Million Playlist Challenge

by Domiziano Scarcelli - 1872664

Introduction

Recommender System for Playlist Continuation

- Challenge started in 2018
- Hosted on AIcrowd
- More than 1000 submissions

Dataset:

- 1 Million Playlists
- 63 Million total songs
- 2 Million unique songs
- 345K unique artists

Playlists created by users from 2010 to 2017

A song recommender is an essential feature to enable an easy discoverability of new songs.

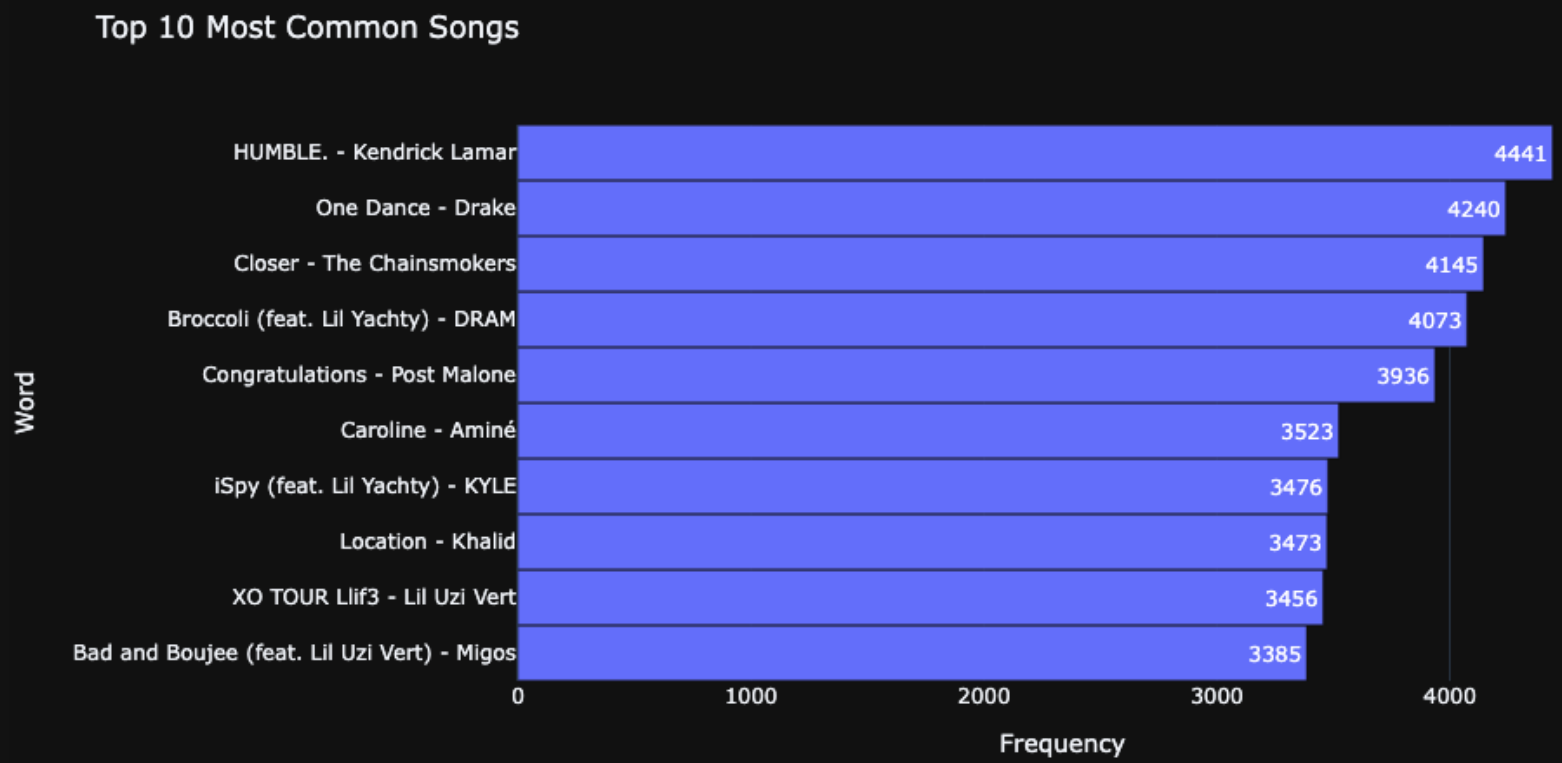
Reduced Dataset

Sampled 10% of the playlists

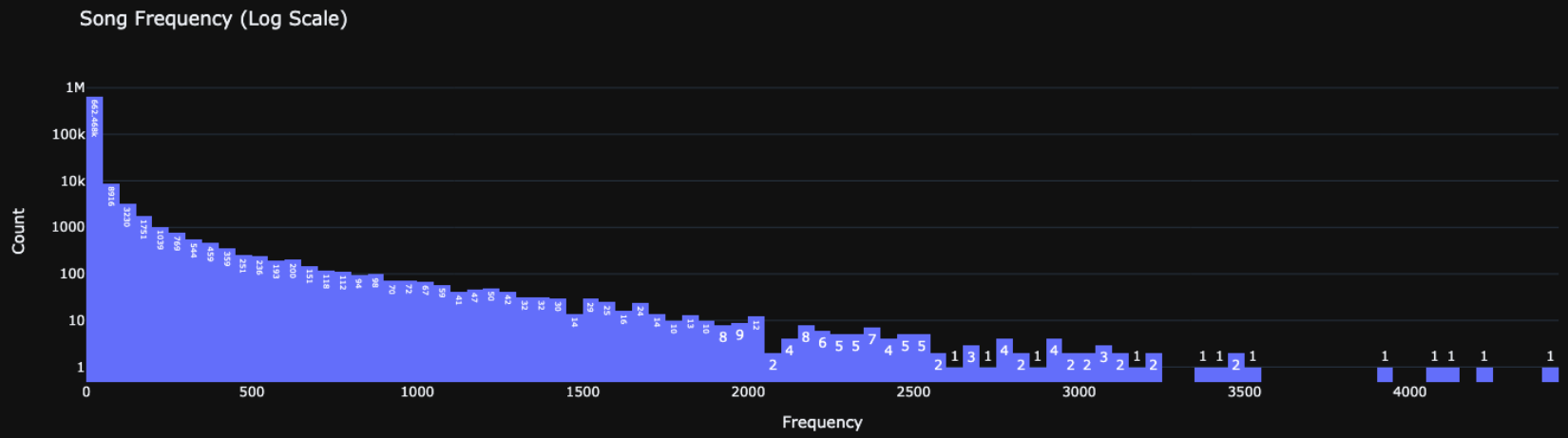
- 100K Playlists
- 681,805 Unique Songs
- 110,063 Artists
- In average:
  - 66 songs per playlist
  - 38 unique artists per playlist

From here, build a distributed recommender system that given a playlist, it recommends new relevant songs that continuate it.

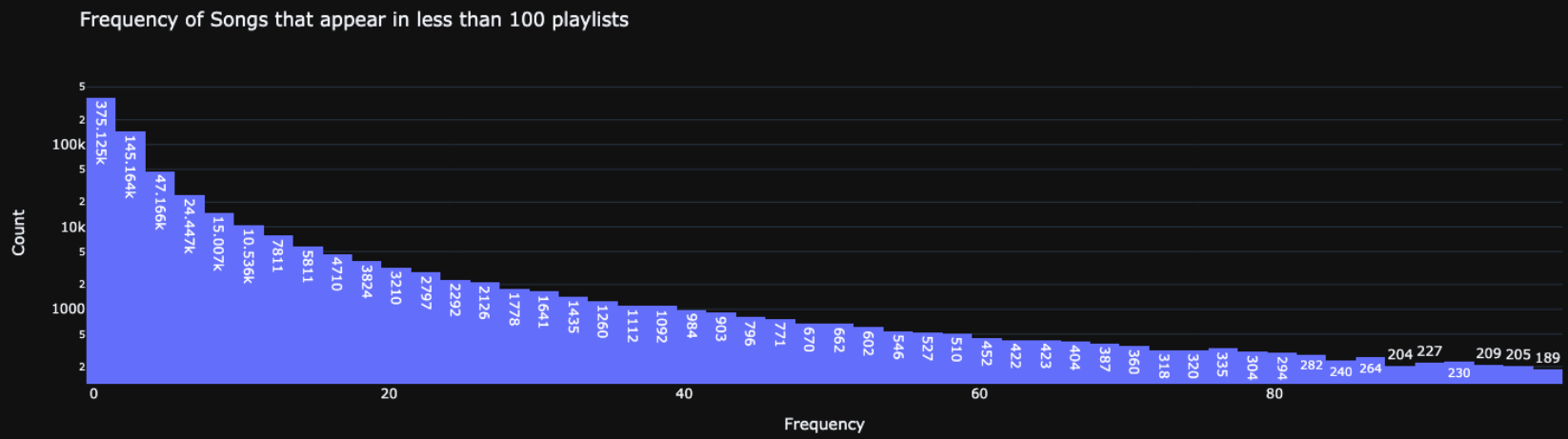
Data Visualization



Songs Frequency (Logarithmic scale)

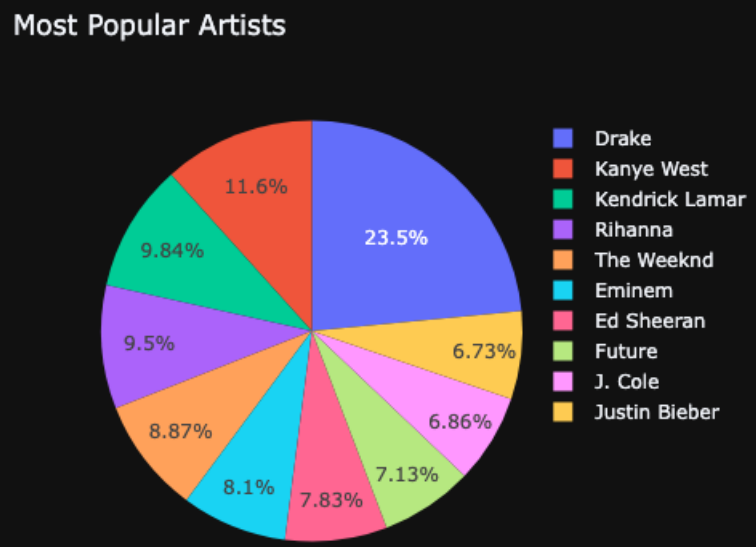


Frequency of songs that appear in less that 100 playlists (Logarithmic Scale)

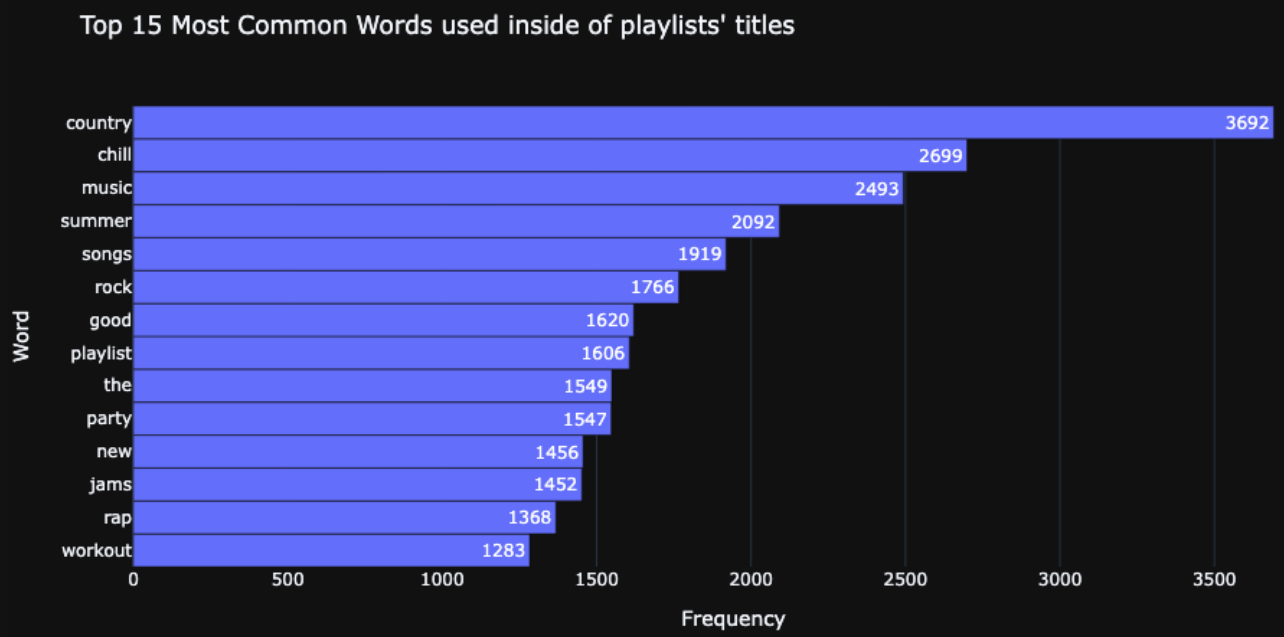


In average each song appears in only 10 playlists!

Most popular artists



Top 15 most used words inside playlist titles



Developed Systems

User-Based Collaborative Filtering;

Item-Based Collaborative Filtering;

Neural Network Approach:

Implemented by using a Denoising Autoencoder developed by the 2nd place solution of the challenge.

How was the data structured

1,000 json files


```
{
"info": {
   "generated_on": "2017-12-03 08:41:42.057563",
   "slice": "0-999",
   "version": "v1"
},
"playlist": {
      "name": "musical",
      "collaborative": "false",
      "pid": 5,
      "modified_at": 1493424000,
      "num_albums": 7,
      "num_tracks": 12,
      "num_followers": 1,
      "num_edits": 2,
      "duration_ms": 2657366,
      "num_artists": 6,
      "tracks": [
         {
```

User-Based Collaborative Filtering - Data Preparation

How the playlists are encoded

1. Map each song in the playlist to a position.

track_uri	pos
track_1	0
track_10	1
track_11	2

2. Create the encoding vector:

   1 in the i-th position, if the song at position i is in the playlist;

   0 otherwise.

$$p = [0, 0, 0, 0, 0, 1, 0, 1, 0, 1]$$

This means that the playlist p has the songs with positions [5, 7, 9]

Average of 66 songs in a playlist, the vectors are 681,805 dimensional and so they are very sparse (99.9903% sparseness).

Memory efficiency via pyspark's `SparseVector`, which stores only the indices and the values.

Generate the Recommendations

The pipeline for the recommendation, given the `SparseVector` of a playlist that has to be continuated, is the following:

1. Compare the playlist with each other playlist, computing the pair-wise similarity using te Jaccard Similarity between their vectors. This will output the similarity value $\in [0, 1]$

track_uri	vector	input_vector	similarity
track_1	indices=1,3,4	indices=0,2,4,10	0.2
track_10	indices=4,6,10	indices=0,2,4,10	0.5
track_11	indices=3,5	indices=0,2,4,10	0.0

2. Take the top-k vectors with the highest similarity value;
3. Aggregate the k vectors, averaging them by their similarity value.

4. Normalize the values dividing by the sum of the k similarity values.

$$p_1 = [0, 1, 1, 0, 0, 1, 0] \quad s_1 = 0.3$$
$$p_2 = [0, 0, 1, 0, 1, 0, 0] \quad s_2 = 0.5$$
$$p_3 = [1, 0, 1, 0, 1, 0, 0] \quad s_3 = 0.45$$

$$p_{\text{agg}} = [0.45, 0.3, 1.25, 0, 0.95, 0.3, 0] \quad s_{\text{sum}} = 1.25$$

$$p_{\text{normalized}} = [0.36, 0.24, 1.0, 0.0, 0.76, 0.24, 0.0]$$

5. From the normalized aggregated vector, remove the songs that already appears in the input playlist;
6. The top-n indices with the highest values will be the recommended songs;

$$n = 3 \quad \text{recommendations} = \{2 : 1.0, 4 : 0.76, 0 : 0.36\}$$

7. Take the `song_uri` of the songs that are mapped into those indices to get the details.

The entire process takes about 30 seconds.

Item-Based Collaborative Filtering - Data Preparation

Differently from User-Based CF, here the tracks are encoded instead of playlists.

Same principle:

1. Map each playlist into a position.

pid	pos
pid_1	0
pid_2	1
pid_3	2

2. Create the encoding vector for each track:

   1 in the i-th position, if the playlist at position i contains the song;

   0 otherwise.

$$s = [0, 0, 0, 0, 0, 1, 0, 1, 0, 1]$$

The song s appears in the playlists 5, 7, 9.

The vector is still very sparse, but its dimensionality is 110,063 instead of 681,805.

Since a playlist appears in an average of 10 playlists, we have a degree of sparseness of 99.9909% (w.r.t. 99.9903% of the user-based cf).

Generate the Recommendations

Given a playlist to continuate, represented as a `DataFrame` containing its songs vectors, the recommendation pipeline is the following:

1. Compute the k-nearest-neighbours for each track in the playlist. This will result in a collection of T dataframes, where $|T|$ is the number of songs in the playlist. A dataframe relevant to the track t has a list of k songs, each one with the distance from t in a separated column;
2. Aggregate each Dataframe $\in T$ in order to have a single dataframe. Since we are sure that the size of T is not big, I first convert the dataframes into python dictionaries

```
{
   track_uri: distance
}
```

The aggregation produce a python dictionary like this:

```
{
   track_uri_1: [0.2],
   track_uri_2: [0.25, 0.45],
   track_uri_3: [0.31, 0.40, 0.36],
   track_uri_4: [0.1],
}
```

3. Convert the dictionary into a pyspark `DataFrame`, averaging the values inside of each list

track_uri	distance
track_uri_1	0.2
track_uri_2	0.35
track_uri_3	0.356
track_uri_4	0.1

4. Remove from the `DataFrame` the songs that already are in the playlist
5. Order the `DataFrame` by ascending distances, and take the top-n tracks as recommendations.

if n = 3, recommendations:

track_uri	distance
track_uri_4	0.1
track_uri_1	0.2
track_uri_2	0.35
track_uri_3	0.356

The entire process takes about 30 to 60 seconds.

K-Neighbours with LSH

- Precise k-neighbours search is too expensive
- *Locally Sensitive Hashing* with pyspark's `MinHashLSH` class.
- Number of hash tables = 20
  - Higher: more precise, less fast
  - Lower: less precise, faster

We can pre-compute the entire set of k-nearest neighbour to be even faster.

This takes a long time, but has to be done just once.

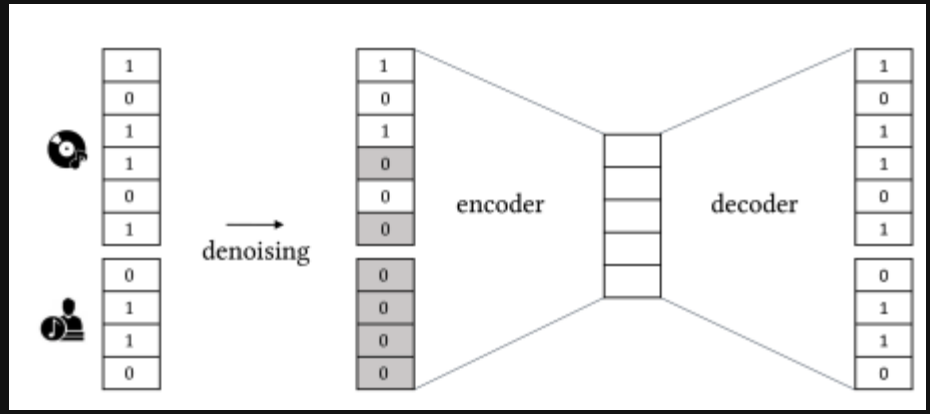
Neural Network Approach - Introduction

Solution taken by the "Hello World" team, which classified in 2nd place in the challenge.

- Denoising Autoencoder that takes Tracks and Artits
- Character level CNN that takes playlist's Title.
- Ensable to make a prediction.

For simplicity, I consider just the Denoising Autoencoder model.

The Autoencoder



Input: Concatenation between songs and artists in the playlist, encoded in the following way:

Songs encoding: the same as User-Based CF;

Artists encoding: 1 if artist in playlist, 0 otherwise.

It reconstruct the input playlist. The intuition is that songs with high values in the reconstructed vectors are relevant for the input playlist.

Noise Generation

Noise to let the model generalize.

Training with Hide & Seek technique:

$$\text{input} = [\underbrace{0, 0, 1, 0, 1, 1}_{\text{songs}}, \underbrace{0, 1, 0}_{\text{artists}}]$$

Each iteration the song vector or the artist vector are masked.

Mask playlist: $\text{input}[\underbrace{0, 0, 0, 0, 0, 0}_{\text{songs}}, \underbrace{0, 1, 0}_{\text{artists}}]$

Mask artists: $\text{input} = [\underbrace{0, 0, 1, 0, 1, 1}_{\text{songs}}, \underbrace{0, 0, 0}_{\text{artists}}]$

The model can learn intra-relationship between artists and tracks.

Dropout with probability p that a node is kept in the network sampled between $(0.5, 0.8)$ as regularization, and to let the model learn inter-relationship between tracks and between artists.

Training

Petastorm to generate DataLoader from pyspark `DataFrame`.

At training time, the model is fed with mini-batches of 100 playlists.

The loss function is the Binary Cross Entropy loss:

$$\mathcal{L}(\mathbf{p}, \hat{\mathbf{p}}) = -\frac{1}{n} \sum_{\mathbf{p} \in \mathbf{P}} p_i \log \hat{p}_i + \alpha \left(1 - p_i\right) \log \left(1 - \hat{p}_i\right)$$

where p_i is the input concatenated vector, and $\hat{p}_i$ is the reconstructed vector.

α = 0.5 is the hyperparameter weighting factor. Balances the importance between observed values (1s) and missing values 0s in the vector.

Same hyperparameters used by the authors of the paper, but smaller learning rate, since I have a smaller dataset to work with.

The whole training (pretrain + train) took about 12 hours on CUDA GPUs.

Validation

In order to do *Early Stopping*, and save the model parameters that achieve the best metrics, at the end epoch the model is evaluated on a validation set.

This is done for both the pretraining (tied weights) and training.

Performance Evaluation

How is the test set built

- Different splits for models with and without training
- If there is no training, split at track level
- It there is training, split at row level, and then at track level

User-based and Item-based CF

Split at track level (75%, 25%)

Original `DataFrame`

pid	vector
0	indices=1,3,4,10,11,23
1	indices=4,6,10,12,34,56
2	indices=3,5,6,8,9,10

Training set (75% of the tracks)

pid	vector
0	indices=1,3,10,23
1	indices=4,10,56
2	indices=3,6,8, 10

Test set (25% of the tracks)

pid	vector
0	indices=4,11
1	indices=6,12,34
2	indices=5,9

Neural Network Based

Create 3 `DataFrames`: Train, Validation, Test

Original `DataFrame`

pid	vector
0	indices=1,3,4,10,11,23
1	indices=4,6,10,12,34,56
2	indices=3,5,6,8,9,10
3	indices=1,2,5,8,10,11
4	indices=0,1,5,8,11,21,34,53

Training Set

(98.5K playlists)

pid	vector
0	indices=1,3,4,10,11,23
1	indices=4,6,10,12,34,56
2	indices=3,5,6,8,9,10

Validation Set

(500 playlists)

pid	vector
3	indices=1,2,5,8,10,11

Test Set

(1,000 playlists)

pid	vector
4	indices=0,1,5,8,11,21,34,53

Then I split at track level the Evaluation and Test set, in order to compute evaluation metrics

No need to further split the Training set.

Evaluation Metrics

Evaluation metrics used for Performance Evaluation

G is the ground truth (tracks in Test set) and R is the set of recommended tracks

R-precision: ratio between correct and incorrect recommended songs

$$\text{Rprec} = \frac{G \cap R_{1:|G|}}{|G|}$$

Normalized Discounted Cumulative Gain: how much the correct recommended songs are up in the list. We define rel_i = 1 if the track with index i is in the ground truth, otherwise rel_i = 0.

$$NDCG = \frac{DCG}{IDCG}$$

$$DCG = rel_1 + \sum_{i=2}^{|R|} \frac{rel_i}{\log_2 i} \quad \text{and} \quad IDCG = 1 + \sum_{i=2}^{|G \cap R|} \frac{1}{\log_2 i}$$

Performance Comparison

Evaluation on 1,000 playlists

User-Based CF

$Rprec$ = 0.1023, $NDCG$ = 0.256

Item-Based CF:

10 Hash Tables: $Rprec$ = 0.0847, $NDCG$ = 0.242

20 Hash Tables: $Rprec$ = 0.0897, $NDCG$ = 0.261

Denoising Autoencoder:

$Rprec$ = 0.1327, $NDCG$ = 0.334

Winners of the challenge:

$Rprec$ = 0.220, $NDCG$ = 0.3858

User-Based better than Item-Based because of lower sparsity!

Conclusions

In general both User-Based and Neural Netork Based have good performances.

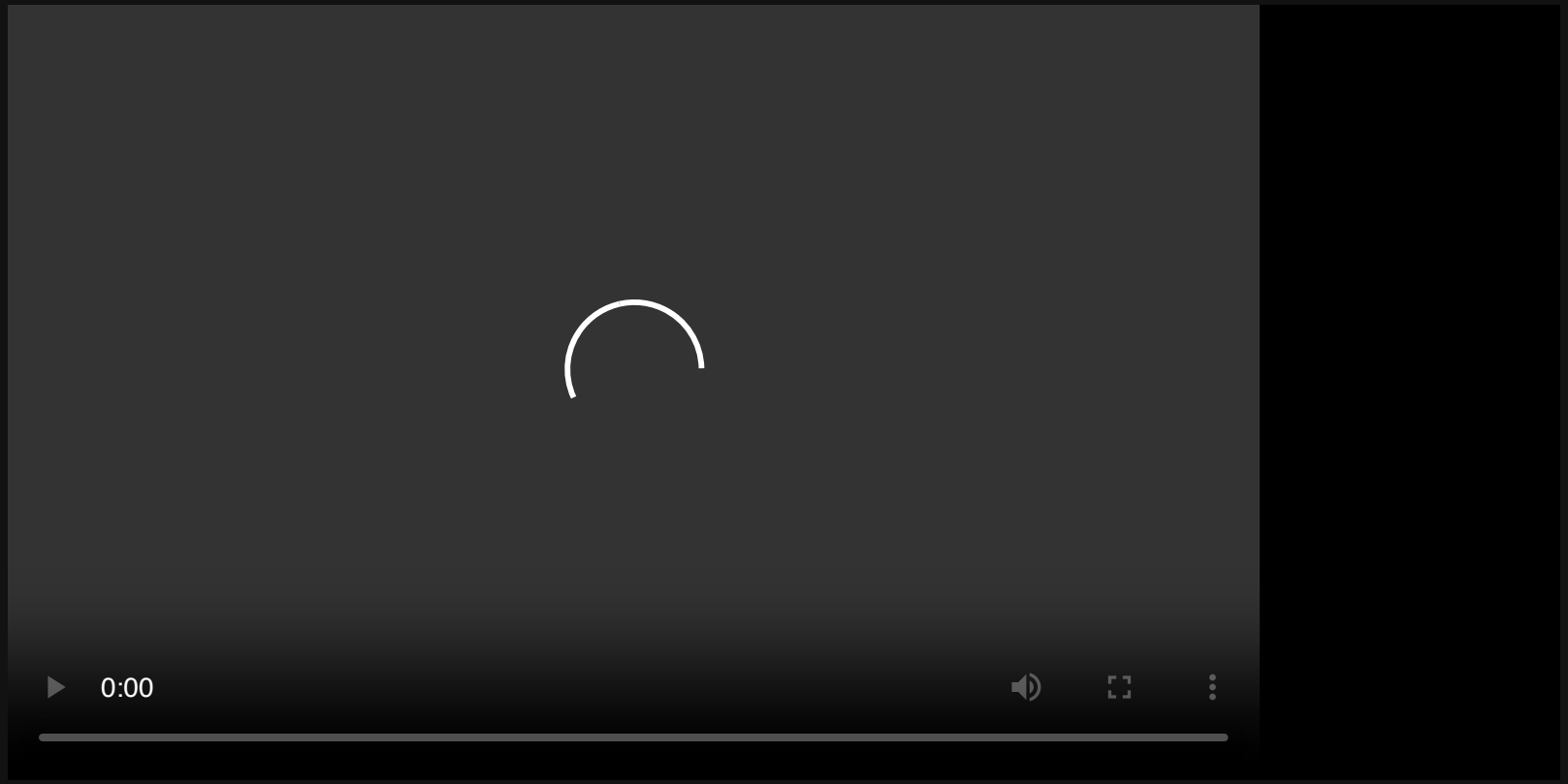
User-Based, even if simple, has good results;

Item-Based could have done better if less sparsity;

Denoising Autoencoder could have learnt better patterns if less unpopular songs.

Better than all the others because of faster inference time.

Web Application Demo



Thank you for the attention

References

AIcrowd Spotify Million Playlist Challenge.

hello world! [Yang et al.]