Autoencoders for Next-Track-Recommendation

Michael Vötter, Eva Zangerle, Maximilian Mayerl, Günther Specht
Databases and Information Systems
Department of Computer Science
University of Innsbruck, Austria
{firstname.lastname}@uibk.ac.at

ABSTRACT

In music recommender systems, playlist continuation is the task of continuing a user's playlist with a fitting next track, often also referred to as next-track or sequential recommendation. This work investigates the suitability and applicability of autoencoders for the task of playlist continuation. We utilize autoencoders and hence, representation learning to continue playlists. Our approach is inspired by the usage of autoencoders to denoise images and we consider the playlist without the missing next-track as a noisy input. Particularly, we design different autoencoders for this specific task and investigate the effects of different designs on the overall suitability of recommendations produced by the resulting recommender systems. To evaluate the suitability of recommendations produced by the proposed approach, we utilize the AotM-2011 and LFM-1b datasets. Based on those datasets, we show that n-grams are a well performing alternative baseline to kNN. Fruther, we show that it is possible to outperform a kNN as well as an n-gram baseline with our autoencoder approach.

1. INTRODUCTION

Recommender systems are applicable to a broad spectrum of domains. The music domain is one such application area, where one specific task is next-track music recommendation. Next-track music recommender systems are recommender systems that aim to find a fitting continuation (next-track) for a given playlist. In general, a playlist is an ordered list of music tracks, where the order is based on time, which means that the first track in the list is expected to be listened first, followed by the second track and so on. In other words, a playlist is a time series of tracks.

Multiple different approaches have been proposed for the playlist continuation task. As mentioned in [7], these approaches are based on a broad spectrum of techniques such as Markov models, collaborative filtering, content similarity as well as hybrids of them. Another traditional approach to compute next-track recommendations is a nearest neighbor search as used in multiple other papers [5, 8, 10].

In the field of music recommendation, deep learning approaches are usually used to include additional features such as textual information [1] or content features [18] in the recommendation process. In contrast, only a few approaches such as [9] directly apply neural network approaches to the playlist continuation task. To fill this gap, we propose a novel autoencoder-based approach that directly applies neural network-based representation learning on the playlist continuation task. The simplest form of an autoencoder is a neural network with a dense input layers and a dense output layer which is trained in an unsupervised manner. Our approach is inspired by the successful application of autoencoders for image denoising [19]. We consider the input playlist (that has to be continued) as a noisy version of the resulting playlist, where a next-track is added as a continuation. We argue that representation learning methods are more suitable to take advantage of the features contained in the playlist structure than e.g., kNN because representation learning methods are specifically designed to learn an effective representation and hence features [2]. To the best of our knowledge, this is the first time that autoencoders are utilized for the playlist continuation task, while the closest related tasks, where autoencoders were used successfully, are collaborative filtering tasks [15, 16, 21].

With this work, we investigate the general applicability of autoencoders for the playlist continuation task. We report the effects of different parameter settings on the overall suitability of recommendations produced by the system and answer the following research questions:

- **RQ1:** How can playlists be vectorized for an autoencoder?
- **RQ2:** Is there an alternative baseline to kNN that better utilizes the order of tracks in a playlist?
- **RQ3:** Which autoencoder design produces competitive results for the next-track music recommendation task?

In [10], Kamehkhosh and Jannach propose to use handcrafted playlists to evaluate next-track music recommender systems. Inspired by that, we use the AotM-2011 dataset as well as the LFM-1b dataset to evaluate the suitability of recommendations produced by our approach. Our experiments show, that the resulting autoencoder approach produces competitive recommendations compared to the kNN baseline.

31^{st} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 11.06.2019 - 14.06.2019, Saarburg, Germany.

The remaining sections of this paper are organized as follows. First, related work will be discussed in Section 2. Afterwards, in Section 3, we describe the algorithm to convert a playlist into a corresponding vector and present our autoencoder approach. Following that, the experimental setup to evaluate our approach by comparing it with a kNN baseline is described in Section 4. Thereafter, we present the results in Section 5 and finally draw a conclusion in Section 6.

2. RELATED WORK

This section gives an overview of next-track music recommendation approaches.

We consider the playlist continuation task to be a special case of the more general task of playlist generation. According to Bonnin and Jannach [5], the preconditions for a playlist generation task are a background knowledge base and target characteristics for the resulting playlist. Based on that, a sequence of tracks (playlist) best fitting the characteristics needs to be found. The playlist generation problem may be converted to the playlist continuation problem by considering all playlists/sessions as the background database and using a target characteristic that describes a fitting track given a playlist to be continued.

Sedhain et al. [15] introduced an autoencoder approach for collaborative filtering. They report that their approach outperforms current state-of-the-art methods such as matrix factorization and neighborhood methods. Zhang et al. [21] use an autoencoder as part of a hybrid collaborative filtering framework able to produce personalized top-n recommendations and rating predictions. For their proposed Semi-AutoEncoder approach, they removed the restriction that the input and output layer must be of the same dimensionality and choose to make the input layer wider than the output layer. This allows to feed the autoencoder with additional feature vectors. In [9], Jannach and Ludewig compare a recurrent neural network (RNN) approach with a kNN approach for the task of session-based recommendations. Their findings show, that the RNN approach is inferior but they believe that further research will probably lead to better RNN configurations that are able to outperform the kNN approach. Nevertheless, this shows that kNN is a strong baseline to compare against.

Jannach et al. [8] present multiple extensions to the kNN approach, which they compare to a kNN baseline. They propose to take additional measures into account with a weighted sum. By using the social tags that are assigned to tracks by Last.fm[1] users, they take content similarities into account. Further, they suggest using numerical features such as tempo, loudness and release year. Additionally, they state that it is possible to take long-term content-based user profiles into account.

The evaluation approach presented by McFee and Lanckriet [12] supports our assumption that playlists and their tracks contain enough information to find fitting next-tracks for a given playlist. They come up with the idea to consider playlist generation as a natural language modeling problem instead of an information retrieval problem. Therefore, they consider a playlist to be equivalent to a sentence in a natural language and tracks to be equivalent to words. Further, they show how techniques known from natural language processing can be used to evaluate playlist generation algorithms.

[1]`https://www.last.fm/`

$$[5,1,3,7,4] \rightarrow \begin{cases} (1,0,1,1,1,0,1) & \text{binary} \\ (2,0,3,5,1,0,4) & \text{order} \\ \left(\frac{2}{5},0,\frac{3}{5},\frac{5}{5},\frac{1}{5},0,\frac{4}{5}\right) & \text{normalized-order} \end{cases}$$

Figure 1: Playlist to encoding transformation.

3. METHODS

In this section, the proposed recommendation approach based on an autoencoder is presented. First, we will explain how playlists are converted into a vector, which is necessary to use them as an input for the autoencoder. Afterwards, the structure and implementation details of the autoencoder are presented in Section 3.2. This includes the general training procedure used and a modified autoencoder layout to overcome overfitting by simulating the continuation task during training.

3.1 Vector Encoding of Playlists

Playlists are usually represented as an ordered lists of tracks. In the special case of the playlist continuation task, the playlist used as input for the algorithms is often referred to as "history". This history can be considered a list of past listening events.

To use playlists as an input for autoencoders it is necessary to convert the ordered list into a vector representation. We propose three different ways to determine the value of each dimension of the generated playlist vector, as presented in the following. An example of all three encodings is shown in Figure 1.

Binary Encoding is the simplest encoding and it is inspired by the (one-hot) vector encoding used in [9]. Each track t in the playlist is converted to the corresponding one-hot encoded track vector $\vec{vt}_t$ where all dimensions, except the one assigned to the track (index t), are set to 0 while the dimension with index t is set to 1. After that, the playlist vector $\vec{p}$ is computed by $\vec{p} = \sum_i \vec{vt}_i$. Note, that the ordering information of the playlist is lost.

Order Encoding is a modified version of the Binary Encoding and includes ordering information. We propose to use the track's index i in the playlist as the value of the dimension t assigned to the track. Therefore, the track vector encoding contains 0 for all dimensions except of the dimension with index t, to which the value i is assigned. To obtain a playlist vector $\vec{p}$, all track vectors are summed up.

Normalized-Order Encoding is an extension to Order Encoding and takes the length of a playlist into account. The playlist vector $\vec{p}$ is normalized by the number of tracks contained in the corresponding playlist, which reduces the effects of the playlist length on the encoding.

3.2 Autoencoders for Playlist Continuation

Autoencoders are an unsupervised learning method used for representation learning [19]. In its simplest form, an autoencoder is a neural network with one input layer, one hidden layer and one output layer, where the input layer is fully connected to the hidden code layer which again is fully connected to the output layer.

In contrast to a representation learning task, where the decoder is removed after the training phase to get the code as an output of the network, we use the whole network

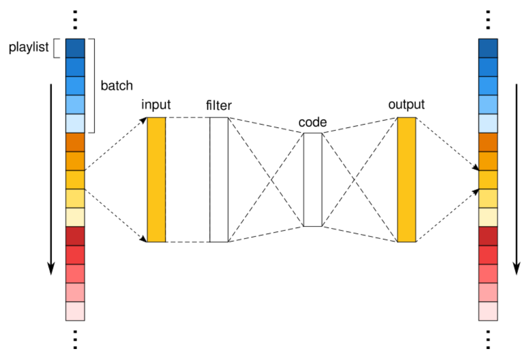

Figure 2: Training workflow for the autoencoders.

to compute recommendations. Recommendations are computed based on the following procedure: Given a playlist that should be continued with a fitting next-track as an input, it is necessary to convert this playlist to a vector. Afterwards, this vector encoded playlist is used as an input for the trained network that produces an output vector represented by the output layer. The output vector holds rating values for all tracks (number of dimensions of the vector) contained in the dataset, which was used to train the network. This output vector is then converted to a prioritized list of tracks by creating a list of indexes (track id's) ordered by their corresponding value in the vector. The index with the highest value is the first in the list while the index with the lowest value is last. Further, all tracks contained in the input playlist are removed from this prioritized list to get next-track recommendations differing from the tracks contained in the input playlist. Mostly, it is necessary to compute a given number of possible continuations. This is achieved by chopping off the list after the given number of tracks.

Using Keras[2], we implemented an autoencoder in Python. Our implementation allows to set the number of epochs, the hidden code layer activation function, the output layer activation function and the used loss function. The input layer size equals the output layer size and is determined automatically base on the dataset. Further, we decided to automatically adapt the code layer size based on the input size divided by 40, which is a result from preliminary experiments. Keep in mind that dense layers are used to build the network, which means that all nodes of one layer are connected to each node of the neighboring layers. Our network consists of one input layer followed by a filtering layer that removes the last track of the input during training. This filtering layer is followed by an optional dropout layer with 0.5 dropout rate that can be disabled. This layer is then followed by a hidden code layer and an output layer.

To train the neural network, a training set is used, which consists of an input and the expected output, which are both the same for autoencoders. The autoencoder learning method is depicted in Figure 2. To train an autoencoder-based on the previously introduced playlist vectors, it first is necessary to create an autoencoder with an input/output layer size fitting the dimensionality of the playlist vectors in the dataset. The training process is further configured to use $\frac{1}{64}$ of the total number of playlists the training set as its batch size and uses Adam as an optimizer because preliminary experiments showed that these settings work well and that they show similar performance compared to other parameter choices.

This setup allows to compare different parameter configurations of an autoencoder where the basic structure of the used neural network, which can be seen in the center part of Figure 2, remains the same.

4. EXPERIMENTS

In this section, we present the setup used for the conducted experiments to evaluate the suitability of recommendations produced by the autoencoder-based approach in comparison to a kNN baseline. In Section 4.1, we introduce the used datasets. Thereafter, in Section 4.2 the kNN baseline recommender is introduced. Followed by an explanation of the n-gram baseline in Section 4.3, continued by the overall experimental setup in Section 4.4.

4.1 Datasets

To evaluate the recommender systems, we aim for datasets that are based on user interaction such as listening logs and playlists because next-track music recommender systems must satisfy the needs of users. Along the lines of previous work [12, 13, 4, 5, 8, 7, 9], we use datasets based on the data gathered from the two music platforms Last.fm[3] and Art of the Mix[4].

Based on the LFM-1b dataset [14] gathered from Last.fm, listening sessions were extracted by Jacob Winder in [20]. These sessions are created by assuming that two listening events of a single user belong to the same listening session if there are no more than 30 minutes between them. The resulting sessions are further filtered. All successive occurrences of the same track are merged into one occurrence. After doing so, all sessions of length one are dropped.

This results in approximately 62 million sessions which we filtered further. In a first step, a session chunk containing the first 3 million sessions with a minimum length of three was created. This chunk shows a high number of different tracks. Reducing the number of tracks contained in a dataset is an important step to keep the size of the resulting neural network low enough to train it in reasonable time. This was achieved by dropping all playlists that contain rarely occurring tracks. We dropped all playlists containing tracks with 840 or fewer occurrences.

Further, we use the AotM-2011 dataset [13] provided as a Python pickle export[5] by Vall et al. [17], as the AotM dataset is often used in literature. Instead of using their split, we merged the training and test set and used five-fold cross-validation, as we also applied five-fold cross-validation on the LFM-1b based dataset. The AotM-2011 dataset contains far fewer playlists and far more tracks than the LFM-1b based datasets (see Table 1). This leads to a playlists/track ratio of 0.220 which is substantially smaller than the ratio of the LFM-1b based datasets.

4.2 Baseline kNN Recommender Systems

A k-Nearest Neighbors (kNN) approach [5] is used as on of the baseline for our experiments. We have chosen kNN as it

[2]`https://keras.io/`

[3]`http://www.last.fm`

[4]`http://www.artofthemix.org`

[5]`https://git.io/fhNfZ`

Table 1: Detailed dataset description.

Dataset	Playlists	Tracks	Playlists/Track
LFM-1b	20,824	2,673	7.790
AotM	2,715	12,355	0.220

has been used for the playlist continuation task in multiple other papers such as [5, 8, 7, 10].

The basic idea behind kNN is to find k different items that are the nearest neighbors of a given item. Neighborhood of items in general is defined based on a distance-function, which allows to take different properties into account. Further, it is necessary to get to a conclusion from those k neighbors which can, for example, be done with a majority vote [11].

In [10], a binary cosine similarity is used as the distance-function. We run a grid search with k's of 10, 20, 50, 100, 200 and 300. We further include the three different ranking functions cosine similarity, item-item similarity and tf-idf similarity defined by the kNN implementation of the implicit Python package[6] (version 0.3.8) in the grid search.

4.3 Baseline N-Gram Recommender System

N-grams are a common statistical model in natural language processing. This technique is for example used in [3] for word predictions. Instead of using sequences of words in sentence, we use sequences of tracks in a playlist. The n parameter specifies the number of successive tracks that are taken into account by the model.

Therefore, the simplest model is a unigram model ($n = 1$) which only counts track occurrences in playlists.

Increasing the number n of the n-gram model makes it possible to consider the previous $n - 1$ tracks for the prediction by calculating probabilities as given in the following equation:

$$P_{\text{n-gram}}\left(t_i | t_{i-(n-1)}\right) = \frac{F\left(t_{i-(n-1)}, \ldots, t_i\right)}{F\left(t_{i-(n-1)}, \ldots, t_{i-1}\right)} \qquad (1)$$

where t_i is the i^{th} track in a sequence of tracks $t_1, t_2, \ldots, t_n$. $F(seq)$ is the frequency of occurrences of a given sequence of tracks seq in the training set. Ranking the tracks by their probabilities (highest first) allows to make predictions based on the probabilities learned by an n-gram model.

4.4 Experimental Setup

We used scikit-learn[7] to run a grid search on the parameters of the approaches. To ensure that the reported results are not bound to a specific train-test split of the datasets, we run a five-fold cross-validation. The k-fold splitting procedure of scikit-learn was used with a fixed random state (seed) to ensure the reproducibility and comparability of the results. The metrics used for the evaluation are *recall* (r) and *mean reciprocal rank* (mrr).

For the evaluation of all recommender systems, a two-step process is used for each of the two presented datasets. In the first step, each recommender system is trained using the training data. For this purpose, a new instance of the recommender system is created for each run and then trained using the training procedure. The second step of the evaluation process computes recommendations with the previously trained recommender. Therefore, each playlist in the training set is decomposed into a history (all tracks except the last one) and the last track [6, 10]. The history is used as input for the recommender system, while the last track is the expected recommendation, based on which the metrics are computed. Recommendation tests of different length are considered to get an impression of the effects of the recommendation test length.

The autoencoder implementation presented in Section 3.2 has a high degree of freedom in terms of modifiable parameters. Therefore, we decided to fix the reduction factor (code layer size), the batch size and the optimizer. The used values were obtained from preliminary experiments. Based on the knowledge gained from those preliminary experiments we determined value ranges for the other parameters used in the grid search.

5. RESULTS

In the following, we first report the recommendation suitability of the different encoding types in Section 5.1. After that, the results achieved by the n-gram baseline are shown in Section 5.2. Finally, we compare the suitability of recommendations produced by our approach on different datasets in Section 5.3.

5.1 Encoding Type

In a first evaluation step, the recommendation suitability of the different encoding types introduced in Section 3.1 was compared using the kNN baseline, based on the AotM and LFM-1b datasets. Table 2 shows the results for kNN using the item-item distance metric. We observe that the normalized-order encoding outperforms both other encodings on both datasets and for different values of k. Interestingly, order encoding without normalization has a negative effect on the performance of the kNN implementation. This can be explained by the fact that the length of a playlist is encoded as well. In addition, order encoding has a larger vector space than binary and normalized-order encoding as the values in each dimension have a bigger range. Due to space reasons we do not include results for the cosine and tf-idf distance metric and other values of k, as these show that normalized-order encoding works best. Further experiments showed, that the Autoencoders show similar behavior when the encoding type is changed.

Based on these findings, we argue that the normalized-order encoding should be used among the three encodings introduced in Section 3.1, which also answers RQ1. Therefore, we use normalized-order encoding for all further evaluations.

5.2 N-Gram Baseline

To evaluate the suitability of an n-gram model we decided to compare a bigram ($n = 2$) and a trigram ($n = 3$) model with the best performing kNN (found using a grid search) configuration on each dataset. The kNN baseline using the item-item as a distance metric with 20 neighbors (kNNi20) works best on the AotM dataset while the cosine distance with 50 neighbors (kNNc50) works best on the LFM-1b dataset. In addition, we give the results of a unigram ($n = 1$) model that always recommends the most popular tracks.

[6]`https://pypi.org/project/implicit/0.3.8/`
[7]`https://scikit-learn.org/`

Table 2: Impact of encoding types measured as recall (r) and mean reciprocal rank (mrr) of kNN.

Dataset	Encoding	k	r/mrr @1	recall @20	mrr @20
AotM	binary	20	0.022	0.056	0.029
		200	0.022	0.062	0.031
	order	20	0.025	0.057	0.031
		200	0.025	0.060	0.032
	norm.-order	20	**0.027**	0.062	**0.035**
		200	**0.027**	**0.065**	**0.035**
LFM-1b	binary	20	0.471	**0.886**	0.593
		200	0.471	0.864	0.590
	order	20	0.435	0.885	0.556
		200	0.437	0.856	0.556
	norm.-order	20	0.618	0.883	**0.695**
		200	**0.619**	0.866	0.694

Table 3, shows that a bi- and trigram is able to outperform a unigram model. Further, it can be seen that those n-gram models work much better on our LFM-1b dataset variation than on the AotM dataset. This can be lead back to the fact that each track on average occurs in 0.22 playlists in the AotM dataset compared to 7.79 occurrences in the LFM-1b dataset, as stated in Table 1. Therefore, common sequences of tracks among playlists are more likely in the LFM-1b dataset than in the AotM dataset. We argue that this is also the reason why the absolute values of all metrics differ that much when comparing the results on both datasets. Compared to both kNN baselines it can be seen that the n-gram models work better on the LFM-1b dataset especially for short recommendation lengths. In contrast, they are less effective on the AotM dataset than kNN models.

It can be seen that n-gram models form a strong baseline for the LFM-1b dataset. Note that the kNN models operate on the normalized-order encoding while the n-gram models utilize the track sequences directly without any encoding, which answers RQ2.

5.3 Autoencoder Approach

In Table 3 we depict the results of our autoencoder approach in comparison to the kNN and n-gram baseline. The results show, that it is possible to outperform a kNN baseline on both datasets using the proposed autoencoder approach. This is especially true for longer recommendation lengths. To give a better overview of the capabilities of our autoencoder approach we give the results of four autoencoder configurations. To distinguish the different parameter configurations of our autoencoder approach we decided to name each configuration (AE1–AE4), where we report results. For each dataset we include results for one autoencoder including the dropout layer (see Section 3.2) and one without dropout. AE1 without the dropout layer and AE2 with the dropout layer are respectively the best performing autoencoder configurations for the AotM dataset, while AE3 (without dropout) and AE4 (with dropout) perform best on LFM-1b according to the mrr@1. AE1 uses tanh as the code layer and output layer activation function with cosine proximity as the loss function and is trained over 5 epochs. AE2 utilizes relu as the code layer activation and softmax as the output layer activation with categorical crossentropy

Table 3: Performance of different recommender systems (RecSys) measured as recall (r) and mean reciprocal rank (mrr).

Dataset	RecSys	r/mrr	recall		mrr	
		@1	@5	@20	@5	@20
AotM	unigram	0.001	0.004	0.016	0.002	0.003
	bigram	0.015	0.027	0.031	0.020	0.020
	trigram	0.015	0.015	0.015	0.015	0.015
	kNNi20	0.027	0.044	0.062	0.033	0.035
	kNNc50	0.018	0.024	0.036	0.020	0.022
	AE1	**0.028**	**0.048**	0.076	**0.035**	0.038
	AE2	**0.028**	0.045	0.073	**0.035**	**0.039**
	AE3	0.027	0.045	**0.080**	0.034	0.037
	AE4	0.027	0.045	0.069	0.034	0.036
LFM-1b	unigram	0.004	0.014	0.046	0.008	0.011
	bigram	**0.748**	**0.841**	0.881	**0.784**	**0.788**
	trigram	0.727	0.771	0.775	0.746	0.746
	kNNi20	0.619	0.786	**0.883**	0.619	0.695
	kNNc50	0.635	0.799	0.869	0.635	0.708
	AE1	0.391	0.684	0.806	0.500	0.515
	AE2	0.619	0.788	0.848	0.686	0.693
	AE3	0.650	0.795	0.849	0.708	0.714
	AE4	0.647	0.806	0.855	0.711	0.717

loss and was trained over 40 epochs. AE3 and AE4 are both trained over 40 epochs use tanh as the code layer activation and cosine proximity as loss. While AE3 is configured with a softmax output activation, AE4 is configured to use sigmoid for output activation.

It can be seen in the results that autoencoders outperform both the kNN and n-gram baselines on the AotM dataset while they are not able to outperform n-gram models on the LFM-1b dataset. Surprisingly, AE1 works best on AotM when trained for 5 epochs. While AE2, AE3 and AE4 reveal similar results per dataset AE1 only produces comparable results on the AotM dataset which answers RQ3. One possible explanation is that it overfits on the particular training set which is tried to be prevented using a dropout layer.

In the above section, the recommendation suitability impact of multiple configurations of an autoencoder were presented. Additionally, results of the best performing configurations on different datasets are given. The results show, that autoencoders can be used for the playlist continuation tasks when configured correctly.

6. CONCLUSIONS

In this work, we proposed a novel autoencoder approach for the playlist continuation task. To use playlists as an input for autoencoders, we introduced a procedure to encode playlists as vectors. The evaluation shows that the proposed autoencoder approach outperforms a basic kNN approach. Particularly, the results show that this is the case regardless of the playlists/track ratio of the used dataset.

This work solely focuses on determining if an autoencoder approach can be used for the playlist continuation task. We showed that outperforming basic kNN is possible for datasets, that we consider small in comparison to the amount of data given in a real-world scenario. One possible source of improvement is the training procedure.

Autoencoders are usually trained to reconstruct the input which we modified slightly. We introduced a filtering layer in the training phase that removes the last track of the input. This trains the autoencoder to "reconstruct" playlists including the last track (next-track) filtered in the input. Additionally, it would be possible to specifically designing a loss function for the continuation task. Strub et al. [16] present a loss function that disregards unknown values to train autoencoders as a collaborative filtering method. Applying a similar loss function to our training procedure is part of future work.

Additionally, a more advanced training procedure could lead to a well performing deep autoencoder. One way of creating a deep autoencoder would be to first train an autoencoder with one input and one output layer (as the ones proposed in this work) and then use the learned code as an input to train another autoencoder. After that, it is possible to split the first autoencoder into the encoder and decoder parts and insert the second autoencoder in between. The resulting autoencoder then can be fine-tuned and extended in the same way. This process is like the one proposed by Vincent et al. [19], where they stack denoising autoencoders. Using such an advanced training procedures for deep autoencoders is part of future work.

In addition, a user-study to evaluate the approaches should be conducted in future work. This is important to get an impression of the user-perceived quality of the approaches.

7. REFERENCES

[1] T. Bansal, D. Belanger, and A. McCallum. Ask the GRU: Multi-task Learning for Deep Text Recommendations. In *10th ACM Conf. on Rec. Sys.*, RecSys, pages 107–114, 2016.

[2] Y. Bengio, A. Courville, and P. Vincent. Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[3] S. Bickel, P. Haider, and T. Scheffer. Predicting Sentences using N-Gram Language Models. In *Empirical Methods in NLP*, 2005.

[4] G. Bonnin and D. Jannach. Evaluating the quality of playlists based on hand-crafted samples. In *14th Conf. of the Intl. Society for Music Information Retrieval*, ISMIR, pages 263–268, 2013.

[5] G. Bonnin and D. Jannach. Automated Generation of Music Playlists: Survey and Experiments. *ACM Comput. Surv.*, 47(2):26:1–26:35, 2014.

[6] S. Craw, B. Horsburgh, and S. Massie. Music Recommenders: User Evaluation Without Real Users? In *24th Intl. Joint Conf. on Artificial Intelligence*, IJCAI. AAAI, 2015.

[7] D. Jannach, I. Kamehkhosh, and G. Bonnin. Biases in Automated Music Playlist Generation: A Comparison of Next-Track Recommending Techniques. In *24th Conf. on User Modeling, Adaptation and Personalization*, UMAP, pages 281–285. ACM, 2016.

[8] D. Jannach, L. Lerche, and I. Kamehkhosh. Beyond "Hitting the Hits": Generating Coherent Music Playlist Continuations with the Right Tracks. In *9th ACM Conf. on Rec. Sys.*, RecSys, pages 187–194. ACM, 2015.

[9] D. Jannach and M. Ludewig. When Recurrent Neural Networks Meet the Neighborhood for Session-Based Recommendation. In *11th ACM Conf. on Rec. Sys.*, RecSys, pages 306–310. ACM, 2017.

[10] I. Kamehkhosh and D. Jannach. User Perception of Next-Track Music Recommendations. In *25th Conf. on User Modeling, Adaptation and Personalization*, UMAP, pages 113–121. ACM, 2017.

[11] J. M. Keller, M. R. Gray, and J. A. Givens. A fuzzy K-nearest neighbor algorithm. *IEEE Transactions on Sys., Man, and Cybernetics*, SMC-15(4):580–585, 1985.

[12] B. McFee and G. Lanckriet. THE NATURAL LANGUAGE OF PLAYLISTS. In *12th Intl. Society for Music Information Retrieval Conf.*, ISMIR, 2011.

[13] B. McFee and G. R. Lanckriet. Hypergraph Models of Playlist Dialects. In *13th Intl. Society for Music Information Retrieval Conf.*, volume 12 of *ISMIR*, pages 343–348, 2012.

[14] M. Schedl. The lfm-1b dataset for music retrieval and recommendation. In *2016 ACM on Intl. Conf. on Multimedia Retrieval*, ICMR, pages 103–110. ACM, 2016.

[15] S. Sedhain, A. K. Menon, S. Sanner, and L. Xie. AutoRec: Autoencoders Meet Collaborative Filtering. In *24th Intl. Conf. on World Wide Web*, WWW, pages 111–112. ACM, 2015.

[16] F. Strub, R. Gaudel, and J. Mary. Hybrid recommender system based on autoencoders. In *1st Workshop on Deep Learning for Rec. Sys.*, DLRS, pages 11–16. ACM, 2016.

[17] A. Vall, H. Eghbal-zadeh, M. Dorfer, M. Schedl, and G. Widmer. Music Playlist Continuation by Learning from Hand-Curated Examples and Song Features: Alleviating the Cold-Start Problem for Rare and Out-of-Set Songs. In *2Nd Workshop on Deep Learning for Rec. Sys.*, DLRS, pages 46–54. ACM, 2017.

[18] A. van den Oord, S. Dieleman, and B. Schrauwen. Deep content-based music recommendation. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Sys. 26*, NIPS, pages 2643–2651. Curran Associates, Inc., 2013.

[19] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and Composing Robust Features with Denoising Autoencoders. In *25th Intl. Conf. on Machine Learning*, ICML, pages 1096–1103. ACM, 2008.

[20] J. Winder. Session-Based Track Embedding for Context-Aware Music Recommendation. Master's thesis, University of Innsbruck, 2018.

[21] S. Zhang, L. Yao, X. Xu, S. Wang, and L. Zhu. Hybrid Collaborative Recommendation via Semi-AutoEncoder. In *Intl. Conf. on Neural Information Processing*, ICONIP, pages 185–193. Springer, 2017.