Song Recommendation Engine

Ryan Mc Neil
Golvis Tavarez

Background & Research Question

- Song recommendation engines are integral to modern audio streaming platforms.
- Traditional methods include collaborative filtering and content-based filtering.
- Recent advances incorporate deep learning methods. Particularly, Convolutional Neural Networks (CNNs) have come to the fore as powerful methods for analyzing audio features and capturing complex patterns in audio data.
- Altogether, these systems serve to enhance user engagement and uncover new musical preferences.

Collaborative Filtering

- Collaborative filtering (CF) is a technique used int recommendation systems that predicts user preferences based on the preferences or activity of other users.
- CF operates under the assumptions that if users have shown similar tastes in the past, they are likely to agree on future preferences.
- Collaborative filtering can user-based, recommending items by looking at the preferences of similar users, or item-based, identifying items that are frequently interacted with together.
- Collaborative filtering can also be memory-based (user-based or item-based nearest-neighbor methods), or model-based (using matrix factorization and SVD).

Our Collaborative Filtering Model

A User-Item Interaction Approach

1. Similarity Matrix Calculation
    a. Calculate cosine similarity between songs using audio features
    b. Create a matrix representing song-to-song similarities

2. User-Based Recommendation Generation
    a. Recommendation Generation:
    b. Identify user's listened tracks
    c. Calculate average similarity to other songs
    d. Sort and filter recommendations
    e. Return top N personalized suggestions

TF-IDF

- Term Frequency-Inverse Document Frequency (TF-IDF) is a technique used in text analysis to evaluate the importance of a word in a document relative to a collection of documents. In short, for a given work, TF counts the occurrences of a word in a document and IDF counts the inverse of its occurrences across the corpus, reducing the importance of words that are common across many documents.

- In text classification TF-IDF transforms textual data into numerical vectors, allowing computation of similarity for classification and comparison based on content.

Encoder Models

- Encoder models (like BERT or GPT) are neural network models designed to "understand" textual data. These models encode text into dense vector representations, capturing *semantic* information in a way that TF-IDF cannot.

- Encoder models enable more nuanced text comparison and excel in capturing the relationship between words and their context within sentences.

Our TF-IDF/Encoder Model

Preprocssing

- Sample data [1]
- Index: Concatenation of song & artist
- Document: Song lyrics text
- Drop duplicates (lives, edits)
- Tokenize w/ stopwords including vocalizations.

Transform data

- For TF-IDF we use TF-IDF vectorizer to create a sparse matrix of terms by documents.
- For an encoder (we use BERT), we encode the lyrics data to embeddings & convert those to tensors.

Recommend Songs

- For TF-IDF calculate cosine similarity between documents to produce a similarity matrix.
- For Encoder model create a matrix of embeddings, then calculate the cosine distance between vector embeddings.
- Recommend the top n songs by cosine similarity.

[1]Data source: https://www.Kaggle.com/datasets/deepshah16/song-lyrics-dataset

Our TF-IDF/Encoder Model

TF-IDF-based Model

```python
def recommend_songs(query_song, songs_and_artists, similarity_matrix, top_n=5):
    """
    Recommend songs based on lyrics similarity.

    Parameters:
    - query_song: str, the title of the song to query
    - song_titles: list of song titles
    - similarity_matrix: precomputed cosine similarity matrix
    - top_n: int, number of recommendations to return

    Returns:
    - list of recommended songs
    """
    # Find the index of the query song
    try:
        idx = songs_and_artists.index(query_song)
    except ValueError:
        return f"Song '{query_song}' not found in the dataset."

    # Get similarity scores for the song
    similarity_scores = list(enumerate(similarity_matrix[idx]))

    # Sort by similarity score (descending) and exclude the query song
    similarity_scores = sorted(similarity_scores, key=lambda x: x[1], reverse=True)
    similarity_scores = [s for s in similarity_scores if s[0] != idx]

    # Retrieve top N recommendations
    top_songs = [(songs_and_artists[i], score) for i, score in similarity_scores[:top_n]]

    return top_songs
```

Executed at 2024.12.09 13:10:15 in 5ms

Encoder-based Model

```python
# Define a function to recommend songs based on similarity
def recommend_songs_encoder(query_song, songs_and_artists, embeddings, top_n=5):
    """
    Recommend songs based on lyrics similarity using BERT.

    Parameters:
    - query_song: str, the title of the song to query
    - song_titles: list of song titles
    - embeddings: precomputed embeddings matrix
    - top_n: int, number of recommendations to return

    Returns:
    - list of recommended songs
    """
    try:
        idx = songs_and_artists.index(query_song)
    except ValueError:
        return f"Song '{query_song}' not found in the dataset."

    # Compute cosine similarity with the query song
    query_embedding = embeddings[idx]
    similarity_scores = util.pytorch_cos_sim(query_embedding, embeddings)[0]

    # Sort by similarity score
    top_indices = similarity_scores.argsort(descending=True)[1:top_n+1]  # Exclude query song
    top_songs = [(songs_and_artists[i], similarity_scores[i].item()) for i in top_indices]

    return top_songs
```

Executed at 2024.12.09 13:11:37 in 5ms

Our TF-IDF/Encoder Model

- The TF-IDF-based model seems weaker. Cosine similarities tend to be a maximum of ~0.25.
- The embedding model recommends songs with cosine similarities upwards of ~0.5.
- Embedding models are likely better for this sort of task given the value of *semantic* meaning in music, versus exact verbiage.
- Once the data is processed, recommendation time is very quick for both models.
- In production, simply storing the embedding matrix would take up an unreasonable amount of memory to store:
  - Memory = # of embeddings * vector size * size of each value (bytes)
  - For an embedding vector of 768, the total songbase of Spotify (~100 million songs) [2], and 4 bytes per value:
    - 100,000,000 * 768 * 4 = 307.2 GB
  - But the similarity matrix would take up:
  - $100{,}000{,}000^2$ * 4 bytes = 40PB
- We could instead:
  - Rather than recomputing the entire matrix, the system could compute similarities dynamically for specific queries, with candidates determined by collaborative filtering or the CNN model outputs.
  - Shrink very small values to 0 and employ sparse matrices.
  - Switch from 32-bit floats (4 bytes) to 8-bit integers (1 byte) with some processing.
  - Distribute this smaller matrix & associated operations using RDDs.
- For commercial servers, any size under ~512GB would be reasonable for RAM.

Convolutional Neural Networks

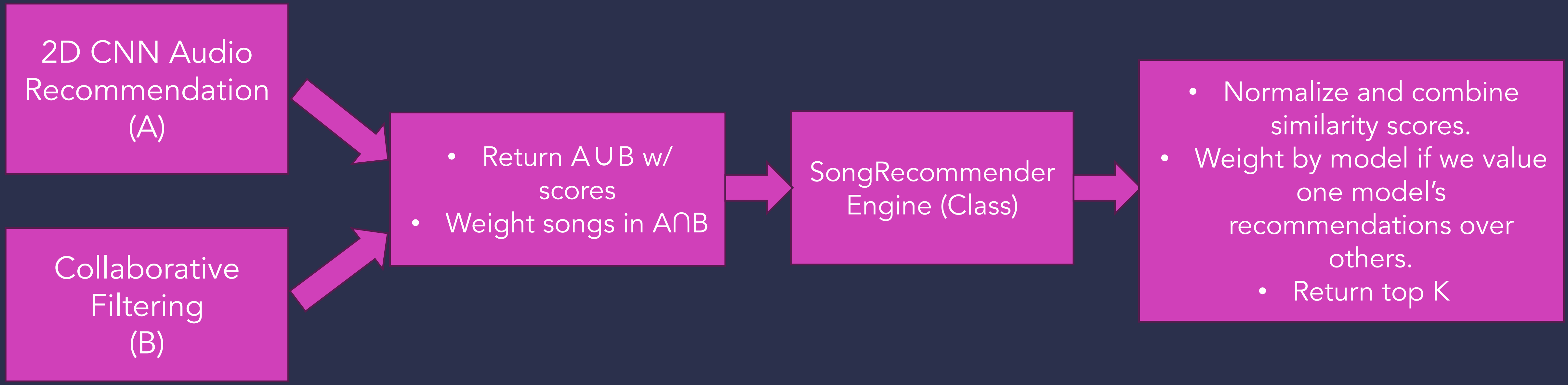
- Convolutional Neural Networks (CNNs) were originally designed for image processing. Have been adapted for audio analysis. Effective at learning high-level semantic representations of audio.
- CNNs process audio data that has been transformed into spectrograms, which visually represent sound frequencies over time. Other features, like pitch & timbre are extracted by the convolutional layers.
- Useful for a variety of audio processing tasks, these models are employed by services like Spotify to compare audio tracks for similarity for the purpose of recommending.

Combine Class

```python
class SongRecommender:
    def __init__(self):
        self.lyrics_df = None
        self.tfidf_similarity_matrix = None
        self.encoder_embeddings = None
        self.songs_and_artists = None
        self.spotify_df = None
        self.spotify_similarity_matrix = None
        self.stop_words = self._get_stop_words()
        self.drop_words = self._get_drop_words()

    def _get_drop_words(self): ...

    def _get_stop_words(self): ...

    def load_lyrics_data(self, data_folder): ...

    def _preprocess_lyrics_df(self, df): ...

    def build_tfidf_model(self): ...

    def build_encoder_model(self, model_name='all-MiniLM-L6-v2'): ...

    def load_spotify_data(self, file_path): ...

    def recommend_by_tfidf(self, query_song, top_n=5): ...

    def recommend_by_encoder(self, query_song, top_n=5): ...

    def recommend_by_collaborative_filtering(self, target_user, n=5): ...
```

```python
def main():
    # Initialize recommender
    recommender = SongRecommender()

    # Load and prepare data
    recommender.load_lyrics_data("lyrics_dataset/csv")
    recommender.build_tfidf_model()
    recommender.build_encoder_model()
    recommender.load_spotify_data('spotify_data.csv')

    # Example recommendations
    query_song = "Coldplay - The Scientist"

    # Get recommendations using different methods
    tfidf_recommendations = recommender.recommend_by_tfidf(query_song)
    encoder_recommendations = recommender.recommend_by_encoder(query_song)
    cf_recommendations = recommender.recommend_by_collaborative_filtering(
        'c1a6910ecac9fd5e5348326675fb6ca6'
    )

    # Print results
    print("TF-IDF Recommendations:")
    for song, score in tfidf_recommendations:
        print(f"- {song} (similarity: {score:.2f})")

    print("\nEncoder Recommendations:")
    for song, score in encoder_recommendations:
        print(f"- {song} (similarity: {score:.2f})")

    print("\nCollaborative Filtering Recommendations:")
    for artist, track in cf_recommendations:
        print(f"- {track} by {artist}")

if __name__ == "__main__":
    main()
```

Hypothetical Design



Questions

Works Cited

1. Shah, D. (2018). "Song Lyrics Dataset." *Kaggle*. https://www.Kaggle.com/datasets/deepshah16/song-lyrics-dataset/
2. Spotify. (2024). "Company Info." *Spotify Newsroom*. https://newsroom.spotify.com/company-info/