The Million Song Recommendation System with Three Methods

Project Report

BYGB 7990 – Big Data Analytics – Spring 2020

Group 1
Jie Lu
Chaoying Bao
Tram Ngoc Le
Yihao Li

Instructor: Prof. Yilu Zhou

04/30/2020

FORDHAM UNIVERSITY
THE JESUIT UNIVERSITY OF NEW YORK

I. Executive Summary

This project was aimed to build a useful and comprehensive music recommendation system. The implementation was based on the Million Song Dataset and the musiXmatch dataset, which included data about user listening history, song metadata, artist, artist similarity, and lyrics. Different algorithms were employed, including popularity based, collaborative filtering based, and content-based methods, and generated three different recommendation lists — Hot Song List, Personal List, and New User List. They were compared and combined to build a recommending strategy for different scenarios and different users. The whole system was built with Python and PySpark on the Google Cloud Platform and AWS. In our final structure, the ALS algorithm was used for a personal recommendation, and lyric-based and artist-based similarity recommendation was employed to solve the cold-start issue.

II. Business Problem

An increasing number of online companies are utilizing recommendation systems to increase user interaction and enrich business potential. Use cases of recommendation systems have been expanding rapidly across many aspects of eCommerce and online media over the last 4-5 years, and we expect this trend to continue. The whole media and entertainment industry are moving online at a dramatic speed. For streaming service magnates like Spotify and Netflix, a sound recommendation system not only increases user interaction and improves the experience of customers, but also makes up their most competitive strength among all.

In this project, we want to focus on the streaming music industry. The potential benefits of a state of art recommendation system are valuable for the streaming music industry. Some possible benefits include:

- Improve user retention: a lot of users prefer a platform that could recommend new content according to their tastes. This is one of the core competencies of a streaming service. A sound recommendation system will increase the user retention rate.
- Improve user engagement: users will also spend more time in the application or on the website when they are accurately recommended for great content.
- Understand the customers' taste and the changing trend of their tastes: a streaming service nowadays needs enormous money on contents and copyright. Knowing the changing trend of customers' tastes will ensure a steady future of the company.

The recommender algorithm in use:

- Collaborative filtering: including user-user, item-item, and some more advanced methods like ALS.
- Content-based filtering: focusing on the knowledge about items themselves to recommend other items that have similar attributes
- Demographic/knowledge-based: using knowledge about users to recommend items
- Hybrid filtering: a combination of two or more algorithms

In the streaming music industry, collaborative filtering and content-based filtering are already widely implemented. Some companies also use Contextual Recommendations to recommend songs based on mood, time, season, and other “context”.

In this project, we want to implement several algorithms and develop a strategy to utilize those algorithms to build an innovative music recommendation system under different scenarios and for different users. We will also try to bring out new ideas on algorithms for future research based on the implementing experience in this project.

III. Dataset Description

The Million Song Dataset was originally started as a collaborative project between The Echo Nest and LabROSA. The dataset contains metadata, or derived features, for one million songs. The size of the entire data scales is up to 280GB. The Million Song Dataset is also a cluster of seven complementary datasets provided by the community. In this project, besides the subset, Taste Profile and musiXmatch are used to build the music recommendation system.

Regarding the Million Song subset, besides 1,000,000 unique tracks ID data, the song metadata, the links between artist ID and the tags, and artists’ similarity were extracted using the SQLite databases. For artists, the artists’ location data was also used.

The Taste Profile subset contains user listening history. The dataset scales up to 488MB, including 1,019,318 unique users, 384,546 unique MSD songs, and 48,373,586 user-song-play count triplets.

The musiXmatch dataset provides official lyrics collection of the Million Song Dataset in the bag-of-words format as well as the full mapping of MSD IDs to musiXmatch IDs. The mapping comes as a text file, including 779 thousand matches. The lyrics are directly associated with Million Song Dataset tracks regarding similar artists, tags, years, and audio features. We extracted the lyrics with the SQLite database. Each track is described as the word-counts for a dictionary of the top 5,000 words across the set. The split of training and testing was done according to the split for tagging, regarding tagging test artists. There are 210,519 training bag-of-words, 27,143 testing ones. The dataset contains track ID, mxm ID, word of lyrics, and frequency of the words features.

After mapping all the musiXmatch IDs, track ID, and song ID from listening history, the final dataset scaled up to 3GB. The table below provides a brief features summary after scraping:

Table 1. Feature Summary

Dataset	**Features**
Listeing_history	User_id, Track_id, Frequency
Tracks_metadata	Track_id, title, song_id, release, artist_id, artist_mbid, artist_name, duration, artist_familiarity, artist hottttnesss, year
Lyrics	Track_id, mxm_tid, word, count, is_test
Seeds	Song_id, track_id, mxm_id
Artist Similarity	Target, new

According to the initial data exploratory analysis of listening history, approximately 59.4% of the tracks were played at least once.

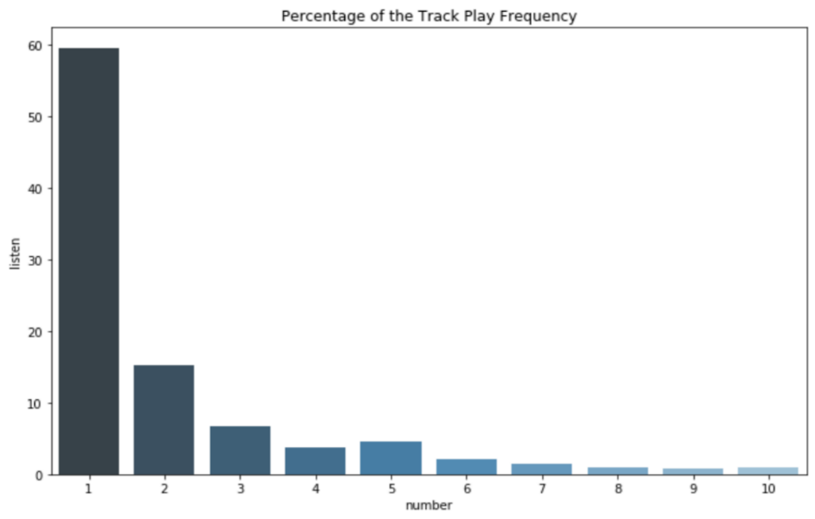


Figure 1. Track Play Frequency Distribution

Another finding, according to the cumulative curve of listening frequency, was that almost 90% of the songs were played between one to five times.

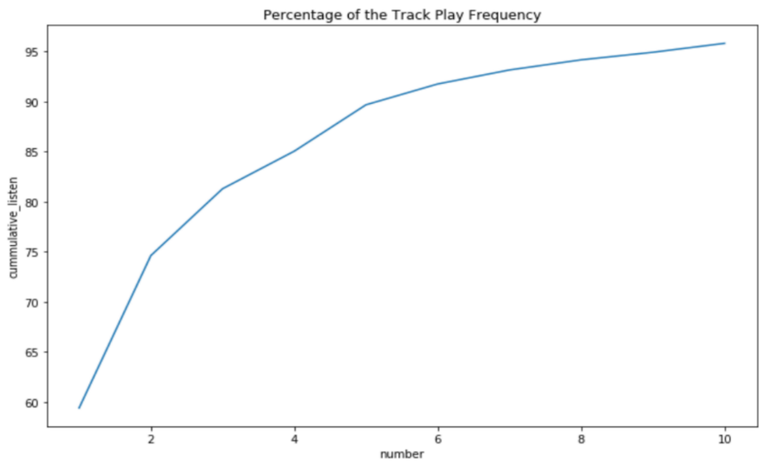


Figure 2. Track Play Frequency Cumulative Distribution

IV. System Design

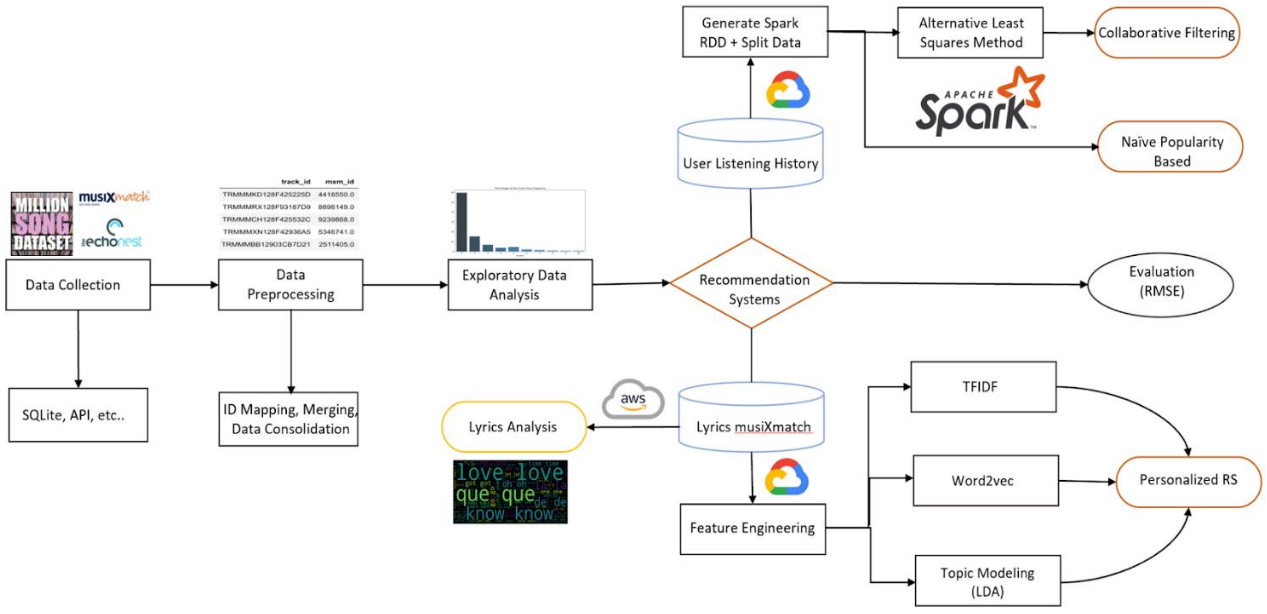


Figure 3. Flow Chart of System Design

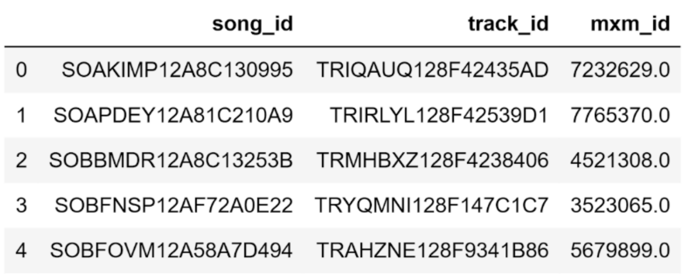
V. Data Preprocessing

Data processing is divided into 2 steps: data consolidation between taste profile, 1 Million unique track ID and MusiXmatch ID, and table joins. Then information regarding seeds, listening history, track metadata, artists information, artist's similarity, and lyrics were saved into separate files for further analysis.

In order to consolidate data of the user listening history, lyrics and unique tracks from MSD, we utilized three datasets from MSD subset (unique_tracks), Taste Profile (train_triplets), and musiXmatch ID mapping (mxm_779k_matches) to map song ID and musiXmatch ID. The processed was divided into two steps:

- Step 1: Map song id to track id
- Step 2: Map track id to musiXmatch id

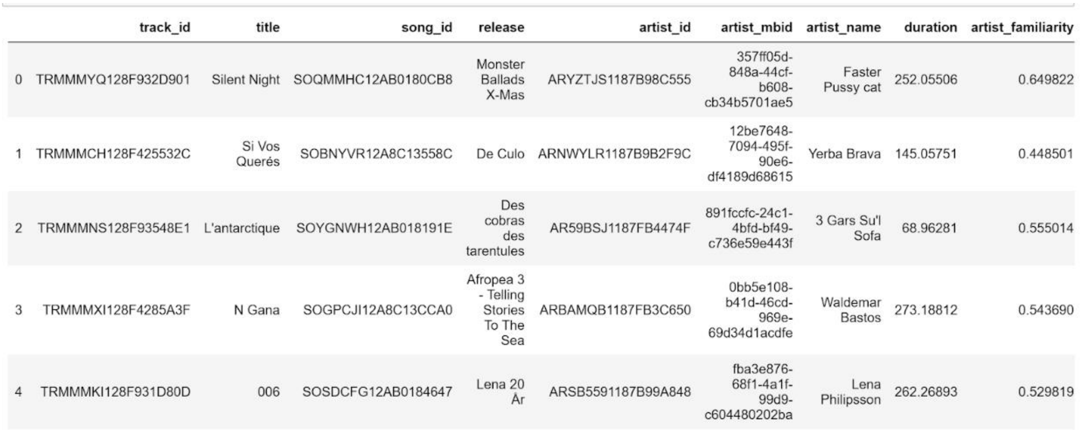
In conclusion, the final dataset covered 90% of the songs in the original listening history, or Taste Profile data. We saved those IDs as seeds and taste profiles containing user ID, track ID, and frequency as listening history, separately.



	song_id	track_id	mxm_id
0	SOAKIMP12A8C130995	TRIQAUQ128F42435AD	7232629.0
1	SOAPDEY12A81C210A9	TRIRLYL128F42539D1	7765370.0
2	SOBBMDR12A8C13253B	TRMHBXZ128F4238406	4521308.0
3	SOBFNSP12AF72A0E22	TRYQMNI128F147C1C7	3523065.0
4	SOBFOVM12A58A7D494	TRAHZNE128F9341B86	5679899.0

Figure 4. Seeds Profile Overview

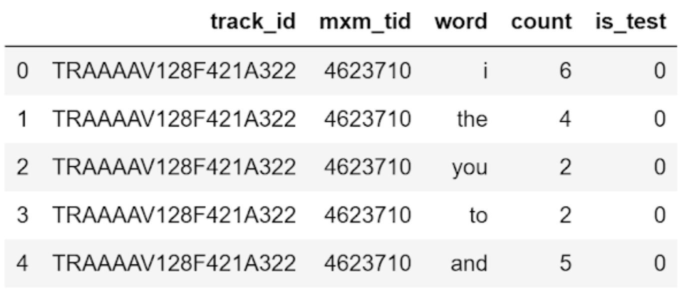
Next, we extracted metadata of tracks from SQLite database ("track_metadata.db) and saved these as "track_metadata.csv."



	track_id	title	song_id	release	artist_id	artist_mbid	artist_name	duration	artist_familiarity
0	TRMMMYQ128F932D901	Silent Night	SOQMMHC12AB0180CB8	Monster Ballads X-Mas	ARYZTJS1187B98C555	357ff05d-848a-44cf-b608-cb34b5701ae5	Faster Pussy cat	252.05506	0.649822
1	TRMMMCH128F425532C	Si Vos Querés	SOBNYVR12A8C13558C	De Culo	ARNWYLR1187B9B2F9C	12be7648-7094-495f-90e6-df4189d68615	Yerba Brava	145.05751	0.448501
2	TRMMMNS128F93548E1	L'antarctique	SOYGNWH12AB018191E	Des cobras des tarentules	AR59BSJ1187FB4474F	891fccfc-24c1-4bfd-bf49-c736e59e443f	3 Gars Su'l Sofa	68.96281	0.555014
3	TRMMMXI128F4285A3F	N Gana	SOGPCJI12A8C13CCA0	Afropea 3 - Telling Stories To The Sea	ARBAMQB1187FB3C650	0bb5e108-b41d-46cd-969e-69d34d1acdfe	Waldemar Bastos	273.18812	0.543690
4	TRMMMKI128F931D80D	006	SOSDCFG12AB0184647	Lena 20 År	ARSB5591187B99A848	fba3e876-68f1-4a1f-99d9-c604480202ba	Lena Philipsson	262.26893	0.529819

Figure 5. Track Metadata Overview

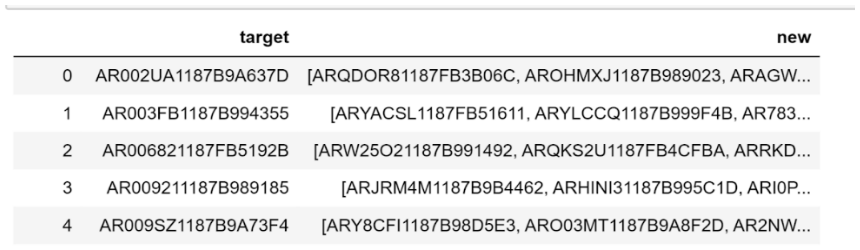
For Lyrics information, we extracted data from "lyrics.db" SQLite database and placed the SQL results in a data frame. We saved the word list of lyrics and the overall lyrics information separately. However, when building a content-based recommendation system, we only used the lyrics table for lyrics information, demonstrated as below:



	track_id	mxm_tid	word	count	is_test
0	TRAAAAV128F421A322	4623710	i	6	0
1	TRAAAAV128F421A322	4623710	the	4	0
2	TRAAAAV128F421A322	4623710	you	2	0
3	TRAAAAV128F421A322	4623710	to	2	0
4	TRAAAAV128F421A322	4623710	and	5	0

Figure 6. Lyrics Overview

For Artists Similarity information, we fetched target artist ID and similar artist ID from "artist_similarity.db" SQLite database. We grouped all the similar artist IDs by the target artist ID and stored the information into a new data frame.



	target	new
0	AR002UA1187B9A637D	[ARQDOR81187FB3B06C, AROHMXJ1187B989023, ARAGW...
1	AR003FB1187B994355	[ARYACSL1187FB51611, ARYLCCQ1187B999F4B, AR783...
2	AR006821187FB5192B	[ARW25O21187B991492, ARQKS2U1187FB4CFBA, ARRKD...
3	AR009211187B989185	[ARJRM4M1187B9B4462, ARHINI31187B995C1D, ARI0P...
4	AR009SZ1187B9A73F4	[ARY8CFI1187B98D5E3, ARO03MT1187B9A8F2D, AR2NW...

Figure 7. Artist Similarity Overview

In conclusion, we prepared five csv files, including "listening_history.csv", "seed.csv", "lyrics.csv", "tracks_metadata.csv", and "artist_similarity.csv", for building collaborative filtering and content-based (personalized) music recommendation systems.

VI. Algorithm Development

i. Popularity Benchmark

As a benchmark for recommending songs, we believe recommending popular songs is better than random guesses. We filtered data, counted the total frequency, and selected the top 10 heated tracks. This would be our baseline saver if, unfortunately, there is no data for a new user (cold start).

ii. Collaborative Filtering

Collaborative filtering is probably one of the most used approaches for the recommendation system, or at least as a necessary component of some advanced recommendation structures. However, it has a surprisingly simple requirement for data. It needs nothing else but users' historical preference on a set of items. The main assumption here is that the users (listeners in our case) who have agreed in the past are very likely to agree in the future. For multiple users, it assumes if user A liked the same product as user B, A would also like to have similar taste as B for other B's favorites.

In terms of the ratings, it often requires some metrics to indicate user's preference, which can either be explicitly available data such as 5 stars or thumbs-up on the product, or an implicitly derived score such as clicks, number of purchases or other data recorded in the cookie. In our case, we would take a traditional way to use the listening frequency as our rating. Our assumption when lacking explicit rating is that a higher listening frequency equates to a higher rating.

Alternating Least Squares (ALS) is the model we used to fit our data and find similarities among different users.

For ALS, we constructed a matrix with frequency of each song played by each user. Certainly, this matrix is sparse with tons of missing value, as a limited number of users have listened to only a limited set of songs. The high-level idea is to approximate the matrix by factorizing it as the product of two matrices: the user matrix that represents each user, while the item matrix describes properties of each track.

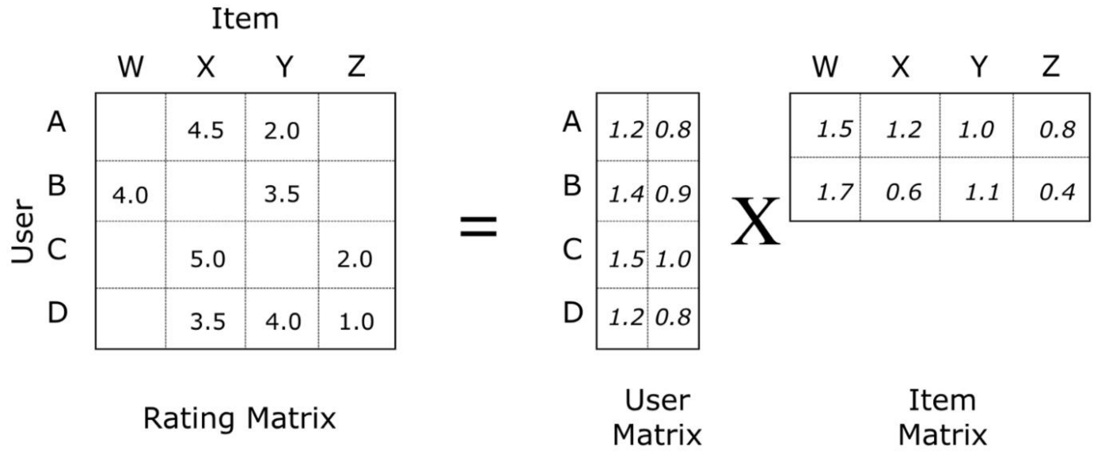


Figure 8. Rating Matrix Factorization

For a reasonable result, we would like to have two matrices created such that the error for the known user-song pairs is minimized. The error here refers to the rooted mean squared error. (RMSE) In a more detailed technical level, ALS would first fill the user matrix with a random value, and then optimize the song matrix value by minimizing the error. After that, it “alters”, which means it would hold the song matrix and optimize the value of the user’s matrix. This step minimizes two loss functions alternatively and should achieve some optima.

When it comes to the implementation, we used the Spark ML pipeline with the following settings. We picked up a big set of the parameters and ran the grid-search with cross-validation on this space. Some important parameters to tune include:

- maxIter: the max number of iterations
- rank: the number of latent factors, which basically determined the shape of matrix
- regParam: the regularization parameter

iii. Content-Based Filtering

1. Algorithm Design

In our system, content-based filtering recommended similar songs based on the cosine similarity of different features of lyrics to a user’s profile. Each songs’ lyrics were characterized by their (1) TFIDF, (2) Word2vec, (3) Topic Model with LDA. The features were represented in matrices, and the cosine similarity was calculated between matrices.

Since our content-based filtering was designed for new users who came to search for a specific song, there was no existing user’s profile. The input song’s name and artist’s name were taken as the user’s profile and then this specific song’s index was matched in the cosine similarity matrix to return the top 10 similar songs.

What is more, similar artists were also taken into account. The similarity scores of this specific artist’s similar artists’ songs for the specific song this user searched for were matched in the cosine similarity matrix, and the top 10 similar songs belong to the similar artists were returned. Finally, along with the specific song this user searched for, the algorithm recommended 20 other songs that were closest in cosine distance to the user’s input song.

Three different cosine similarities were used to recommend 20 songs, respectively, and the overlapping songs among three results were taken as our final recommended songs.

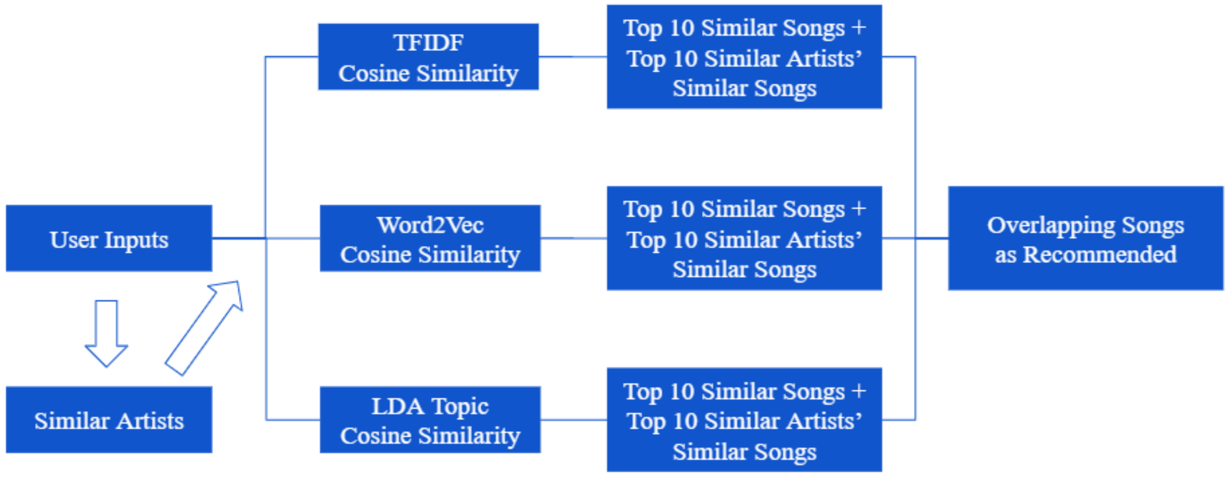

TFIDF
Cosine Similarity
Top 10 Similar Songs + Top 10 Similar Artists' Similar Songs
User Inputs
Word2Vec
Cosine Similarity
Top 10 Similar Songs + Top 10 Similar Artists' Similar Songs
Overlapping Songs as Recommended
Similar Artists
LDA Topic
Cosine Similarity
Top 10 Similar Songs + Top 10 Similar Artists' Similar Songs

Figure 9. Content-Based Filtering Algorithm Development

2. Features Creation

(1) TFIDF for Lyrics

After stopword removal and further Porter stemming, the TD-IDF Vectorizer in the python scikit-learn package was used to calculate the TFIDF of each song's lyrics. However, there was a limit in our lyric dataset. There were only bag-of-words in the dataset rather than the original lyrics. The actual lyrics were protected by copyright, and Million Song Dataset did not have permission to redistribute them. Therefore, only the unigram could be used.

After getting the TFIDF matrix, the cosine_similarity in the python scikit-learn package was used to calculate the cosine similarity scores among different tracks based on the TFIDF.

(2) Word2vec for Lyrics

Word2vec is a two-layer neural net that processes text by "vectorizing" words. Its input is a text corpus, and its output is a set of vectors: feature vectors that represent words in that corpus. While Word2vec is not a deep neural network, it turns text into a numerical form that deep neural networks can understand.

The purpose and usefulness of Word2vec are to group the vectors of similar words in vector space. That is, it detects similarities mathematically. Word2vec creates vectors that are distributed numerical representations of word features, features such as the context of individual words. It does so without human intervention.[2]

The data size was so large that calculating the cosine similarity based on the high dimensional sparse TFIDF matrix was time-consuming and required a lot of memory. Thus, the Word2vec was used in our second method. However, there was the same limit. Only the unigram could be used, which made the Word2vec model less accurate since there was no context.

The Word2Vec model in the python gensim package was used to create vectors for each song's lyrics. The words appearing only once were ignored, the total dimension of the word vectors was set 200, and the window size was set as 1 because there were only unigrams in the lyrics.

After getting the Word2vec matrix, the cosine_similarity in the python scikit-learn package was used again to calculate the cosine similarity scores among different tracks based on the vectors.

(3) LDA for Lyrics

Topic modeling is a type of statistical modeling for discovering the abstract "topics" that occur in a collection of documents. Latent Dirichlet Allocation (LDA) is an example of the topic model and is used to classify text in a document to a particular topic. It builds a topic per document model and words per topic model, modeled as Dirichlet distributions.[5] Thus, the big idea behind LDA is that each document can be described by a distribution of topics, and each topic can be described by a distribution of words.

We assumed that there must be some similar topics among all songs. Limiting the number of topics could also help us save time and memory when dealing with a large dataset. Therefore, the LDA was used in our third method. LDA ignores syntactic information and treats documents as bags of words, so the unigram format in our dataset doesn't matter too much in this method.

The LDA Model in the python gensim package was used to create a topic matrix for all songs' lyrics. Only the top 10 topics were selected in the model.

After getting the topic model matrix, the cosine_similarity in the python scikit-learn package was used again to calculate the cosine similarity scores among different tracks based on the topics' weights.

(4) Artist Similarity

The Million Song Dataset offered a dataset containing similarity among artists, which expanded the content of our content-based filtering. This data was used after the cosine similarity was calculated. The similar artists' songs were into account, even though they were not in the top 10 similar songs list because we assumed that similar artists would attract the same audiences in some way.

3. Content-Based Filtering Building

When all the features were ready, two key functions were defined to accomplish the recommendation task. The first function was defined to return the top 10 recommended songs simply based on the cosine similarity. The second function was defined to find the other top 10 recommended songs based on the artist similarity combined with cosine similarity. Finally, these two functions would create a DataFrame that contained the user input song along with the other 20 recommended songs. Three DataFrames would be created based on three cosine similarities, and they were merged to return the overlapping songs, which could be the most similar songs to the user input song.

VII. Results and Evaluation

i. Lyrics Analysis

Besides recommendation system building, we conducted further exploration of lyrics by implementing the LDA model and visualizing the most popular 30 words in lyrics based on the time, including songs from the 2000s, 90s, 80s, 70s, 60s, and 50s.

1. Topic Modeling Visualization (20 topics)

Regarding the LDA model in lyrics analysis, it is noted that the topic modeling visualization was slightly different from the result in the previous content-based recommendation system as the lyrics were not processed with stemming and the model defined 20 topics instead of 10 topics. We wanted to keep the original word form for a more comprehensive lyrics topic demonstration. We removed stopwords, generated token dictionary class and built a corpus. We fitted the corpus and dictionary into the LDA model with 20 topics.

We found some interesting results from LDA clustering of 20. The interactive illustration of clusters can be reviewed in our jupyter notebook. The graph has shown that the cluster 5,10,18, and 16 as well as cluster 20,19 and 14 were separated from other clusters in different dimensions. The reason for this separation was the difference of language as those clusters were not English. On the other hand, cluster 11, 17, 12, and 7 contains lyrics with darker meaning or religious meaning. For instance, we could refer cluster 11 to religion as the most relevant words are “world”, “life”, “live”, “god”, “us”, “heaven”, “god”, “jesus”, “soul”, “angle”, and so on. However, topic in cluster 12 could be related to some darker meaning, or rap song, since the lyrics contained very negative words, such as the “f” word, “kill”, “dead”, “hate”, “hell”, “gun”, “shit”, “bitch”, “war”, “sick”, “shot”, or “murder”. Those clusters were placed on the fourth dimension (lower right side) of the graph. The first dimension (upper right side) included topics with more positive vibes, such as love or party. For instance, cluster 15 could be referred to as a “dance party” as the lyrics included mostly “oh”, “ooh”, “ah”, “yeah”, “shake”, “mama”, “yes”, “babi”, “ohh” and so on. There are multiple clusters relating to “love” mixed together since this topic tends to contain multiple emotions. For example, we can refer to cluster 8 as “happy love” as the lyric contains “love”, “want”, “need”, “feel”, “like”, “kiss”, “true”, “touch”, “give” and so on. Cluster 1 and cluster 2 can refer to a “sad love” or “break up” as the lyrics contains “away”, “see”, “day”, “feel”, “dream”, “fall”, “eye”, “heart” , “time”, “never”, “know”, “think”, and so on, which can demonstrate the longing or sad feeling.

In general, we can get a good class of 20 topics with the LDA model. Each topic is approximately well-defined with different themes, such as religion, dance party, dark drama, happy love, and sad love.

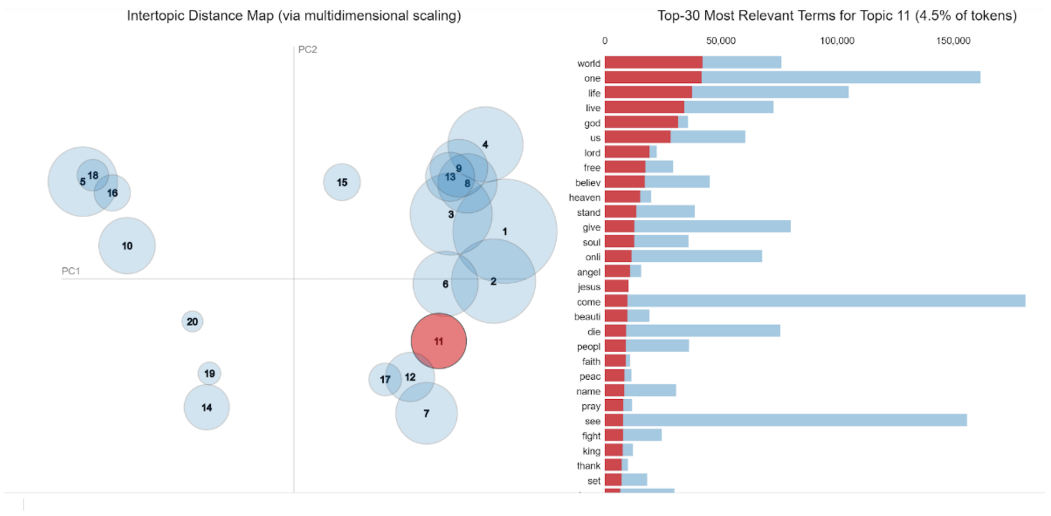


Figure 10. Topic Clusters

2. Lyrics Analysis through the Time

The second analysis of lyrics we conducted related to the timeline of the songs. We examined the changes of lyrics from songs in the 2000s, 90s, 80s, 70s, 60s and 50s. We merged the lyrics table with the year from the track metadata table. The lyrics were preprocessed with stemming, stopwords removal and tokenization. We plotted a bar chart of the top 30 words with the highest frequency for each period.

We found some interesting results based on the observation of the top 30 words in lyrics over the time. "Love" seems to be the greatest inspiration in songs of the 50s, 60s, 70s, 80s and 90s as the top words repeated over the past more than 50 years. From 2000 to 2011, the word "love" took the second place while the word "know" became the most popular word. In general, the word "love", "know", "like", "got" were always popular from the 50s to the 2000s.

Comparing the most popular words in lyrics of the 60s and 2000s, the lyrics in the 60s tended to bring a more subtle vibe than the 2000s did. The 60s lyrics had "babi", "know", "love", "yeah", "oh", and "got", which demonstrated the priority in love and the lover. At the same time, the 2000s lyrics had "love", "la", "know", "got", "de", and "come", which portrayed a bustling vibe and prioritized love and party at the same time.

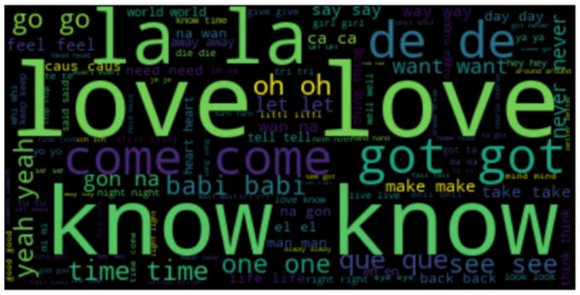


Figure 11. WordCloud for 1960s

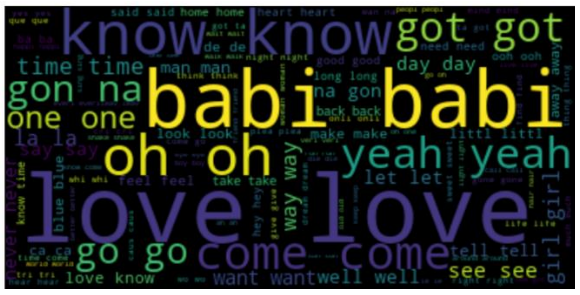


Figure 12. WordCloud for 2000s

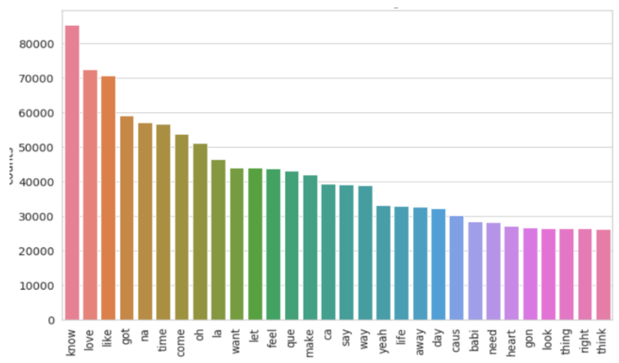


Figure 13. Song of 2000s

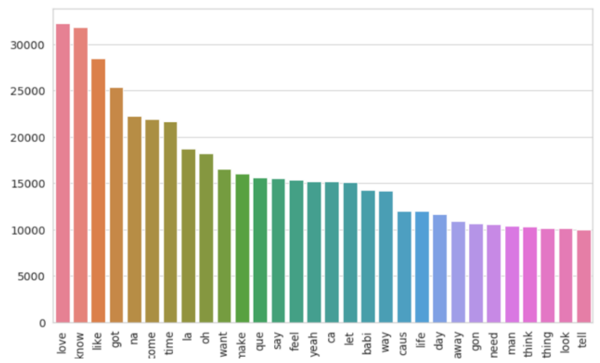


Figure 14. Song of the 1990s

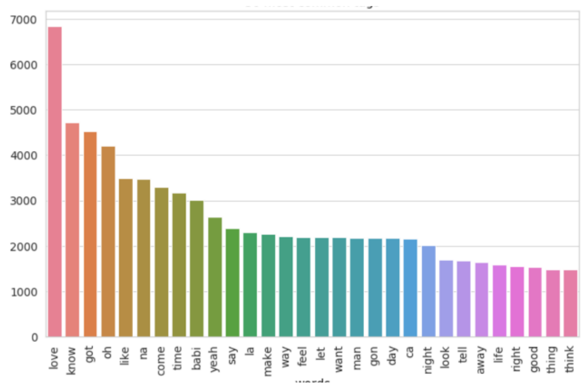


Figure 15. Song of 1980s

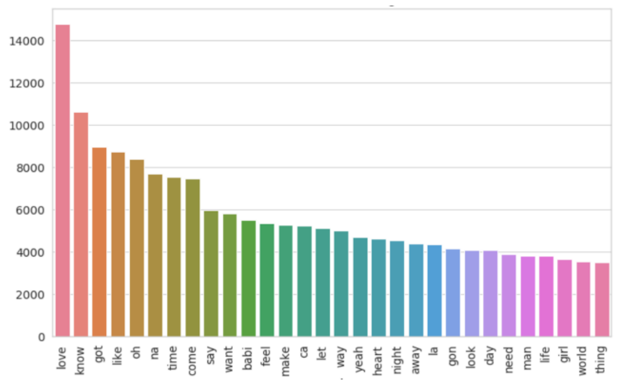


Figure 16. Song of the 1970s

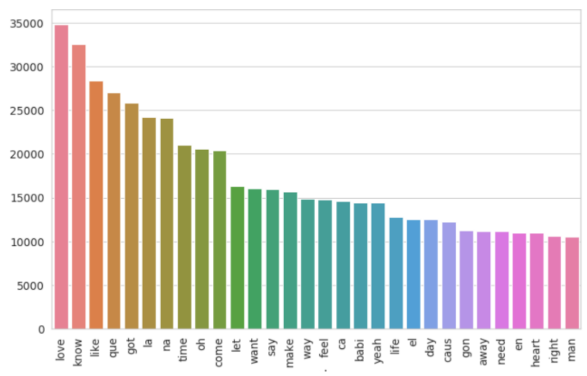


Figure 17. Song of 1960s

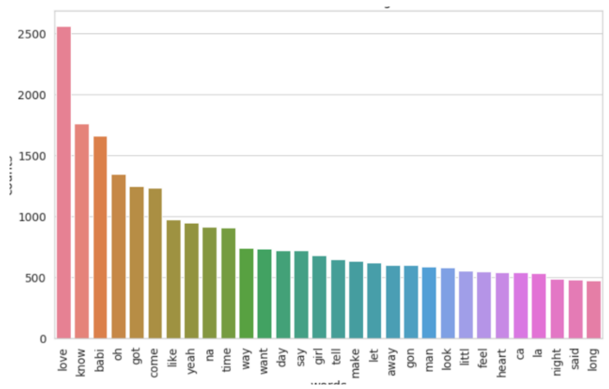
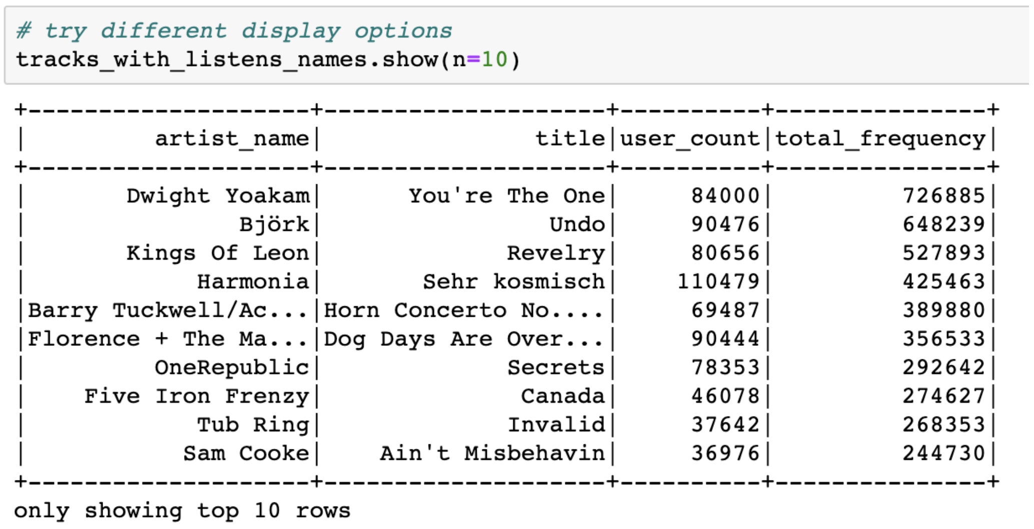


Figure 18. Song of the 1950s

ii. Recommendation Systems

1. Collaborative Filtering:

Here is the benchmark of our top 10 most listened songs.



```
# try different display options
tracks_with_listens_names.show(n=10)
```

```
+--------------------+--------------------+----------+---------------+
|         artist_name|               title|user_count|total_frequency|
+--------------------+--------------------+----------+---------------+
|       Dwight Yoakam|      You're The One|     84000|         726885|
|               Björk|                Undo|     90476|         648239|
|       Kings Of Leon|             Revelry|     80656|         527893|
|            Harmonia|       Sehr kosmisch|    110479|         425463|
|Barry Tuckwell/Ac...|Horn Concerto No....|     69487|         389880|
|Florence + The Ma...|Dog Days Are Over...|     90444|         356533|
|         OneRepublic|             Secrets|     78353|         292642|
|    Five Iron Frenzy|              Canada|     46078|         274627|
|            Tub Ring|             Invalid|     37642|         268353|
|           Sam Cooke|    Ain't Misbehavin|     36976|         244730|
+--------------------+--------------------+----------+---------------+
only showing top 10 rows
```

Figure 19. Top 10 Songs Based on Popularity

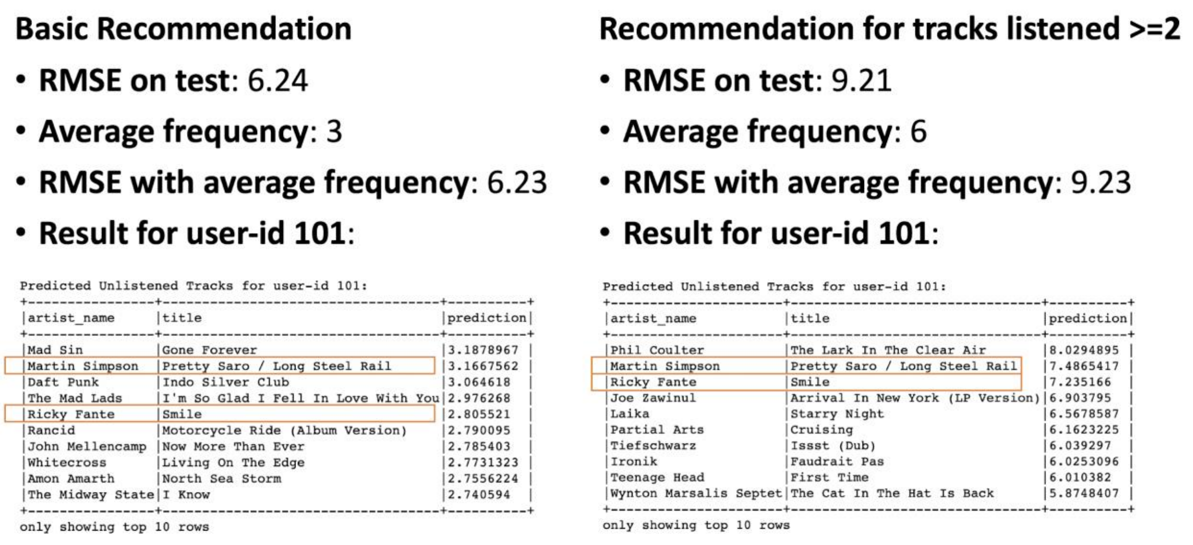
We split our data into train, validation, and test three parts. 60% as train, 20% as validation, and 20% as test. We ran grid search and cross-validation on the validation set to determine the best parameters of the recommendation system, and only recommend the results to the new users in the test set once to avoid information leakage. The process is implemented from scratch and is locked in a pipeline. This work is constructed to avoid overfitting problems.

As for Spark 2.0, when asked to provide a rating for new users never seen before, the ALS can only yield NAN value. Therefore, it makes it impossible to adapt Spark ML's Cross Validator if we would like to check the RMSE. We chose to let the algorithm drop NAN values by default with a customized cross validation process before using RMSE for evaluation.

We did the following configuration steps:

- Set appropriate parameters for users, items, and rating
- Fit ALS and transform the table to generate the prediction column
- Run predictions against the validation set and check the error
- Finalize the model with the best RMSE score

In our exploratory analysis step, we noticed that more than half of the songs in the sample are listened at least once. Thus, we infer songs played more than once can better represent users' tastes and run the ALS on two versions of the datasets.



Basic Recommendation

- **RMSE on test**: 6.24
- **Average frequency**: 3
- **RMSE with average frequency**: 6.23
- **Result for user-id 101**:

```
Predicted Unlistened Tracks for user-id 101:
+----------------+------------------------------------+----------+
|artist_name     |title                               |prediction|
+----------------+------------------------------------+----------+
|Mad Sin         |Gone Forever                        |3.1878967 |
|Martin Simpson  |Pretty Saro / Long Steel Rail       |3.1667562 |
|Daft Punk       |Indo Silver Club                    |3.064618  |
|The Mad Lads    |I'm So Glad I Fell In Love With You|2.976268  |
|Ricky Fante     |Smile                               |2.805521  |
|Rancid          |Motorcycle Ride (Album Version)     |2.790095  |
|John Mellencamp |Now More Than Ever                  |2.785403  |
|Whitecross      |Living On The Edge                  |2.7731323 |
|Amon Amarth     |North Sea Storm                     |2.7556224 |
|The Midway State|I Know                              |2.740594  |
+----------------+------------------------------------+----------+
only showing top 10 rows
```

Recommendation for tracks listened >=2

- **RMSE on test**: 9.21
- **Average frequency**: 6
- **RMSE with average frequency**: 9.23
- **Result for user-id 101**:

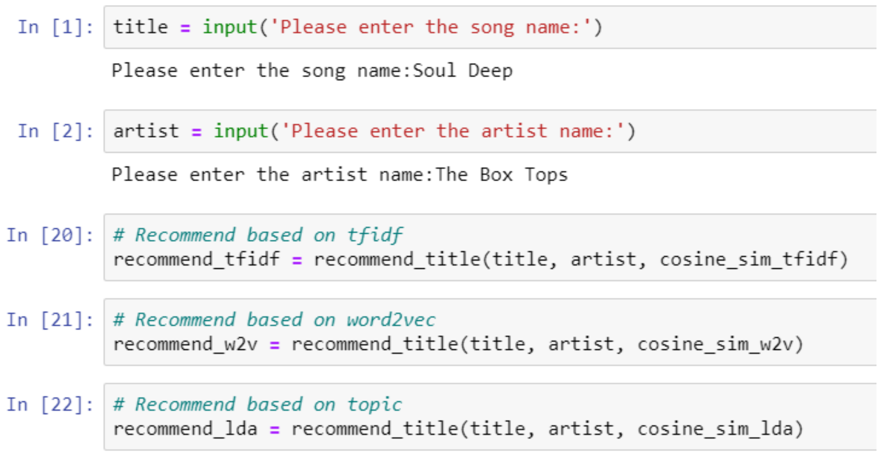
```
Predicted Unlistened Tracks for user-id 101:
+----------------------+--------------------------------+----------+
|artist_name           |title                           |prediction|
+----------------------+--------------------------------+----------+
|Phil Coulter          |The Lark In The Clear Air       |8.0294895 |
|Martin Simpson        |Pretty Saro / Long Steel Rail   |7.4865417 |
|Ricky Fante           |Smile                           |7.235166  |
|Joe Zawinul           |Arrival In New York (LP Version)|6.903795  |
|Laika                 |Starry Night                    |6.5678587 |
|Partial Arts          |Cruising                        |6.1623225 |
|Tiefschwarz           |Issst (Dub)                     |6.039297  |
|Ironik                |Faudrait Pas                    |6.0253096 |
|Teenage Head          |First Time                      |6.010382  |
|Wynton Marsalis Septet|The Cat In The Hat Is Back      |5.8748407 |
+----------------------+--------------------------------+----------+
only showing top 10 rows
```

Figure 20. Results Comparison of Model with All Data and Frequency >=2 Data

Except for cross-validation, another common measurement is to compare the model with fake test data where every rating is the average number of frequencies from the train data. While the model excluding the 1-time listened songs returns a slightly higher RMSE score on the test dataset, it behaves better when compared with the average frequency. And this may hint the recommendation makes more sense. The highlighted tracks can be of the highest recommendation quality, as they are the overlap between two ALS recommendation models.

2. Content-Based Filtering

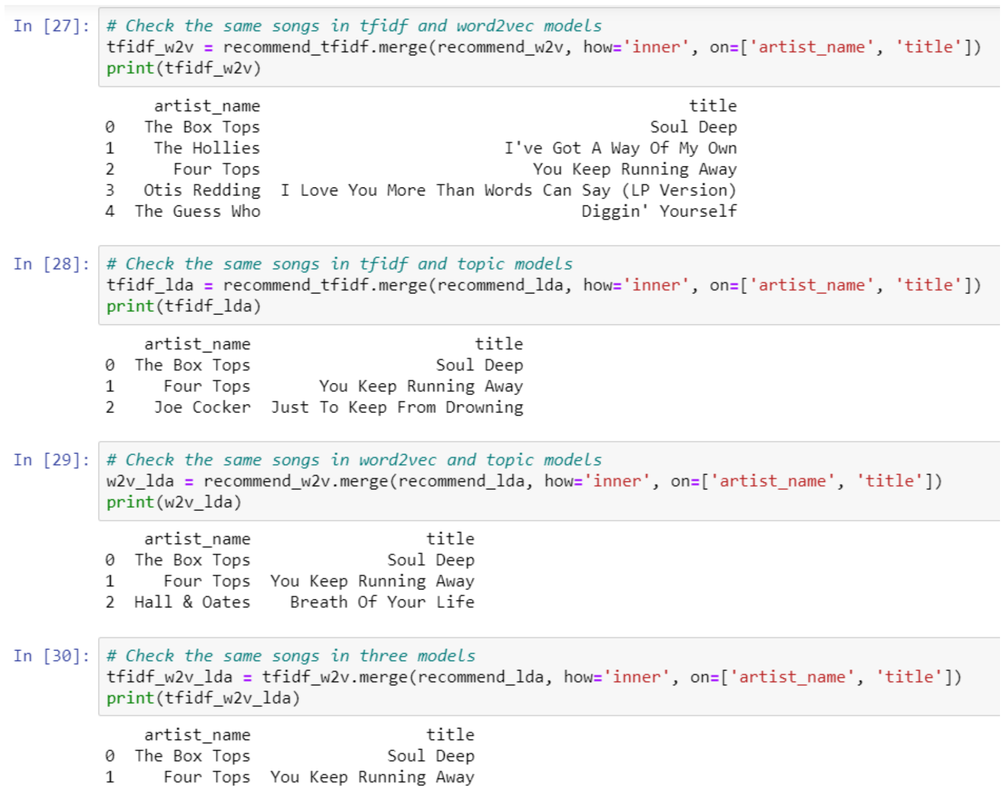
We simulated a new user coming to search for a song to see the recommended results.



```
In [1]: title = input('Please enter the song name:')

        Please enter the song name:Soul Deep

In [2]: artist = input('Please enter the artist name:')

        Please enter the artist name:The Box Tops

In [20]: # Recommend based on tfidf
         recommend_tfidf = recommend_title(title, artist, cosine_sim_tfidf)

In [21]: # Recommend based on word2vec
         recommend_w2v = recommend_title(title, artist, cosine_sim_w2v)

In [22]: # Recommend based on topic
         recommend_lda = recommend_title(title, artist, cosine_sim_lda)
```

Figure 21. Simulation of A New User

Three DataFrames were created after we ran the function, and there were 4 same songs between TFIDF and Word2vec methods, 2 same songs between TFIDF and LDA methods, 2 same songs between Word2vec and LDA methods, and 1 same song among three methods. Therefore, our final recommended song related to the Box Tops' Soul Deep was Four Tops' You Keep Running Away.



```
In [27]: # Check the same songs in tfidf and word2vec models
         tfidf_w2v = recommend_tfidf.merge(recommend_w2v, how='inner', on=['artist_name', 'title'])
         print(tfidf_w2v)

             artist_name                                        title
         0   The Box Tops                                   Soul Deep
         1    The Hollies                       I've Got A Way Of My Own
         2     Four Tops                          You Keep Running Away
         3   Otis Redding  I Love You More Than Words Can Say (LP Version)
         4  The Guess Who                              Diggin' Yourself

In [28]: # Check the same songs in tfidf and topic models
         tfidf_lda = recommend_tfidf.merge(recommend_lda, how='inner', on=['artist_name', 'title'])
         print(tfidf_lda)

            artist_name                      title
         0 The Box Tops                  Soul Deep
         1    Four Tops      You Keep Running Away
         2   Joe Cocker  Just To Keep From Drowning

In [29]: # Check the same songs in word2vec and topic models
         w2v_lda = recommend_w2v.merge(recommend_lda, how='inner', on=['artist_name', 'title'])
         print(w2v_lda)

            artist_name                  title
         0 The Box Tops              Soul Deep
         1    Four Tops  You Keep Running Away
         2 Hall & Oates    Breath Of Your Life

In [30]: # Check the same songs in three models
         tfidf_w2v_lda = tfidf_w2v.merge(recommend_lda, how='inner', on=['artist_name', 'title'])
         print(tfidf_w2v_lda)

            artist_name                  title
         0 The Box Tops              Soul Deep
         1    Four Tops  You Keep Running Away
```

Figure 22. Overlapping Songs among Three Methods

VIII. Conclusions and Lessons Learned

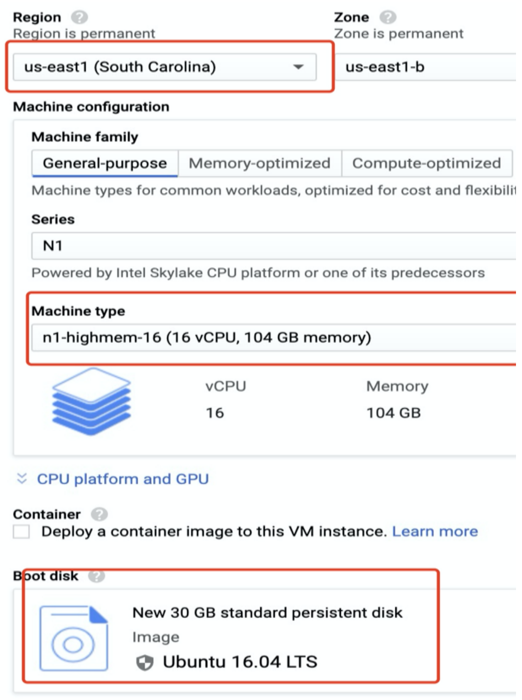
In this project, we successfully built a recommendation system using a dataset with 1,019,318 unique users and 384,546 unique songs.

We used the ALS algorithm, which combines user and item knowledge, for our collaborative filtering recommender. This recommender will be used for old users with enough listening history to generate personalized recommendations. We also combined several aspects of song knowledge, including artist similarity, TF-IDF, Word2vec, and LDA modeling for lyric similarity, to build a content-based recommender. The content-based recommender is for new users with only one or a few searches and listening history. Similar songs for the current song will be recommended.

Considering that Spotify has about 2 million monthly active users, our project is close to the monthly magnitude of the industry-level. Yet we suffered a lot in the implementing process and still face some obstacles to improving our system.

i. Lessons learned

There are several lessons we learned when working with cloud and pyspark. First, spark has a high dependency on memory usage. During the testing phase, our virtual instance with 30GB memory crushes from time to time. It would save some cost to choose a high memory specialized instance on google cloud. To run cross validation on the 3GB listening history data, it would be secure to choose the following configuration. It is very helpful to use the Unix command "free -m" to check available memory in time. In addition, we learned that we need to cache the datasets whenever they are likely to be used more than once:



Region
Region is permanent
us-east1 (South Carolina)

Zone
Zone is permanent
us-east1-b

Machine configuration

Machine family
General-purpose | Memory-optimized | Compute-optimized
Machine types for common workloads, optimized for cost and flexibili

Series
N1
Powered by Intel Skylake CPU platform or one of its predecessors

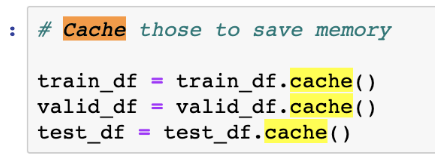
Machine type
n1-highmem-16 (16 vCPU, 104 GB memory)

vCPU 16
Memory 104 GB

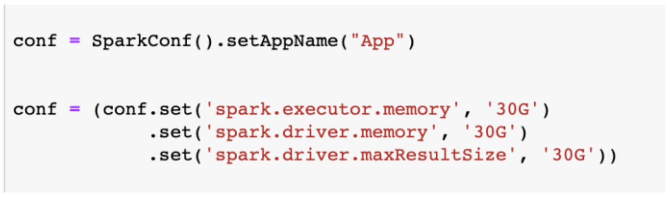
CPU platform and GPU

Container
Deploy a container image to this VM instance. Learn more

Boot disk
New 30 GB standard persistent disk
Image
Ubuntu 16.04 LTS



```
# Cache those to save memory

train_df = train_df.cache()
valid_df = valid_df.cache()
test_df = test_df.cache()
```



```
conf = SparkConf().setAppName("App")


conf = (conf.set('spark.executor.memory', '30G')
        .set('spark.driver.memory', '30G')
        .set('spark.driver.maxResultSize', '30G'))
```

Figure 23. Useful Tips

However, overcoming memory issues on cloud alone is not sufficient. We will also need to change some default spark settings to permit more executing memory and driver memory resources to be adapted by spark.

In addition, debugging with spark is quite suffering, as the error message is not informative at all. A common mistake is that the data type is not fixed or switched during the processing. Due to the Java development environment, different methods are very strict on input data type. We need to remind ourselves of casting the data types quite often. For example, ALS function in spark only allows integer input. It would return error even when the data type is big integer and so on. In the same vein, the raw ideas from the original data frame are no longer useful that we have to generate new integer id pairs.

```
from pyspark.sql import functions as F

# change ids to int
user_id = history_df.select('user_id')\
                    .distinct()\
                    .select('user_id', F.monotonically_increasing_id()\
                    .alias('new_userid'))


track_id = history_df.select('track_id')\
                     .distinct()\
                     .select('track_id', F.monotonically_increasing_id()\
                     .alias('new_trackid'))
```

Figure 24. Change ID's Data Type to Int

```
# ALS has strict requirement for data type
# bigint is very problematic
# make bigint to int

train_df = train_df.withColumn("new_userid", train_df["new_userid"].cast(IntegerType()))
train_df = train_df.withColumn("new_trackid", train_df["new_trackid"].cast(IntegerType()))

valid_df = valid_df.withColumn("new_userid", valid_df["new_userid"].cast(IntegerType()))
valid_df = valid_df.withColumn("new_trackid", valid_df["new_trackid"].cast(IntegerType()))

test_df = test_df.withColumn("new_userid", test_df["new_userid"].cast(IntegerType()))
test_df = test_df.withColumn("new_trackid", test_df["new_trackid"].cast(IntegerType()))
```

Figure 25. Change ID's Data Type from Bigint to Int

ii. Further Steps

1. Hybrid system

Currently, we are still facing some obstacles towards a final hybrid recommendation system. At this stage, our main difficulty comes from the lack of valid lyrics data. As for further steps in the future, we would be interested in combining the collaborative filtering and content-based recommendation system to provide hybrid recommendations. By taking the complementary advantages from various recommendation algorithms, we believe a hybrid solution such as using overlappings would provide users with suggestions of higher quality. For example, a hybrid system based on lyrics and listening history could make sure the users not only like the explicit topic of the track but also like the genre from a taste perspective.

As a matter of fact, more dimensions of recommendation are always better. Google naturally combined plenty of recommendation strategies in its wide and deep recommendation system with neural networks and ensemble methods. We are not able to bring this project to such a top level, but this precious practice undoubtedly helped us lay a solid foundation for potential industrial works involving recommendation systems or distributed systems.

2. New ideas for future exploration

User2Vec: We use Word2vec for NLP in this project to analyze the knowledge about songs. What if we consider songs as words and users as documents? The listening history of a user is the text content of the document. Then we can apply Word2vec and Doc2vec to find similar songs and users.

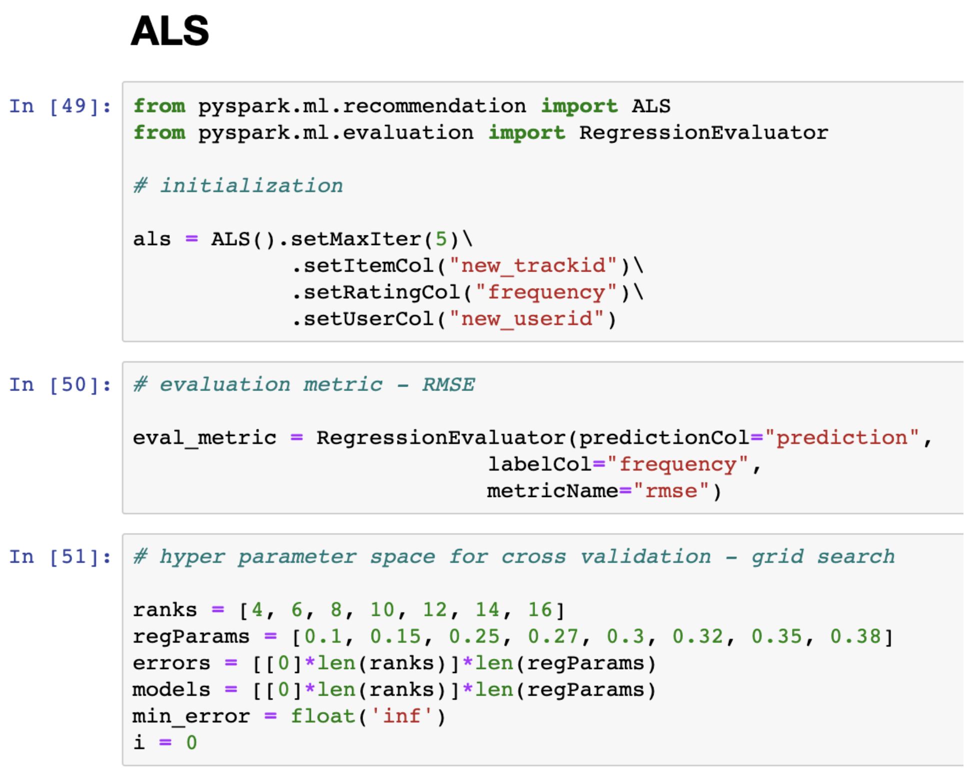
Graph algorithms: Graph database is another new trend for online shopping recommendation systems. When we model songs, artists, and users in a graph database, we will be able to use graph algorithms to find the relationship among them. This also allows us to consider the knowledge about songs, artists, and users at the same time.

Content-based filtering using music audio: In this project and most current business implementation, content-based filtering in the music industry means lyric-based or text-based. However, machine learning and deep learning are also having great processes in audio processing. Analyzing the music audio directly could be a new direction for content-based filtering.

IX. References

1. A Beginner's Guide to Word2Vec and Neural Word Embeddings. (n.d.). Retrieved from https://pathmind.com/wiki/word2vec
2. Content-based Filtering. (2012, January 24). Retrieved from http://recommender-systems.org/content-based-filtering/
3. Karantyagi. (n.d.). karantyagi/Restaurant-Recommendations-with-Yelp. Retrieved from https://github.com/karantyagi/Restaurant-Recommendations-with-Yelp
4. Li, S. (2018, June 1). Topic Modeling and Latent Dirichlet Allocation (LDA) in Python. Retrieved from https://towardsdatascience.com/topic-modeling-and-latent-dirichlet-allocation-in-python-9bf156893c24
5. MODELING. (n.d.). Retrieved from https://xindizhao19931.wixsite.com/spotify2/modeling
6. Welcome! (n.d.). Retrieved from http://millionsongdataset.com/

Appendix



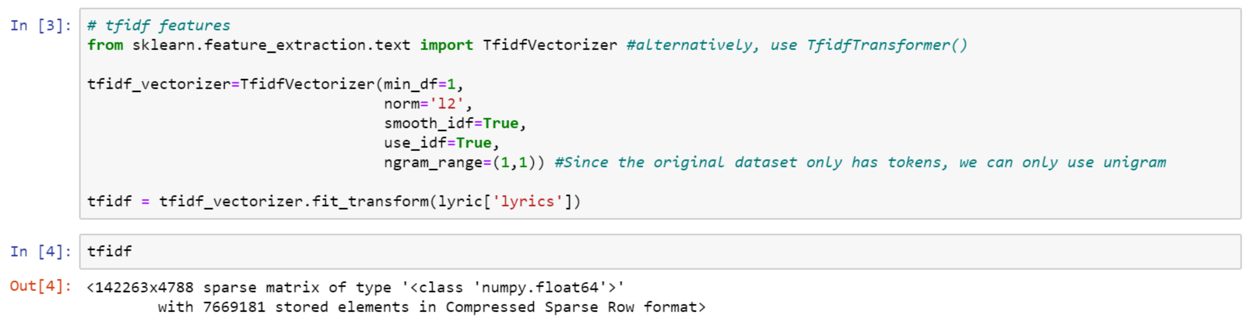
ALS

```
In [49]: from pyspark.ml.recommendation import ALS
         from pyspark.ml.evaluation import RegressionEvaluator

         # initialization

         als = ALS().setMaxIter(5)\
                     .setItemCol("new_trackid")\
                     .setRatingCol("frequency")\
                     .setUserCol("new_userid")
```

```
In [50]: # evaluation metric - RMSE

         eval_metric = RegressionEvaluator(predictionCol="prediction",
                                           labelCol="frequency",
                                           metricName="rmse")
```

```
In [51]: # hyper parameter space for cross validation - grid search

         ranks = [4, 6, 8, 10, 12, 14, 16]
         regParams = [0.1, 0.15, 0.25, 0.27, 0.3, 0.32, 0.35, 0.38]
         errors = [[0]*len(ranks)]*len(regParams)
         models = [[0]*len(ranks)]*len(regParams)
         min_error = float('inf')
         i = 0
```

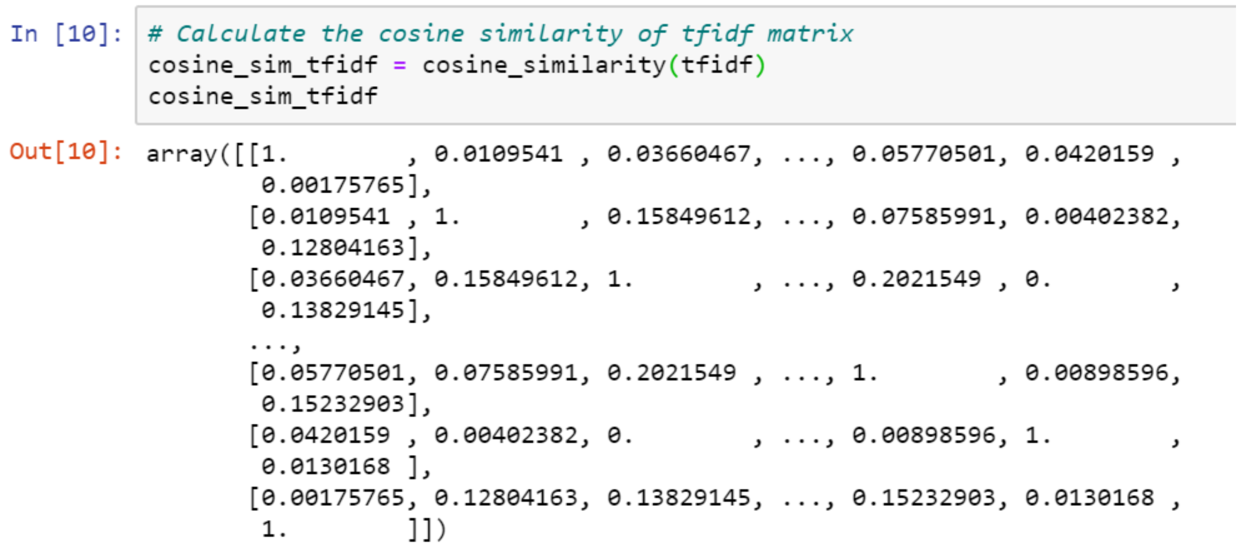
Figure 26. Collaborative Filter Building



```
In [3]: # tfidf features
        from sklearn.feature_extraction.text import TfidfVectorizer #alternatively, use TfidfTransformer()

        tfidf_vectorizer=TfidfVectorizer(min_df=1,
                                         norm='l2',
                                         smooth_idf=True,
                                         use_idf=True,
                                         ngram_range=(1,1)) #Since the original dataset only has tokens, we can only use unigram

        tfidf = tfidf_vectorizer.fit_transform(lyric['lyrics'])
```

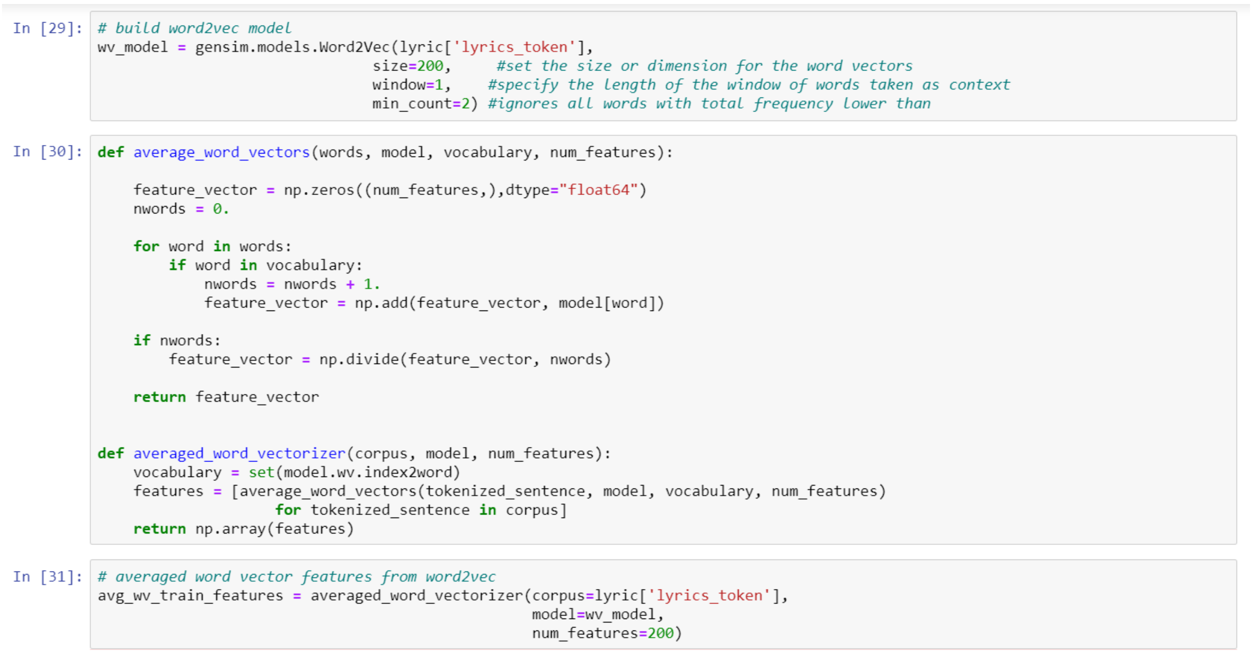
```
In [4]: tfidf

Out[4]: <142263x4788 sparse matrix of type '<class 'numpy.float64'>'
                with 7669181 stored elements in Compressed Sparse Row format>
```

Figure 27. TFIDF Matrix Creation



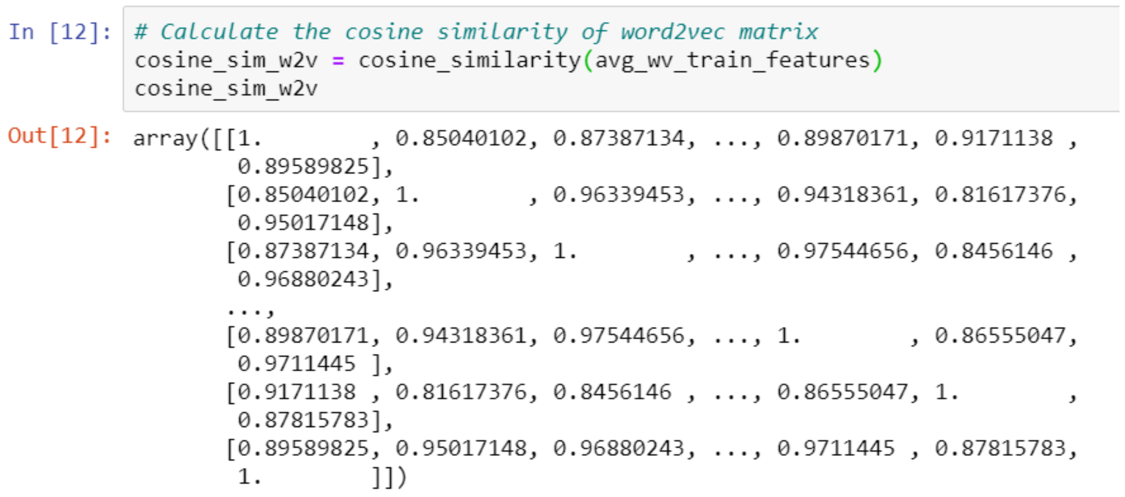
```
In [10]: # Calculate the cosine similarity of tfidf matrix
         cosine_sim_tfidf = cosine_similarity(tfidf)
         cosine_sim_tfidf

Out[10]: array([[1.        , 0.0109541 , 0.03660467, ..., 0.05770501, 0.0420159 ,
                 0.00175765],
                [0.0109541 , 1.        , 0.15849612, ..., 0.07585991, 0.00402382,
                 0.12804163],
                [0.03660467, 0.15849612, 1.        , ..., 0.2021549 , 0.        ,
                 0.13829145],
                ...,
                [0.05770501, 0.07585991, 0.2021549 , ..., 1.        , 0.00898596,
                 0.15232903],
                [0.0420159 , 0.00402382, 0.        , ..., 0.00898596, 1.        ,
                 0.0130168 ],
                [0.00175765, 0.12804163, 0.13829145, ..., 0.15232903, 0.0130168 ,
                 1.        ]])
```

Figure 28. Cosine Similarity Matrix Creation Based on TFIDF



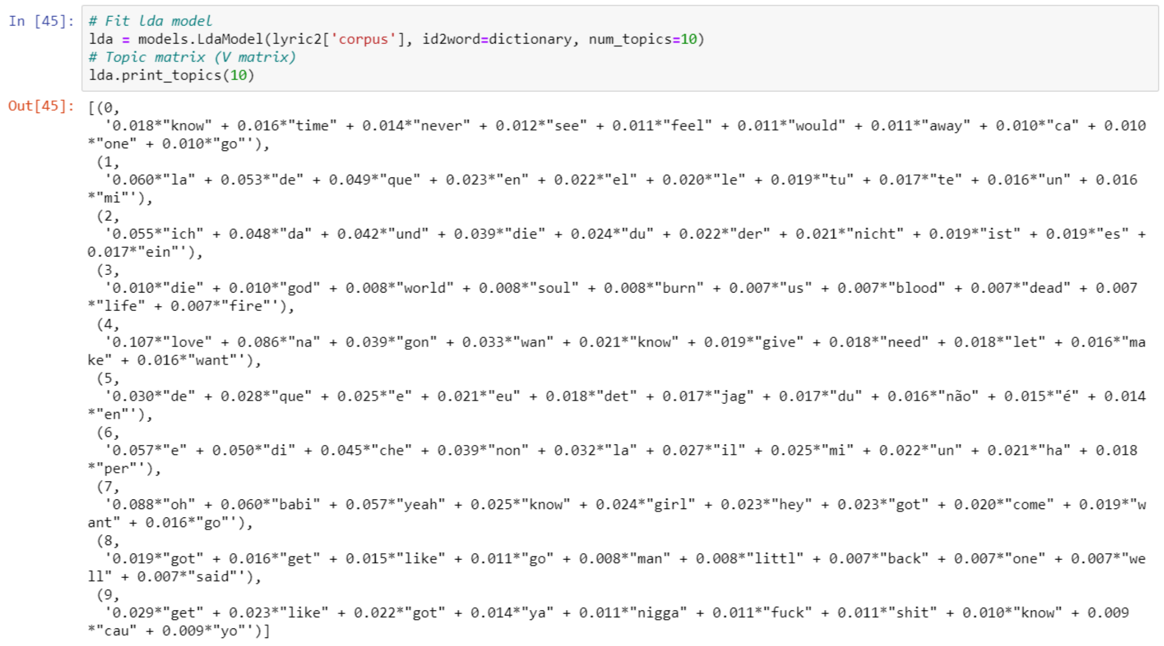
```
In [29]: # build word2vec model
         wv_model = gensim.models.Word2Vec(lyric['lyrics_token'],
                                           size=200,      #set the size or dimension for the word vectors
                                           window=1,     #specify the length of the window of words taken as context
                                           min_count=2) #ignores all words with total frequency lower than

In [30]: def average_word_vectors(words, model, vocabulary, num_features):

             feature_vector = np.zeros((num_features,),dtype="float64")
             nwords = 0.

             for word in words:
                 if word in vocabulary:
                     nwords = nwords + 1.
                     feature_vector = np.add(feature_vector, model[word])

             if nwords:
                 feature_vector = np.divide(feature_vector, nwords)

             return feature_vector


         def averaged_word_vectorizer(corpus, model, num_features):
             vocabulary = set(model.wv.index2word)
             features = [average_word_vectors(tokenized_sentence, model, vocabulary, num_features)
                         for tokenized_sentence in corpus]
             return np.array(features)

In [31]: # averaged word vector features from word2vec
         avg_wv_train_features = averaged_word_vectorizer(corpus=lyric['lyrics_token'],
                                                          model=wv_model,
                                                          num_features=200)
```

Figure 29. Word2vec Matrix Creation



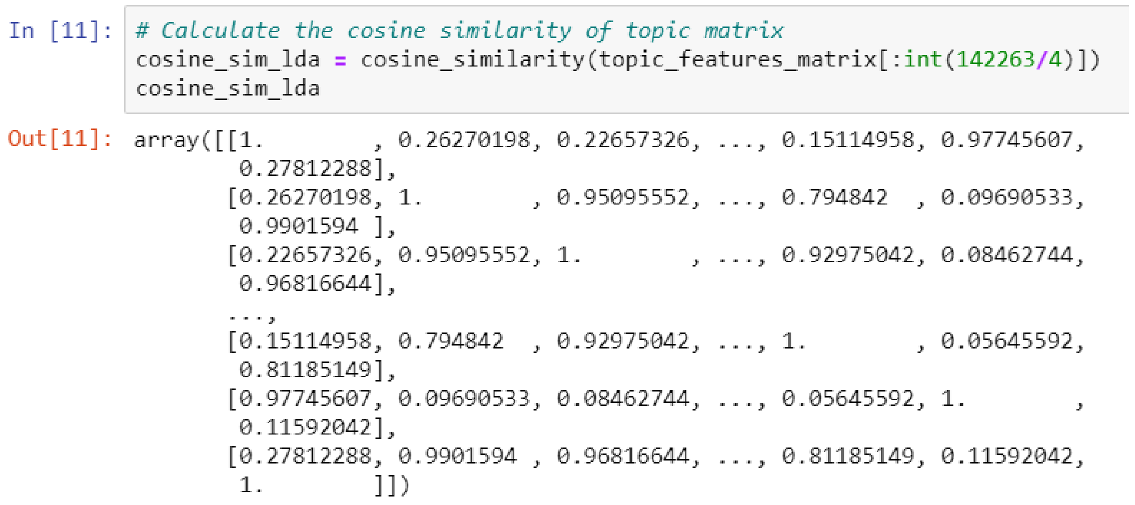
```
In [12]: # Calculate the cosine similarity of word2vec matrix
         cosine_sim_w2v = cosine_similarity(avg_wv_train_features)
         cosine_sim_w2v

Out[12]: array([[1.        , 0.85040102, 0.87387134, ..., 0.89870171, 0.9171138 ,
                 0.89589825],
                [0.85040102, 1.        , 0.96339453, ..., 0.94318361, 0.81617376,
                 0.95017148],
                [0.87387134, 0.96339453, 1.        , ..., 0.97544656, 0.8456146 ,
                 0.96880243],
                ...,
                [0.89870171, 0.94318361, 0.97544656, ..., 1.        , 0.86555047,
                 0.9711445 ],
                [0.9171138 , 0.81617376, 0.8456146 , ..., 0.86555047, 1.        ,
                 0.87815783],
                [0.89589825, 0.95017148, 0.96880243, ..., 0.9711445 , 0.87815783,
                 1.        ]])
```

Figure 30. Cosine Similarity Matrix Creation Based on Word2vec



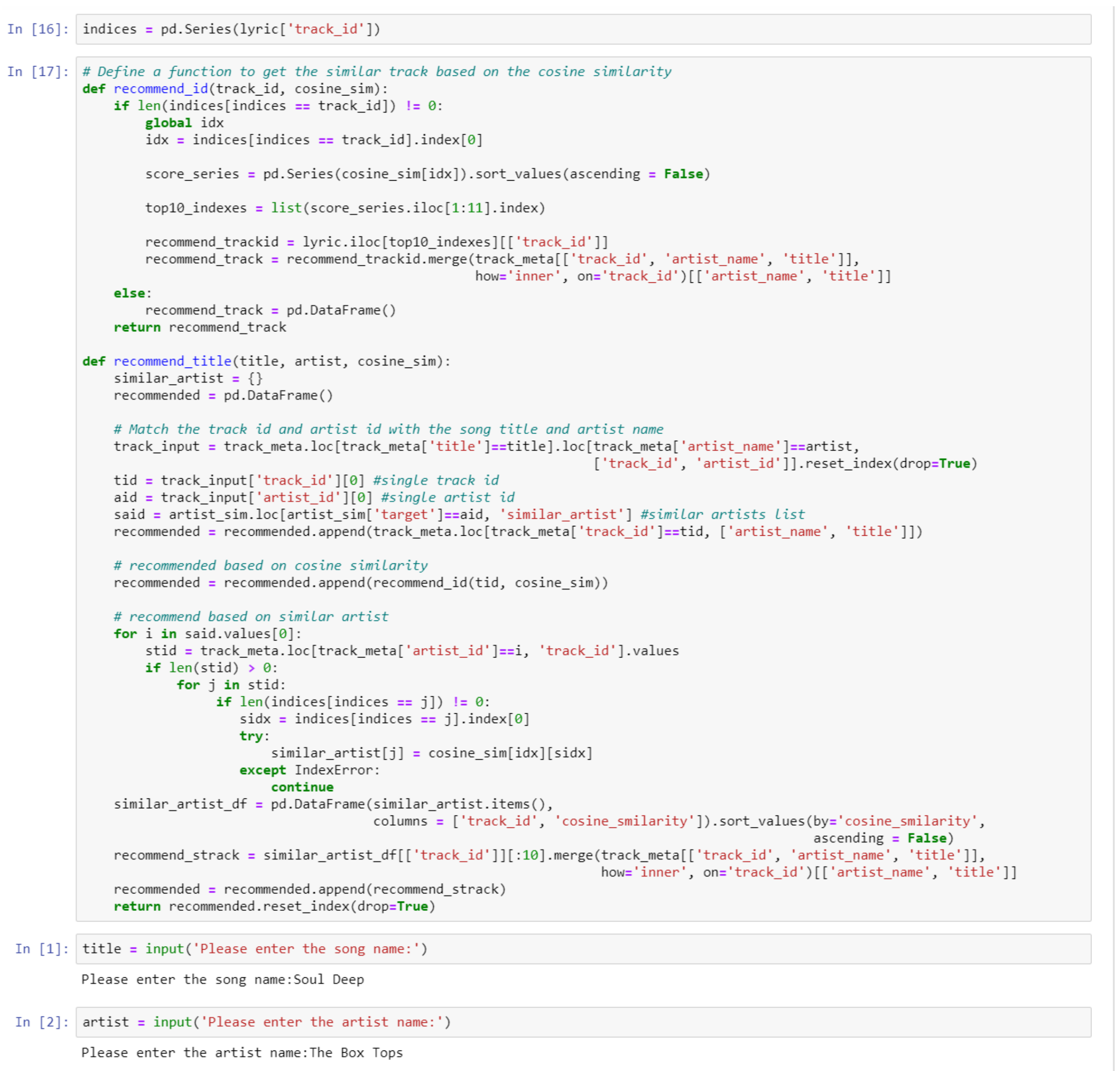
```
In [45]: # Fit lda model
         lda = models.LdaModel(lyric2['corpus'], id2word=dictionary, num_topics=10)
         # Topic matrix (V matrix)
         lda.print_topics(10)

Out[45]: [(0,
           '0.018*"know" + 0.016*"time" + 0.014*"never" + 0.012*"see" + 0.011*"feel" + 0.011*"would" + 0.011*"away" + 0.010*"ca" + 0.010
          *"one" + 0.010*"go"'),
          (1,
           '0.060*"la" + 0.053*"de" + 0.049*"que" + 0.023*"en" + 0.022*"el" + 0.020*"le" + 0.019*"tu" + 0.017*"te" + 0.016*"un" + 0.016
          *"mi"'),
          (2,
           '0.055*"ich" + 0.048*"da" + 0.042*"und" + 0.039*"die" + 0.024*"du" + 0.022*"der" + 0.021*"nicht" + 0.019*"ist" + 0.019*"es" +
          0.017*"ein"'),
          (3,
           '0.010*"die" + 0.010*"god" + 0.008*"world" + 0.008*"soul" + 0.008*"burn" + 0.007*"us" + 0.007*"blood" + 0.007*"dead" + 0.007
          *"life" + 0.007*"fire"'),
          (4,
           '0.107*"love" + 0.086*"na" + 0.039*"gon" + 0.033*"wan" + 0.021*"know" + 0.019*"give" + 0.018*"need" + 0.018*"let" + 0.016*"ma
          ke" + 0.016*"want"'),
          (5,
           '0.030*"de" + 0.028*"que" + 0.025*"e" + 0.021*"eu" + 0.018*"det" + 0.017*"jag" + 0.017*"du" + 0.016*"não" + 0.015*"é" + 0.014
          *"en"'),
          (6,
           '0.057*"e" + 0.050*"di" + 0.045*"che" + 0.039*"non" + 0.032*"la" + 0.027*"il" + 0.025*"mi" + 0.022*"un" + 0.021*"ha" + 0.018
          *"per"'),
          (7,
           '0.088*"oh" + 0.060*"babi" + 0.057*"yeah" + 0.025*"know" + 0.024*"girl" + 0.023*"hey" + 0.023*"got" + 0.020*"come" + 0.019*"w
          ant" + 0.016*"go"'),
          (8,
           '0.019*"got" + 0.016*"get" + 0.015*"like" + 0.011*"go" + 0.008*"man" + 0.008*"littl" + 0.007*"back" + 0.007*"one" + 0.007*"we
          ll" + 0.007*"said"'),
          (9,
           '0.029*"get" + 0.023*"like" + 0.022*"got" + 0.014*"ya" + 0.011*"nigga" + 0.011*"fuck" + 0.011*"shit" + 0.010*"know" + 0.009
          *"cau" + 0.009*"yo"')]
```

Figure 31. Topic Model Creation



```
In [11]: # Calculate the cosine similarity of topic matrix
         cosine_sim_lda = cosine_similarity(topic_features_matrix[:int(142263/4)])
         cosine_sim_lda

Out[11]: array([[1.        , 0.26270198, 0.22657326, ..., 0.15114958, 0.97745607,
                 0.27812288],
                [0.26270198, 1.        , 0.95095552, ..., 0.794842  , 0.09690533,
                 0.9901594 ],
                [0.22657326, 0.95095552, 1.        , ..., 0.92975042, 0.08462744,
                 0.96816644],
                ...,
                [0.15114958, 0.794842  , 0.92975042, ..., 1.        , 0.05645592,
                 0.81185149],
                [0.97745607, 0.09690533, 0.08462744, ..., 0.05645592, 1.        ,
                 0.11592042],
                [0.27812288, 0.9901594 , 0.96816644, ..., 0.81185149, 0.11592042,
                 1.        ]])
```

Figure 32. Cosine Similarity Matrix Creation Based on Topic Model



```
In [16]: indices = pd.Series(lyric['track_id'])
```

```
In [17]: # Define a function to get the similar track based on the cosine similarity
def recommend_id(track_id, cosine_sim):
    if len(indices[indices == track_id]) != 0:
        global idx
        idx = indices[indices == track_id].index[0]

        score_series = pd.Series(cosine_sim[idx]).sort_values(ascending = False)

        top10_indexes = list(score_series.iloc[1:11].index)

        recommend_trackid = lyric.iloc[top10_indexes][['track_id']]
        recommend_track = recommend_trackid.merge(track_meta[['track_id', 'artist_name', 'title']],
                                                  how='inner', on='track_id')[['artist_name', 'title']]
    else:
        recommend_track = pd.DataFrame()
    return recommend_track

def recommend_title(title, artist, cosine_sim):
    similar_artist = {}
    recommended = pd.DataFrame()

    # Match the track id and artist id with the song title and artist name
    track_input = track_meta.loc[track_meta['title']==title].loc[track_meta['artist_name']==artist,
                                                                 ['track_id', 'artist_id']].reset_index(drop=True)
    tid = track_input['track_id'][0] #single track id
    aid = track_input['artist_id'][0] #single artist id
    said = artist_sim.loc[artist_sim['target']==aid, 'similar_artist'] #similar artists list
    recommended = recommended.append(track_meta.loc[track_meta['track_id']==tid, ['artist_name', 'title']])

    # recommended based on cosine similarity
    recommended = recommended.append(recommend_id(tid, cosine_sim))

    # recommend based on similar artist
    for i in said.values[0]:
        stid = track_meta.loc[track_meta['artist_id']==i, 'track_id'].values
        if len(stid) > 0:
            for j in stid:
                 if len(indices[indices == j]) != 0:
                    sidx = indices[indices == j].index[0]
                    try:
                        similar_artist[j] = cosine_sim[idx][sidx]
                    except IndexError:
                        continue
    similar_artist_df = pd.DataFrame(similar_artist.items(),
                                     columns = ['track_id', 'cosine_smilarity']).sort_values(by='cosine_smilarity',
                                                                                            ascending = False)
    recommend_strack = similar_artist_df[['track_id']][:10].merge(track_meta[['track_id', 'artist_name', 'title']],
                                                                  how='inner', on='track_id')[['artist_name', 'title']]
    recommended = recommended.append(recommend_strack)
    return recommended.reset_index(drop=True)
```

```
In [1]: title = input('Please enter the song name:')

        Please enter the song name:Soul Deep
```

```
In [2]: artist = input('Please enter the artist name:')

        Please enter the artist name:The Box Tops
```

Figure 33. Content-Based Filter Building