TF-IDF score of a term in a specific document. The TF-IDF representation reflects the importance of each term in each document while considering the overall frequency of the term across all documents.
3. **TF-IDF Representation:**
   * After executing this code, the variable **tfidf** contains the TF-IDF representation of the original text data. This representation is commonly used in natural language processing (NLP) and information retrieval tasks because it provides a way to measure the importance of terms in a document collection while taking into account their frequency and distribution.
   * The TF-IDF representation is often used for tasks like text classification, document retrieval, and information retrieval, where you want to capture the relevance of specific terms in documents relative to the entire corpus.

<https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html>

1. **Creating a TruncatedSVD Object:**
   * **lsa = TruncatedSVD(n\_components=50, algorithm='arpack')** creates an instance of the **TruncatedSVD** class. This object will be used to perform the dimensionality reduction (SVD) on the TF-IDF matrix.
   * **n\_components=50**: This parameter specifies the number of components (dimensions) to reduce the TF-IDF matrix to. In this case, it reduces it to 50 dimensions. Reducing dimensions is useful for various purposes, such as improving efficiency in downstream analysis or capturing the most important latent semantic features.
   * **algorithm='arpack'**: This parameter specifies the algorithm to use for singular value decomposition (SVD). 'arpack' is one of the available algorithms for SVD. It's a good choice for sparse matrices, such as TF-IDF matrices, because it is efficient in handling sparse data.
2. **Fitting the LSA Model:**
   * **lsa.fit(tfidf)** fits (applies) the LSA model to the TF-IDF matrix (**tfidf**). This means it calculates the singular value decomposition and reduces the dimensionality of the TF-IDF matrix while preserving the most important semantic information.
   * SVD is a mathematical technique used to factorize a matrix into three matrices: U, Σ (sigma), and V^T (transpose of V). LSA retains the top singular values and corresponding vectors to reduce the dimensionality of the original matrix.
3. **Dimensionality Reduction:**
   * After fitting the LSA model, the TF-IDF matrix is transformed from its original high-dimensional space to a lower-dimensional space (in this case, 50 dimensions). The resulting representation captures the latent semantic structure of the data.
   * LSA is particularly useful for reducing the dimensionality of text data while preserving the underlying semantic relationships between terms and documents. It helps in capturing the core semantic meaning of the text.
4. **Why 'arpack' Algorithm?**
   * The 'arpack' algorithm is chosen because it is specifically designed for handling large sparse matrices efficiently. Sparse matrices are common in natural language processing when working with TF-IDF representations of text data. 'arpack' is an iterative method that can compute a few singular values and vectors without explicitly computing the full SVD, making it computationally efficient for large and sparse datasets.

TF-IDF Representation:

Before computing cosine similarity, each video in the dataset is represented as a vector in a high-dimensional space based on its TF-IDF values. Each dimension of the vector corresponds to a unique term (word) in the entire corpus of video descriptions and titles.

Cosine Similarity:

Cosine similarity is a measure of similarity between two vectors in a multi-dimensional space. It calculates the cosine of the angle between two vectors and produces a value between -1 and 1. In the context of document similarity, higher cosine similarity values indicate greater similarity between documents.

Algorithm Explanation:

The algorithm works as follows for each video that a user has seen (seen\_videos):

Find the index of the user's seen video in the dataset (video\_index). Calculate the cosine similarity between the TF-IDF vector of the user's seen video and the TF-IDF vectors of all other videos in the dataset. This results in a similarity score for each video, indicating how similar it is to the user's seen video.

Enumerate and sort these similarity scores to find the top 20 most similar videos (excluding the user's seen video itself). Create a dictionary called video\_scores to store video titles and their scores. Iterate through the top similar videos and add them to the video\_scores dictionary. If a video title is already in the dictionary, increase its score (e.g., by 0.3). This step accounts for videos that are similar to more than one of the user's seen videos.

Deep Neural Network:

**Label Encoding Users and Videos:**

* **user\_enc = LabelEncoder()** and **item\_enc = LabelEncoder()** create instances of the **LabelEncoder** class for users and videos, respectively. Label encoding is used to convert categorical values (in this case, user IDs and video titles) into numerical labels.
* **refined\_dataset['user'] = user\_enc.fit\_transform(refined\_dataset['UserID'].values)** applies label encoding to the 'UserID' column in the dataset and stores the encoded values in a new column named 'user'. Each unique user ID is mapped to a unique numerical label.
* **refined\_dataset['video'] = item\_enc.fit\_transform(refined\_dataset['Title'].values)** applies label encoding to the 'Title' column (video titles) in the dataset and stores the encoded values in a new column named 'video'. Each unique video title is mapped to a unique numerical label.

Input Layers:

Two input layers are defined, one for users and one for movies. These layers are initialized with the shape of (1,), indicating that each input will be a single integer representing a user or movie.

Embedding Layers:

Embedding layers are used to convert the user and movie IDs into dense vectors of size n\_factors. These vectors are initialized with the 'he\_normal' initializer and regularized with L2 regularization to prevent overfitting. n\_users and n\_movies represent the total number of users and movies, respectively.

Reshape Layers:

The output of the embedding layers is reshaped to have a shape of (n\_factors,) to prepare them for concatenation.

Concatenation Layer:

The user and movie embeddings are concatenated together into a single vector. This combines the information from both the user and movie to make a recommendation.

Dropout Layers:

Dropout layers are added to reduce overfitting. They randomly set a fraction of input units to zero during training.

Dense Layers:

Several dense layers are added to the model:

The first dense layer has 32 units and ReLU activation.

The second dense layer has 16 units and ReLU activation.

The third dense layer has 100 units with a softmax activation. The softmax activation is commonly used in recommendation systems for multi-class classification tasks. It assigns probabilities to different classes (in this case, movie recommendations).

Model Definition:

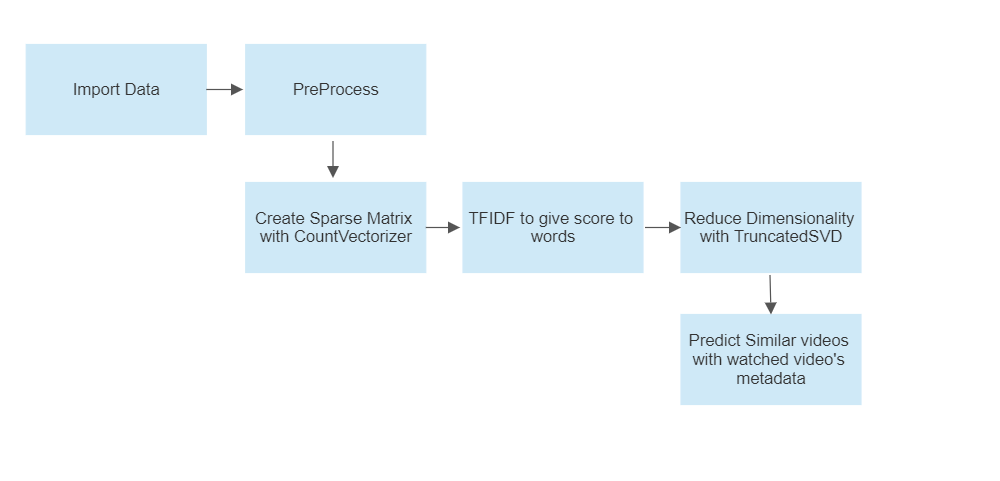
The model is defined with the input layers for users and movies and the output layer with the softmax activation.

Compilation:

The model is compiled with the Stochastic Gradient Descent (SGD) optimizer, sparse categorical cross-entropy loss (suitable for multi-class classification), and accuracy as a metric.

**Flow Chart:**

Cosine Similarity:



Deep Neural Network:

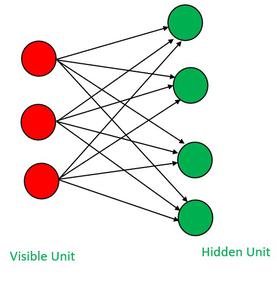
Architecture:

A diagram of a process flow

Description automatically generated

**FURTHER BUILDING:**

**RBM:**



**A Restricted Boltzmann Machine (RBM) can be a powerful tool for video recommendation systems, especially when dealing with collaborative filtering and capturing complex patterns in user preferences. Here's how an RBM can be advantageous for video recommendation:**

1. **Implicit Feedback Handling: RBMs are well-suited for handling implicit feedback data, which is common in video recommendation systems. Implicit feedback includes user actions like views, clicks, and time spent on a video. RBMs can model these interactions effectively and capture user preferences even when explicit user ratings are not available.**
2. **Representation Learning: RBMs can automatically learn meaningful representations of videos and user preferences in a low-dimensional latent space. This can help in capturing complex and abstract features that might not be evident in raw data. For videos, these features could include content type, genre, visual style, or even mood.**
3. **Collaborative Filtering: RBMs can naturally perform collaborative filtering by learning user-item interactions. They can identify similar users and videos based on patterns of interactions, leading to personalized recommendations. This makes them particularly useful in scenarios where users have varying tastes.**
4. **Serendipity: RBMs have a stochastic component in their learning, which means they can introduce an element of serendipity into recommendations. This can help users discover videos they might not have found through traditional means, leading to a more engaging recommendation system.**

**However, it's important to note that RBMs also have some challenges:**

**Complexity: RBMs can be more complex to train and tune compared to simpler recommendation algorithms. Proper hyperparameter tuning and training strategies are essential.**

**Cold Start Problem: RBMs may struggle with the cold start problem, where new users or videos have limited interaction data. Hybrid models combining RBMs with other techniques can mitigate this issue.**

**Interpretability: RBMs are often seen as black-box models, making it challenging to interpret why a particular recommendation was made. This can be an issue in applications requiring transparency.**

**DBN:**

A diagram of a structure

Description automatically generated

1. **Hierarchical Feature Learning: DBNs consist of multiple layers of Restricted Boltzmann Machines (RBMs) stacked on top of each other. This hierarchical structure allows them to learn increasingly abstract and complex features from the data. In the context of video recommendation, this means that DBNs can capture intricate patterns and relationships in user behaviors and video content features.**
2. **Automatic Feature Extraction: DBNs can automatically discover and extract relevant features from raw data, including video content and user interaction history. This can be highly beneficial when dealing with large and unstructured video datasets, as the DBN can learn to represent videos and users in a more meaningful way.**
3. **Collaborative and Content-Based Recommendations: DBNs can combine collaborative filtering and content-based recommendation approaches within the same model. The lower layers can focus on user-user and video-video interactions (collaborative filtering), while the upper layers can incorporate video content features (content-based recommendation). This hybrid approach can result in more accurate and diverse recommendations.**
4. **Personalization: DBNs excel at personalization. They can capture individual user preferences and adapt to changes in user behavior over time. This is crucial for providing tailored video recommendations to users with varying tastes and interests.**
5. **Implicit Feedback Handling: Similar to RBMs, DBNs can handle implicit feedback effectively, making them suitable for scenarios where users interact with videos through actions like clicks, views, and watch duration rather than explicit ratings.**
6. **Scalability: With proper training strategies and parallelization, DBNs can scale to handle large video libraries and user bases. This scalability is essential for real-world video recommendation systems.**

**Conclusion:**

In conclusion, leveraging a knowledge-based model is a highly effective strategy for addressing the "cold start" problem in video recommendation systems. These models offer a reliable solution by making recommendations based on explicit knowledge about videos, such as metadata, genres, and content attributes. In scenarios where new users lack historical interaction data or new videos have limited engagement history, knowledge-based models stand out as a dependable option. They not only provide relevant suggestions to users without a history but also ensure diversity in recommendations by considering various video attributes. Furthermore, the interpretability of knowledge-based recommendations enhances user trust and engagement, making them an invaluable tool for mitigating the challenges of cold start scenarios.

In the realm of personalized video recommendation, cosine similarity emerges as a powerful metric and technique. Its ability to measure similarity between users or videos based on historical interactions or content features is well-suited to capture user preferences and provide tailored suggestions. By calculating cosine similarity, recommendation systems can identify users with similar tastes or videos with shared attributes, resulting in more accurate and personalized recommendations. This approach significantly enhances user satisfaction by offering content that aligns with individual preferences, ultimately leading to higher engagement and retention rates.

When it comes to general video recommendation, deep neural networks (DNNs) equipped with sophisticated architectures and algorithms offer a versatile and powerful solution. These DNN-based models can capture intricate patterns in user behavior and video content, providing a holistic approach to video recommendations. The flexibility and scalability of DNNs enable them to handle vast datasets with millions of users and videos, ensuring robust recommendations. DNNs excel in scenarios where personalized recommendations are essential but also extend their capabilities to general recommendations by leveraging the collective intelligence of user interactions. By collaborative filtering, DNNs contribute to a comprehensive and user-centric video recommendation experience, making them a top choice for general video recommendation systems.