STATS 780
Project Report

Korede Adegboye - 400014008

14 April, 2022

Contents

Introduction

Problem Statement

The music platform publicly known as Spotify enables users to group songs together – create playlists. Traditionally, when the end of the playlist approaches, users are tasked to select another playlist, album, or song to listen to. This alteration could be time-consuming and can be seen as a distraction (eg, when driving, working, etc.). The data set contains a million playlists and their respective features. It is provided by Spotify and is publicly sourced from (Spotify 2018). The data is extracted from a subset of Spotify playlists and thus, reliable conclusions could be drawn. Much of the literature on the data set is given by the participants of the challenge. (Ludewig et al. 2018b) remarks that recommendation systems with a deep learning framework are most commonly used to solve this issue. Where it increases user experience by personalization and provides a seamless continuation of a user's playlists – curated playlists. Nevertheless, an agreeable disadvantage of this approach is the computational time and the resources required. My main concern in this study is to assess with whether or not, highly complex algorithms are justified relative to the application. An alternate aim of this project is to acquire an in-depth understanding of current issues and methods that arise with recommender systems. This study will give foundational value to the continuous development of recommender systems.

Proposed Solution

The proposed solution lies with a lightweight approach to provide automated recommendations given a user's playlist. We can generally define this as unsupervised learning and algorithms. Amid unlabeled data, the model looks to discover patterns and hidden structures. I believe that they help solve the problem since they look towards various ways to classify similarities of songs and playlists that do not already exist in the current playlist. Additionally, this procedure is concerned with information retrieval.

Methods

I intend to deliver a subset of recommended songs for a given playlist by utilizing exploratory data analysis, methodology, modeling, and analyzing the results. The comparison criterion aims to provide relative performances of slightly different methods to facilitate music recommendation systems. Used by (Spotify 2018), R-Precision is a score based on the proportion of relevant songs discovered. Afterward, the size of the training data, interpretability, and the number of features will be taken into account to evaluate the suitability of such a concept.

The basis of this scheme is given by unsupervised nearest neighbors search (NNS). I will govern this with 2 different approaches which are collaborative-filtering and content-based filtering. Collaborative filtering is chosen to be playlist based and content filtering will be track(item)-based. In NNS, the goal is to find the points closest to given point (x_i). This closeness measure is denoted

by various distance measures (ie., jaccard, cosine, euclidean, etc). Note, to train the model, k does not need to be tuned, rather it is selected at the time the suggestion is needed. Let us begin by outlining collaborative filtering. Say we have N_p playlists. Recommending a song for a playlist p_0 by the songs t_i. The indcator variable I_k is 1 if the a playlist p_k contains t_i and 0 other wise. We also denote the function $d(p_0, p_k)$ to be the distance (or similarity) measure.

$$rank(t_0, p_0) = \sum_{k \in N_p} d(p_0, pk) * 1_k$$

, where N_K is the set of K-neighbors (playlists) to consider at t_0 in the training set.

Now for the content-based filtering approach, we have

$$rank(t_0, p_0) = ...N(t_0) = ...d(t_0, t_i)$$

Essentially, given the songs t_0 that are contained in playlist p_0 will obtain the respective similarities of audio features. Note $t_0 \neq t_i$. The ... refer to filtering methods that we are unable to translate to symbols at this time (See Algorithm 1, Algorithm 2).

If the method were to be applied in a live environment (eg, Spotify application), interest is in the closet point, but when conveying recommendation metrics (quality of suggestion), k neighbors (tuning parameter) are adjusted for. For content-based filtering, we can tune K such that we obtain the most popular vote. Hence, NNS can suggest the next song by finding groups based on similar playlists or songs given dissimilarity (distance). Where, similar objects are found to be of smaller values – closer to zero. Considering songs are constantly added to Spotify, and users expand their playlists then we expect that the complexity of NNS will grow with the number of samples. Run time is $O(n * p)$ where n is the number of training samples and p is the number of features. The advantages of NNS, is its ability to fit arbitrarily complex functions, it can achieve a range of model complexities and the predictions ae easy to inspect. In contrast, the disadvantages are attributed to being computationally expensive and as more variables are added the performance decreases.

Algorithm 1 Collaborative filtering

1. Let x_0 denote the playlist the user is currently listening to
2. For $i = 1, \ldots k$, playlists:
   (a) Via a distance/similairy measure calculate the similarity of the i playlist to that of x_0
   (b) return the nearest playlist and the index
3. Go through each song in the nearest playlist to x_0. Filter for the next song to play, by choosing the songs that are played from the nearest player after a song that a user has played. So if $song_i$ is found then choose $song_{i+1}$ as the a reccomended song.

While we are not constrained to these distance (or similarity) measures, here is a selection of those

Algorithm 2 Content-based filtering

1. Let x_0 denote the playlist the user is currently listening to
2. For $i = 1, \dots k$, songs within the playlists:
   (a) Via a distance/similarity measure calculate the similarity of the ith song to that of songs not contained in the playlist
   (b) return the k nearest songs for each $song_i$ and the index
3. Choose the most popular songs that appear. Select the song(s) with the most similarity to reccommend

that we think will perform well with our data. For sets A, B we have

Jaccard Distance	$dist(A,B) = \frac{\lvert A \cap B \rvert}{\lvert A \cup B \rvert} = \frac{\sum A_i B_i}{\sum A_i^2 + \sum B_i^2 - \sum A_i B_i}$
Cosine Similarity	$sim(A,B) = \frac{\lvert A \cap B \rvert}{\sqrt{\lvert A \rvert \lvert B \rvert}} = \frac{\sum A_i B_i}{\sqrt{\sum A_i^2 * \sum B_i^2}}$
Euclidean Distance	$\lvert A - B \rvert = \sqrt{\sum (A_i - B_i)^2}$

Table 1: Playlists – Collaborative-based filtering. $n < p$, $k = \{$no: 0, yes:1$\}$. Source: Spotify milltion playlist data set

Feature extraction is not built into the method itself. So, an alternative would be to assess the result of principal component analysis (PCA). It is primarily a dimension reduction technique, where interest is in obtain a low-dimensional representation with as much variation of the data as possible. The principle components are produced such that an orthogonal directions and optimization allow the largest variances of the data to be detected.

Reproducible results will be attained through the random seed used to take samples and other functions in `python`. The interpretation of the results is straightforward as we expect the output to give the songs that are most similar (ie, the k nearest songs). In regards to the specific approaches, the time complexity of the collaborative approach given the playlists will be more than that of the content-based approach because it considers a search with more dimensions. To achieve the optimal bias-variance trade off and attain an accurate evaluation of these methods, we will use a train-validation split. Before applying the methods, we suspect the curse of dimensionality to be an issue. Recall, it is a problem caused by high dimensional data (a lot of features), where the number of samples needed to obtain accurate results is exponential. Evidently this is evident in NNS, as with more dimensions the further apart we expect playlists to be.

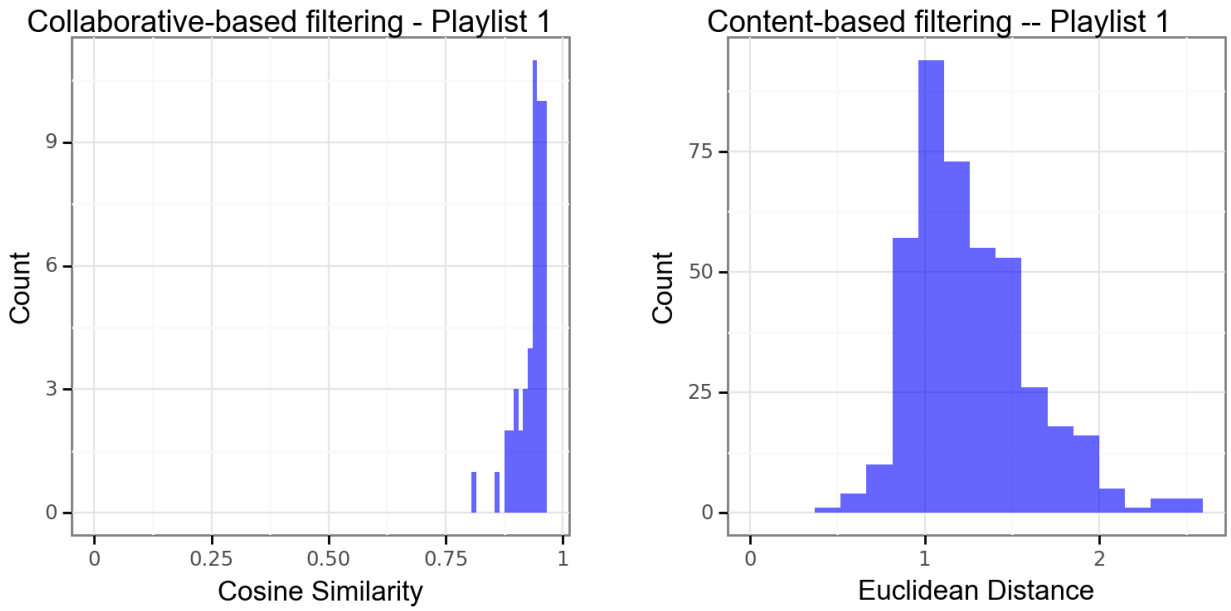

Figure 1: 3-dimensional vs 2-dimensional scatter plots to show how the neighborhood similarities decrease with higher dimension – cure of dimensionality

Results

Preliminary Results - Exploratory Data Analysis

The original data set is 40Gb of data, due to the large file and expected run times, a sample of 1000 playlists and/or 67503 songs (non-unqiue observations) will be considered. The listing of playlists is assumed to be random (eg, no noticeable order). Originally, the data is in JSON format, so a conversion to CSV is done. Through this process, the dictionary of songs and meta data features are extracted from the playlist. At the moment, we are considering 2 separate data sets for the playlist summaries and the songs of the playlist. This is done so summary metrics do not have to be re-calculated as often. Which results in a trade-off between space and time. There exist 11 and 9 variables for the playlist summaries and songs, respectively. The prominent variables for the songs are playlist identification, track name, artist name, and the track uniform resource identifier (uri), while the remaining aid in finding more information about the song.

Given the respective data sets (playlist summaries and songs), the majority of the data types are numeric (counts) and strings. To summarize the data, the distribution given by the number of artists is right-skewed (Figure 3, Suppl.). This indicates that we should expect fpr most playlists to show similarities. This is analogous for the number of albums and the number of songs. Majority of the playlist descriptions are missing. Note, use of descriptions also requires text analysis, which is not of concern for this study. Accustomed to the context outliers are not removed, since there is a sufficient amount of the count metrics outside of the there respective inter-quantile ranges.

	Song_1	Song_2	$\cdots$	Song_p
Playlist_1	k_1	k_{12}	$\cdots$	k_{1p}
Playlist_2	k_{2p}	k_{2p}	$\cdots$	k_{2p}
$\vdots$	$\vdots$			
Playlist_i	k_{ip}	k_{ip}	$\cdots$	k_{ip}
$\vdots$	$\vdots$		$\cdots$	
Playlist_n	k_{np}	k_{np}	$\cdots$	k_{np}

Table 2: Playlists – Collaborative-based filtering. $n < p$, $k = \{\text{no: } 0, \text{ yes:1}\}$. Source: Spotify milltion playlist data set

Given the approaches and the aim, it is acceptable to give the listener another song no matter the circumstance. In application, we suggest the monitoring of metrics to further understand and develop logic towards these situations. By visual inspection Figure 4 suggests there is a strong positive correlation between the number of tracks, number of albums and number of artists. This will be closely analyzed during feature extraction/selection. While text analysis could be done on the user's given names attributes, redundant variables should be removed to reduce space. Thus, playlist name, collaboration status, description, and smg duration are removed from the data sets. The most popular songs among playlists reveals that at most 60 out of the 1000 playlists contain the same song (Figure 5, Supplm.). This again gives insight to a general understanding of the similarities/differences across playlists. It can be extended to an investigation regarding which method is best for playlists that contain one to many of the most popular songs. Table 2 is constructed by a cross tabulation of whether or not a song exist in a playlist. This gives 1000 observations (playlists) and 34443 features, to be used for collaborative filtering.

To retrieve audio features, the Spotify API enables us to input up to 100 track uri's for each call. Such a process is only done once, since the features of a song are stationary. Afterwards, each songs features is inner joined to there respective playlists. This gives insight to the construction of Table 3. The audio features are seen as complimentary to the playlist, and thus an additional exploratory analysis in regards to the audio features is omitted. Although, we would like to note that it is plausible for audio analysis to assist with further development of our solution. Therefore, we use Table 3 for content-based filtering.

Application of methods

Given the dynamic size of playlists and their creation, the data structure for collaborative filtering is accessed. We determined that sparse matrices are memory-efficient data structures since many of the values are zero. This is the case since we have binary values. Sparse matrices remove any zero elements. This is also useful since the curse of dimensionality exists in this data set. Jaccard and cosine similarities were considered and determined to give analogous results. Jaccard dissimilarity is applicable here because it provides a proportion of the number of songs that 2 playlists have

	playlist ID	loudness	$\cdots$	danceability	energy	track ID
1	1	K_{12}	$\cdots$	$\cdots$	K_{1p}	track_1
1	1	K_{12}	$\cdots$	$\cdots$	K_{1p}	track_2
$\vdots$	$\vdots$					
2	2	K_{22}	$\cdots$	$\cdots$	K_{2p}	track_1
$\vdots$	$\vdots$		$\cdots$			
n	n	K_{n2}	$\cdots$	$\cdots$	K_{np}	track_1

Table 3: Playlists – Collaborative-based filtering. $n < p$, $k = \{\text{no: } 0, \text{ yes:1}\}$. Source: Spotify milltion playlist data set

Track		Audio Feature		
pos (in playlist)	artist name	danceability	energy	
track uri	artist uri	key	loudness	instrumentalness
album uri	duration ms	mode	speechiness	acousticness
pid	track artist	liveness	valence	tempo
track name	tempo	type	uri	analysis url
album name		time signature	duration ms	id

Table 4: Playlists – Collaborative-based filtering. $n < p$, $k = \{\text{no: } 0, \text{ yes:1}\}$. Source: Spotify milltion playlist data set

in common out of the total amount of songs. Jaccard distance performed slightly better but we discovered that Jaccard distances are not compatible with sparse matrices. Thus we proceed with cosine similarity moving forward. It is given by finding how much 2 playlists may overlap – taking their dot products.

Knowing that KNN does not perform well in high dimensions, feature selection was perceived as a valuable step to gaining performance. This was conducted on the audio features. By principal component analysis, the number of dimensions was reduced from 14 to 12. Those 12 components explain 98.84% of the variation (Figure 6, Suppl.). Of the first 4 components, we can report that song energy, danceability, artist popularity, and key are among the most impactful. We tried euclidean, Minkowski, and Manhattan distances, where consistent results are given. Therefore, the euclidean distance was the distance chosen. Since there remains a varying magnitude of values normalization was applied.

By Table 5, the average r-precision metric determined collaborative filtering to be relatively well-performed.

$$r - precision = \frac{\text{number of relevant songs retrieved}}{\text{total number of songs retrieved}} = \frac{\sum_{i=1}^{n} I_i}{n}$$

The average r-precisions involves 50 randomly selected playlists. It indicates the proportion of the top-R retrieved documents that are relevant, where r is the number of relevant documents for the

	Avg R-Precision
Collaborative	0.01510
Content (w/o PCA)	0.00182
Content (w/ PCA)	0.00141

Table 5: Evaluation

current query (N. 2009). On average, when the NNS determines a song to be relevant it does correctly 1.5% of the time. Thus far, this evaluation suggests NNS to not be the optimal approach for recommending songs. To further understand the results, distributions of the distances/similarities are given in Figure 2. For collaborative-based filtering, the playlists appear to be dissimilar. From an algorithmic viewpoint, there are just enough tail values (relatively close) to extract the most similar playlists. The distribution of content-based filtering is analyzed in the same manner. Notice, however, that we can spot ties among the measures. This is of concern since it is difficult to differentiate which is most relevant. Through evaluation, I also could conduct an interactive(informal) test of the results. While inputting a playlist into the methods, the song recommended differed substantially from what one would expect. Note that, among 5 different playlists (tests) I did not experience this result for all.

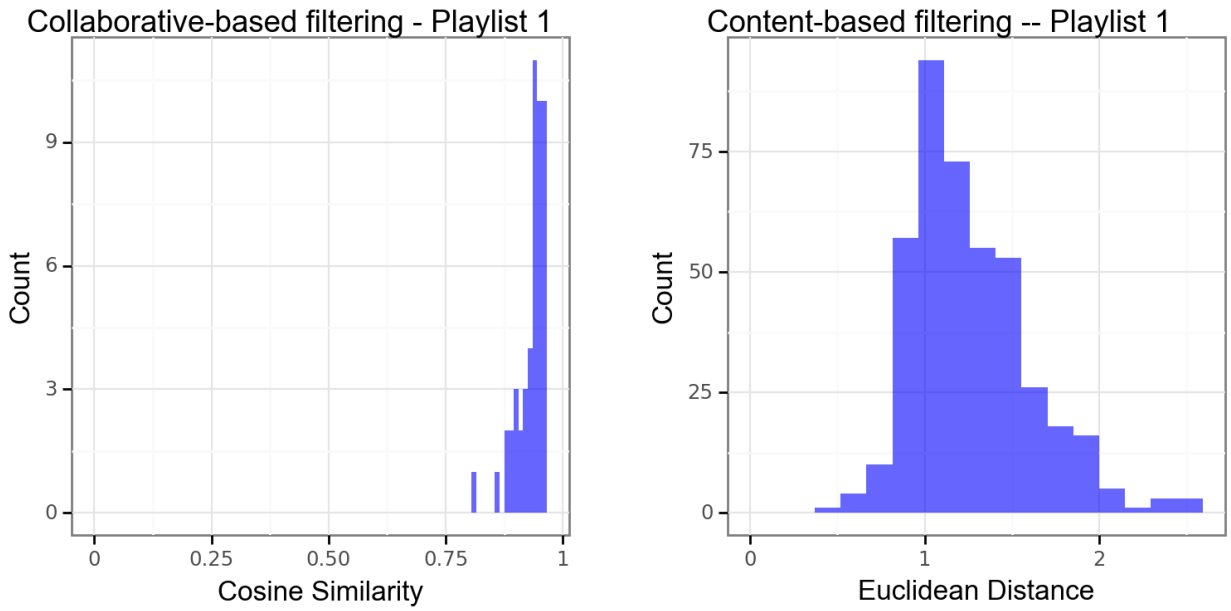

Figure 2: Collaborative vs Content based filtering visualization of results. 1000 randomly selected playlists for training and metrics given by 50 random selected playlists

Conclusion

The methods considered have given a baseline for patterns within a user playlist. When a user has a few songs within their playlist, a collaborative approach is useful. When there are many songs, then use content-based filtering. A hybrid of the two would be suitable. For instance, after the nearest playlist is found, NNS can then be conducted on songs in that playlist relative to the current user's playlist. The authors of the paper expand on this course of action. They mention hybridization to be a combination of lightweight solutions. The authors were able to achieve an r-precision of approximately 18% with 990,000 playlists (2.3 million tracks). Interpretation of "similarity" can be misleading. Songs can be deemed to be "close" but upon listening to the recommended songs, it is much different. Therefore, if correct both methods give fairly interpretable results but require different logic to get relevant recommendations.

Simpler techniques rely on algorithmic approaches, where further filtering is needed to get precise recommendations. Another consideration for filtering is the popularity among nearest neighbors. For instance, by taking 20 neighbors for each song it is likely that if a playlist is homogeneous the same neighbors will appear again. Naturally, this is suggestive of a rank-based measure that is included in the motivated paper. Another feature I should have considered was the main genre of the song, where we believe this is would be informative for many playlists.

The unsatisfactory results obtained are largely due to the sample size used. This remark indicates that the learning rate for the Nearest Neighbor search is slow. In the case of the collaborative approach, it will benefit from more playlists because due to the law of large numbers it is plausible that similar playlists can be found with few differences. But in the case of content-based filtering, there appear to be many songs that can be observed to be similar. While code is widely available for recommender systems, each problem and understanding remains different. Therefore, the algorithms that I programmed may not be up to par with industry standards but I believe coding from scratch gives more insight into the decisions made when it comes to recommendation systems – familiarity. I take this project experience as one that is coherent to the applications of a project lifecycle and can thus be further developed. There is ongoing research for unsupervised dimensionality reduction techniques. Some are situated with unsupervised neighborhood component analysis for distance metric learning.

References

C. Langensiepen, A. Cripps, and R. Cant. 2018. *Using PCA and k-Means to Predict Likeable Songs from Playlist Information.* 2018 UKSim-AMSS 20th International Conference on Computer Modelling; Simulation (UKSim). https://ieeexplore.ieee.org/document/8588173.

Ihler, Alex. 2010. *Evaluating Recommender Systems.* University of California Irvine – Department of Mathematics. https://www.math.uci.edu/icamp/courses/math77b/lecture_12w/pdfs/Chapter%2007%20-%20Evaluating%20recommender%20systems.pdf.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2021. *An Introduction to Statistical Learning V2: With Applications in r.* Springer.

Ludewig, Malte, Iman Kamehkhosh, Nick Landia, and Dietmar Jannach. 2018a. *Effective Nearest-Neighbor Music Recommendations.* https://github.com/rn5l/rsc18.

———. 2018b. "Effective Nearest-Neighbor Music Recommendations." In, 1–6. https://doi.org/10.1145/3267471.3267474.

N., Craswell. 2009. *R-Precision.* Encyclopedia of Database Systems. Springe. https://doi.org/10.1007/978-0-387-39940-9_486.

Spotify. 2018. *AIcrowd | Spotify Million Playlist Dataset Challenge | Challenges.* AIcrowd. https://www.aicrowd.com/challenges/spotify-million-playlist-dataset-challenge.

Wong, Phoebe. 2009. *Spotify Million Playlist Challenge 2018 - k-NN Collaborative Filtering, Content-Based Filtering, ALS Matrix Factorization, KMeans Clustering.* Github. https://github.com/phoebewong/spotify-teamNPK.

Supplementary Material

```
library(reticulate)
options(reticulate.repl.quiet = TRUE)
knitr::knit_engines$set(python = reticulate::eng_python)
```

```
# data processing
import json
import os.path
import spotipy
from spotipy.oauth2 import SpotifyClientCredentials
import numpy as np
import pandas as pd
import datetime
import warnings
from random import sample
from random import seed
# viz
import seaborn as sns
from plotnine import *
import matplotlib.pyplot as plt
import patchworklib as pw
# method
from sklearn.neighbors import NearestNeighbors
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.neighbors import NeighborhoodComponentsAnalysis
from sklearn.preprocessing import StandardScaler
from pca import pca
from scipy.sparse import csr_matrix
```

```
# data to csv and transformation
# os.path.isfile(fname)
listOfPlaylists = [] # list of dataframes
fileName = "spotify_million_playlist_dataset/data/mpd.slice.0-999.json"
data = json.load(open(fileName))
playlists = pd.DataFrame(data["playlists"])
for i in range(len(playlists)):
  playlist = playlists.iloc[i]
  currPID = playlist['pid']
  songs = playlist['tracks']
  currTracks = pd.DataFrame.from_dict(songs)
  currTracks['pid'] = currPID
  listOfPlaylists.append(currTracks)

playlist_summaries = playlists.drop(columns = "tracks", axis = 1)
playlist_df = pd.concat([listOfPlaylists[i] for i in range(len(listOfPlaylists))], ignore_index=True)

playlist_df.to_csv('data/mpd_0-999.csv', index = False)
playlist_summaries.to_csv('data/pSummaries_0-999.csv', index = False)
```

```
# Exploratory data analysis - 1
playlists = pd.read_csv("data/mpd_0-999.csv")
pSummaries = pd.read_csv("data/pSummaries_0-999.csv")
## variable names
list(playlists.columns)
```

```
## ['pos', 'artist_name', 'track_uri', 'artist_uri', 'track_name', 'album_uri', 'duration_ms', 'album_name', 'pid']
```

```
list(pSummaries.columns)
## summaries - omits strings columns..
```

```
## ['name', 'collaborative', 'pid', 'modified_at', 'num_tracks', 'num_albums', 'num_followers', 'num_edits', 'duration_ms', 'num_artists', 'description']
```

```
pDescribe = playlists.describe()
psDescribe = pSummaries.describe()
## number of observations
playlists.shape
```

```
## (67503, 9)
```

```
pSummaries.shape
## data types
```

```
## (1000, 11)
```

```
playlists.dtypes
```

```
## pos             int64
## artist_name    object
## track_uri      object
## artist_uri     object
## track_name     object
## album_uri      object
## duration_ms     int64
## album_name     object
## pid             int64
## dtype: object
```

```
pSummaries.dtypes
# correlation analysis
```

```
## name             object
## collaborative      bool
## pid               int64
## modified_at       int64
## num_tracks        int64
## num_albums        int64
## num_followers     int64
## num_edits         int64
## duration_ms       int64
## num_artists       int64
## description      object
## dtype: object
```

```
pSummaries["collaborative"] = pSummaries["collaborative"].astype(int)
pSummaries_slim = pSummaries[['collaborative', 'modified_at', 'num_tracks',
  'num_albums', 'num_followers', 'num_edits', 'duration_ms', 'num_artists']]
cormat = pSummaries_slim.corr()
cormat = cormat.reset_index()
cormat1 = cormat.melt(id_vars = 'index')

# outliers analysis - NA

# handling missing values
playlists.isna().sum()
```

```
## pos            0
## artist_name    0
## track_uri      0
## artist_uri     0
## track_name     0
## album_uri      0
## duration_ms    0
## album_name     0
## pid            0
## dtype: int64
```

```
pSummaries.isna().sum()
```

```
## name               0
## collaborative      0
## pid                0
## modified_at        0
## num_tracks         0
```

```
## num_albums         0
## num_followers      0
## num_edits          0
## duration_ms        0
## num_artists        0
## description      980
## dtype: int64
```

```
# Exploratory data analysis - 2 not needed ??
```

```
(
ggplot(pSummaries, aes(x='num_artists'))
+ geom_histogram(fill = "blue", alpha = 0.6, bins = 24)
+ labs(y = "Count", x = "Number of Artists",
  title = "Histogram - Number of artists per playlist")
+ theme_bw()
).draw()
```

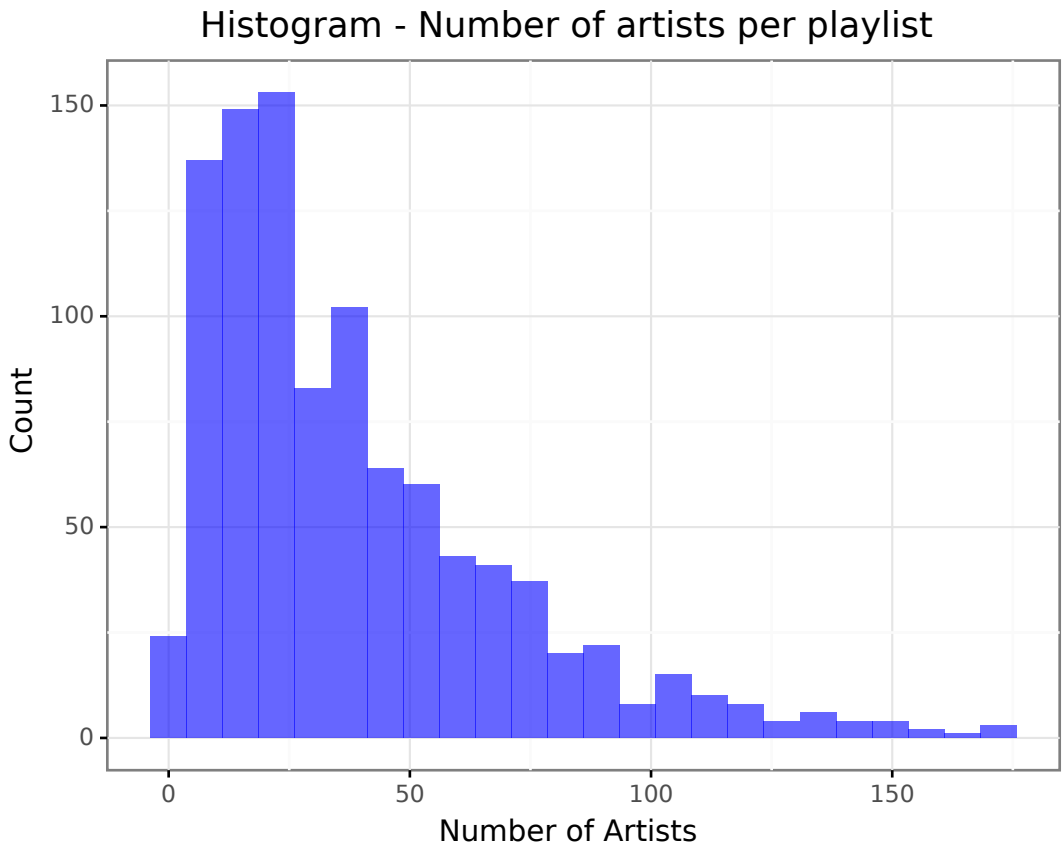

Figure 3: Histogram – Number of artists per playlists

```
# correlation
p3 = (
ggplot(cormat1, aes(x = "index", y = "variable", fill = "value"))
+ geom_tile()
+ labs(x = "", y = "", fill = "Corr")
+ scale_fill_gradient(low="white", high="blue")
+ theme(axis_text_x = element_text(angle = 90))
)
# ggsave(p3, "images/corrPlot.png")
```

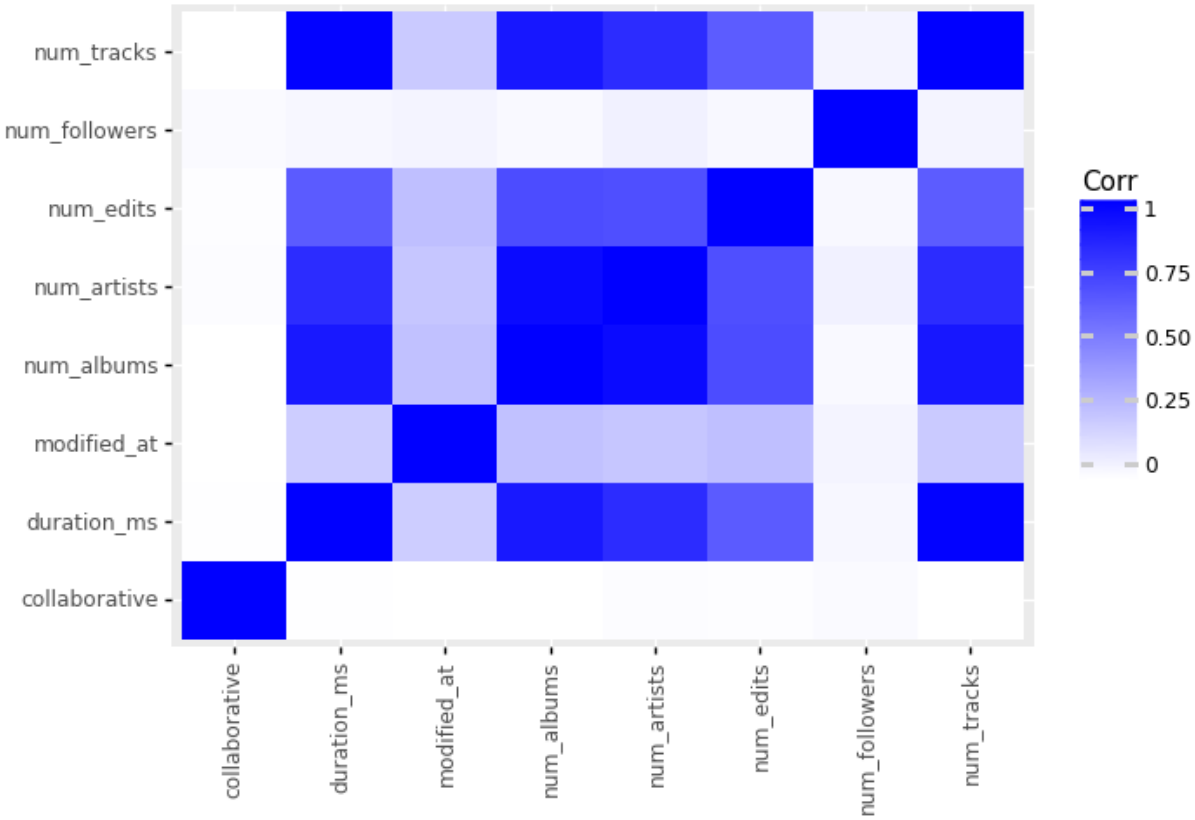

Figure 4: Correlation plot of 1000 playlist summaries

```
# Most popular songs in existing playlists - bar plot
playlists["track_artist"] = playlists["track_name"] + " - " + playlists["artist_name"]
counts = pd.DataFrame(playlists.track_artist.value_counts()[:10])
counts = counts.reset_index()
p5 = (
ggplot(data = counts)
+ geom_bar(aes(y = "track_artist", x = "index"),
    stat = "identity", fill = "blue", alpha = 0.6)
+ coord_flip()
+  theme_bw()
+ labs(title = "Top 10 Songs given across 1000 playlists",
          x = "Count",  y = "Song - Artist")
)

# ggsave(p5, "images/top10.png")
```

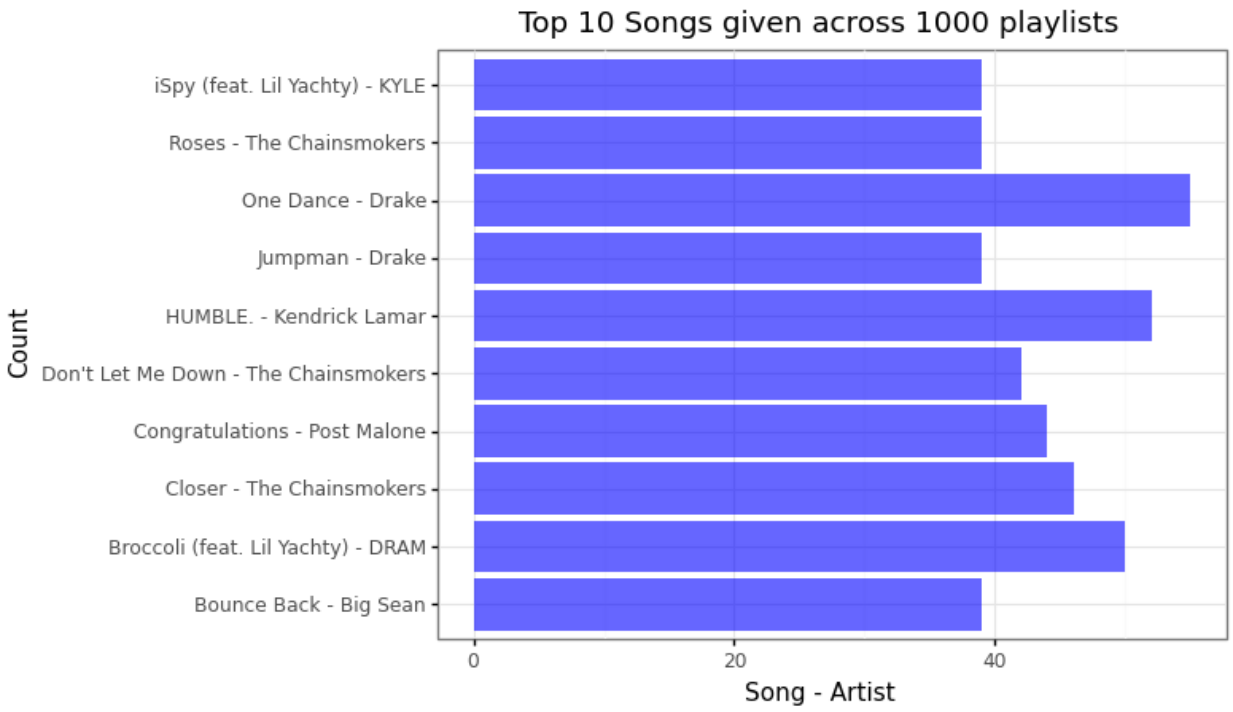

Figure 5: Top 10 Songs from 1000 playlist summaries. Source:

```
# get 25% songs from each playlist
collabF = pd.crosstab(playlists.pid, playlists.track_uri)
train, val = train_test_split(playlists, test_size=0.25, random_state=1, stratify = playlists.pid)
train_bin = pd.crosstab(train.pid, train.track_uri)
train_bin_sparse = csr_matrix(train_bin)
knnRec = NearestNeighbors(metric = "cosine", algorithm = "brute", n_neighbors = 5)
nbrs = knnRec.fit(train_bin_sparse)
```

```
# function for getting recommended songs and scores
def getCollabFilteringPrecision(pid, nbrs, playlists, train_bin, k):
  metric, idx = nbrs.kneighbors(np.array(train_bin.loc[pid]).reshape(1,-1), n_neighbors=k)
  currVal = val[val.pid == pid].track_uri
  alreadyPlayed = train_bin.iloc[idx[0][0]]
  playlistToConsider = train_bin.iloc[idx[0][1]]
  recTracks = []
  remaining = []
  # get the songs played after each relevant song
  # or get based on the artists that occurs most often?
  for i in range(len(playlistToConsider)-1):
```

```
    if (playlistToConsider[i] and alreadyPlayed[i]) and not playlistToConsider[i+1]:
      recTracks.append(playlistToConsider.index[i+1])
    elif (playlistToConsider[i] and not alreadyPlayed[i]):
      remaining.append(playlistToConsider.index[i])

  recTracks.extend(remaining)
  recTracks = recTracks[: len(currVal)]
  rprecision = currVal.isin(recTracks).sum()/currVal.shape[0]
  return rprecision, metric
```

```
# collaborative-based filtering
warnings.filterwarnings('ignore')
# get avg r-precision for 50 random playlists
seed(1)
randomPID = sample(train_bin.index.tolist(), 50)
avgPrec = 0.0
for pid in randomPID:
  avgPrec = avgPrec + getCollabFilteringPrecision(pid, nbrs, playlists, train_bin, 2)[0]
avgPrec
```

```
## 0.7551371768532785
```

```
avgPrec = avgPrec/50
avgPrec
```

```
## 0.015102743537065571
```

```
metric = getCollabFilteringPrecision(0, nbrs, playlists, train_bin, 50)[1]
metrics = pd.DataFrame({"metric": metric[0][1:]})
p2 = (
ggplot(data = metrics)
+ geom_histogram(aes(x = "metric"), fill = "blue", alpha = 0.6, binwidth = 0.01)
+ labs(title = "Collaborative-based filtering - Playlist 1",
       x = "Cosine Similarity",  y = "Count")
+ theme_bw()
+ xlim(0, None)
)

# ggsave(p2, filename = "pres/resCollabFiltering.png")
```

```
# get song features
## connect to api
CLIENT_ID = '3a54c80d02cd449db2714b2d90603a08'
CLIENT_SECRET = '872cafa58750401f80797f104a8815a5'
spotify = spotipy.Spotify(client_credentials_manager=SpotifyClientCredentials(CLIENT_ID, CLIENT_SECRET))

len(playlists.track_uri.unique())
iter = list(range(100,len(playlists.track_uri.unique()), 100))
iter.append(len(playlists.track_uri.unique()))
uniqueSongs = playlists.track_uri.unique()
allSongs = []
start = 0
for end in iter:
  results = spotify.audio_features(uniqueSongs[start:end])
  currTracks = pd.DataFrame.from_dict(results)
  allSongs.append(currTracks)
  start = end
## combine all
allSongs_df = pd.concat([allSongs[i] for i in range(len(allSongs))], ignore_index=True)
# export to csv
allSongs_df.to_csv('data/songFeatures.csv', index = False)
```

```
# preprocessing
warnings.filterwarnings('ignore')
audioFeatures = pd.read_csv("data/songFeatures.csv")
audioFeaturesAndP = pd.merge(left = playlists[["pid", "track_uri", "artist_name"]], right = audioFeatures,
  left_on = "track_uri", right_on = "uri", how = "left")
# artist popularity -- feature engineering
pop = pd.DataFrame({"artist_count":audioFeaturesAndP.artist_name.value_counts()})
pop["artist_name"] = pop.index
```

```
audioFeaturesAndP = pd.merge(left = audioFeaturesAndP, right = pop, on = "artist_name", how = "left")

contentF = audioFeaturesAndP.drop(["id", "track_href", "analysis_url", "type", "uri", "artist_name"], axis = 1)
trainAF, valAF = train_test_split(contentF, test_size=0.25, random_state=1, stratify = playlists.pid)
trainAF = trainAF.reset_index()
trainAFSlim = trainAF.drop(["track_uri", "pid", "index"], axis = 1)

x = StandardScaler().fit_transform(trainAFSlim)
X = pd.DataFrame(data=x, columns=trainAFSlim.columns)
```

```
# Disable printing
class blockPrint():
  def __enter__(self):
    self._original_stdout = sys.stdout
    sys.stdout = open(os.devnull, 'w')

  def __exit__(self, exc_type, exc_val, exc_tb):
    sys.stdout.close()
    sys.stdout = self._original_stdout
```

```
# Principal component analysis
with blockPrint():
  model = pca()
  # Fit transform
  out = model.fit_transform(X)

print(out["topfeat"])
```

```
##       PC           feature   loading  type
## 0    PC1            energy -0.506492  best
## 1    PC2      danceability -0.567884  best
## 2    PC3      artist_count  0.490994  best
## 3    PC4               key  0.633591  best
## 4    PC5       duration_ms  0.485910  best
## 5    PC6          liveness  0.789089  best
## 6    PC7  instrumentalness -0.579658  best
## 7    PC8    time_signature -0.632413  best
## 8    PC9       duration_ms  0.687742  best
## 9   PC10  instrumentalness  0.580839  best
## 10  PC11       speechiness  0.619864  best
## 11  PC12      danceability -0.597504  best
## 12   PC1          loudness -0.495175  weak
## 13   PC4              mode -0.600678  weak
## 14   PC1      acousticness  0.461091  weak
## 15  PC12           valence  0.449223  weak
## 16   PC6             tempo -0.507597  weak
```

```
model.plot()
```

```
## (<Figure size 1500x1000 with 0 Axes>, <AxesSubplot:>)
##
## meta NOT subset; don't know how to subset; dropped
```

```
# function for getting recommended songs and scores
def getContentFilteringPrecision(pid, nbrsCont, trainAF, train_norm):
  playlistIdx = trainAF.index[trainAF.pid == pid].tolist()
  # for all tracks in the playlist find k most similar songs
  songsToConsider = []
  for idx in playlistIdx:
    metric, locs = nbrsCont.kneighbors(np.array(train_norm.loc[idx]).reshape(1,-1), n_neighbors=30)
    currConsiderations = pd.DataFrame({"metric": metric[0].tolist(), "location": locs[0].tolist()})
    songsToConsider.append(currConsiderations)

  alreadyPlayedTracks = trainAF[trainAF.pid == pid].track_uri
  songsToConsiderFiltered = pd.concat([songsToConsider[i] for i in range(len(songsToConsider))], ignore_index=True)
  trainAF["index"] = trainAF.index
  songsToConsiderFiltered = pd.merge(trainAF, songsToConsiderFiltered, left_on = "index", right_on = "location", how = "inner")
  # remove songs already in playlist
  songsToConsiderFiltered = songsToConsiderFiltered[~songsToConsiderFiltered.track_uri.isin(alreadyPlayedTracks)]
```

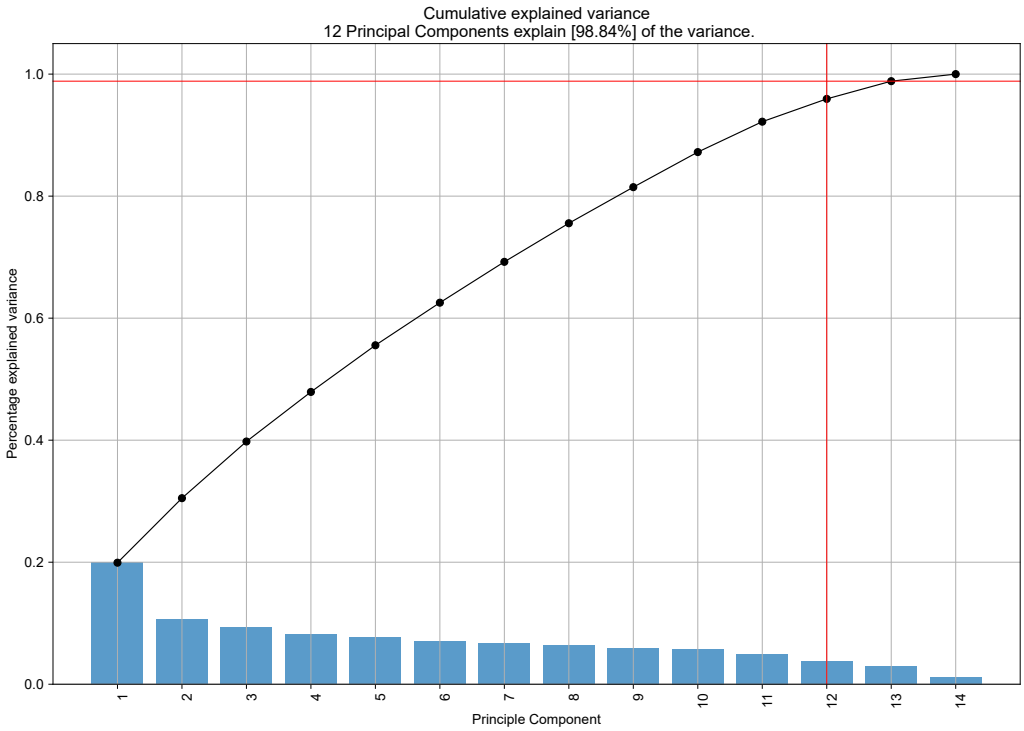

Figure 6: Cumulative Explained variance for Principal Component Analysis

```
  songsToConsiderFiltered = songsToConsiderFiltered.sort_values(by = ["metric", "artist_count"])
  # get the songs that appear more than once and sort by sum of distance
  filter2 = pd.DataFrame(songsToConsiderFiltered.track_uri.value_counts())
  filter2 = filter2[filter2.track_uri > 1].index
  filter3 = songsToConsiderFiltered[songsToConsiderFiltered.track_uri.isin(filter2)]
  # filter3 = pd.DataFrame(filter3.groupby('track_uri')['metric'].sum()) # for case of ties
  filter3 = filter3.drop_duplicates("track_uri")
  filter3 = filter3.sort_values(by = "metric")
  currValAF = valAF[valAF.pid == pid].track_uri
  numTracks = currValAF.shape[0]
  rprecision = currValAF.isin(filter3[:numTracks].track_uri).sum()/numTracks
  return rprecision, songsToConsiderFiltered
```

```
# content-based filtering
## tried minkowski and manhattan distancees
warnings.filterwarnings('ignore')
knnCont = NearestNeighbors(metric = "euclidean", algorithm = "brute")

## no PCA
nbrsCont = knnCont.fit(X)
seed(1)
randomPID = sample(train_bin.index.tolist(), 50)
avgPrec = 0.0
for pid in randomPID:
  avgPrec = avgPrec + getContentFilteringPrecision(pid, nbrsCont, trainAF, X)[0]

avgPrec = avgPrec/len(randomPID)
avgPrec

## with PCA
```

```
## 0.0018218623481781376
```

```
nbrsCont = knnCont.fit(model.transform(X))
## avg precision for 50 random playlists
```

```
## [pca] >Processing dataframe..
```

```
avgPrec = 0.0
with blockPrint():
  for pid in randomPID:
    avgPrec = avgPrec + getContentFilteringPrecision(pid, nbrsCont, trainAF, model.transform(X))[0]

# avgPrec
avgPrec = avgPrec/len(randomPID)
avgPrec
```

```
## 0.0014143920595533499
```

```
toPlot = getContentFilteringPrecision(0, nbrsCont, trainAF, X)[1]
toPlot = toPlot.drop_duplicates("track_uri")
toPlot = pd.DataFrame({"metric": toPlot.metric})

p1 = (
ggplot(data = toPlot)
+ geom_histogram(aes(x = "metric"), fill = "blue", alpha = 0.6)
+ labs(title = "Content-based filtering -- Playlist 1",
       x = "Euclidean Distance",  y = "Count")
+ xlim(0, None)
+ theme_bw()
)
p1.draw()
# ggsave(p1, filename = "pres/resContentFiltering.png")
```

```
# arrange plots
g1 = pw.load_ggplot(p1, figsize=(3,3))
g2 = pw.load_ggplot(p2, figsize=(3,3))
g12 = (g2|g1)
g12.savefig("images/g12.png")
```