RecSys Challenge 2018: playlist continuation based on shared neighborhood-methods, matrix factorization and audio-feature classification

Blauensteiner, Timon Cedric
timon.blauensteiner@uni-konstanz.de

Metz, Yannick
yannick.metz@uni-konstanz.de

August 29, 2018

Abstract

As part of the ACM RecSysChallenge 2018 in cooperation with Spotify, we created a recommendation system for music. We predict songs for different metadata and number of seed tracks using a combination of different approaches: Neighborhood-exploration, in combination with a latent feature representation for larger numbers of seed tracks and ranking of tracks based on an audio feature neural network. The neighborhood approach is an efficient way to generate good results, which can be improved inexpensively by matrix factorization and matching of audio features. With our approach, we reached a top 20% placement at the competition.

1 Challenge

1.1 Overview

Spotify is a subscription-based music streaming service where users can create private or public playlists of arbitrary songs. After the user specifies a playlist name Spotify helps the user to pick songs for their playlist by suggesting songs to be added to the playlist. Those recommended songs are shown in groups of ten and can be previewed to decide for the user if the song matches the playlist and should be added. Figure 1 illustrates how the feature looks like in the Spotify service.

In the context of the RecSysChallenge 2018 Spotify asks academics to compete on a public leaderboard.

The task is to build a recommender system that recommends 500 unique songs (or tracks) for a given playlist based on tracks already in the playlist called *seed tracks* and a playlist name. Overall, recommendations for 10 different configurations with 1,000 playlists each (i.e. for a total of 10,000 playlists) have to be created:

- Playlist name only
- Playlist name, first track
- Playlist name, first 5 tracks
- First 5 tracks (no playlist name)
- Playlist name, first 10 tracks
- First 10 tracks (no playlist name)
- Playlist name, first 25 tracks
- Playlist name, random 25 tracks
- Playlist name, first 100 tracks
- Playlist name, random 100 tracks

To validate the quality of the contestant's recommendations Spotify uses existing playlists, choosing some tracks as seed tracks, keeping some *holdout tracks* and measuring how well the recommendations based on playlist name and seed tracks match the holdout tracks of each playlist.

Those holdout tracks are used to compute three scores measuring different properties which are then combined via Borda Count. Borda count ranks the n teams in a leaderboard by each score separately and combines them by giving for rank position r each $n - r$ points to a team [1] (i.e. a team ranked number one in all three scores has a total of $3n - 3$ points, a team ranked last in all three scores has 0 points, and a team ranked number one in one and number two in the two other scores has $3n - 5$ points).

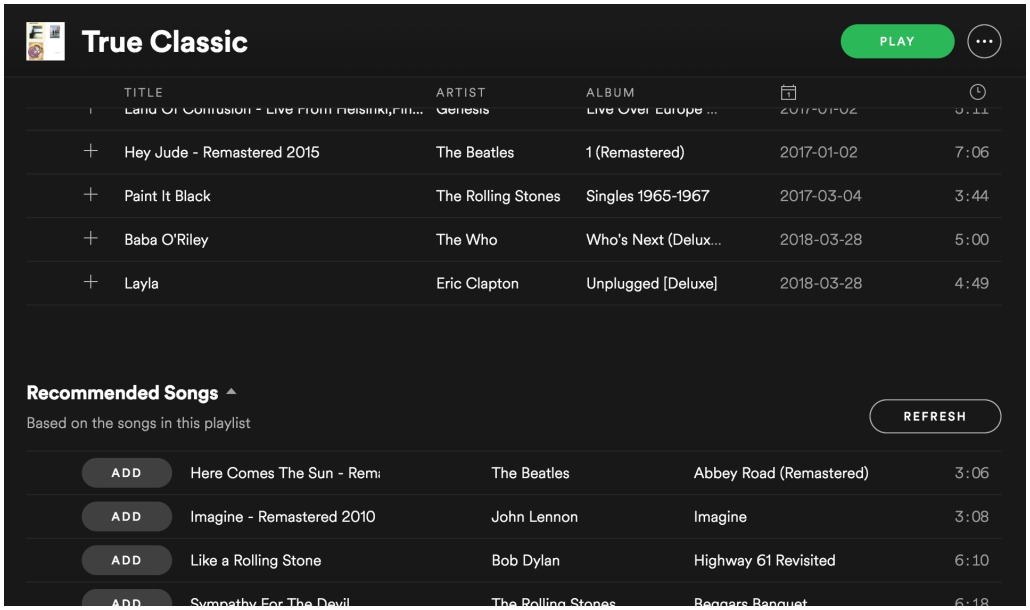


Figure 1: Screenshot of the Spotify Desktop App. The playlist name is *True Classic*, the first list shows some of the songs in the playlist and at the bottom some recommended songs can be seen, with the refresh button to request ten more songs to be recommended.

There are two leaderboards, the main track where only the provided data set can be used for training recommender models and the creative track where any additional public and freely available data can be used for refining the models.

1.2 Scores

G is the set of holdout tracks and R the (ordered) set of recommended tracks, with $R_{1:n}$ specifying the first n recommendations. The submission scores are the means of each score per playlist. $A(T)$ is the set of artists for a set of tracks T.

R precision R precision measures the proportion of matching tracks within the first $|G|$ recommendations. If a recommended track does not match but the artist does 0.25 is credited:

$$Rprec = \frac{|G \cap R_{1:|G|}| + 0.25|A(G) \cap A(R_{1:|G|} \setminus G)|}{|G|}$$

NDCG **N**ormalized **d**iscounted **c**umulative **g**ain measures the quality of order of the recommendations in the range of $[0, 1]$ with 1 for the perfect order:

$$DCG = \sum_{i=1}^{|R|} \frac{R_i \in G}{\log_2(i+1)}; \quad IDCG = \sum_{i=1}^{|G|} \frac{1}{\log_2(i+1)}; \quad NDCG = \frac{DCG}{IDCG}$$

Song clicks Song clicks is a Spotify specific score essentially measuring the position of the first match. Its core idea is to measure how many times the list of 10 recommendations in the Spotify service needs to be refreshed until the first song will be added to the playlist:

$$clicks = \left\lfloor \frac{\underset{i=1..500}{\text{argmin}}\ R_i \in G - 1}{10} \right\rfloor$$

2 Data set

To solve the RecSysChallenge Spotify provides a dataset of one million user-generated playlists (short MPD) sampled from billions of playlists with specific criteria [2]:

- Creator is US resident, at least 13 years old
- public playlist with 5 to 250 tracks
- common playlist name (shared by other playlists in the MPD)
- 3+ artists, 2+ albums, 1+ followers
- a minimum number of listeners (not specified further)
- contains no local (non-Spotify) track
- created between 01/01/2010 and 12/01/2017
- has no offensive playlist name or *adult-oriented* created by minors

2.1 Data schema

The training data was provided as 10,000 individual JSON-files, a way to store nested data structures. For each playlist, a number of attributes were given:

attribute	description
name (*)	Necessary title of the playlist, may contain special characters
pid	Unique identifier of playlist
num_tracks *	Number of tracks
num_albums/num_artists	Number of unique albums/artists in playlist
modified_at	Date of the last modification (given in UNIX-time)
num_followers	Number of users who follow the playlist besides the owner
collaborative	Whether a playlist is curated by multiple users simultaneously
num_edits	How often the playlist has been modified since creation
duration_ms	Combined length of all tacks in ms
(description)	Optional argument: Longer description given by users

Table 1: Attributes provided for each playlist in the dataset. The attributes marked with a * are also available for the challenge playlists.

Naturally, not every attribute is meaningful in the context of playlist continuation. While the description attribute may contain useful information, it is only available for a small fraction (around 1.7%) of all playlists. Additionally, most attributes (such as *collaborative*, *modified_at* or *num_followers*) are not available for the challenge data and thus cannot be used for predicting directly.

Each playlist then contains a number of tracks, and for each track we also get some information:

attribute	description
pos	Position of a track in the playlist
track_name/track_uri	Name of the track/Unique identifier of track
album_name/album_uri	Name of album/Unique identifier of album
artist_name/artist_uri	Name of the artist/Unique identifier of the artist
duration_ms	Duration of song in milliseconds

Table 2: Attributes provided for every single track. Here the given attributes are identical between training and challenge dataset.

2.2 Exploration

We wanted to get insights from different statistics and distributions in the MPD. That can be used to plan out a sufficient recommendation strategy and are important for resource management.

We looked at the distribution of playlist lengths (Figure 2), how often single tracks appear across all playlists (Figure 3), and how many common tracks (or neighbors) a track has in playlists (Figure 4).

First, we looked at the distribution of playlist lengths:

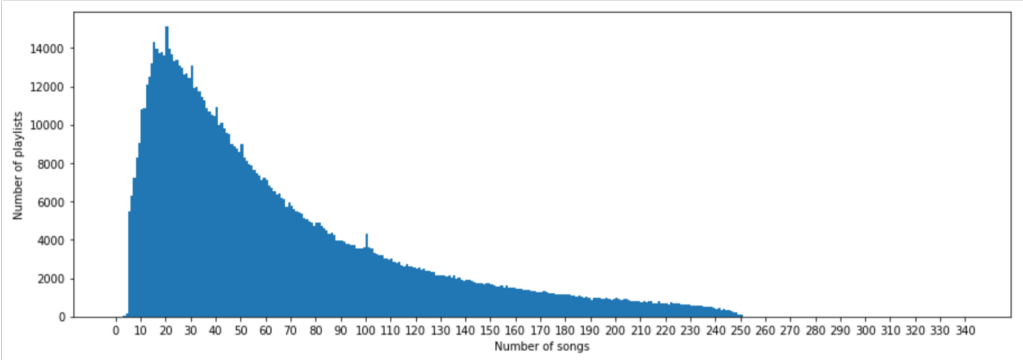


Figure 2: A large part of the playlists in the training set contains around 30-50 tracks. The number of tracks in a playlist is limited to 250 tracks.

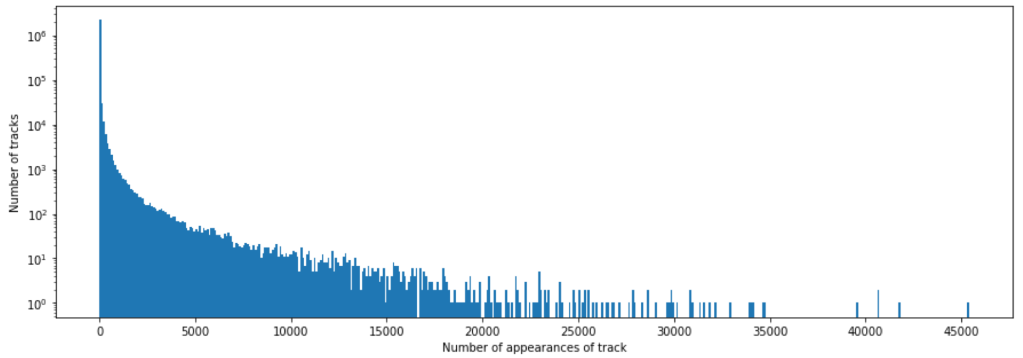


Figure 3: How often a track appears in the million playlists? The statistic shows us that is not sufficient to just recommend popular tracks as they only account for a small number of tracks in playlists. E.g. the most occurring track, ".HUMBLE" by the American rapper Kendrick Lamar appears around 45,000 times, so roughly in every 20th playlist.

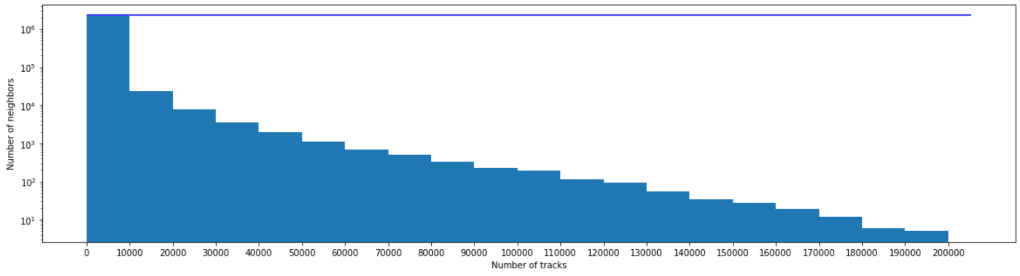


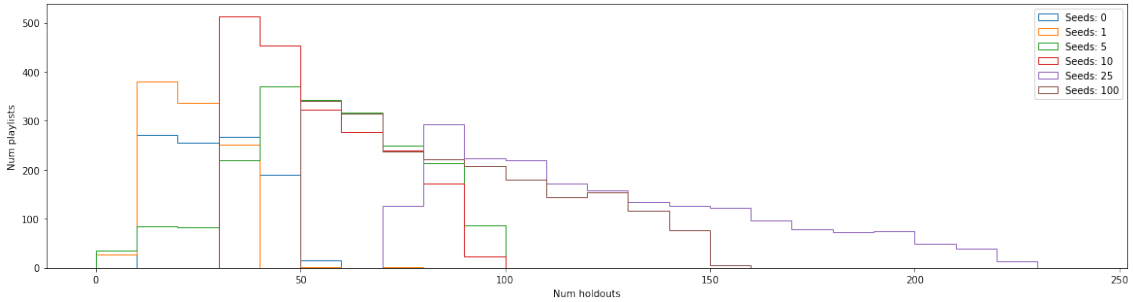
Figure 4: This figure shows the number of direct "neighbors", tracks that appear together with a song, in a playlist. For the tracks in the first bin, the average neighborhood size exceeds 1 million tracks. This is important for our main graph-based approach, we looked at the first-order neighborhood size of a track. This corresponds to the number of tracks that occur in one of the same playlists of a track. A larger neighborhood size means that we have to access more potential candidates or filter out tracks during the process.

2.3 Data import and preparation

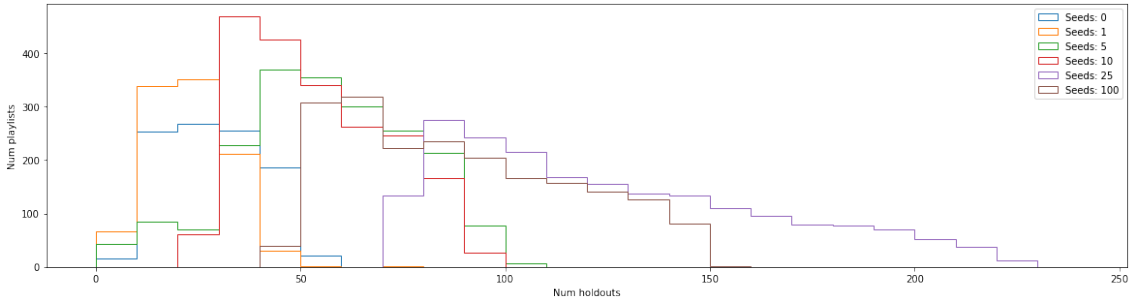
The first step was to import the data from files into a common data structure for fast access and simple data handling:

- First, we created a mapping to translate each character-based Spotify track URIs (e.g. *spotify:track:7KXjTSCq5nL1LoYtL7XAwS*) to an internal numerical ID. This internal id can serve as an index for internal models and lists, and saves a lot of memory because we can save playlists just as a list of integers (e.g. *20453, 12309, 232, ...*) instead of lists of strings.
- Depending on the model, we then prepared the data structure needed to build the model, e.g. a list containing all playlists.
- Furthermore, additional metadata of the playlists were imported for later analysis (e.g. parsing playlist titles, see for example section 3.1.)
- Spotify themselves provide an automated song analysis for most of their hosted tracks. This analysis includes 17 meaningful features such as danceability, speechiness or loudness of a track. We used the Spotify Web API to crawl these audio features for each of our tracks.

During the import process, we realized that there are a large number of duplicates, meaning two or multiple URIs that point to the exact same track, sometimes even to the same track in the same album. We experimented with removing the duplicate links, e.g. by changing all IDs to the most frequent ID for a certain song, but this actually had negative effects on the score. All in all, it was not necessary to do much data cleansing and preprocessing, e.g. no need to remove missing values, as the dataset was already prepared well by Spotify.
In order to quickly test different approaches and how hyper-parameters influence our predictions, we had to create a test set representing the challenge test set. To better fit the challenge set we analyzed for each of the 10 configurations the distribution of holdout tracks bucketed in steps of 10. We then drew random MPD playlists and checked if one of the 10 configurations still has a bucket left where one more representative is needed. That gave us a test set, we called "the golden test set", with a similar distribution of holdout tracks per configuration as in the challenge set (Figure 5).



(a) Challenge set



(b) Golden test set

Figure 5: Comparison of holdout distribution between challenge and test set. Different seed lengths have a different distribution of the *number of holdout tracks*.

3 Models

The system we try to build belongs to a group of software called "recommender systems". They are used in a wide array of fields, including music, movies, consumer products, restaurants and much more [3]. As a field with many commercial and practical applications, there is a vital research community. There are two general approaches for recommender systems:

- Collaborative filtering, which builds models from the user's past behavior as well as similar decisions by other users. It then uses these to predict items a user might be interested in. Collaborative filtering does not require actual knowledge about the items it predicts.
- Content-based filtering uses a set of discrete characteristics of an item to recommend similar items.

For the main track, we were bound to collaborative filtering, as we do not have additional information about the items themselves. For the creative track, we can mix in a content-based approach. Such a combination of both is called a *hybrid recommender system*.

We used a number of different models to solve the challenge in a satisfying way. In this chapter, we introduce each model (or model category) that we experimented with during the development process. Consequently, for the models that did not end up as part of the final solution, we explain the limitations preventing it. In section 3.8, we describe which models we used for our final submission and how different approaches were combined in a complete recommendation pipeline.

3.1 Recommend without seed tracks: NLP on playlist names

A tenth of the challenge playlists have no seed track, but only a playlist name. A different approach than for recommendations with seed tracks is required to create recommendations for those playlists. The approach takes all training data playlists and then tokenizes and stems/lemmatizes the playlist names.

For the main track, external data is not allowed. Therefore we created a simple tokenization and stemming function: All latin characters are transformed to their lowercase version, all non-latin characters are replaced with spaces, groups of spaces are reduced to a single space. The space is then used as a token delimiter. Each token is stemmed by a regular expressions matching:

```
.{3,}ing$|.{3,}ings$|.{3,}ly$|.{3,}ed$|.{2,}[^aeious]s$|ses
```

All endings (ing, ings, ly, ed, s, ses) are then removed.

For the creative track, external data can be used. Therefore we decided to use the Python library spaCy (https://spacy.io) with an existing English model for tokenization and lemmatizing.

For each existing token, a token-specific top list is then created. They count the number of playlists a track appears in with that one particular token in its playlist name.

When recommending tracks for a given playlist name the name is then tokenized and stemmed/lemmatized, and for each token, the corresponding top lists are then merged into one recommendation list:

$$\text{score}(track) = \sum_{token \in \text{tokenize}(playlist\_name)} \log_2(\text{frequency}(token, track))$$

Afterward, the 500 tracks with the highest scores are recommended.

Some words (e.g. music) appear in many playlists and do not have any meaning helping to recommend tracks. To determine this list of stop words to be removed before recommending we decided to take a subset of 1,000 tracks of the golden test set and compared the scores of the approach with and without pruning a particular token averaged over all 1,000 test items. A token that had a negative effect on at least two out of the three scores were added to a stop word list and pruned.

Selecting the stop words of the 100 most used tokens in the challenge seeds gave us 21 stop words: *music, good, song, the, my, mix, up, out, it, hop, stuff, road, a, august, upbeat, pump, to, drive, jamz, time, fav* The ten most used tokens that have been kept: *summer, jam, workout, rap, party, christmas, new, s, feel, vibe*

By design, the list of stop words varies based on the choice of the test set. For the final submission, we decided to take only *playlist, music, for, me, to, this, song, spotify, i, and* since those stop words appeared to be stable and led to score improvements for both the test set and the leaderboard.

3.2 Decision rule mining

Finding potentially interesting tracks for a playlist based on the existing content of a playlist and by looking at associations to other playlists with similar content can be seen as a problem domain of association rule mining. The goal is to find rules stating if a playlist contains a set of songs A, it also likely will contain the set of songs B. This is done by first finding frequent itemsets, in this case a set of tracks that appear in the same playlist frequently, and then finding associations $A \rightarrow B$ between different itemsets with a high support and a high confidence. We tried out implementations of two standard methods for finding frequent itemsets: The Apriori algorithm [4] and an improved approach called FP-growth [5]. While decision rule mining is a potentially interesting approach, we did not use them for two main reasons:
Firstly, computing frequent itemsets for large fractions or even the whole set of playlists at the same time would have been infeasible with our computing infrastructure. On the one hand, we cannot set the minimum support of itemsets too low, or we would miss many rare (but potentially meaningful) associations as many tracks only appear very infrequently (compare to Figure 3). On the other hand, setting a low minimum support leads to a very large number of potential frequent itemsets, in our case dozens of millions of sets. Only computing rules for smaller parts of the playlist database would have been possible but did take a very long time during our experiments. Secondly, during research about recommendation systems, association rule mining was rarely the method of choice for our kind of task, e.g. due to the huge number of different items, and other alternatives were preferred.

3.3 Shortest path

As an early idea of a graph-based approach, we thought of how to create recommendations for tracks that never or rarely appear together. We thought of recommending songs that *lie in between the two genres/types of music*. For this, we thought of seeing songs as vertices and edges between songs as co-occurrences with low weights for a high co-occurrence $w(u, v) = \frac{1}{\#\text{ playlists with u and v}}$. This approach gave promising results for two songs from totally different genres to have a slow *transition* from the genre of one song to another when we checked some examples manually. Unfortunately, this approach proved to be unfeasible for the MPD. We reached our computational and memory limits already by loading only 1% of the MPD. This is mainly due to the fact that the more playlists are involved the more likely an edge between two vertices exist, leading to a highly connected graph on many playlists.

3.4 Frequent neighbor tracks

Frequent neighbor tracks is a graph-based approach and can be intuitively described as *given a seed track: which are the most probable tracks to appear together in a playlist with this seed track?*. It is based on a bipartite graph between songs and the playlists they are included in. For each seed track the probability of the tracks appearing together with that particular seed track are then computed and summed over all seed tracks to get an overall ranking of the most probable tracks to appear with those seed tracks.

Algorithm 1 Additive frequent neighbor tracks

```
function RECOMMEND(Seeds)
    rank ← dict()
    for all s ∈ Seeds do
        neighbors ← counter()
        for all p ∈ Playlists(s): t ∈ Tracks(p) \ Seeds: increment neighbors[t]
        for all (track, count) ∈ neighbors do
            prob ← count / |Playlists(s)|
            if track ∈ ranks ∨ prob > 0.01: rank[track] ← rank[track] + prob
        end for
    end for
    return First 500 tracks of sort_desc(rank)
end function
```

By looking at Algorithm 1 we can directly see that all data to be loaded before recommending are the relationships between tracks and playlists. The runtime of the actual recommendation depends on the number of seeds and the number of tracks a seed appears together within playlists. To reduce the memory usage during recommending and computational time when sorting only tracks that have a probability of more than 1% for at least one seed track are considered.

Among all tested models for collaborative filtering by track occurrences this model is by far the fastest to initialize (7min, bottle neck is mainly I/O) and takes the least memory (less than 6GiB). Depending on the number of seed tracks and their neighbors a recommendation takes $< 1s$ to $20s$ on a 4GHz Intel machine. This can be easily parallelized using the bipartite graph as a read-only resource.

3.5 Matrix factorization

One of the most widely used techniques for many different recommender systems is called matrix factorization [6, 7]. The goal of matrix factorization is to find common latent, lower-dimensional features, here called factors, for each item and transactions (in this case track and playlist). This enables us to compute similarities between tracks and playlists.
First, we have to compute these latent factors by a process called non-negative matrix factorization. The playlist-track matrix describing the training data is decomposed into two smaller matrices. The shape of these matrices is determined by a hyperparameter f, that determines the number of latent features/dimensions. A higher value for f allows for a more accurate reconstruction of the original matrix but requires more computing time and memory. The values of the two matrices cannot be computed directly. Instead, they can be approximated via gradient descent or here Alternating Least Squares [8]. Using regularizers is important to counter over-fitting to the training data and enabling new recommendations for challenge playlists instead of just reproducing training playlists.
After having computed the latent track and playlist factors for the training data, we can compute recommendations for the challenge playlists. As those playlists do not exist in the training data, we have to recompute playlist factors P_c for each of the playlists. With the computed playlist and track factors, we get a confidence score $r_{ij} = (P_c)_i T_j^T$ for each track and can then choose the tracks with the highest confidence that are not already in the playlist, as recommendations.
We used a python library focused on implicit feedback (feedback that is only binary and does not rely on ratings for items) for our implementation [9]. During the training of the models, we had to find fitting parameters f and λ, the number of latent features and the magnitude of regularization. Training a model took relatively long (up to multiple hours), so it was challenging to tune the model to perfect accuracy. We also experimented with building a whole model and models for only fractions of the dataset, with smaller models allowing for a higher number of latent factors and a potentially higher accuracy, but also requiring a new computation for each new model. We ended up with one model for the entire dataset, 500 latent factors and a regularization value of 0.01 . Model fitting and computing of the recommendations took around 6 hours on a 16-Core CPU and required around 14GB of RAM during model fitting. While performing comparable with the Frequent neighbor tracks approach (compare section 3.4), we could achieve a slight improvement in accuracy by combining both models.

3.6 Denoising autoencoders

Another innovative approach we tried are the so-called Denoising autoencoders [10, 11]. Autoencoders are neural networks, that can be used to extract latent features (compare to section 3.5) by being trained to replicate the input as output with hidden layers containing fewer neurons than input and output. In our context, we use the purposeful addition of noise to the input as a mean to create recommendations. During training, we deactivate part of the input, suppressing tracks actually included in the playlist, but require the model to output the actual, complete playlist. This way, the model learns to "complete" playlists and recommend tracks that fit an input of tracks. While being an intriguing concept, we couldn't achieve great results with this approach. We used an adapted variant of an existing implementation [12]. Despite training one model for more than 8 hours on a GPU, accuracy stayed significantly lower than our existing models. Reasons may be the sheer amount of different items and the hardware limitations. In the reference papers, which showed very promising results, the number of unique items was significantly lower ($\sim$ 100.000) and they had 4 professional-grade GPUs for training, while we had a single consumer-grade one.

Maybe running the model for longer times would have made a difference, but potential errors in the implementation, as well as incomplete preparation and pre-processing, may have played a role as well.

3.7 Pairwise song feature classification

For the creative track, we could use audio analysis features as additional input data. Spotify themselves provided a total of 17 different features for almost every track in the dataset via their Web API. We could use those to achieve further improvement over our existing solution. We thought about different approaches on how to use the data, from clustering to find genres to nearest-neighbor approaches. But we ended up with a pairwise classification as it might be able to learn more complex relationships between feature values, while still being easy to implement. We used a simple feed-forward neural network with 2*17 input units, 3 hidden layers, ReLu activation, batch normalization and dropout. As input data, we used a pair of two tracks respectively. We randomly drew a playlist, and then up to three songs of a playlist. For each track, we sampled another track from the same playlist to create a correct training example and a song from the dataset not contained in the playlist to create a "non-fit" example. We trained the network on around 100 million track pairs.
We used the pairwise classification to rank the 500 recommended tracks from previous approaches. For the pairs between seed and recommended tracks, confidence scores are computed, and then the rank out of 500 is added for previous scores and feature classification for the final rank.

3.8 Recommender pipeline

Finally, we want to give a quick overview of our whole approach, and the two main steps: Firstly, building and preparing the models. Secondly, doing the actual recommendations for the challenge playlists. Figure 6 shows all of the different models combined into the recommendation pipeline, while Figure 7 describes which method was used for different seed sizes.

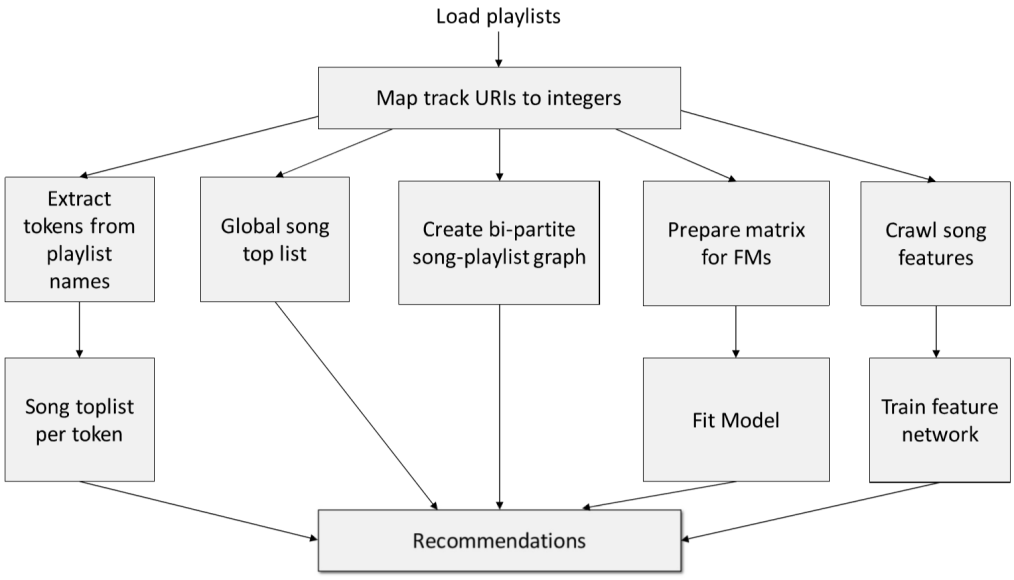


Figure 6: Illustration of all components, that we ended up using in the final recommender system. This graphic shows how the models are prepared for the actual recommendation process.

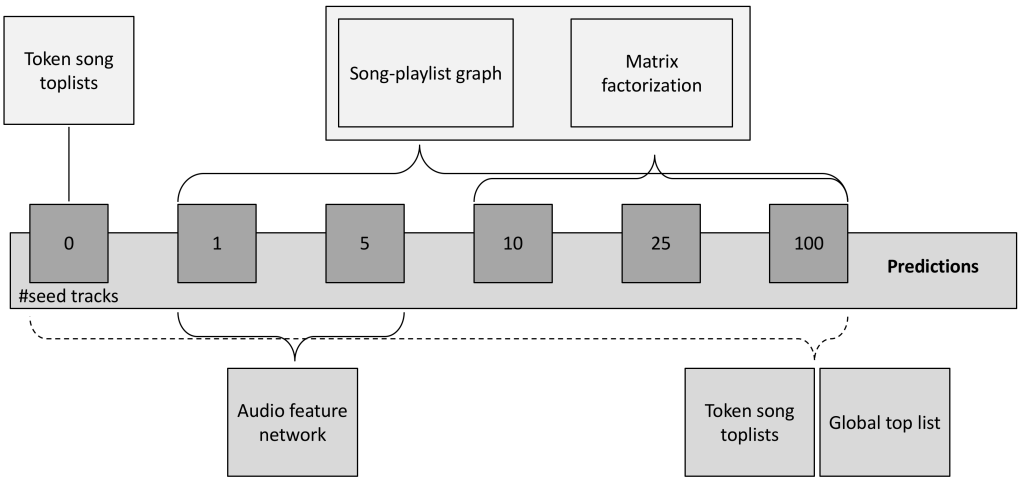


Figure 7: This visualization shows which approach was used for which seed configuration. E.g. Token song toplist was used primarily for playlist with zero seeds.The song-playlist graph and the matrix factorization method were used together for seed lengths of 10,25 and 100. Recommendations were alternately drawn from the two approaches. In case the other methods could not recommend enough songs, token song toplist and the global toplist were used as fallback methods to complete the list of 500 recommendations.

4 Results

With our combination of different models we achieved a top 20% placement in the main track, and a top 10 placement in the creative track. Table 3 shows the scores of the top 3 ranked teams and our scores for the main track, table **??** for the creative track.

On a positive side note, the actual scores seemed to be relatively closely related to our own golden test set. While the actual numeric values were not a perfect fit, testing on the golden test set generally gave a strong indication on which methods were preferable, even with small margins between scores. Upload to the leaderboard was only possible once every 24h, so the possibility to test in between was quite useful.

Figure 7 illustrates the performance of three different approaches on the golden test set. This visualization gives a great overview of how well the algorithms perform and especially how well correct tracks are sorted in the recommendations.

Rank	R Precision	NDCG	Click scores
1	0.224	0.395	1.78
2	0.223	0.393	1.90
3	0.215	0.385	1.78
19	0.195	0.350	2.22

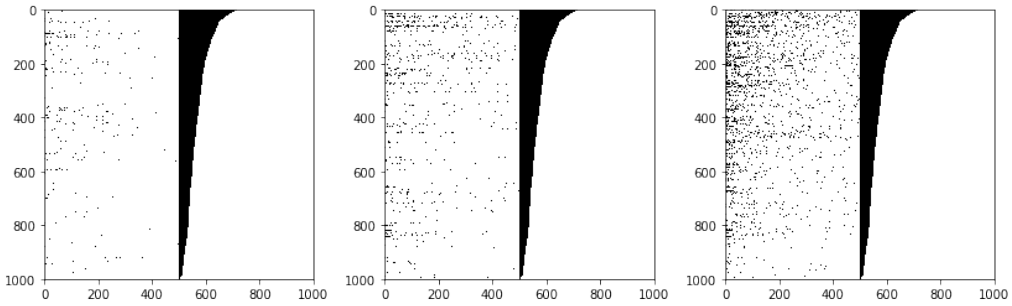
Rank	R Precision	NDCG	Click scores
1	0.223	0.394	1.78
2	0.220	0.385	1.93
3	0.209	0.375	2.05
8	0.195	0.350	2.13

Table 3: Main track/Creative track scores: The left table shows the three different scores (compare to section 1.2) for the three top-ranked teams in the challenge, and our submission at rank 19. A total of 111 teams handed in a solution. The right table shows the results of the creative track with our submission at rank 8 (out of 31).

5 Conclusion

During the challenge we learned about the difficulties with really large data sets and the possibilities available to have good recommendations with collaborative filtering. Due to our limited resources we were forced to find an approach that is feasible with those limitations. This might not always lead to the best results (in the leaderboard), but is usually more practically usable in business-context.

Although not practicable from a business perspective, extensive (cloud) computing resources are necessary in order to win such a challenge. We spent a lot of time optimizing our models,



(a) Most frequent tracks (b) Playlist names (c) Frequent seed neighbors

Figure 8: Comparison of correct matches with different approaches with a subset of the golden test set. Each line is a playlist, playlists are sorted by number of holdouts. Each black pixel in x direction means the xth recommended track has a match in the holdouts. Left side of each plot is the actual match, right side is an optimal match (every track of holdout matched and positioned at the beginning of all 500 recommendations).

which was missing to try out more or combinations of models.

We also noticed that the recommender system with our scores give pretty good recommendations already and new, creative approaches like shortest path give interesting results that are not rewarded by the scoring mechanisms. The score is calculated based on existing playlists and not if the user would pick those songs if they'd be offered to them. That means that the scores actually only tell how well the recommender can predict songs of an existing recommender system, since Spotify had recommender systems in place before already, creating a bias towards those recommendations, even though a new recommendation might be better from a user-perspective. In order to measure a new recommender system appropriately it is better to apply an A/B test comparing the old recommender system along the new one and possibly some random songs (to check if randomness performs better than the existing recommender systems in certain scenarios).

6 Future work

Playlist names The recommendation based on playlist names could be further improved by weighting each token instead of simply removing stop words. This could be done by scoring each token with a distinctiveness weight, similar to inverse document frequency. Furthermore each token could be seen as a feature for Matrix Factorization together with the songs in a playlist. Some first tests gave us embeddings for tokens with similar meanings, but it turned out to not be usable with restricted computational resources.

FMs Tokens as FM features could not only be useful for recommendations without seed tracks but also improve the quality of recommendations with seed tracks...

Use of additional hardware resources Many of our tested approaches, like the matrix factorization or the denoising auto-encoders could have benefited from use of more extensive hardware, e.g. machines by using cloud services.

Tuning of merging strategies Instead of choosing the merging strategy manually, we could have used more sophisticated methods such as neural networks.

Better audio feature classification The audio feature classification approach was not yet fully developed and could have achieved better results, e.g. by classifying the combination of seed tracks, or training with an objective other than just binary affiliation to a playlist.

References

[1] The million playlist dataset - recsys challenge. https://recsys-challenge.spotify.com/rules. Accessed: 2018-08-29.

[2] The million playlist dataset - recsys challenge. https://recsys-challenge.spotify.com/dataset. Accessed: 2018-08-08.

[3] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor. *Recommender Systems Handbook*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.

[4] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[5] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 1–12, New York, NY, USA, 2000. ACM.

[6] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, August 2009.

[7] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 69–77, New York, NY, USA, 2011. ACM.

[8] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 263–272. Ieee, 2008.

[9] Python Library for Fast Python Collaborative Filtering for Implicit Feedback Datasets . https://github.com/benfred/implicit. Accessed: 2018-08-21.

[10] A. G. Abad and L. I. Reyes-Castro. Collaborative Filtering using Denoising Auto-Encoders for Market Basket Data. *ArXiv e-prints*, August 2017.

[11] Yao Wu, Christopher DuBois, Alice X. Zheng, and Martin Ester. Collaborative denoising auto-encoders for top-n recommender systems. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, WSDM '16, pages 153–162, New York, NY, USA, 2016. ACM.

[12] Implementation of Deep Learning model for Recommendation System in Tensorflow . https://github.com/77abe77/Collaborative-Denoising-Autoencoder. Accessed: 2018-08-22.