university of groningen / faculty of science and engineering / bernoulli institute

Music Recommendation System for Spotify Playlists

Advanced Machine Learning Semester Project

Matthijs van der Lende, s4325621, m.r.van.der.lende@student.rug.nl
Niclas Müller-Hof, s4351495, n.j.muller-hof@student.rug.nl
Matthijs Jongbloed, s4357833, m.d.jongbloed@student.rug.nl
Mik Claessens, s4370244, m.h.m.claessens@student.rug.nl

Abstract: This paper presents a hybrid music recommendation system for Spotify playlist continuation that integrates collaborative filtering and content-based filtering to address challenges such as data sparsity and cold starts. Leveraging a large-scale Spotify playlist dataset, we implement a model-based collaborative filtering approach using singular value decomposition (SVD) alongside a content-based method that learns track representations via Word2Vec embeddings from track attributes. The two methods are combined using a weighted ensemble, and extensive hyperparameter tuning and evaluation—through a train-test split of tracks within playlists—demonstrate that the hybrid model achieves a test accuracy of 0.41, outperforming the standalone collaborative (0.34) and content-based (0.16) systems. Our analysis highlights both the strengths of this integrated approach, which effectively captures latent relationships between tracks and playlists, and its limitations, such as the static treatment of playlists as user proxies and reliance on high-level audio features. The study concludes by suggesting future improvements, including the incorporation of deeper audio representations and explicit user feedback, to further enhance personalization in music recommendation systems.

Contents

1 Introduction

Modern digital platforms are increasingly confronted with vast collections of content, such as music, movies or short videos, making it difficult for users to discover new and relevant items. To address this challenge, *recommender systems* have emerged as a critical technology within information filtering, designed to guide users towards items of likely interest (Ko, Lee, Park, & Choi, 2022). Such systems have found widespread application accross diverse domains including music discovery (Iwahama, Hijikata, & Nishida, 2004), movie suggestions (Salter & Antonopoulos, 2006), e-commerce product recommendations (Weihong & Yi, 2006), and selection of educational materials (Imran & Abdullah, 2010).

A variety of machine learning paradigms underpin recommender systems. In this project we focus on a hybrid recommendation system that uses *model-based collaborative filtering*, involving the singular value decomposition (SVD) and *content-based filtering* based on `word2vec` word embeddings. Below, we summarize collaborative and content-based filtering and outline their known benefits and drawbacks, motivating the hybrid strategy we adopt in our methodology.

1.1 Content-Based filtering

Content-based filtering leverages the intrinsic properties of items to generate recommendations; for music recommendations, this may include metadata (e.g., artist name, genre, popularity) or learned feature embeddings (e.g., from audio signals or lyrical content). The system builds a user profile based on the features of items the user has already engaged with, and suggests new items that are *similar* to those in the user's profile. Because these approaches rely primarily on item attributes, they are especially effective at recommending items that share notable characteristics with those the user has liked in the past, thereby inherently solving one facet of the *cold start problem* for items: as long as an item's features can be extracted, it can be recommended to matching user profiles even before widespread ratings are available. Moreover, a user with just a few initial inputs can start receiving relevant suggestions immediately (Pazzani & Billsus, 2007). However, content-based methods often struggle to promote diversity, as they tend to recommend items closely related to the user's established preferences, leading to a narrower set of suggestions. Additionally, while item cold starts are addressed, the approach still requires sufficient user interaction history to capture taste, and it seldom introduces "novel" content that diverges from a user's known interests, thus limiting opportunities for broader exploration.

1.2 Collaborative filtering

In contrast, collaborative filtering (CF) exploits patterns across user interactions and assumes that users who agreed in the past will continue to show similar preferences in the future (Im & Hars, 2007). That is, if two users share a similar history of item engagement, items favored by one user may be relevant to the other. CF systems operate on user-item rating or interaction matrices, identifying neighborhoods of similar users (user-based CF) or similar items (item-based CF). By filling in missing entries in this matrix, CF models can predict items likely to match a user's tastes without analyzing any specific item attributes.

A classic implementation of CF is *memory-based*, where the entire user-item interaction matrix is directly used to produce recommendations in real time. Two main variants exist: User-Based Collaborative Filtering finds a set of users ("neighbors") who share similar interaction patterns with the active user. Items liked by these similar users are then recommended. Item-Based Collaborative Filtering focuses on item similarities; items that resemble those the user has interacted with are suggested. The other form of CF is *model-based*, which uses machine learning techniques to learn a model that predicts missing ratings or interactions, rather than use the interaction matrix directly. Commonly used ML techniques include Bayesian networks, clustering models, latent factor models (singular value decomposition, principal component analysis), latent Dirichlet allocation and Markov decision process-based models (Su & Khoshgoftaar, 2009).

Despite its intuitive appeal and widespread success, CF suffers from several well-known problems:

- **Sparsity**: In large-scale systems, most users only interact with a small fraction of available items, leading to sparse user-item matrices that degrade recommendation quality.
- **Cold Start**: Newly registered users lack historical interactions, making it challenging to tailor recommendations. Similarly, new items without ratings or interaction data are difficult to recommend.
- **Gray Sheep**: Some users exhibit atypical or eclectic tastes that do not align with any identifiable community of other users.

1.3 Hybrid Methods

Hybrid recommender systems combine content-based and collaborative filtering in order to mitigate the limitations of each approach. They incorporate both *item features* and *interaction patterns* to generate more accurate, diverse, and robust recommendations. Several hybridization techniques exist (Burke, 2002):

- **Weighted Hybridization**: Linear or weighted combination of content-based and CF scores.
- **Switching Hybridization**: Dynamically switches between content-based and CF methods based on certain criteria (e.g., data availability).
- **Cascade Hybridization**: Prioritizes one recommender's results and uses the second as a re-ranking or refinement layer.
- **Mixed Hybridization**: Presents recommendations from both methods simultaneously.
- **Feature Combination**: Treats CF features as inputs to a content-based model, or vice versa.
- **Feature-Augmentation**: Uses the output of one approach to enrich the feature set of the other.
- **Meta-Level**: Constructs a model based on the model generated by another system (e.g., user embeddings from CF fed into a content-based algorithm).

This integrated strategy is popular among leading commercial platforms, which often rely on hybrid solutions to serve diverse user preferences at scale (Gomez-Uribe & Hunt, 2016). Such an approach also offers greater resilience to both cold start and sparsity issues, while still leveraging content attributes to increase recommendation diversity.

In this project, we focus on building a music recommendation system for playlist continuation using large-scale data. The goal is to generate relevant additions for a given playlist by modeling user and item relationships at scale. To address the complexities involved, we design an hybrid recommendation model that can capture patterns in existing playlists and identify meaningful connections among a vast array of tracks. By analyzing various factors that influence user preferences, we aim to provide a framework capable of handling high-volume data while delivering personalized, high-quality recommendations.

1.4 Dataset

The recommendation task at hand is *playlist continuation*, whereby for each playlist we aim to suggest the top n most relevant additional tracks. As a simplification, each unique playlist is treated as a separate "user." To realize this, we started with the Spotify 1-million playlist dataset (Papreja, Venkateswara, & Panchanathan, 2020; Chen, Lamere, Schedl, & Zamani, 2018), which comprises public playlists gathered through the Spotify developer API. The dataset is cleaned by removing infrequent songs (those appearing in fewer than three playlists), eliminating duplicates, and retaining only playlists with lengths between 10 and 5000 tracks, we ended up with 745,543 playlists, 2,470,756 tracks, and 2680 unique genres. Originally, Spotify's audio features could not be accessed due to deprecated API calls; fortunately, a separate Kaggle dataset* containing 1.2 million tracks with audio attributes extracted before deprecation allowed us to recover these features. We then matched the songs in our playlists to those within the Kaggle dataset, producing a final "matched" dataset of 134,712 songs. An association table was created to map each playlist identifier (pid) to the corresponding track Uniform Resource Identifiers (URIs), enabling efficient lookups of which songs belong to which playlist. In order to create proper train test splits and have enough data to make a prediction we have another requirement to have at least 50 or 100 songs inside a playlist. The distribution of playlist size can be seen in Figure 1.1. As can be seen there are less playlist with many songs which makes our dataset very limited for playlist with more than 50 or 100 songs. In fact there are only 124 playlist with more than 100 songs.

The resulting "matched" dataset offers a range of metadata and audio features that provide a multifaceted view of each track. For each track the following metadata is available:

- Album: ID, name, URI.
- Artist(s): ID, name, URI.
- Track: ID, name, duration, popularity, explicit flag, URI.

*`https://www.kaggle.com/datasets/rodolfofigueroa/spotify-12m-songs`

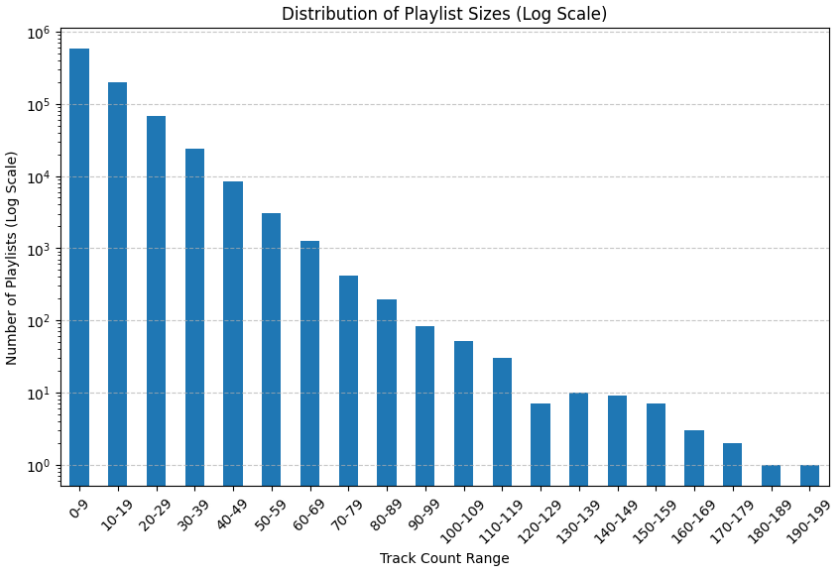


Figure 1.1: The distribution for how many songs each playlist has. (Log scale)

In addition to basic metadata, the Kaggle dataset (obtained before the Spotify audio features API was deprecated) provides detailed musical attributes for each track:

- *Danceability*: A measure (0.0 to 1.0) of how suitable a track is for dancing, based on a combination of musical elements such as tempo, rhythm stability, beat strength, and overall regularity.
- *Energy*: Also scaled from 0.0 to 1.0, capturing the intensity and activity of a track; higher energy often implies a faster, louder, or more dynamic piece of music.
- *Key*: The musical key the track is in, represented as integers (0 = C, 1 = C#, ..., 2 = D).
- *Loudness*: The overall loudness of a track, measured in decibels (dB). Typical values range between -60 dB and 0 dB.
- *Mode*: Indicates whether a track is in a major (1) or minor (0) key.
- *Speechiness*: Assesses the presence of spoken words; higher values (closer to 1.0) imply more speech-like qualities.
- *Acousticness*: Reflects how acoustic a track is, with 1.0 indicating a high likelihood the track is purely acoustic.
- *Instrumentalness*: Predicts whether a track contains no vocals. The closer the value is to 1.0, the greater the likelihood it is purely instrumental.
- *Liveness*: Estimates the probability that a track was performed live; higher values indicate a stronger audience or "live" presence in the recording.
- *Valence*: A measure of the musical positiveness conveyed by a track, with 1.0 being more positive or "happy."
- *Tempo*: The speed of the track, measured in beats per minute (BPM).
- *Year / Release Date*: The recorded year and specific date when the track was released.

By combining these metadata fields with the rich set of audio features, we can capture both high-level contextual information (e.g., artist identity, popularity) and deeper musical characteristics (e.g., danceability, energy). This wealth of data forms the basis for both content-driven and hybrid recommendation approaches, ensuring that suggested tracks not only align with a playlist's existing style but also offer opportunities for meaningful discovery.

We computed the Pearson correlation coefficients between different audio features available in the dataset and visualized them using a heatmap as seen in Figure 1.2. This provides insights into which features are interdependent and how they might collectively influence playlist continuation. The heatmap reveals several interesting relationships.

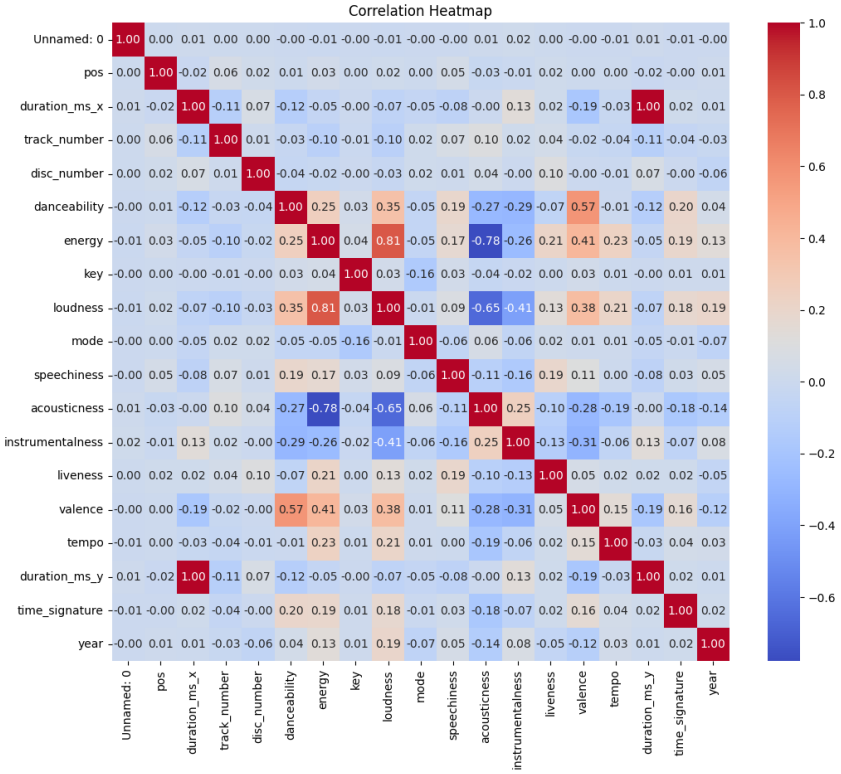


Figure 1.2: Correlation Heatmap of Song Features: This heatmap visualizes the relationships between various song attributes from the Kaggle dataset. Strong positive correlations (closer to 1) indicate features that tend to increase together, while strong negative correlations (closer to -1) highlight inverse relationships.

2 Methodology

2.1 Recommendation System

First we cover how we designed our recommendation system specifically for the task of recommending new tracks to Spotify playlists. Recommendation system commonly talk about a set of users to which we want to recommend a set of items to. In this case we identify users with playlists $\mathcal{P} = \{p_1, p_2, \ldots, p_{N=887{,}060}\}$, and the items are all the tracks $\mathcal{T} = \{t_1, t_2, \ldots, t_{M=134{,}712}\}$. Each playlist p is itself a subset of $\mathcal{T}$. Additionally as described in Section 1.4, each track has associated with it a set of attributes. So $t_i \in A_1 \times \ldots \times A_m$. Given a playlist, the recommendation algorithm assigns a score to each track. A recommendation would then consists of the top n *new* tracks with the highest score. Essentially what both the collaborative and content-based filtering components of the recommender learn is a function $h : \mathcal{P} \times \mathcal{T} \to \mathbb{R}^{\geq 0}$. Key observations: We assume our dataset $(\mathcal{P}, \mathcal{T})$ is static, and h is learned exclusively in an unsupervised or self-supervised fashion. In this formulation there is no ground truth target variable. We do not know in advance these recommendation scores for tracks and the filtering algorithms aim to discover patterns or relationships within the data itself. In practice, this means that the recommendation system is built around uncovering latent structures in the playlists and track attributes. For example, collaborative filtering leverages implicit interactions between playlists and tracks, identifying commonalities among playlists with shared tracks, while content-based filtering focuses on the inherent characteristics of the tracks themselves.

2.1.1 Data Splitting

Most machine learning tasks involve dataset splitting into a train set, validation set and test set. Here $(\mathcal{P}, \mathcal{T})$ can be split too, but it only makes sense on a per playlist basis. Essentially, the recommendation system will always have access to the entire track dataset $\mathcal{T}$, after all it must know what is the possible set of items it can recommend. For each playlist $p \in \mathcal{P}$, we can split off a subset $p_{\text{test}} \subset p$ for testing, forming $\mathcal{P}_{\text{test}}$. So we remove tracks from the playlist. $\mathcal{P}_{\text{train}}$ contains then all the adjusted playlists

$p_{\text{train}} = p \setminus p_{\text{test}}$. For hyperparameter tuning, we can then perform the same procedure again on $\mathcal{P}_{\text{train}}$ to obtain a validation and reduced train set, or more generally apply K-fold cross-validation. To ensure that there are enough tracks after splitting we filtered out playlists with less than 100 tracks. This reduces the number of playlists to 124. We have included in Appendix A.1 the results from the experiment described in the following section but for the case where we filter out playlists with less than 50 tracks, which increases the number of playlists to 5132.

The idea behind doing the splitting like this is that we can evaluate a recommendation system by having it recommend tracks on this reduced playlists, and then check whether it matches the tracks that were removed from the complete playlist. The intuition is that this simulates how well the system predicts relevant tracks that a user ends up actually adding to their playlist. We can then take the average of the evaluation for each playlist. See Table 2.1 for an example.

Table 2.1: An example of the data-splitting procedure using a 50% split. The recommender recommends tracks based on the second table, after which one can examine whether the recommendations match those in the third table.

Playlist	Tracks
p_1	t_1, t_2, t_3, t_4
p_2	t_5, t_6, t_7, t_8
p_3	$t_9, t_{10}, t_{11}, t_{12}$
p_4	$t_{13}, t_{14}, t_{15}, t_{16}$
p_5	$t_{17}, t_{18}, t_{19}, t_{20}$

$\mapsto$

Playlist	Tracks
p_1	t_1, t_2
p_2	t_5, t_6
p_3	t_9, t_{10}
p_4	t_{13}, t_{14}
p_5	t_{17}, t_{18}

Playlist	Tracks
p_1	t_3, t_4
p_2	t_7, t_8
p_3	t_{11}, t_{12}
p_4	t_{15}, t_{16}
p_5	t_{19}, t_{20}

2.1.2 Collaborative Filtering

For our model-based collaborative filtering algorithm we must first consider the user-item interaction matrix, that for each playlist $p \in \mathcal{P}$ gives a rating or score for each possible item $t \in \mathcal{T}$. We define each element r_{ij} of the user-item matrix $R \in \mathbb{R}^{N \times M}$ (rows corresponding to playlists and columns corresponding to tracks) as

$$r_{ij} = \begin{cases} 1 & \text{if track } t_j \text{ is in playlist } p_i \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

Before we do anything one will quickly realize that $R \in \mathbb{R}^{887060 \times 134712}$ (assuming for simplicity no data splitting), so a matrix with about 119.5 billion elements. This is quite problematic in terms of space complexity. If in the best case we use only a single bit the matrix would still require about 14.9GBs of memory. However not all is lost, we can exploit the fact that R by definition is a *sparse* matrix. Most elements of the matrix will be zero because most playlist only have a small subset of all tracks. Specifically, we used the compressed sparse row (CSR) format for storing sparse matrices. The CSR format stores only the non-zero values, along with their row and column indices. This reduces the space complexity from $\mathcal{O}(NM)$ to $\mathcal{O}(L)$, where L is the number of playlist track interactions (e.g., number of nonzero entries).

The model based matrix factorization algorithm that we applied uses the SVD which was used to obtain missing ratings non-interacted items. When applied to the user-item matrix we obtain $R = U\Sigma V^\top$ where $U \in \mathbb{R}^{N \times N}$ contains the left singular vectors and $V \in \mathbb{R}^{N \times N}$ the right singular vectors $V \in \mathbb{R}^{M \times M}$ and $\Sigma \in \mathbb{R}^{N \times M}$ is the diagonal matrix containing the singular values. We use the top k latent factors (the singular vectors corresponding to the top k singular values): $\hat{R} = U_k \Sigma_k V_k^\top \in \mathbb{R}^{N \times M}$ where $U_k \in \mathbb{R}^{N \times k}$ can be interpreted as a playlist/user-feature matrix and $V_k \in \mathbb{R}^{M \times k}$ a track/item-feature matrix. The prediction ratings matrix is the reconstructed matrix $\hat{R} = U_k \Sigma_k V_k^{\top}$[†], which fills in the missing values for tracks that have no interaction with a given playlist. The recommendation for a given playlist is done by taking the top n tracks, filtering out tracks that are already in the playlist, with the highest predicted score $\hat{r}_{ij}$ where $j = 1, \ldots, n$.

As an example consider

$$R = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}. \tag{2.2}$$

[†]Due to the size of $\hat{R}$, which is no longer a sparse matrix, we only computed the relevant row on the fly $\mathbf{u}_k^p \Sigma_k V_k^\top \in \mathbb{R}^{1 \times M}$, which computes the predicted ratings for a playlist p for all M tracks.

We compute $\hat{R} = U_k \Sigma_k V_k^\top$ with $k = 2$:

$$\hat{R} = \begin{bmatrix} 1.1 & 0.1 & 0.75 & -0.25 \\ 0.1 & 1.1 & -0.25 & 0.75 \\ 0.85 & 0.85 & 0.35 & 0.35 \end{bmatrix}. \tag{2.3}$$

The intuitive interpretation is: playlist 1 gets recommended a new track 2 because playlist 3 also has track 1 but also track 2. Playlist 3 gets recommended track 3 and 4 because it shared tracks with playlist 1 and 2 which do have those tracks. The SVD here discovers latent relationships between the playlists and the tacks which allows us to predict interactions for unseen items based on playlists sharing certain tracks. Note that in contrast to the content-based approach we do not analyze the content of the tracks themselves.

2.1.3 Content-Based Filtering

Content-based filtering uses the attributes of the tracks to make recommendations. Assume we have some model that can give us feature vectors that capture the semantic meaning of the track in a semantic embedding space in $\mathbb{R}^D$, so a function $f_\theta : \mathcal{T} \to \mathbb{R}^D$ with parameters $\theta \in \mathbb{R}^P$. A recommendation can then be made based on a cosine similarity between vectors in the word embedding space. The corresponds to the aim of content-based filtering is to recommend new items with attributes similar to those that the user likes.

Let $f_\theta(t) = \mathbf{f}_t$ denote the feature vector of track $t \in \mathcal{T}$. We can now construct a playlist vector $\mathbf{f}_p = \frac{1}{|p|} \sum_{t \in p} \mathbf{f}_t$. A recommendation is then simply the top n tracks that are most similar to the playlist vector in the embedding space. It is common to use the cosine similarity:

$$d_{\cos}(\mathbf{f}, \mathbf{f}^*) = \cos(\theta) = \frac{\mathbf{f} \cdot \mathbf{f}^*}{\|\mathbf{f}\| \|\mathbf{f}^*\|} \in [-1, 1], \tag{2.4}$$

which is equal to the cosine of the angle between the vectors. Similarity is measured by to what degree the new track vector in the embedding space are pointing in the same direction as the playlist vector.

For constructing f_θ, there are a lot of options. For the purposes of the project we focused on computing word embeddings derived from the songs attributes. We make an assumption that these attributes are sufficient to characterize each track, without analyzing the lyrics, which saves significantly on computational complexity.

2.1.4 Learning Word Embeddings

To learn word embeddings we constructed a corpus of sentences based on the track attributes. For each track, we constructed a sentence using the string representation of the following attributes: “name“, “artists“, “danceablity“, “energy“, “tempo“, “release_date“, “acousticness“, “speechiness“, “instrumentalness“, “liveness“. The audio features are numeric with values in $[0, 1]$, so to create a proper string we discritize them as follows: For an attribute a we prepend the string “low“ if the value is in $[0, 0.4]$, “medium“ if in $[0.4, 0.7]$ and “high“ if in $[0.7, 1]$. So for example a “danceability“ of 0.7 becomes the string “high_danceability“. The corpus $\mathcal{C}$ derived from $\mathcal{T}$ becomes the set of track sentences. In choosing the track attributes we had filtered out attributes which had a high (> 0.5) negative or positive correlation with another attribute.

After obtaining $\mathcal{C}$, we used `Word2Vec` (Mikolov, Chen, Corrado, & Dean, 2013) to learn static word embeddings in $\mathbb{R}^D$. We do not require contextual understanding of words, as the track attributes meaning does not depend on their position in a sentence or the surrounding words. Therefore, using a computationally intensive model like `BERT` is unnecessary, and `Word2Vec`'s efficient, static embeddings are sufficient for our task.

The neural network architecture for `Word2Vec` is a shallow neural network with one hidden layer. The input layer takes in an one-hot encoded vector that has the same size as the vocabulary $|\mathcal{V}|$, which are the number of unique strings or tokens. The hidden layer contains D units, which is equal to the dimensionality of your word embeddings. After training, the word embeddings are stored in the learned weight matrix $W_{\text{emb}} \in \mathbb{R}^{|V| \times D}$ from the input layer to the hidden layer. Each row of W_{emb} corresponds to the learned D-dimensional embedding for a specific word in the vocabulary. At inference time, the word embeddings can be retrieved by computing $\mathbf{w}_n = W_{\text{emb}}^\top \mathbf{o}_n$, where $\mathbf{o}_n \in \{0, 1\}^{|V|}$ is the one-hot encoded representation of $w_n \in \mathcal{V}$. Two learning tasks, continuous bag of words (CBOW) and skip-gram are used

to learn W_{emb}. For both training tasks, the goal is to same, we wish to learn an embedding $\mathbf{w}_i \in \mathbb{R}^D$ for each $w \in \mathcal{V}$.

The first learning task, skip-gram, predicts context words given a target word. The goal is to maximize the probability[‡] of context words given the target word. Assuming all context words are independent this probability is given by:

$$\prod_{t=1}^{T} \prod_{-C \leq j \leq C, j \neq 0} \mathbb{P}(w_{t+j}|w_t), \tag{2.5}$$

where T is the total number of words in the corpus and C is the context window size (e.g., $C = 5$ words before and after the target). Using a form of gradient descent we maximize this probability by minimizing the average negative log likelihood:

$$\mathcal{L}(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-C \leq j \leq C, j \neq 0} \log \mathbb{P}(w_{t+j}|w_t), \tag{2.6}$$

and where the probabilty of a context word w_{t+j} given a target word w_i is computed using the softmax function:

$$\mathbb{P}(w_{t+j}|w_t) = \frac{\exp(\mathbf{w}_t \cdot \mathbf{w}_{t+j})}{\sum_{w \in \mathcal{V}} \exp(\mathbf{w}_t \cdot \mathbf{w})}. \tag{2.7}$$

The linguistic motivation behind skip-gram is that words appearing in similar contexts have similar meaning. To maximize the probability of context words given a target word, the neural network must make the embedding vectors of words appearing in similar context closer compared to words outside the context window due to the softmax in (2.7). Equivalently, related words $\mathbf{w}_t \cdot \mathbf{w}_{t+j}$ get a higher dot product, compared to unrelated words $\mathbf{w}_t \cdot \mathbf{w}$.

CBOW is similar to skip-gram but predicts the target word given its surrounding context words:

$$\mathbb{P}(w_t|w_{t-C}, \ldots, w_{t+C}) = \frac{\exp(\mathbf{w}_t \cdot \mathbf{h})}{\sum_{w \in V} \exp(\mathbf{w} \cdot \mathbf{h})}, \tag{2.8}$$

where $\mathbf{h} = 1/2C \sum_{j=-C}^{C} \mathbf{w}_{t+j}$ is the average of the embeddings of the context words.

To get a single embedding vector for a track, we averaged the word embeddings for each string in its sentence representation. To summarize:

$$f_\theta(t) = \mathbf{f}_t = \frac{1}{|S_t|} \sum_{w \in S_t} \mathbf{w}, \tag{2.9}$$

where S_t is the set of strings in the sentence representation of track t, and $\mathbf{w} \in \mathbb{R}^D$ is the word embedding of string w.

2.1.5 Hybrid Model

The hybrid model combines the strengths of collaborative and content-based filtering to overcome their individual limitations. To ensure that the scores of the collaborative filtering recommendation and content-based recommendation are within the same range we use the softmax function to turn the vector of recommendation scores for each track into a probability vector. We then took a simple approach and combined the scores of the collaborative filtering recommendation $h_{\text{collab}} : \mathcal{P} \times \mathcal{T} \to [0, 1]$ and content-based filtering recommendation $h_{\text{content}} : \mathcal{P} \times \mathcal{T} \to [0, 1]$ from both methods using a weighted sum:

$$h_{\text{hybrid}}(p, t) = \alpha \cdot h_{\text{content}}(p, t) + (1 - \alpha) \cdot h_{\text{collab}}(p, t), \tag{2.10}$$

where $\alpha \in [0, 1]$ balances the contribution of each method. α is the content weight and $1 - \alpha$ is the collaborative weight.

[‡]The probability of a word/token $\mathbb{P}(w)$, is shorthand for the more mathematically correct $\mathbb{P}(W = w)$, where W is a random variable with sample space $\mathcal{V}$.

2.2 Evaluating Recommendation Systems

For each playlist $p_{\text{test}} \in \mathcal{P}_{\text{test}}$, p_{test} is the subset of tracks reserved for testing, and $k = |p_{\text{test}}|$ represents the number of test tracks in the playlist. The recommender model generates a ranked list of n recommended tracks for playlist p, denoted as R_p.

The accuracy for playlist p, denoted as accuracy_p, is calculated as the fraction of correctly recommended tracks (hits) out of the total number of test tracks n. Mathematically, this is expressed as:

$$\text{accuracy}_p = \frac{|R_p \cap p_{\text{test}}|}{n},$$

where $|R_p \cap p_{\text{test}}|$ represents the cardinality of the intersection between the recommended tracks R_p and the test tracks p_{test}.

To evaluate the overall performance of the model, the average accuracy across all playlists in the test set $\mathcal{P}_{\text{test}}$ is computed. This is given by:

$$\text{avg\_accuracy} = \frac{1}{|\mathcal{P}_{\text{test}}|} \sum_{p \in \mathcal{P}_{\text{test}}} \text{accuracy}_p,$$

where $|\mathcal{P}_{\text{test}}|$ is the total number of playlists in the test set. This metric provides a measure of the model's ability to correctly recommend tracks across all playlists.

2.3 Hyperparameter Tuning

We selected the following hyperparameter set for tuning: For h_{collab}, we simply considered the number of latent factors k. For h_{content}, we took relevant `Word2Vec` hyperparameters such as D, the dimensionality of the word embeddings, window size C, the number of epochs e (e.g., the number of training passes over the text corpus), and $sg \in \{0, 1\}$, with $sg = 1$ indicating the use of skip-gram and $sg = 0$ for CBOW. We included D, which determines the expressiveness of the embeddings, and C, which controls the context size for capturing semantic or syntactic relationships. sg determines the training algorithm. The hyperparameter set for h_{hybrid} is $\Lambda = \{\alpha, k, D, C, e, sg\}$, where α is the hyperparameter for weighting h_{collab} and h_{content} ratings. We performed hyperparameter tuning using 2-fold cross validation with the dataset splitting approach described in section 2.1.1. See Table 3.1 for the range of values we used. We tuned the hybrid model by tuning h_{collab} and h_{content} separately and when the optimal hyperparameters for those were found we proceeded to tune α.

Table 2.2: Hyperparameters for Hybrid Recommender h_{hybrid}.

Hyperparameter	Range
k (Number of Latent Factors)	{5, 20, 50, 100}
D (Dimensionality of Embeddings)	{50, 100, 200}
C (Context/Window Size)	{5, 10, 15}
e (Number of Epochs)	{10, 20, 30}
sg (Skip-Gram vs CBOW)	{0, 1}
α (Ensemble weight)	{0.2, 0.5, 0.8}

2.4 Experiment Setup

After obtaining the optimal hyperparameters Λ_{opt} for the recommender, we fitted our recommender h_{hybrid} on $(\mathcal{P}_{\text{train}}, \mathcal{T})$. To summarize, this involves for h_{collab}, computing the SVD of the interaction matrix R in CSR format, and for h_{collab} learning the word embedding matrix W_{emb} using `Word2Vec`. After this was done we evaluated h_{collab} and h_{content} separately on $(\mathcal{P}_{\text{test}}, \mathcal{T})$, before evaluating h_{hybrid} on $(\mathcal{P}_{\text{test}}, \mathcal{T})$. At inference time, to get recommendation scores for all tracks for a playlist p, h_{collab} computes the relevant row $\mathbf{u}_k^p \Sigma_k V_k^\top$ in the predicted ratings matrix $\hat{R} = U_k \Sigma_k V_k^\top$ and h_{content} computes the track embeddings[§] $\{\mathbf{f}_t \mid t \in \mathcal{T}\}$ and playlist embedding $\mathbf{f}_p$ and computes a cosine similarity between $\mathbf{f}_p$ and each $\mathbf{f}_t$.

[§]It is more efficient to precompute the track embeddings $\mathbf{f}_t$ instead of computing them at inference time.

3 Results

The results of the hyperparameter tuning experiment can be found in Figure 3.1 and 3.2. The results of the experiment on the optimized recommenders can be found in Table 3.2. See also Appendix A.2 for a recommendation example and Appendix A.3 for a visualization of the semantic embedding space.

The hyperparameter tuning experiment yielded distinct performance patterns across different configurations. See Table 3.1 for the optimal hyperparameters found. Figure 3.1 shows that for content-based recommendations using `word2vec`, models with larger window sizes ($C = 15$) consistently outperformed those with smaller contexts ($C \in \{5, 10\}$). The optimal vector size emerged at 100 dimensions, achieving peak validation accuracy of 0.242 when combined with 30 training epochs and Skip-Gram architecture ($sg = 1$).

Three configurations dominated the top performance tier: 1) $D = 100/C = 15/e = 30$ (0.242 accuracy), 2) $D = 100/C = 10/e = 30$ (0.232), and 3) $D = 50/C = 15/e = 30$ (0.221). Skip-Gram ($sg = 1$) models consistently outperformed CBOW ($sg = 0$) counterparts by 12-18 percentage points across all parameter combinations. Training duration showed positive correlation with accuracy, with 30-epoch models outperforming 10-epoch versions by 6-9 percentage points.

Figure 3.2 reveals parameter sensitivity for the collaborative and hybrid recommender. h_{collab} achieved peak accuracy of 0.34 at $k = 20$ latent factors, with performance degrading at both smaller and larger k. For h_{hybrid}, the content-collaborative weighting parameter α showed maximal accuracy (0.41) at $\alpha = 0.8$.

Table 3.2 presents final evaluation results using optimized parameters. The hybrid system achieved superior test accuracy (0.41), outperforming the standalone collaborative (0.34) and content-based (0.16) recommendation system.

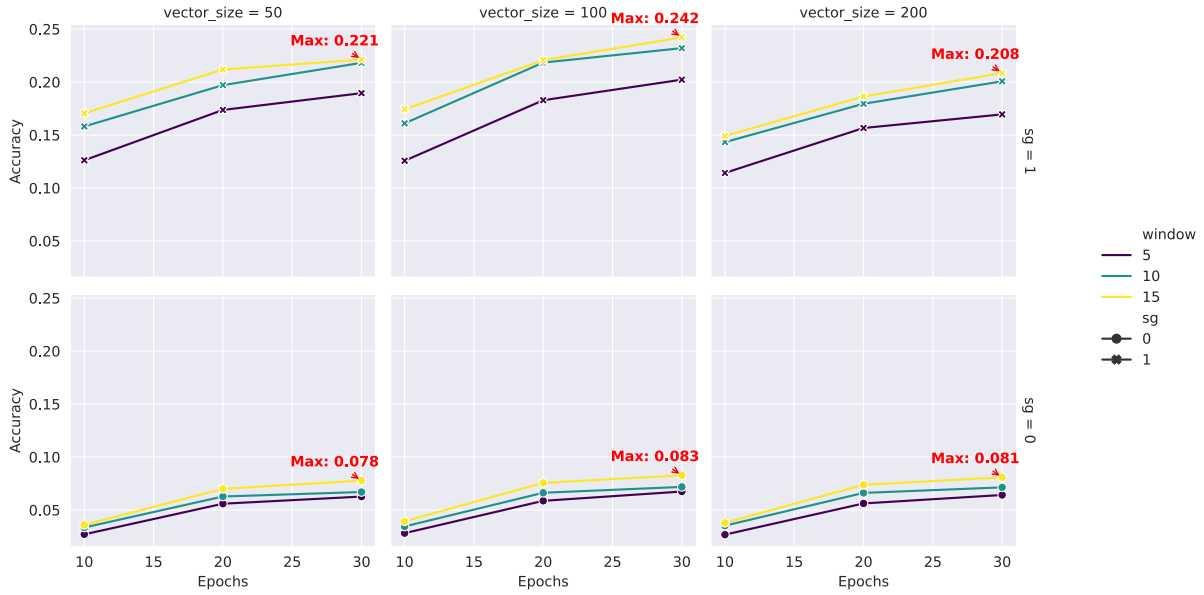
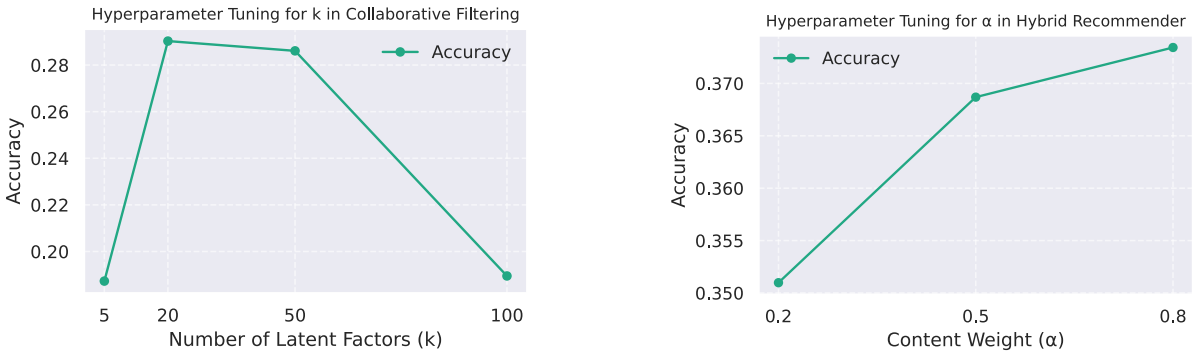


Figure 3.1: Hyperparameter tuning results for h_{content} for different hyperparameter configuration of `word2vec`. vector_size=D, window=C.

Table 3.1: Optimal hyperparameters found from the hyperparameter tuning experiment.

Hyperparameter	Range
k (Number of Latent Factors)	20
D (Dimensionality of Embeddings)	100
C (Context/Window Size)	15
e (Number of Epochs)	30
sg (Skip-Gram vs CBOW)	1
α (Ensemble weight)	0.8



(a) Accuracy vs. k

(b) Accuracy vs. α

Figure 3.2: Effect of Hyperparameters on Average Accuracy. Left: Tuning k for h_{collab}. Right: Tuning α for the h_{hybrid} given the optimal hyperparameters found for h_{collab} and h_{content}.

Table 3.2: Final Test Accuracy for the Recommender Systems using the optimal hyperparameters obtained from the hyperparameter tuning.

Recommender System	Test Accuracy
Collaborative Filtering	0.34
Content-Based	0.16
Hybrid	0.41

4 Discussion

4.1 Conclusion

The results of this study highlight the nuanced trade-offs involved in designing effective music recommendation systems. Unlike traditional classification tasks, where an objective accuracy can be used as an objective, recommender systems must provide recommendations that are loosely relevant to the user without explicitly being given what a relevant item is. Our experiment generally shows that our hybrid recommendation system, including the standalone collaborative filtering and content-based filtering recommenders, when asked to recommend songs on playlists where some of the tracks are removed, was able to recommend a certain percentage (20-40%) of the tracks that had been removed from playlists. This indicates that the recommender was able to recommend new tracks that are actually relevant out of the potential 134,712 tracks available in our dataset (see also Appendix A.2) .

It is worth emphasizing that this accuracy should not be interpreted the same way as in classification, while in classification we would want the highest possible accuracy on new samples, here maximizing the accuracy is not as important. The recommendations only need to be relevant, but relevance is not an objective measure. In fact, you may not want very high accuracy as you might also want to recommend novel songs that are slightly different in some ways.

Some takeaways from the hyperparameter tuning experiment: For content-based filtering the CBOW algorithm performs significantly worse than skip-gram when learning the word embeddings. In our corpus, each "sentence" is a fixed set of attribute strings (e.g., high_danceability, medium_energy, the song name), creating minimal/no syntactic/sequential context. CBOW relies on averaging neighboring words, which loses meaning when "context" is just a bag of unrelated categorical bins and the artist and song name. This is presumably why skip-gram works more effectively, as it does not average the context surrounding the target word and predicts the context words given the target word instead.

Increasing the dimensionality of feature vectors as captured by the k and D hyperparameters does initially improve performance as they increase, but decrease when they reach a certain threshold. Which can be attributed to the curse of dimensionaltity. For `Word2Vec`, it appears that larger context sizes improves performance. Most likely, a larger context window ($C = 15$) improves performance because it allows the model to capture all attribute relationships within a track's fixed "sentence" (e.g., linking 'high_danceability' to both the artist and 'medium_energy').

It is interesting that although the content recommendation system performs worse than the collaborative one, the hybrid performs better while giving relatively higher weight (0.8) to the content recommendation.

This could be due to error decorrelation, which means the models are making different errors. Collaborative filtering misses niche or cold-start tracks, while content-based filtering may struggle with popularity trends. Combining them compensates for each other's weaknesses, improving overall performance.

4.2 Limitations

Unlike our recommendation system, recommendation systems used in production by Spotify and YouTube operate in dynamic, non-static environments where the catalog of items (e.g., songs, videos) and user preferences evolve continuously. This introduces challenges not captured by our static dataset: (1) user preferences may shift over time, (2) long-term metrics like retention or repeat engagement cannot be measured, and (3) real-time feedback loops—where recommendations influence user behavior and vice versa—are absent. Future work could explore simulating non-static environments, such as incorporating temporal dynamics or evaluating the system's ability to adapt to evolving user tastes and new content.

A mores specific and important limitation that is especially relevant for the content based recommendation model is the reliance on high-level audio features like 'danceability' and 'accousticness'. These may provide too shallow a representation of songs compared to more detailed features that can be obtained by directly extracting features from audio data by means of spectral analysis. Spectral features can represent variations in instrumentation, harmonic complexity and dynamics much more directly than the features we had access to for this project. Incorporating deeper audio representations could improve recommendation quality by enabling finer-grained understanding of songs.

One of the fundamental design decisions behind the recommendation system described in this report is to treat playlists as proxies of users. This makes intuitive sense as playlists do usually reflect the musical preferences of the users that composed them. There can however be several reasons for why this need not be the case: Playlists are often context-specific (e.g., for workouts or parties), which can bias recommendations toward situational tastes rather than a user's overall preferences. Additionally, a single playlist typically represents only a subset of a user's musical interests. Finally collaborative or third-party-curated playlists may further misalign with individual preferences, introducing noise. This final point is particularly relevant for our implementation due to our choice of filtering out playlists based on a minimum number of tracks. Longer playlists are potentially more likely to be curated by 3rd parties. As a consequence our model's hyperparameters may be more optimized for generalized or context-specific trends, than personalized listening behaviors. As a result, while the system may perform well on the task of playlist continuation, its ability to deliver truly personalized recommendations remains limited.

Related to this is the risk of emphasizing existing patterns in the data, such as popular tracks or common genre associations, at the expense of novel or surprising recommendations that users might find engaging. For example, the system may struggle to recommend lesser-known tracks or cross-genre connections that align with a user's unique tastes but are underrepresented in the dataset.

4.3 Recomendations

For future projects it may be interesting to investigate mechanisms capable of balancing popularity and novelty in recommendations. An example of this could be to incorporate diversity metrics that prevent recommendation systems from continuously suggesting the same set of songs. By diversifying the music recommendations and exposing users to a broader range of genres and artists, user experience could be improved significantly.

A potential other future direction of this project is to move beyond treating playlists as proxies for users by incorporating explicit user data, such as listening history, skip behavior, and explicit feedback (e.g., likes/dislikes). Incorporating explicit feedback, such as thumbs-up or thumbs-down ratings, would provide direct signals about user preferences, enabling more accurate personalization. This shift would allow the system to tailor recommendations to preferences of individual users.

References

Burke, R. (2002, 11). Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, *12*. doi: doi:10.1023/A:1021240730564

Chen, C.-W., Lamere, P., Schedl, M., & Zamani, H. (2018). Recsys challenge 2018: Automatic music playlist continuation. In *Proceedings of the 12th acm conference on recommender systems* (pp. 527–528).

Gomez-Uribe, C. A., & Hunt, N. (2016, December). The netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manage. Inf. Syst.*, *6*(4). Retrieved from https://doi.org/10.1145/2843948 doi: doi:10.1145/2843948

Im, I., & Hars, A. (2007, November). Does a one-size recommendation system fit all? the effectiveness of collaborative filtering based recommendation systems across different domains and search modes. *ACM Trans. Inf. Syst.*, *26*(1), 4–es. Retrieved from https://doi.org/10.1145/1292591.1292595 doi: doi:10.1145/1292591.1292595

Imran, K., & Abdullah, N. (2010, 12). Learning materials recommendation using good learners' ratings and content-based filtering. *Educational Technology Research and Development*, *58*, 711-727. doi: doi:10.1007/s11423-010-9155-4

Iwahama, K., Hijikata, Y., & Nishida, S. (2004). Content-based filtering system for music data. *2004 International Symposium on Applications and the Internet Workshops. 2004 Workshops.*, 480-487. Retrieved from https://api.semanticscholar.org/CorpusID:10176896

Ko, H., Lee, S., Park, Y., & Choi, A. (2022). A survey of recommendation systems: recommendation models, techniques, and application fields. *Electronics*, *11*(1), 141.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). *Efficient estimation of word representations in vector space.* Retrieved from https://arxiv.org/abs/1301.3781

Papreja, P., Venkateswara, H., & Panchanathan, S. (2020). Representation, exploration and recommendation of playlists. In *Machine learning and knowledge discovery in databases: International workshops of ecml pkdd 2019, würzburg, germany, september 16–20, 2019, proceedings, part ii* (pp. 543–550).

Pazzani, M. J., & Billsus, D. (2007). Content-based recommendation systems. In P. Brusilovsky, A. Kobsa, & W. Nejdl (Eds.), *The adaptive web: Methods and strategies of web personalization* (pp. 325–341). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from https://doi.org/10.1007/978-3-540-72079-9_10 doi: doi:10.1007/978-3-540-72079-9_10

Salter, J., & Antonopoulos, N. (2006). Cinemascreen recommender agent: combining collaborative and content-based filtering. *IEEE Intelligent Systems*, *21*(1), 35-41. doi: doi:10.1109/MIS.2006.4

Su, X., & Khoshgoftaar, T. M. (2009). A survey of collaborative filtering techniques. *Advances in Artificial Intelligence*, *2009*(1), 421425. Retrieved from https://onlinelibrary.wiley.com/doi/abs/10.1155/2009/421425 doi: doi:https://doi.org/10.1155/2009/421425

Weihong, H., & Yi, C. (2006, 09). An e-commerce recommender system based on content-based filtering. *Wuhan University Journal of Natural Sciences*, *11*, 1091-1096. doi: doi:10.1007/BF02829216

A Additional Results

A.1 Results with Larger Number of Playlists

In Figure A.1, A.2 and Table A.2 can be seen the results from a more computationally expensive experiment using the same setup as described in section 2.3 and 2.4 but with 5132 playlists with a minimum of 50 tracks as opposed to 124 playlists with a minimum of 100 tracks. The overall conclusion that can be drawn are drastically different from the experiment we performed on a smaller number of playlists (note that the number of potential tracks to recommend remains the same). Two differences worth mentioning: there is a slight increase in performance when increasing the number of latent factors k to 100 compared to the 124 playlist experiment. This makes some sense as the number rows in the interaction matrix is equal to the number of playlists. Additionally, there is a decent performance drop for both skip-gram and CBOW in content-based filtering, though the performance gap between the two remains similar. Our reasoning for this occurring is that there are a larger number of smaller playlists included, and since our track embedding vectors only provide a very rough approximation of the meaning of the track, it is possible that the model struggles more to accurately represent playlists in the embedding space. It also highlights a potential flaw in our evaluation metric. If the test playlist is small, say in the extreme case it contains only one track, then the recommender will either get an accuracy of 1 and an accuracy of 0. This does not occur in practice because there are a minimum of 50 tracks though this is something worth considering.

What is interesting is that although there is a performance drop in content-based filtering, weighing its recommendation scores (at least ≤ 0.5 weight) in the hybrid recommender still gives a higher accuracy than just sticking with collaborative filtering.

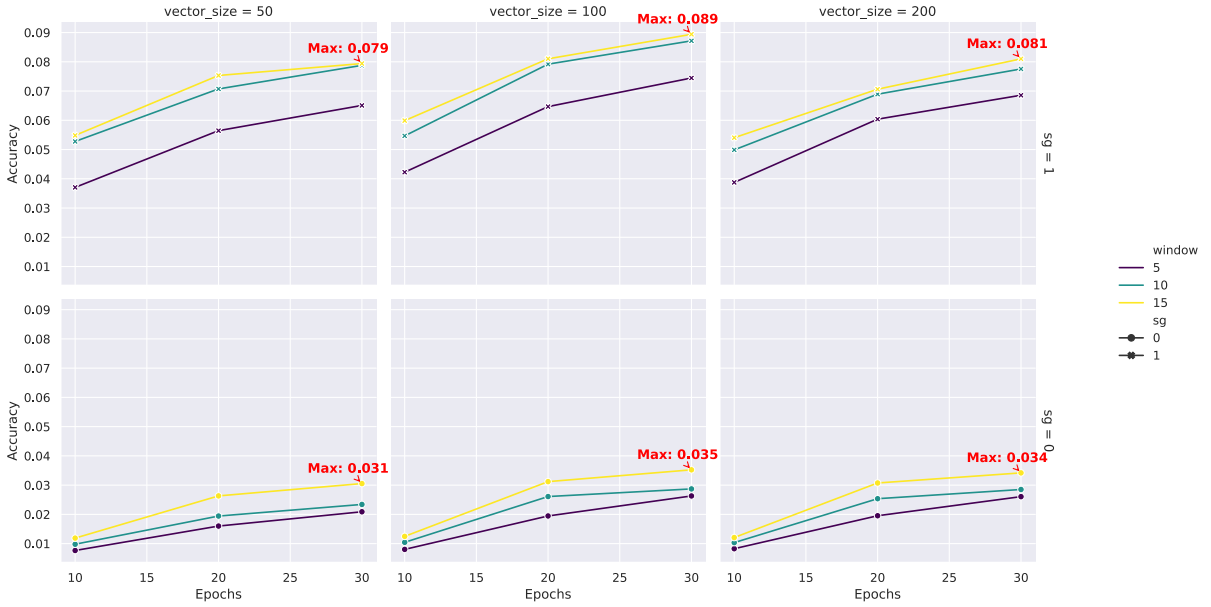
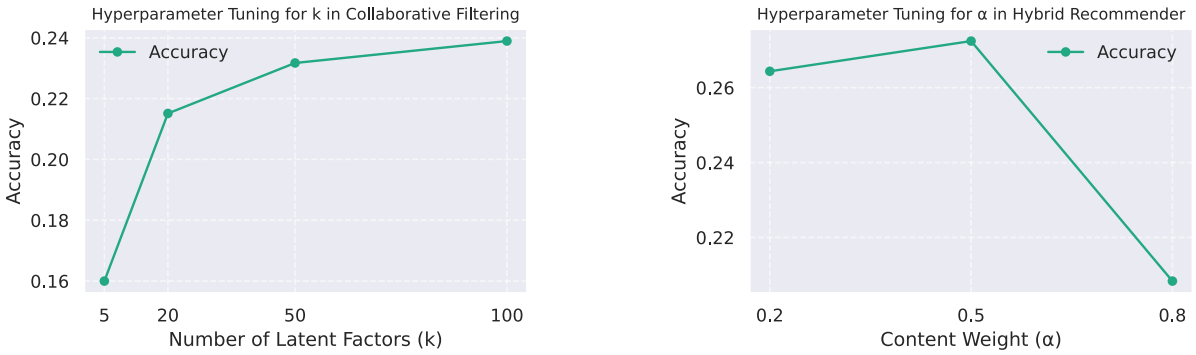


Figure A.1: Hyperparameter tuning results for h_{content} for different hyperparameter configuration of `word2vec`. vector_size=D, window=C.

Table A.1: Optimal hyperparameters found from the hyperparameter tuning experiment.

Hyperparameter	Range
k (Number of Latent Factors)	100
D (Dimensionality of Embeddings)	100
C (Context/Window Size)	15
e (Number of Epochs)	30
sg (Skip-Gram vs CBOW)	1
α (Ensemble weight)	0.5



(a) Accuracy vs. k **(b) Accuracy vs. α**

Figure A.2: Effect of Hyperparameters on Average Accuracy. Left: Tuning k for h_{collab}. Right: Tuning α for the h_{hybrid} given the optimal hyperparameters found for h_{collab} and h_{content}.

Table A.2: Final Test Accuracy for the Recommender Systems using the optimal hyperparameters obtained from the hyperparameter tuning.

Recommender System	Test Accuracy
Collaborative Filtering	0.27
Content-Based	0.05
Hybrid	0.29

A.2 Recommendation Example

Table A.3 shows concrete recommendation example using a playlist from our dataset. We used the hybrid recommender with the optimal hyperparameters from section 3. One observation is that "Money Won't Pay" and ''I'll Fall" appear to be Japanese pop songs (jpop) and one of the recommended songs, あかつき, by PASSEPIED is also jpop.

Table A.3: Example Playlist and Recommendations
All tracks available on Spotify for verification

Original Playlist Tracks	
Danse macabre	Camille Saint-Saëns, Slovak Radio Symphony Orchestra
Bent (Roi's Song)	DIIV
Mess Me Around	The Babies
Heart It Races (Cover)	Dr. Dog
Money Won't Pay	bo en, Augustus
I'll Fall	bo en, Coris
Crying in Public	Chairlift
We're Not Just Friends	Parks, Squares and Alleys
Weak	SWV
Brazil	Declan McKenna
Bethlehem	Declan McKenna
Recommended Tracks	
あかつき	PASSEPIED
Blindly	Kina Grannis
Complicate It	Tapping The Vein
Astro	Neil Finn
ANA	Mallive

A.3 Visualizing the Embeddings

Figure A.3 shows a 2D projection of the $D = 100$ embedding space learned by `Word2Vec` (skip-gram). The embeddings are visualized using t-SNE (t-Distributed Stochastic Neighbor Embedding), which preserves

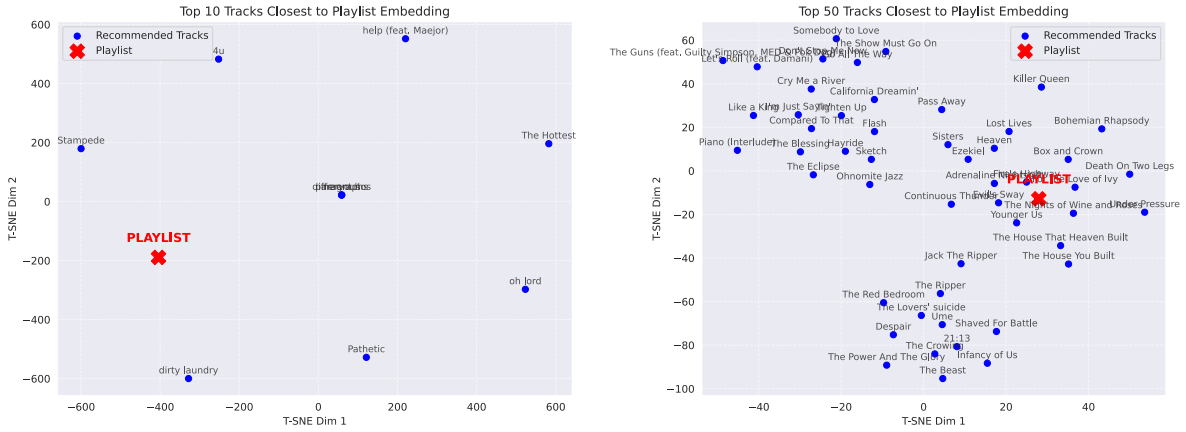


Figure A.3: Visualization of the $D = 100$ embedding space learned by `Word2Vec` (using skip-gram).

local structure while reducing dimensionality. Clusters in the plot suggest that the model captures meaningful relationships between tracks, potentially reflecting genre or artist similarities.

B Use of Large Language Models

Large Language Models (LLMs) such as ChatGPT and DeepSeek were used to assist in the creation of the tables and figures in the report. The LLM was given raw results from the experiment and for the tables was asked to generate a nice looking LaTeX table. For the plots it was given some preliminary Python code alongside the data and was asked "What is the best way to present this information?". After generation, the tables and Python code for the plots were adjusted as desired.