**VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY**

**INTERNATIONAL UNIVERSITY**



**PROJECT REPORT**

**Recommend Music System (Group 13)**

# CONTENT

## TASK

## I. Overview

## II. Dataset

2.1. Missing Data and Outliers

2.2. Outliers

2.3. Train/ test/ Validation

## III. Classification

3.1. Model: Colaborative Filtering

3.2. Model: K-Nearest Neighbors

3.3. Evaluation

## IV. Comparison

## V. References

**TASK**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Student ID** | **Contribution** | **Task** |
| Chau Khac Dinh Phong | ITDSIU20076 | 50% | Data preprocessing  (clean, train, split), visualize, build model, report, presentation |
| Ung Thi Hoai Thuong | ITDSIU20028 | 50% | Prepare data, presentation, build model, report |

## I. Overview

In this report, we delve into the comparative analysis of two models: Collaborative Filtering and K Nearest Neighbor to be able to find music playlists that are equivalent to the user's preferences. To find out user preferences and behavior, in this case, we use Spotify's music data set to analyze and provide music suggestions to users.

## II. Dataset

The data consists of 22 columns with attributes such as danceability, energy, key, loudness, etc., and a total of 42,305 entries. Particularly important columns may include:

* Danceability, energy, loudness, speechiness, acousticness, instrumentalness, liveness, valence, tempo: the properties of music measured by Spotify.
* Key, mode, time\_signature: theoretical musical properties.
* Duration\_ms: length of the track in milliseconds.
* Genre: music genre
* Song\_name, title: song name.
* ID, uri, track\_href, analysis\_url: links and IDs to refer to Spotify resources.

The Unnamed: 0 and title columns have appears to be unnecessary and also has missing data.

A screen shot of a computer

Description automatically generated

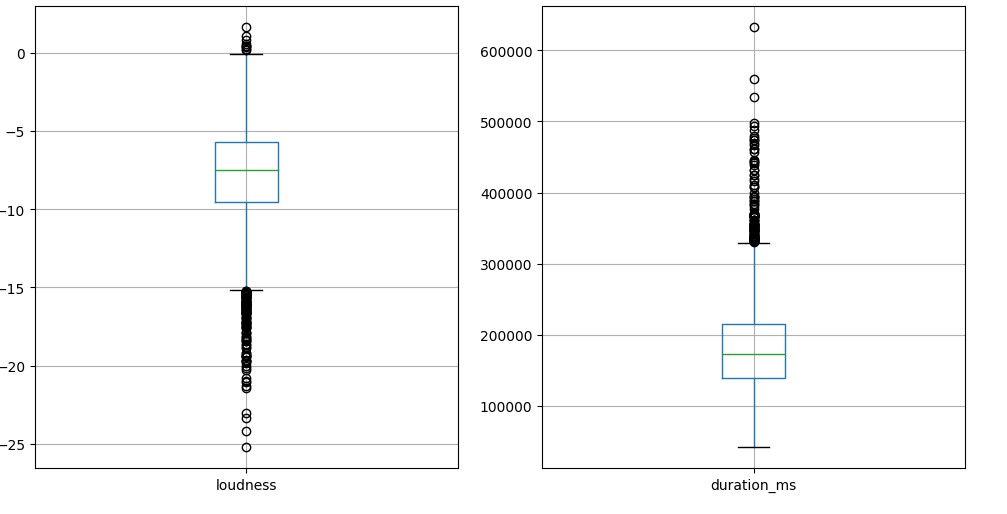
### **2.1. Missing Data**

Missing Data: The ‘Unnamed’ and ‘title columns’ have missing data. It may be necessary to consider whether to fill in values or remove these rows depending on the intended use.The data has been cleaned and updated as follows:

* Unnecessary column ('Unnamed: 0') has been removed.

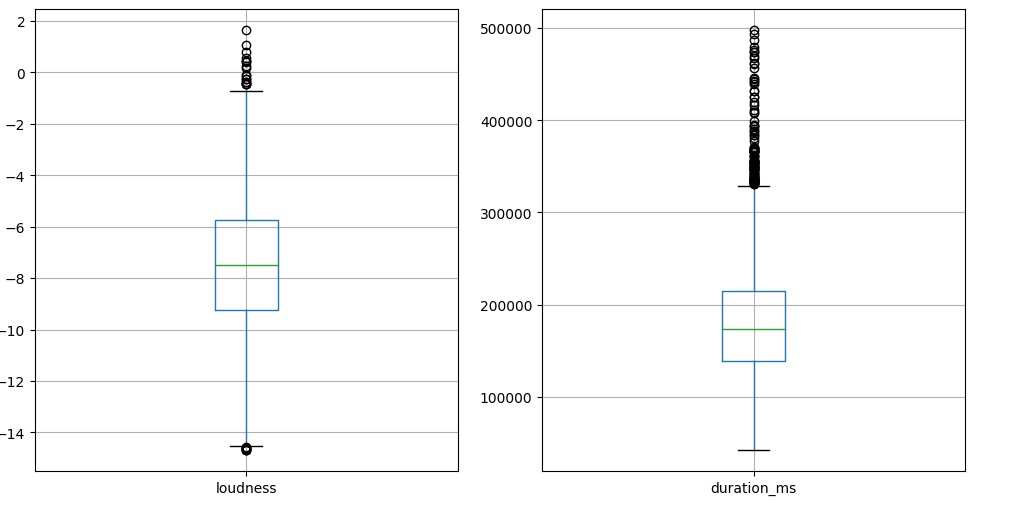
### **2.2. Outliers**

When looking at distribution plots or boxplots of variables, we often look for signs of outliers that may affect the quality of the model or analysis. "Loudness" and "duration\_ms" can often show some obvious exceptions like very low volume or unusual song lengths, which can skew the results. We will check for exceptions in loudness and song length (duration\_ms) to see if there are any abnormalities.



* Loudness: There are some outliers at the lower end of the distribution, showing that some songs have very low loudness levels compared to the majority of the data.
* Duration\_ms: There are also some big outliers in song length, with some songs having significantly longer lengths than average.

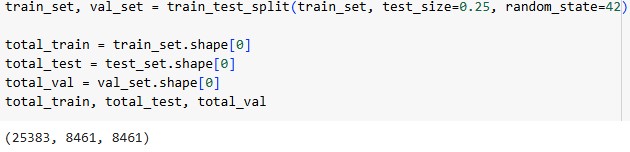
 Solution: We can handle these outliers by removing them or replacing them with the median value of the corresponding column.

  Loudness: Low loudness outliers have been replaced by medians, making this distribution more stable.

 Duration\_ms: Song length exceptions have been adjusted to a more reasonable value, eliminating values that are very large compared to normal.

#### 2.3. Train/ test/ validation

Split the data into training, testing, and validation sets. After dividing the data set, we obtain the results:



* Training set: 25,383 samples (about 60% of total data)
* Validation set: 8,461 samples (about 20% of total data)
* Test set: 8,461 samples (about 20% of total data)

The training set will be used to build and fine-tune the model, the validation set will be used to evaluate the model and fine-tune the parameters, and finally, the test set will be used to evaluate the final performance. of the model in a realistic environment.

## III. Classification

### **3.1. Model: Colaborative Filtering**

 Colaborative Filtering is a method of recommending system based on song characteristics without relying on other users' reviews or preferences, encompassing the following steps:

### 1. Initialize the Data Matrix

First, we load and preprocess the data. Then, select musical attributes to use as input features and create feature and label datasets.

### 2. Standardize the Data Matrix

Normalize data to ensure musical attributes are on the same scale

3. Calculate Similarity

Used to calculate the similarity matrix between songs.

### 4. Predict Ratings

Predicts and returns similar songs based on input song names.

 Below is the code snippet demonstrating these steps for the Colaborative Filtering:

import pandas as pd from sklearn.preprocessing import StandardScaler from sklearn.metrics.pairwise import cosine\_similarity from sklearn.model\_selection import train\_test\_split import numpy as np import matplotlib.pyplot as plt import re from sklearn.neighbors import NearestNeighbors

|  |
| --- |
| # Load data data = pd.read\_csv('/content/genres\_v2 (1).csv')    # Drop unnecessary columns and handle missing values data\_cleaned = data.drop(columns=['Unnamed: 0', 'title'])    # Select musical attributes to use as input features  features = ['danceability', 'energy', 'key', 'loudness', 'mode',  'speechiness',  'acousticness', 'instrumentalness', 'liveness', 'valence', 'tempo']    # Sampling a subset of data to reduce memory usage data\_subset = data\_cleaned.sample(n=15000, random\_state=42) data\_subset = data\_subset.drop\_duplicates(subset='id', keep='first')    # Divide the data into training set and test set train\_set, test\_set = train\_test\_split(data\_subset, test\_size=0.2,  random\_state=42) train\_set, val\_set = train\_test\_split(train\_set, test\_size=0.25,  random\_state=42) # 0.25 x 0.8 = 0.2    # Create data sets of features (features) and labels (labels) X\_train = train\_set[features] y\_train = train\_set['genre'] X\_val = val\_set[features] y\_val = val\_set['genre'] X\_test = test\_set[features] y\_test = test\_set['genre'] # Normalizing feature data scaler = StandardScaler()  X\_train = scaler.fit\_transform(X\_train)  X\_val = scaler.transform(X\_val)  X\_test = scaler.transform(X\_test)    feature\_data = scaler.fit\_transform(data\_subset[features]) print(scaler) |

|  |
| --- |
| print(X\_train) print(X\_val) print(X\_test)  # Calculating cosine similarity matrix similarity\_matrix = cosine\_similarity(feature\_data) print(similarity\_matrix)    # Creating a DataFrame for the similarity matrix similarity\_df = pd.DataFrame(cosine\_similarity(feature\_data),  index=data\_subset['id'], columns=data\_subset['id'])  #Predict the next song def get\_similar\_songs\_by\_name(song\_name, similarity\_df, data,  top\_n=7):  song\_name = clean\_song\_name(song\_name) if 'normalized\_song\_name' not in data.columns:  data['normalized\_song\_name'] =  data['song\_name'].apply(clean\_song\_name)    song\_data = data[data['normalized\_song\_name'] == song\_name] if song\_data.empty:  return f"Song name '{song\_name}' not found in the data."    song\_id = song\_data.iloc[0]['id'] if song\_id not in similarity\_df.index:  return f"Song ID {song\_id} not found in the similarity data."    sim\_scores = similarity\_df.loc[song\_id, :] sim\_scores\_sorted = sim\_scores.sort\_values(ascending=False) top\_similar\_ids = sim\_scores\_sorted.iloc[1:top\_n+1].index top\_similar\_songs = data[data['id'].isin(top\_similar\_ids)]    return top\_similar\_songs[['song\_name', 'genre']]    def clean\_song\_name(song\_name):  """Clean and normalize the input song name."""  # Ensure the song name is a string if song\_name is None: |

|  |
| --- |
| return "" # Return an empty string or some placeholder text for  None values song\_name = str(song\_name).strip() # Convert to string and strip  whitespace if not song\_name:  return "" # Return an empty string if song\_name is effectively  empty after stripping  # Convert to lower case before normalization song\_name = song\_name.lower()  # Normalize space around punctuation marks and ensure exactly one space follows any punctuation song\_name = re.sub(r'\s\*([,;.!?])\s\*', r'\1 ', song\_name).strip()  # Ensure single spaces between words song\_name = ' '.join(song\_name.split()) return song\_name      def get\_similar\_songs\_by\_name(song\_name, similarity\_df, data,  top\_n=7):  song\_name = clean\_song\_name(song\_name) if 'normalized\_song\_name' not in data.columns:  data['normalized\_song\_name'] =  data['song\_name'].apply(clean\_song\_name)    song\_data = data[data['normalized\_song\_name'] == song\_name] if song\_data.empty:  return f"Song name '{song\_name}' not found in the data."    song\_id = song\_data.iloc[0]['id'] if song\_id not in similarity\_df.index:  return f"Song ID {song\_id} not found in the similarity data."    sim\_scores = similarity\_df.loc[song\_id, :] # Make sure this is a  Series sim\_scores\_sorted = sim\_scores.sort\_values(ascending=False) #  This should now work without error top\_similar\_ids = sim\_scores\_sorted.iloc[1:top\_n+1].index |
| top\_similar\_songs = data[data['id'].isin(top\_similar\_ids)]    return top\_similar\_songs[['song\_name', 'genre']]  # Example of use user\_input\_song\_name = input("Enter a song name to find similar songs: ") similar\_songs = get\_similar\_songs\_by\_name(user\_input\_song\_name,  similarity\_df, data\_subset) print(similar\_songs) |

#### 3.2. Model: K-Nearest Neighbors

* K-Nearest Neighbors (KNN) for song suggestions: K-Nearest Neighbors (KNN) is a simple yet powerful algorithm in machine learning, widely used for classification and regression tasks. In this context, we will use KNN to find similar songs based on musical features
* Below is the code snippet demonstrating these steps for the K Nearest Neighbor:

### 1. Choose the number of nearest neighbors (K)

The user pre-defines the number K, i.e. the number of nearest neighbors to consider. In this example, we choose K=5.

### 2. Calculate distance

For each new data point that needs classification, KNN calculates the distance between this point and all data points in the training set. This distance is often measured using methods such as Euclidean, Manhattan or cosine distance.

### 3. Find K nearest neighbors

Select the K data points with the smallest distance to the new data point. These nearest neighbors are the songs that have the most similar features to the song to be searched.

### 4. Majority Voting

In the K nearest neighbors, the new data point will be labeled according to the majority of the labels of these neighbors. For example, if the majority of the K neighbors are of the "pop" genre, the new song will be classified as "pop".

|  |
| --- |
| import pandas as pd from sklearn.preprocessing import StandardScaler from sklearn.metrics.pairwise import cosine\_similarity from sklearn.model\_selection import train\_test\_split import numpy as np from sklearn.neighbors import NearestNeighbors import re    # Load data data = pd.read\_csv('/content/genres\_v2.csv')    # Drop unnecessary columns and handle missing values data\_cleaned = data.drop(columns=['Unnamed: 0', 'title']) data\_cleaned = data\_cleaned.dropna()    # Normalize feature data features = ['danceability', 'energy', 'key', 'loudness', 'mode',  'speechiness',  'acousticness', 'instrumentalness', 'liveness', 'valence', 'tempo']    scaler = StandardScaler() feature\_data = scaler.fit\_transform(data\_cleaned[features])    # Match the indices of feature\_data with data\_cleaned data\_cleaned = data\_cleaned.reset\_index(drop=True)    # Train KNN model knn = NearestNeighbors(n\_neighbors=5, algorithm='auto',  metric='cosine') knn.fit(feature\_data)    # Function to get similar songs using KNN def get\_knn\_similar\_songs(song\_index, knn\_model, data, k=5): |

|  |
| --- |
| distances, indices =  knn\_model.kneighbors([feature\_data[song\_index]], n\_neighbors=k+1) similar\_song\_indices = indices[0][1:] # Skip the first index as it is the item itself similar\_songs = data.iloc[similar\_song\_indices] return similar\_songs    # Example of use: Finding similar songs for a given song index song\_index = 0 # Index of the song you want to find similar songs for similar\_songs = get\_knn\_similar\_songs(song\_index, knn,  data\_cleaned) print(similar\_songs[['song\_name', 'genre', 'id']])    # Splitting data into train and test sets train\_set, test\_set = train\_test\_split(data\_cleaned, test\_size=0.2,  random\_state=42)  X\_train, X\_test = train\_test\_split(feature\_data, test\_size=0.2, random\_state=42)    # Function to find similar songs by name using the similarity dataframe def get\_similar\_songs\_by\_name(song\_name, similarity\_df, data,  top\_n=7):  song\_name = clean\_song\_name(song\_name) if 'normalized\_song\_name' not in data.columns:  data['normalized\_song\_name'] =  data['song\_name'].apply(clean\_song\_name)    song\_data = data[data['normalized\_song\_name'] == song\_name] if song\_data.empty:  return f"Song name '{song\_name}' not found in the data."    song\_id = song\_data.iloc[0]['id'] if song\_id not in similarity\_df.index:  return f"Song ID {song\_id} not found in the similarity data."    sim\_scores = similarity\_df.loc[song\_id] |
| sim\_scores\_sorted = sim\_scores.sort\_values(ascending=False) top\_similar\_ids = sim\_scores\_sorted.iloc[1:top\_n+1].index top\_similar\_songs = data[data['id'].isin(top\_similar\_ids)]    return top\_similar\_songs[['song\_name', 'genre']]    # Creating a DataFrame for the similarity matrix similarity\_matrix = cosine\_similarity(feature\_data) similarity\_df = pd.DataFrame(similarity\_matrix,  index=data\_cleaned['id'], columns=data\_cleaned['id'])    # Example of use: Finding similar songs by name user\_input\_song\_name = input("Enter a song name to find similar songs: ") similar\_songs = get\_similar\_songs\_by\_name(user\_input\_song\_name,  similarity\_df, data\_cleaned) print(similar\_songs) |

#### 3.3. Evaluation

* To evaluate the model, we use two indices: Average Precision and Average Recall to evaluate the effectiveness of the models in the recommendation and classification system.
* Average Precision, indicates the precision indicating the proportion of suggested songs.
* Average Recall, which determines what percentage of true matches the model retrieved from all possible matches.

### 1. Collaborative Filtering

|  |
| --- |
| def evaluate\_model(similarity\_df, data, test\_set, top\_n=7):  """Evaluate the collaborative filtering model.""" total\_precision = 0 total\_recall = 0 count = 0 |
| for index, row in test\_set.iterrows(): song\_name = row['song\_name'] true\_genre = row['genre'] similar\_songs = get\_similar\_songs\_by\_name(song\_name, similarity\_df, data, top\_n)    if isinstance(similar\_songs, str): # Song not found, skip this example continue    true\_positive = (similar\_songs['genre'] == true\_genre).sum() precision = true\_positive / len(similar\_songs) if len(similar\_songs) > 0 else 0 recall = true\_positive / top\_n    total\_precision += precision total\_recall += recall count += 1    avg\_precision = total\_precision / count if count != 0 else 0 avg\_recall = total\_recall / count if count != 0 else 0    return avg\_precision, avg\_recall avg\_precision, avg\_recall = evaluate\_model(similarity\_df, data\_subset, test\_set) print(f"Average Precision: {avg\_precision:.2f}") print(f"Average Recall: {avg\_recall:.2f}") |

 *The results we collected :*



### 2. K Nearest Neighbor

# Evaluation function def evaluate\_knn\_model(knn\_model, test\_data, feature\_data, data, k=5):

|  |
| --- |
| total\_precision = 0 total\_recall = 0 count = 0 for index, row in test\_data.iterrows(): try:  song\_index = data.index[data['id'] == row['id']].tolist()[0] if song\_index >= len(feature\_data): continue distances, indices =  knn\_model.kneighbors([feature\_data[song\_index]], n\_neighbors=k+1)  similar\_song\_indices = indices[0][1:] similar\_songs = data.iloc[similar\_song\_indices]['genre'] true\_positive = sum(similar\_songs == row['genre']) precision = true\_positive / len(similar\_songs) recall = true\_positive / 1 total\_precision += precision total\_recall += recall count += 1 except IndexError: print(f"An index error occurred with song\_index:  {song\_index}") avg\_precision = total\_precision / count if count != 0 else 0 avg\_recall = total\_recall / count if count != 0 else 0 return avg\_precision, avg\_recallThe result we collected : |

 *The result we collected:*



## IV. Comparison

|  |  |  |
| --- | --- | --- |
|  | **Collaborative Filtering** | **K Nearest Neighbor** |
| Average Precision | 0.20 | 0.36 |
| Average Recall | 0.20 | 1.80 |

The comparison of the two models, Collaborative Filtering and K Nearest Neighbor (KNN), shows that while they have similar average precision scores, they differ significantly in terms of recall. Collaborative Filtering has an average precision of 0.20, matching its average recall score, indicating consistent but limited performance in identifying relevant items. In contrast, KNN not only has a slightly higher precision of 0.36 but excels with a substantially higher recall of 1.80, suggesting it is much more effective at retrieving a larger proportion of relevant items.

**Conclusion:**

The superior recall of the KNN model indicates it is much more effective in identifying relevant items, making it a better choice for scenarios where capturing as many relevant recommendations as possible is more critical than the exact precision of those recommendations. This makes KNN particularly useful in environments where the cost of missing a relevant item is higher than the cost of including some irrelevant ones.

## V. References

1. Dataset: [Dataset of songs in Spotify (kaggle.com)](https://www.kaggle.com/datasets/mrmorj/dataset-of-songs-in-spotify)
2. Google Collab Link:

[Recommend Music System - Colab (google.com)](https://colab.research.google.com/drive/11WgQ2EJH06qznm_f6jxk0FbHrUYJd8wv?usp=sharing)

1. Document : [K-Nearest Neighbor(KNN) Algorithm - GeeksforGeeks](https://www.geeksforgeeks.org/k-nearest-neighbours/)

[Collaborative Filtering in Machine Learning -](https://www.geeksforgeeks.org/collaborative-filtering-ml/)

[GeeksforGeeks](https://www.geeksforgeeks.org/collaborative-filtering-ml/)