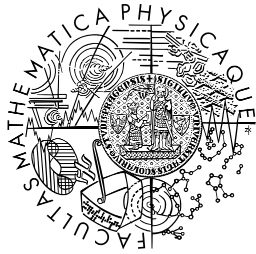

**FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University**

BACHELOR THESIS

Michaela Vystrčilová

Similarity methods for music recommender systems

Department of Software Engineering

Supervisor of the bachelor thesis:	Mgr. Ladislav Peška, Ph.D.
Study programme:	Computer Science
Study branch:	IOI

Prague 2019

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

I would hereby like to thank my supervisor Mgr. Ladislav Peška, Ph.D. for his time, feedback, advice and for the knowledge he has shared with me.

I am also grateful to the people who supported and encouraged me while writing this thesis.

Title: Similarity methods for music recommender systems

Author: Michaela Vystrčilová

Department: Department of Software Engineering

Supervisor: Mgr. Ladislav Peška, Ph.D., department

Abstract: Traditional music recommender systems rely on collaborative-filtering methods. This, however, puts listeners who do not enjoy mainstream songs at a disadvantage because CF systems depend on popularity patterns. Content-based recommendation methods might be useful in solving this issue. Since tag-based searches are a widespread tool to aid traditional music recommendation, this paper presents content-based methods measuring similarity between songs with focus on methods utilizing song's lyrics and audio recordings. First, we evaluated the accuracy of several approaches based on lyrics and audio information on real user playlists and found that lyrics-based methods yield competitive results to audio-based methods. Results also revealed that both categories include methods that are 100 times more accurate compared to random suggestions and that they have potential for even better results. After the evaluation phase, we selected well-performing methods and implemented them in a web application aiming on recommending novel music to the users based on their content-based profile.

Keywords: music recommendation, unsupervised feature learning, audio-based song similarity, lyrics-based song similarity

Contents

Introduction **5**

1 Data **9**
1.1 Datasets . . . 9
1.1.1 Lyrics dataset . . . 9
1.1.2 User-information dataset . . . 9
1.2 Final dataset statistics . . . 10

2 Lyrics-based methods **13**
2.1 Text embedding methods . . . 13
2.1.1 Bag of Words . . . 13
2.1.2 Tf-idf . . . 13
2.1.3 Word2Vec . . . 14
2.1.4 Doc2Vec . . . 14
2.1.5 Self organizing maps . . . 15
2.2 Related work . . . 16
2.3 Text representation choices . . . 16

3 Audio-based methods **19**
3.1 Basic audio representation methods . . . 19
3.1.1 Raw waveform . . . 19
3.1.2 Spectrograms . . . 20
3.1.3 Mel Spectrograms . . . 21
3.1.4 Mel Frequency Ceptral Coefficients . . . 21
3.2 Simple audio representation methods . . . 22
3.2.1 PCA . . . 22
3.3 Deep audio representation methods . . . 23
3.3.1 Convolutional neural networks . . . 23
3.3.2 Deep belief networks . . . 24
3.3.3 Recurrent neural networks . . . 24
3.3.4 Autoencoders . . . 24
3.4 Related work . . . 25
3.5 Audio implementation choices . . . 25
3.5.1 Basic audio representation choices . . . 25
3.5.2 Deep Audio representation choices . . . 25

4 Experiments **27**
4.1 Expectations . . . 27
4.2 Experimentation protocol . . . 27
4.3 Text method experiments . . . 29
4.3.1 Tf-idf experiments . . . 29
4.3.2 PCA on Tf-idf . . . 29
4.3.3 Word2Vec experiments . . . 30
4.3.4 SOM experiments . . . 31
4.4 Simple audio method experiments . . . 31

4.4.1 Audio preparation . . . 31
4.4.2 Raw Mel-spectrograms . . . 32
4.4.3 Raw MFCCs . . . 32
4.4.4 PCA with spectrograms . . . 33
4.4.5 PCA with mel-spectrograms . . . 33
4.5 Deep audio experiments . . . 34
4.5.1 Architecture . . . 34
4.5.2 Inputs and outputs . . . 35
4.5.3 GRU network with spectrogram input . . . 36
4.5.4 LSTM network with spectrogram input . . . 36
4.5.5 GRU and LSTM networks with Mel-spectrogram input . . . 37
4.5.6 GRU and LSTM networks with MFCC input . . . 37
4.6 Similarity metrics . . . 38
4.7 Evaluation . . . 38
4.7.1 Desired recommender-system features . . . 38
4.7.2 Evaluation measures . . . 38
4.8 Text method results . . . 41
4.8.1 Tf-idf results . . . 42
4.8.2 PCA_Tf-idf results . . . 42
4.8.3 W2V results . . . 43
4.8.4 SOM results . . . 44
4.9 Simple audio representation results . . . 46
4.9.1 Raw mel-spectrogram results . . . 46
4.9.2 Raw MFCCs results . . . 47
4.9.3 PCA on spectrograms . . . 48
4.9.4 PCA on Mel-spectrograms . . . 49
4.10 Deep audio representation results . . . 50
4.10.1 GRU network with spectrogram input . . . 50
4.10.2 LSTM network with spectrogram input . . . 50
4.10.3 GRU and LSTM networks with Mel-spectrogram input . . . 51
4.10.4 GRU and LSTM networks with MFCC input . . . 52
4.11 Discussion . . . 53
4.11.1 Expectations vs. reality . . . 53
4.11.2 Relative results . . . 54
4.11.3 Result interpretation . . . 55
4.11.4 Additional findings . . . 56

5 Web Application **63**
5.1 Analysis . . . 63
5.2 Implementation . . . 63
5.2.1 Technologies . . . 64
5.2.2 Design . . . 64
5.2.3 Similarity measure implementation . . . 66
5.2.4 Base data import . . . 70
5.3 Configuration options . . . 70
5.3.1 Similarity method configurations . . . 70
5.3.2 Email . . . 70
5.3.3 Server . . . 71

Conclusion **74**

Bibliography **75**

List of Figures **79**

List of Tables **81**

A Attachments **83**
A.1 First Attachment 83

Introduction

Millions of songs online provide an opportunity to find great songs for people with all kinds of music tastes. However, only a small fraction of all the songs that are produced becomes popular. Those are the ones people are being exposed to the most. They are promoted on various platforms such as YouTube[1] or Spotify[2], and played across all radio stations sometimes several times a day. After a while, older songs become less popular, one could use the term *"overplayed"* and other (usually new) songs take their place. But what if a person's next favorite song already existed, it just did not become popular? It is unlikely to hear unpopular but possibly likeable songs for people with unusual music preferences on the radio. Radio stations try to target as many listeners as possible. A recommender system that collects data about what a user listens to could on the other hand specifically target the person's taste and help anyone discover tracks perfectly tailored for them without being dependant on their popularity.

The suggestions recommender systems provide for basically any online content are crucial. With the amount of songs, movies, books, clothes, electronics and many more, it would be extremely time consuming for a person to go through all of the items in order to find what they are looking for. Recommendation systems are trying to make it easier for people to find what they want. They even try to predict, what they will be looking for next or what they might want but do not know it yet.

There are three main method groups to generate (not only) music recommendations for users. First are collaborative-filtering methods (CF) where recommendations are based on the preferences of like-minded users. Second are content-based methods where recommendations are based on the song content (tags, audio, lyrics, ...) and the third group are hybrid methods combining the first two together.

Generally, CF methods for all kinds of recommendation systems appear to be researched more extensively [1] however, there are certain drawbacks of these approaches. Most obviously, there is a problem with new, unrated songs because no user has viewed or liked them, so they cannot be recommended to like-minded users with a method based only on collaborative filtering. This is called the cold-start problem. Also, the recommendations tend to be dependent on user popularity patterns. Nevertheless, with enough user data, collaborative filtering methods generally outperform content-based methods [2].

Due to these observations, there are not many applications that would recommend songs based solely on their content and to the best of our knowledge, there is no music recommendation application that would recommend songs to its users based only on lyrics. As this is a logical consequence of the findings above, we believe that a recommender system based exclusively on content-based methods could be helpful for users with an unusual taste since it would not be popularity-dependant. We decided to introduce such a recommender application.

In order to create a content-based music recommender application we need to decide on the source(s) of content information. A basic CB recommender is

[1]https://www.youtube.com
[2]https://www.spotify.com

attribute-based. Common song attributes are the genre, the artist, creation year and so on. Nonetheless, we decided not to use these simple CB attributes in this thesis, because almost all music related application allow users to search based on tags so it would not bring anything new really.

Instead of simple CB attributes, we chose lyrics and audio as the sources of content information. The audio channel of the song is probably the ultimate low-level content information of every song. People listen to music because it is a pleasant sound and it is likely that it is audio features that define, whether a person likes a song or not. On the other hand, processing a song's audio channel to acquire meaningful features is an expensive and complex task with many options and hyperparameters that need to be set.

Song lyrics, i.e., a textual transcript of the vocals in the song is somewhat less informative, however it may still possess valuable information, for which, multiple processing methods were already developed. The process of transforming raw lyrics into some meaningful attributes is less demanding compared to the processing of raw audio. There is an intuitive a notion that their performance might be doubtful, however, many studies evaluate them on their genre classification accuracy [3] or compare them to collaborative filtering systems [4] which does not always mimic actual user behaviour.

Recommendation is mostly based on similarity between items which can be defined in various ways. Both lyrics and audio need pre-processing before establishing similarity.

Goals

The goals of this thesis are:

- to determine whether lyrics and audio-based methods mimic actual user behaviour and are relevant in recommender systems
- create a web application where these methods will be implemented to provide a recommender system which is not dependant on song popularity

In order to do so we take the following steps. We describe various ways of pre-processing text and audio signals in an unsupervised manner for content-based song similarity calculations. Then we select some of them based on previous studies and their features, evaluate their performance on real user playlists, compare the results and then implement fitting methods in a web application.

Although it originally seemed that processing both content modalities are similar, during our work on the thesis, it turned out that the complexity and diversity of the pre-processing steps exceeded our expectations. It includes language, text representation and similarity metrics for lyrics-based methods and audio extraction, audio representation and similarity metrics for audio-based similarity.

We decided to focus on unsupervised learning of song feature representation of both audio and text. This includes encoding a song into a vector so that a standard similarity-based recommendation technique can be used to evaluate similarity of two arbitrary songs without having any information about genre or other tags. The vectors can also be used for more advanced algorithms using for example Recurrent neural networks to calculate similarity. That is however above the scope of this work.

The web application's main purpose is to introduce the user to new songs he has not listened to yet based on a song similarity method he selects. The songs are provided by the application's default database but adding songs is possible too and its distance to other songs is taken into account for recommendations.

1. Data

1.1 Datasets

1.1.1 Lyrics dataset

We chose the *55000+ Song Lyrics dataset* from Kaggle.com[1] to obtain lyrics data. The Kaggle dataset originally contained 57,650 English songs. Its lyrics are scraped from LyricsFreak[2]. Extremely long and short lyrics were removed as well as all non-ASCII symbols from the lyrics. Figure 1.1 shows the first two entries of the dataset.

artist	song	link	lyrics
ABBA	She's My Kind Of Girl	/a/abba/ ahes+my+kind+of+ girl_20598417.html	Look at her face, it's a wonderful face And it means something special to me Look at the way that she smiles when she sees me How lucky can one fellow be? She's just my kind of girl, she ma...
ABBA	Andante, Andante	/a/abba/ andante+andante_ 20002708.html	Take it easy with me, please Touch me gently like a summer evening breeze Take your time, make it slow Andante, Andante Just let the feeling grow Make your fingers soft and light Let yo...

Figure 1.1: First two entries of the 55000+ Lyrics Dataset

1.1.2 User-information dataset

To evaluate text and audio based methods on real-life user data, we had to select a dataset containing song information and lyrics as well as a dataset with information about users and their played tracks. First we tried to match the lyrics dataset onto the *Thisismyjam* dataset[3]. However we were able to match only 6800 songs with lyrics as well as user data. We then tried the *Echo Nest Taste Profile Subset*[4] [5] dataset available on the Milion Song Dataset (MSD) website. The Echo Nest Taste Profile Subset provides 48,373,586 triplets of *user id, song id* and *the number of times the user has played a song.* This then had to be mapped onto the MSD dataset to get a name and artist for each song and then mapped onto our lyrics dataset.

After removing songs we did not have lyrics for, we ended up with 16,594 unique songs and 45,054 unique users. Even though it is a significant reduction it still provides enough data to carry out all the desired experiments. For each of the 16,594 songs we also acquired a mono .wav file.

[1] https://www.kaggle.com/mousehead/songlyrics
[2] https://www.lyricsfreak.com
[3] http://www.thisismyjam.com
[4] https://labrosa.ee.columbia.edu/millionsong/tasteprofile

1.2 Final dataset statistics

Overall our final dataset had 160,454 entries containing a user id, artist, song title and lyrics. We extracted two of different datasets, suited for different tasks throughout the thesis.

- The *Song dataset* denoted as SD in this work. It contains the 16,594 unique songs with their metadata (title, artist, lyrics and audio), the users were omitted.
- The *User dataset* denoted as UD. Each one of its 110,826 entries consists of a userID, song name and artist. It contains 11,123 unique users who have at 4 songs assigned to them so it is a dataset of all playlists of length at least 4.

Since the evaluation method is aimed to reveal the missing entries based on the implemented recommendation techniques[5], we studied the dataset and especially the playlist's lengths in more detail.

Here are some important remarks:

- Each user only has one playlist. This means there is a one to one mapping between users and playlists and the terms are used interchangeably.
- We do not know which songs the user has played most recently.
- Users with only one song are not useful for the purpose of evaluation.

When analyzing our dataset, it turned out, that out of 45,054 playlists, there are 22,257 with only one song, which left us with 22,797 we could use. We however decided to use only playlists of length at least four for evaluation because it still leaves us with enough unique playlists — 11,123 to be specific — and allows us to study deeper connections than song to song similarities.

The distribution of the lengths for useful playlists is shown in more detail in Figure 1.2. We can see that most of the playlists are short, almost a third of them only contains two songs. The average number of songs per playlists (including those containing only one song) is 3.56.

The average number of playlists a song from our dataset belongs to is 10.84. The distribution and the most popular songs are depicted in Figure 1.3. The by far most popular song with a total of 816 plays was *Royals* by *Lorde*. Second came *Radioactive* by *Imagine Dragons* with 674 users who played it. All other songs have been played by less than 500 users.

[5]Chapter 4

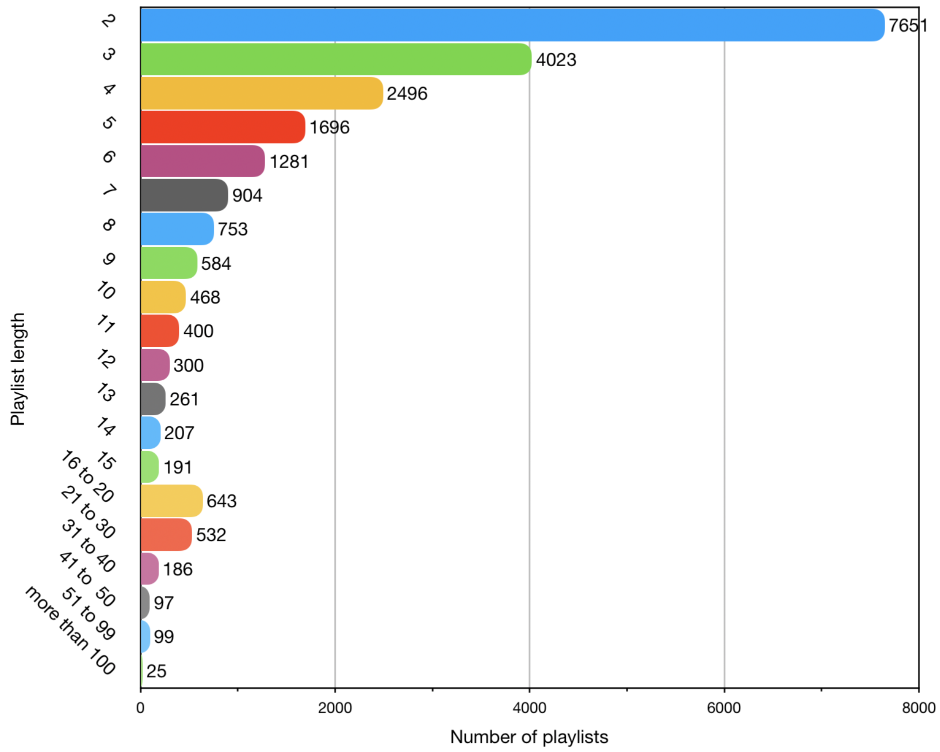

Figure 1.2: Playlists' lengths histogram

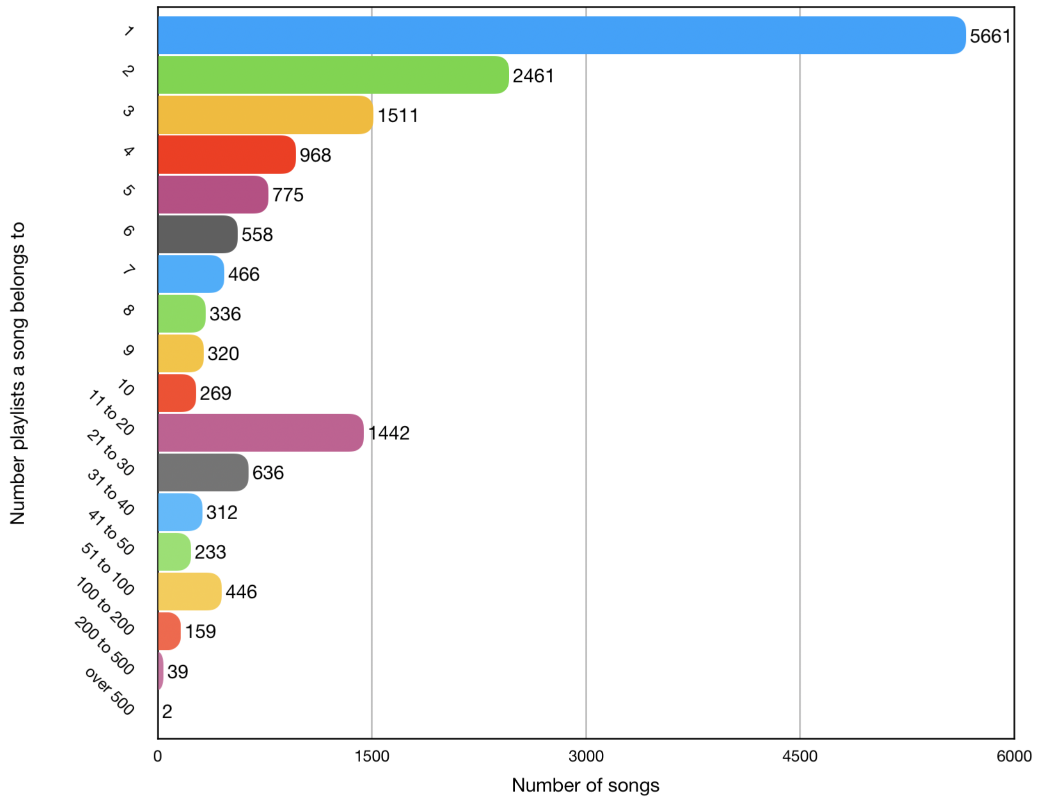

Figure 1.3: The histogram of playlist counts per individual songs

2. Lyrics-based methods

In this chapter we will briefly describe some of the most prominent methods to represent songs based on their lyrics. After specifically focusing on the positive and negative aspects of these methods, we will select suitable candidates for testing and potentially for our web application.

The reason to explore lyrics-based methods in this thesis is based on several factors. It is the belief of the authors that although the utilization of song lyrics is not completely unexplored as shown in Section 2.2, there is space for innovative research. For example, to the best of our knowledge, there are no recommender systems that would rely solely on lyrics analysis. An advantage of these methods could lie in providing relevant recommendations that would be more variable compared recommendations from other, more traditional techniques.

2.1 Text embedding methods

2.1.1 Bag of Words

The Bag of Words commonly referred to as BoW is a text representation method which counts how many times a word appears in a document. In the context of this thesis, it counts, how many times a word appears in the song lyrics. BoW represents each text (song) as a word-count vector where each index corresponds to the number of times a certain word appeared in it. An advantage of this encoding is its simplicity. The simplicity, however, brings some drawbacks. It for example ignores the fact that some words which can be found in most documents have a smaller informative value than others that only appear in a small fraction of the documents.

2.1.2 Tf-idf

Term Frequency-Inverse Document Frequency is another way of transforming documents into vectors. Unlike the BoW, Tf-idf does not measure only the counts of words in a document but it also measures their relevance. As can be deduced from the title term frequency, first the number of appearances of a word t in each document d proportional to the number of all words in that document - *tf(t,d)* - is computed. Then comes the inverse document frequency part - *idf* - where the words are weighted as seen in Formula 2.1. The words that appear frequently in most documents have lower weights than those who only appear in some.

$$idf(t) = log\frac{1 + n_d}{1 + df(t)} + 1 \tag{2.1}$$

Formula 2.2 is then the final Tf-idf formula multiplying the word frequencies with their weights:

$$tf\text{-}idf(t,d) = idf(t) * tf(t,d) \tag{2.2}$$

In both of the formulas above t is the word d is the document $df(t)$ is the number of documents containing the word t and n_d is the total number of documents.

2.1.3 Word2Vec

Word2Vec is a two-layer neural network trained to encode the linguistic context of a word introduced by Tomas Mikolov [6]. Each word has an assigned vector in a vector space of typically hundreds of dimensions generated from a large corpus. The position of a word corresponds to its context, meaning, that words that share common context are closer to each other.

There are two possible Word2Vec architectures, the continuous bag-of-words (CBOW) and the continuous skip gram. The CBOW predicts the current word from the words surrounding it — the context. It does not keep the order of the surrounding context words. The skip gram does, which makes it slower but also more effective, especially for infrequent words [6]. The Skip-Gram architecture takes one word and predicts all the context around it.

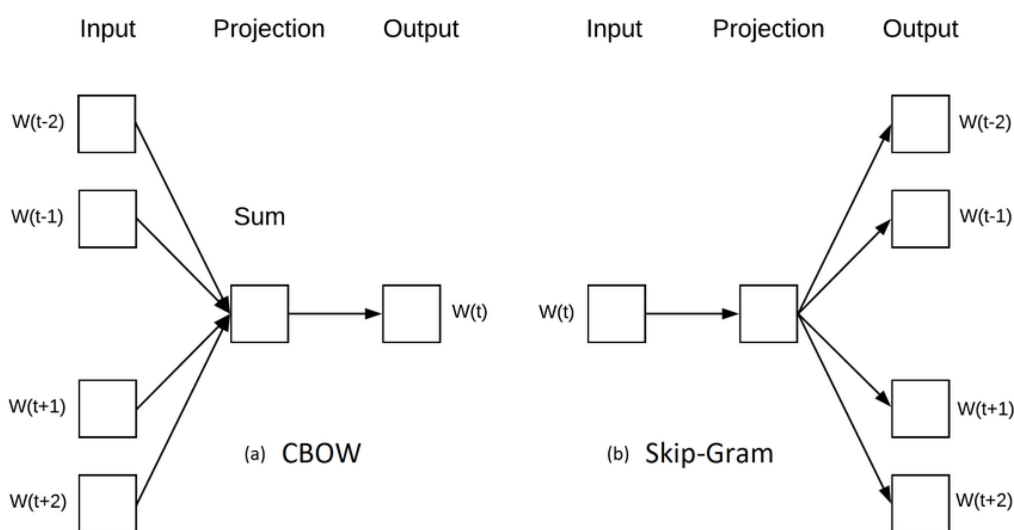

Figure 2.1: The CBOW and Skip-gram Word2Vec architectures from [7]

Multiple things have to be taken into account when training a W2V model. The information value of words that occur in all training documents is quite low so they can be removed to increase training speed. The dimensionality of the space also elevates accuracy only to a certain point so some threshold has to be set. Another parameter is the context window, which determines, how many words before and after a given word are included as its context.

2.1.4 Doc2Vec

Doc2Vec is an unsupervised algorithm that learns the feature representation of texts with varying lengths and encodes them into vectors of the same length. As the name suggests it is heavily based on the idea of Word2Vec. It was also first presented by the same group of researches in this paper [8]. The main idea of the method is to use the Word2Vec model but add one more vector to represent the paragraph as a whole. As in the Word2Vec model, there are two architectures for the Doc2Vec approach. The Distributed Memory (DM) version of Paragraph vector and the Distributed Bag of Words (DBOW) version of the Paragraph vector. The DBOW is faster but does not consider the order of the words as it predicts a random group of words from the paragraph vector. The DM on

the other hand takes previous words and the paragraph vector into account and predicts just one word. This way, because the paragraph does not shift across the text, the DM architecture is able to capture some word order but it requires more time to be trained.

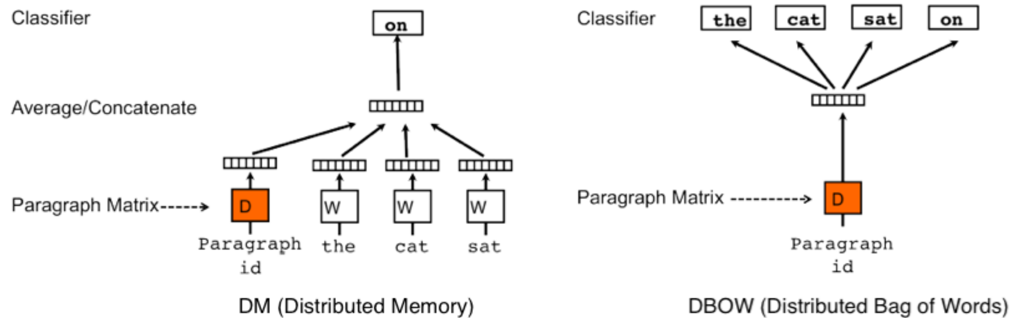

Figure 2.2: The Doc2Vec DM and DBOW architecture taken from [8]

2.1.5 Self organizing maps

A self organizing map (SOM) is a type of a neural network that learns how to reduce the dimension of input data in an unsupervised manner. SOMs were introduced by Teuvo Kohonen [9]. They use competitive dimensionality reduction (meaning the nodes in the SOM network compete to get the right to respond to the input data) which is quite unusual for neural networks as they usually use backpropagation. The models that SOMs compute are (usually) two dimensional spaces of neurons (called *codebook* vectors) where similar examples are close to each other and dissimilar examples further from each other.

The SOM network is trained through an iterative process which is visualized in Figure 2.3. It chooses one sample $\mathbf{x} \in R^n$ from the input training set at random and teaches it to itself. During teaching, the network feeds the chosen sample into all its units. A winner unit is calculated based on a similarity measure (usually Euclidean distance) between $\mathbf{x}$ and the *codebook* vectors. Finally the values of the network units are updated. The best-matching unit is moved a closer to $\mathbf{x}$ and so are all the topological neighbours of the best unit.

The neighbours are defined by a neighbourhood function. It decreases with time and decides how radical the change around the winner will be. There are multiple functions that can be used. One can use the Gaussian kernel around the winner, however this is quite computationally expensive. A good and more efficient function is sometimes called the *"bubble"* function which is constant over the whole neighbourhood of the winner and zero elsewhere [10].

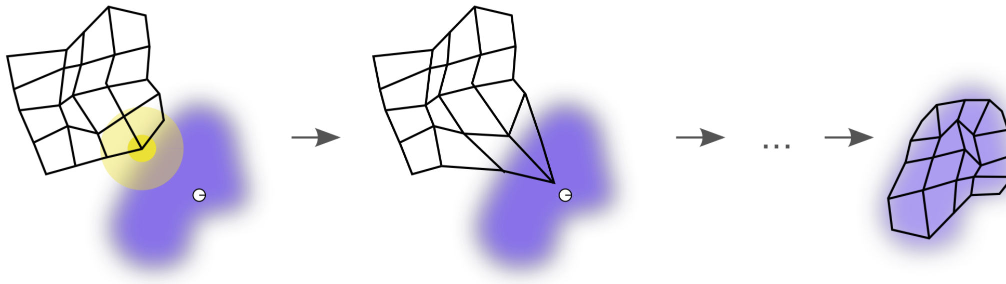

Figure 2.3: Visualization of the training algorithm used for SOM networks. The blue area represents the distribution of the data. The white dot is the randomly selected sample. On the left, the SOM network nodes are randomly spread accross the space. When finding a winner (middle) and its defined neighbourhood (the yellow area) the network moves towards the datapoint and eventually after repeated iterations spreads mimicking the distribution of the data (left). This image is taken from Wikipedia[1]

2.2 Related work

There are several papers on music recommendation based on lyrics. For example [4] has shown, that simple Tf-idf song embedding was 12.6 times more accurate then just random suggestions on the musiXmatch dataset[2]. In [11] the authors compared the Doc2Vec and the SOM algorithm using cosine similarity for vector aggregation on a dataset containing Hindi songs and found that in their experiments, the SOM outperforms Doc2Vec. Paper [3] even studies using intact lyrics as input for Recurrent (LSTM) and Hierarchical neural networks and evaluates it based on genre classification.

2.3 Text representation choices

When choosing methods for our web application there are several factors to consider. Besides the expected accuracy of the algorithms, which is often difficult to estimate since lyrics-based recommendation methods have not been researched extensively, we have to consider the implementation as well as temporal complexity features of all the methods. Also, the fact that we want to focus more on a cross-sectional approach rather than a thorough optimization of one particular algorithm means, we prefer diversity in our chosen algorithms.

The Bag of Words representation could be a good choice to get some kind of baseline results. Nevertheless, since the Tf-idf algorithm is widely based on the BoW and is still quite simple, we choose **Tf-idf** as our baseline. As mentioned at the beginning of this chapter, it was shown to be 12.6 times more accurate on the musiXmatch Dataset (MXD)[3] than just random suggestions, and that is

[1] https://commons.wikimedia.org/wiki/File:Somtraining.svg
[2] https://labrosa.ee.columbia.edu/millionsong/musixmatch
[3] https://labrosa.ee.columbia.edu/millionsong/musixmatch

what we hope to achieve with all of our text methods. A downside of the Tf-idf method is the length of its vectors. Even though they consist mostly of zeros, for our dataset, the length of each of them is over 40,000.

Word2Vec and Doc2Vec are two similar approaches. The issue with Word2Vec when representing a whole document, in our case the lyrics for one song, is the transition between the word vectors and the whole document encoding. A commonly used aggregation method is to define the document vector as the mean of all the word vectors.

Doc2Vec does not suffer from this problem, as its default is suited to represent a complete text. However, the problem with Doc2Vec is the amount of data it needs for training. Because every document is one sample, the number of documents necessary to achieve reasonable results is much higher than for Word2Vec where one sample is one word. What is also convenient with Word2Vec is, that there already exists a pre-trained Word2Vec model from Google[4]. It consists of 3 million words with a 300-dimensional vector for each. Three hundred dimensions is a reasonable number (especially considering the fact that the Tf-idf vectors have over 40,000 dimensions). It was trained on roughly a billion words from a Google News dataset. Therefore we chose the **Word2Vec** method over the Doc2Vec.

We also decided to implement the **SOM** network rather than Doc2Vec to represent songs because of a study showing, that Self organizing maps perform better than a Doc2Vec-based algorithm [11]. It also does not need as much data as the Doc2Vec to be trained.

One more thing we had to chose for the SOM was the form of the input. We decided to try the W2V representation. Mainly because the training of a self organizing map is quite computationally expensive and having vectors with over 40,000 dimensions would make it extremely time-consuming. We did not give up on the Tf-idf representation though. We trained another SOM where used Tf-idf vectors pre-processed by PCA which were reduced to length 4,457.

[4]https://code.google.com/archive/p/word2vec/

3. Audio-based methods

In this section we will describe the possibilities of how to transform an audio signal (in our case from a .wav file) into representations suitable for song similarity calculations. This process consists of many steps and a lot of research has been done on all of them as illustrated in Section 3.4.

The reason we are focusing on audio in this thesis is the notion, that what people care about in a song is its sound. There are patterns in music that are pleasant to the human auditory system, otherwise, music would not be so popular. We believe it is the sound wave that contains these patterns. It is difficult to define what exactly they are, so we hope that with the use of unsupervised machine learning algorithms, we will be able to find them and then locate them in unseen songs as well.

Figure 3.1 illustrates the steps of audio extraction. The blue part of the diagram describes the steps that are taken to acquire various basic music representations which are explained in Section 3.1. Each of these representations can be given as input to a machine learning algorithm as depicted in green from Section 3.2 or deep learning algorithm from Section 3.3 depicted in purple. Both simple and deep learning algorithms yield a final vector representation of the song.

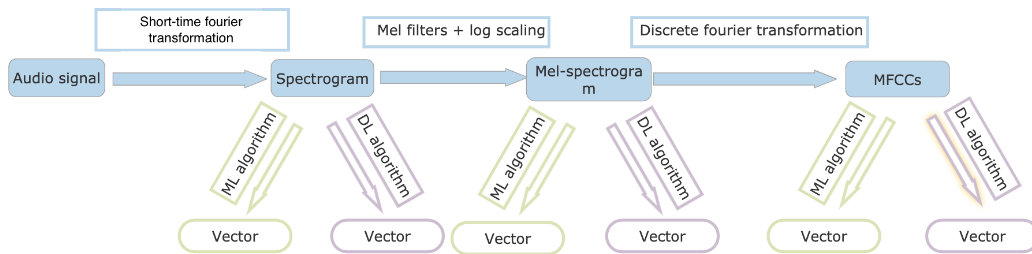

Figure 3.1: A diagram displaying the steps taken in audio extraction and feature learning. ML stands for machine learning and DL for deep learning.

3.1 Basic audio representation methods

3.1.1 Raw waveform

Sound is as vibration that spreads through gas, liquid or solid as a wave of pressure. For humans, the sound we hear has a frequency between 20Hz and 20kHz. Other sound waves are inaudible for humans. The most basic representation of sound as an audio signal is a *waveform*. It captures the variation of pressure over time. As we cannot store infinite data to capture the state of the wave in every moment, we need to establish a *sample rate*. A sample rate is the number of samples per second at which the pressure is recorded as amplitude. Common sample rates are 44,100 Hz and 22,050 Hz that capture oscillation up to 22,050 Hz and 11,025Hz [12].

3.1.2 Spectrograms

Raw waveform data have a lot of data points which make them spaciously demanding. Luckily, they also display strong regularities in their oscillations which gives us a different, more compact possibility to represent audio signal. The signal can be encoded as the strength of oscillations at various frequencies as opposed to amplitudes over time. Such an encoding is called a *spectrum* when sinusoids are used as prototypical oscillations.

A spectrum is obtained from a waveform by applying *Discrete Fourier Transformation*. The signal after DFT is represented by oscillations of a few frequencies spanning the full signal.

However, a problem with this approach is, that for longer recordings, many oscillations are present only over some limited time span or they change frequency. To represent all the oscillations the *Short-Time Fourier Transformation* can be computed. It slices the audio into small often overlapping windows, computes their spectra and then puts them together in a chronological order. This spectra matrix is called the *spectrogram* and for the song 'Someone Like You' by 'Adele' it has the shape of (2206 x 7796). It can be visualized as a graph with frequency on one axis and time on the other axis. The intensity of a frequency is represented by color.

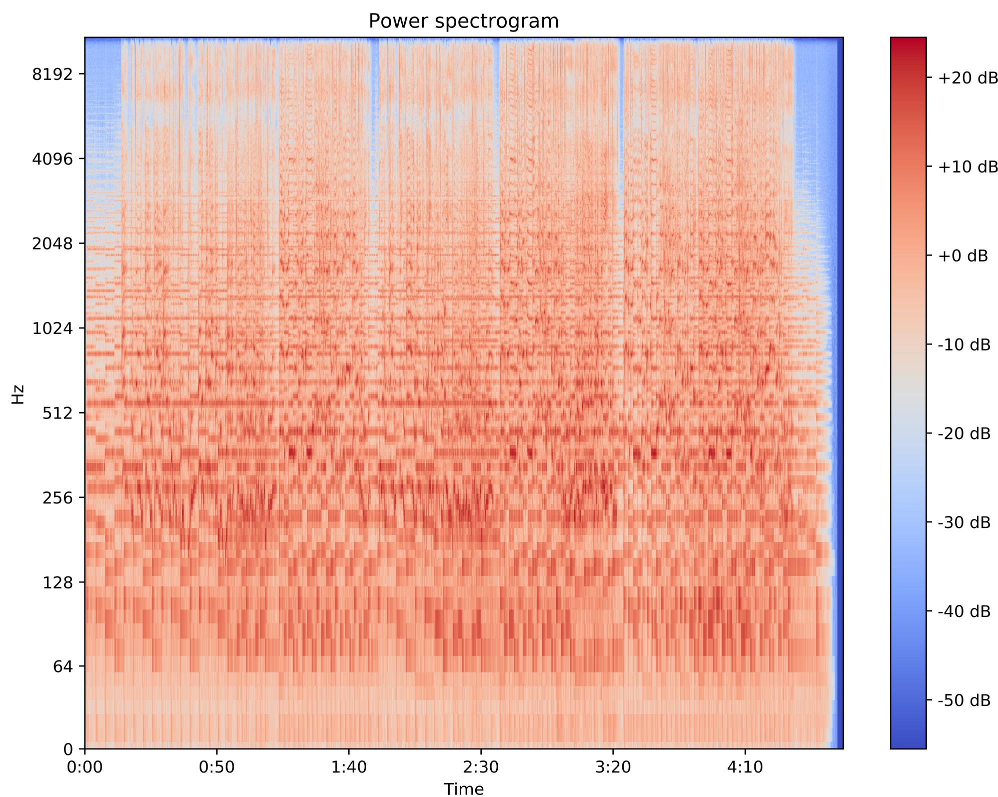

Figure 3.2: Spectrogram of the song 'Someone Like You' by 'Adele'. The intensity of different frequencies over time is converted to decibels.

3.1.3 Mel Spectrograms

Mel-spectrograms are another approach to reducing the dimensionality of the audio data. They are filtered spectrograms. Frequency bands are extracted by applying triangular *Mel-scale* filters to the power spectrum. The *mel scale* after which these spectrograms are called was named in 1937 in a study by Volkmann and Newman [13]. Since then it has been re-formulated multiple times, for example by Umesh at al [14]. It is based on the human perception of pitch and loudness and allows us to convert from Hz to Mels. Mels are more discriminatory at lower frequencies and less at higher frequencies - as is the human ear.

Each of the triangular filters has a response going from 1 to 0. They respond 1 at the center of some frequency and then their response decreases linearly to 0 towards to the place where they meet the neighbouring filters. When these filters are applied to a spectrogram, we get a mel-spectrogram. It is again a matrix of a smaller shape this time (320 x 7796) for the song 'Someone Like You' and it can also be visualized as illustrated in Figure 3.3.

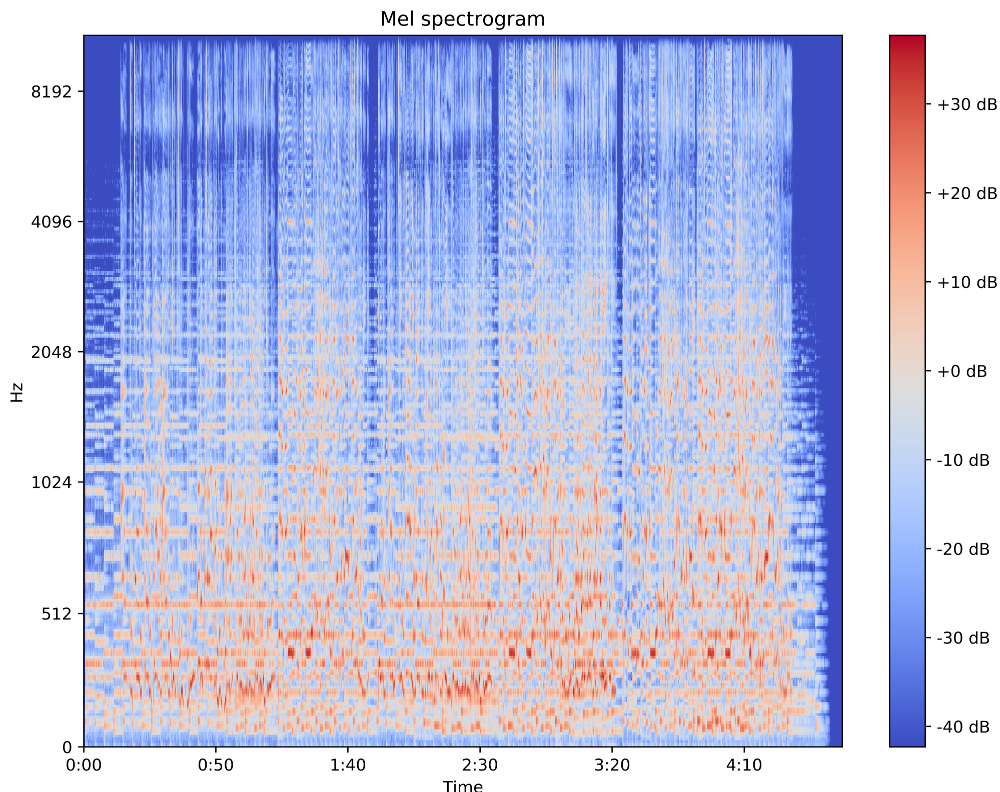

Figure 3.3: Mel-spectrogram of the song 'Someone Like You' by 'Adele'. The intensity of different frequencies over time is converted to decibels.

3.1.4 Mel Frequency Ceptral Coefficients

Mel-Frequency Ceptral Coefficients (MFCC) are another step further in compressing audio features. They are obtained by applying *Discrete cosine transformation* to mel-spectrograms. For the song 'Someone Like You' by 'Adele' which we used

as an example for spectrograms as well as mel-spectrograms, it created a matrix of shape (128 x 7796) which looks as Figure 3.4 illustrates.

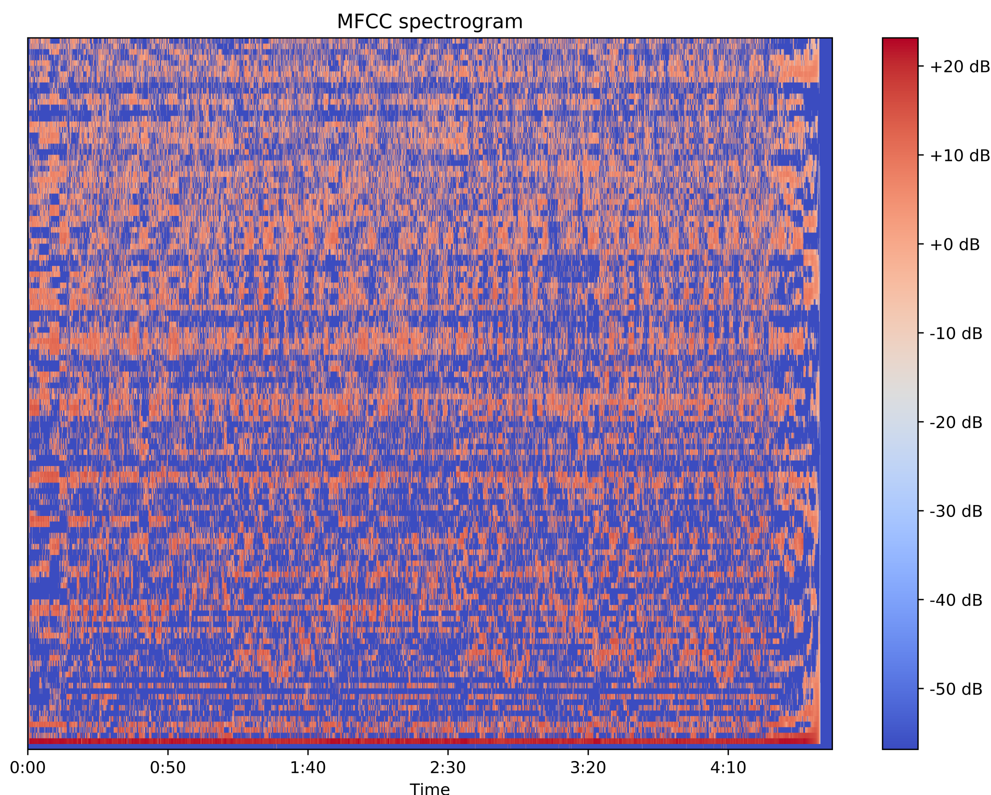

Figure 3.4: MFCCs of the song 'Someone Like You' by 'Adele'. The intensity of different frequencies over time is converted to decibels.

A nice introduction to music signal processing with respect to deep machine learning, where spectrograms, mel-spectrograms and MFCCs are explained in more detail can be found here [12] where we also drew a lot of our information from.

3.2 Simple audio representation methods

3.2.1 PCA

PCA is a common machine learning algorithm used to reduce dimensionality of the feature space that was invented by Karl Pearson in 1901 [15]. It tries to keep features with most variance and discards feature in which all the data points are highly correlated. The data space is transformed in such a way, that the first principal component (PC) has the largest possible variance, the second PC the second largest variance etc.

Mathematically it is an orthogonal linear transformation to a new coordinate system where the base vectors are the principal components. To achieve this, it is first necessary to center the data around the origin. That is done by subtracting the mean of each variable from the data. After that a covariance matrix is computed with its eigenvalues and corresponding eigenvectors. After normalizing

the eigenvectors, they can be interpreted as the new basis vectors. This new basis transforms the covariance matrix, so that it becomes diagonal. Each of the diagonal elements represents the variance of each axis. All the components without any reduction give us the whole information about the input data, only in a vector space with a different basis. Every component explains some portion of the data's variance.

The variance can be calculated by dividing every eigenvalue corresponding to an eigenvector with the sum of all eigenvalues. The dimensionality reduction is dependant on how many components we want. If we want to visualize the input data in a 2D graph, we create a space with only the first two PCs as a base and map all the data onto it. The reason to use this algorithm in our thesis would be mainly to reduce the length of the audio flattened audio matrices.

The PCA assumes that there is linear correlation between features. If there is not, the PCA will not discover it and will loose a lot of information with the dimensionality reduction it performs.

3.3 Deep audio representation methods

3.3.1 Convolutional neural networks

Convolutional neural networks are neural networks that have become extensively researched after AlexNet (a form of CNN) was demonstrated in 2012 and outperformed all other methods for visual classification [16]. As with other neural networks, CNN's biggest advantage is to emulate behavior of unknown non-linear functions. CNNs have the ability to map high-dimensional data into a space of finite categories (with hundreds or thousands of classes). They are widely used in visual imagery tasks.

The idea behind convolutional neural networks is to use *local filters* instead of creating fully connected layers. This has risen from the idea that in images, there are correlated compositions on short scale distances, rather than at large distances. For example when detecting a human face in an image with a tree in the background, the tree does not have much to say about the face, unlike the eyes or the nose by which the face can be identified and are in a much greater proximity to each other. This is also the reason why in the CNN's architecture, the layers are not fully connected.

There are 4 main components that are generally included in a CNN network. The *Convolution layer*, the *ReLU*, the *Maxpooling layer* and the *Fully connected layer* that yields the output. To briefly describe these layers lets start with convolution. The convolutional layer helps to reduce the number of connections and weights. It consists of filters that can be learned. These only take a small number of nearby features into account at a time but extend through the whole input. Each of the filters creates a 2D activation map by computing the dot product of entries of the filter and the input. ReLU (rectified linear unit) is generally used to increase non linear properties of the decision function. Its function is $f(x) = max(0, x)$ which is applied to the results of the convolution to speed up training — compared to previously used functions for example sigmoid functions — without affecting the receptive fields of the convolution layer. The pooling layer also reduces the number of parameters and helps prevent over-

fitting. The most common function to implement is *max pooling*. The features are partitioned into a set of non-overlapping rectangles (if input is 2D) and each of these rectangles is represented by its maximal value. The final layer is usually fully connected. Its neurons have connections to all activations of the previous layer and their activation is then computed as an affine transformation [17].

3.3.2 Deep belief networks

Deep belief networks introduced by Hinton [18] are multiple *Restricted Boltzmann machines* greedily stacked on top of each other. RBMs are shallow two-layer neural nets created by Paul Smolensky in 1986 [19]. The first layer is called the visible layer, the second layer is called the hidden layer. The nodes of each layer communicate with the previous and subsequent layer but there are no connections between nodes of the same layer. Each visible node takes a low level feature to be learned and multiplies it by some weight. The results for each feature of an input sample are then summed, bias is added and the result is passed through an activation algorithm which produces output for each hidden node. These outputs then can be redirected into another hidden layer instead of the output of the neural network.

RBMs also have the ability to reconstruct data without supervision. When the input makes it through both layers it then becomes input for the hidden layer and travels through the neural network in the opposite direction. The activations are multiplied by the same weights and passed to the visible layer where a new bias is added. The output of the visible layer is then compared to the initial input and the network adjusts weights so that it minimizes the difference between the input and the output.

3.3.3 Recurrent neural networks

Recurrent neural networks have one major difference compared to other neural networks. They include feedback loops in their structure which allows them to exhibit dynamic behaviour and makes them useful for processing sequential data. RNNs have a hidden state that is determined by previous states and is updated with every subsequent step. There are many variants of RNNs such as *Fully recurrent, Long short-term memory* introduced by Hochreiter and Schmidhuber in [20], *Gated recurent units* first described in [21], or *Bi-directional* invented by Schuster and Paliwal in [22].

Recurrent neural networks are used for working with sequential data for example in speech recognition [23] or time-series anomaly detection [24].

3.3.4 Autoencoders

Autoencoders are not tied to one type of neural network. They can be build from recurrent layers, convolutional layers or from a simple Multi-layer Perceptron.

The autoencoder has two parts the encoder and the decoder. It is trained in an unsupervised manner. The encoder takes the input data and reduces its dimensions. The decoder then tries to recreate it into its original form without knowing, what the data looked like before the encoder processed it. The idea is,

that over time, the encoder learns how to shrink the data with retaining as much information as possible for the decoder so the decoder it is able to recreate the original form as accurately as possible.

3.4 Related work

For the signal representation, there is obviously the possibility to use raw audio data as input for any machine learning algorithm. This however is a quite uncommon approach and when tested recently in tag prediction [25] it had worse results than standard spectrograms. Multiple studies and experiments have been done using spectrograms as audio representation for classification tasks — for example [26] — or for unsupervised feature learning [2], [27], [28], which is what also our area of interest.

In these studies the inputs were fed into various neural networks. The output of those was then evaluated on music classification or compared to music similarity estimation using similarity metrics directly on spectrograms. We are going to do a similar thing in this thesis. Our evaluation will be done on user playlists and the methods will be compared to each other.

3.5 Audio implementation choices

3.5.1 Basic audio representation choices

The flattened vectors from spectrograms, mel-spectrograms and MFCCs have tens sometimes even hundreds of thousands of features which is a big disadvantage, especially, when we have an intention of implementing them inside the web application. Nevertheless, we decided to test recommendations based at least on **raw mel-spectrograms** and **raw mfccs** to have an idea on how audio data without intervention of machine learning behave. We also decided to use the **PCA** to transform spectrograms, mel-spectrograms because we believe that it could reasonably reduce their vector lengths and remove noise from the data.

3.5.2 Deep Audio representation choices

Neural networks are a quickly developing and expanding field with many various applications. It is extremely time consuming to build an accurate neural network for a specific task. Therefore, we decided to choose our neural network architecture and parameters based on literature relevant to the topic of audio feature learning. The main requirements we had for the algorithms was that they have to work in an unsupervised manner and that they have to reduce the feature space. This pruned the number of possibilities for us considerably, as most architectures are designed for music classification (mostly into genres) or speech and sound recognition.

We decided that **autoencoders** suite the task in this thesis best. It is an unsupervised algorithm. The encoder part does exactly what we desire which is encoding the input sample into a smaller dimension. The decoder transforms it

again into the same vector. The decoder part is only necessary for training and only the encoder is then used for the song representation.

Since we are working with sound data, the choice was to use sequence-to-sequence RNN layers which have the ability to encode sequences (spectrograms, mel-spectrograms and mfccs are m x n matrices). The autoencoder architecture and the choice of layers was mainly inspired by this paper [29] where they used GRU layers with mel-spectrograms as input. However, we want to add more methods into comparison, so we implemented not only neural networks with **GRU** layers but also **LSTM** layers and used spectrograms, mel-spectrograms and also MFCCs as input rather than focusing on one type of layer or input and tuning one particular neural network to give the best results possible. A more detailed description of the architectures is provided in Section 4.5.

4. Experiments

In this chapter we describe the experiments we performed on song preprocessing methods chosen in Chapters 2 and 3. This includes describing every method's input, training (if there was any) and output (meaning the vector-encoded representations for each song in the SD). All this is presented in Sections 4.3, 4.4 and 4.5. Another thing we did not cover yet and will be presented here, in Section 4.6, is describing how the different vector representations will be aggregated into a definition of similarity.

In Section 4.7 we acquaint the reader with evaluation methods, the reasons we decided to use them and the evaluation results for each method. The results are first presented separately for each method (or a group of closely related methods) in Sections 4.8, 4.9 and 4.10, and then summarized and interpreted in Section 4.11 where various graphs and tables illustrating the prominent or interesting trends and tendencies can be found. Before all this however, we start with stating, what the expected outcomes were initially.

4.1 Expectations

Even before reading any literature, we made two main predictions:

- **First:** Audio-based methods will perform better than text-based methods.
- **Second:** More advanced methods will outperform simpler machine learning methods.

By more advanced methods we mean methods that were invented more recently, are more computationally expensive and/or have a more complicated mathematical idea behind them.

The first prediction was mostly based on the intuition, that audio contains more information about a song than the lyrics and it is also what people care about more when listening to music, both of which makes it more relevant for song encoding.

The second prediction was also based on intuition and later supported by reading into this topic. Most of the papers we studied (meaning those in Sections 3.4 and 2.2) were describing neural networks performing audio or text-based recommendation or classification. Their results were then compared to simpler algorithms which they mostly outperformed.

4.2 Experimentation protocol

Every method was trained on either lyrical or audio data of all 16,594 songs from the SD dataset. For each method (except of raw audio methods which do not need any), we created a model. In addition, we created a matrix — the *Representation matrix*, denoted as R_m where m stands for a particular method — after training the method's model. Each R_m has the shape of (16594 x $l(v_m)$) where $l(v_m)$ is the length of the song vector representation for method m. Row $R_{m_{i,*}}$ contains the representation of song s_i from the SD dataset using method m.

We also calculated a *Distance matrix* denoted as D_m for every method. The shape of the D_m is (16594 x 16594) and it is the same regardless of the method. On position $D_{m_{i,j}}$ is the similarity of $R_{m_{i,*}}$ and $R_{m_{j,*}}$. We talk more about similarity in Section 4.6.

To make things easier for the rest of the thesis we now present all the methods that were tested and introduce a nomenclature. There are 3 different parts each method name has — the name of the main algorithm, the name of the input and the length of the output vector. Every method can be uniquely identified by a combination of these three parts. Where there is only one or two parts necessary to uniquely identify the method, we omit those parts which are surplus.
The tested methods are:

- **Tf-idf** which stands for the Tf-idf method.
  Training is described in 4.3.1 and the results in 4.8.1.
- **PCA_Tf-idf** which stands for PCA with Tf-idf vectors as input.
  Training is described in 4.3.2 and the results in 4.8.2.
- **W2V** which represents the Word2Vec method.
  The training is described in 4.3.3 and the results in 4.8.3.
- **SOM_PCA_Tf-idf** and **SOM_W2V** which stand for the SOM network with PCA_Tf-idf vectors as input and the SOM network with W2V vectors as input.
  Training is described in 4.3.4 and the results in 4.8.4.
- **Raw mels** which are raw mel-spectrograms.
  Training described in 4.4.2 and the results in 4.9.1.
- **Raw MFCCs** which are raw MFCCs.
  Training is described in 4.4.3 and the results in 4.9.2
- **PCA_spec_1106** and **PCA_spec_320** which stand for PCA with spectrograms as input and output lengths of 1,106 for the first method and 320 for the second method.
  Training is described in 4.4.4 and the results in 4.9.3.
- **PCA_mel_5715** and **PCA_mel_320** which stand for the PCA with mel-spectrograms as input and the output length of 5,715 for the first method and 320 for the second method.
  Training is described in 4.4.5 and the results in 4.9.4.
- **GRU_spec_20400** and **GRU_spec_5712** which stand for an autoencoder with GRU layers and spectrogram input. The output vectors have length 20,400 for the first method and 5,712 for the second method.
  The architecture is described in 4.5.1, training is described in 4.5.3 and the results in 4.10.1.
- **LSTM_spec_20400** and **LSTM_spec_5712** which stand for an autoencoder with LSTM layers and spectrogram input.
  The architecture is described in 4.5.1, training is described in 4.5.4 and the results in 4.10.2.

- **GRU_mel** which stands for an autoencoder with GRU layers and mel-spectrogram input.
  The architecture is described in 4.5.1, training is described in 4.5.5 and the results in 4.10.3.

- **LSTM_mel** which stands for an autoencoder with LSTM layers and mel-spectrogram input.
  The architecture is described in 4.5.1, training is described in 4.5.5 and the results in 4.10.3.

- **GRU_MFCC** which stands for an autoencoder network with GRU layers and MFCCs as input.
  The architecture is described in 4.5.1, training is described in 4.5.6 and the results in 4.10.4.

- **LSTM_MFCC** which stands for an autoencoder with LSTM layers and MFCCs as input.
  The architecture is described in 4.5.1, training is described in 4.5.6 and the results in 4.10.4.

4.3 Text method experiments

4.3.1 Tf-idf experiments

Input

The lyrics of each song from SD were stripped of all punctuation characters as well as apostrophes and converted into a single string. All lyric-strings were appended into a list of strings which formed the training dataset.

Training

The training dataset was given to the `fit_transform` method as a parameter. The method was called on an instance of `TFidfVectorizer` from Python's `sklearn` package. The results were saved into the R_{Tf-idf}. The `TFidfVectorizer` instance was saved as the model to be potentially used in the proposed web application.

Output

The song representations were vectors of length 40,165. They contained a lot of zeros so it was possible to store them as *sparse vectors*. Sparse vectors however, are more difficult to operate with and therefore we also converted them into dense vectors.

4.3.2 PCA on Tf-idf

Because simple Tf-idf yielded good results we wanted to implement it. The long vectors posed a problem to our web application though. It is sometimes necessary in the application to calculate the distance between a newly added song and all

the songs that are already in the database and this calculation is more complex for longer vectors. To reduce the complexity of this task but still use Tf-idf we decided to try to reduce the dimensions of the vectors using PCA.

Input

As input, we provided the Tf-idf vectors for songs from SD to the PCA which we acquired as described 4.3.1. We did not normalize them.

Training

We first trained a PCA from Python's `sklearn.decomposition.PCA` without any dimensionality reduction. We then chose a space were the explained variance ratio was equal to 90%. This space had 4,457 dimensions. Knowing this, we trained a new PCA instance with 4,457 components which reduced our Tf-idf vector's length from 40,165 to 4,457 and saved it as the model.

Output

The output were vectors of length 4457 and were saved to the R_{PCA_Tf-idf}.

4.3.3 Word2Vec experiments

Input

The input for the W2V model was the same as for the Tf-idf method.

Training

In the case of Word2Vec, we did not perform any training. Instead, we used the a subset of the pre-trained W2V model from Google[1] containing 200,000 most common words which cover all meaningful words in the song lyrics we have. Most common means, that they appeared most in their training documents. If there was a word in a song that was not in the subset it was ignored.

Output

Because the Google model takes always just one word as input and returns its vector of fixed length 300, we had to put the word vectors together into one song-representation vector. We chose a basic approach where we averaged all the word vectors into one final song vector. The song vector's position i contained the average over the values of word vectors' position i. This approach yielded a vector of length 300 which is significantly lower than the Tf-idf vector.

[1] https://code.google.com/archive/p/word2vec/

4.3.4 SOM experiments

Input

We tried the W2V vectors from 4.3.3 as input for the self organizing map, mainly because of their length and also because we were hoping, that the SOM could improve the results of the W2V method. We also tried to train the SOM on the output vectors of the PCA_Tf-idf method. We did not try the raw Tf-idf vectors as they are longer and it would prolong training significantly. Also, it makes sense to use PCA_Tf-idf because it yielded better results than raw Tf-idf.

Training

We used a python library called `minisom` [30] to create the self organizing maps. We build a map with a grid size $5 * |SD|$ and the number of iterations was also 5 times the size of SD. We saved a model after each multiple of 16,594 iterations and interestingly, the representations did not change after 33,187 iterations (which is 2*16,594). It was also necessary to normalize the input vectors and set learning rate to 0.2. Otherwise the songs formed 3 to 5 large clusters on the grid placing thousands of songs on the same coordinate.

Output

The output representation for each song was a vector of length two. It is possible to display the songs on a 2D map which we also did and it can be seen in Figure 4.9.

4.4 Simple audio method experiments

4.4.1 Audio preparation

In order to encode audios of songs, we first needed to define some kind of standard audio-form to make the audio information suitable for the machine learning methods that we are testing in this section and the next. We decided to extract chunks of the same duration from all songs, so the input vectors for each category (spectrograms, mel-spectrograms and mfccs) are also of the same length within each category. Since all songs have different lengths and one complete 3.5 minute long song results in a spectrogram of size 5,214x2,206 which when flattened is a vector of length 11,502,084 we decided to extract only 15 second excerpts from each song to create spectrograms, mel-spectrograms and MFCCs from those. We took 5 seconds between the 15th and 20th second, 5 seconds starting in the middle of the song and 5 seconds starting 15 seconds before the end. We did not start at the beginning and at the end because in some songs, there is silence or applause or some talking before the actual song starts or after it ends.

It was also necessary to decide on some parameters for spectrogram, mel-spectrogram and MFCC extractions such as window width, window overlap and the number of mel-frequency bands. As stated previously, our neural networks were inspired by [29] where they also performed parameter optimisation for the

mel-spectrograms which they used as input for neural networks. We decided to use the reported parameters to create the spectrograms, mel-spectrograms in this thesis. We use spectrograms, mel-spectrograms and MFCCs as input for not only neural networks but also the PCA.

The resulting choices were following. We set window width w to 0.2 and window overlap w_o to $0.5w = 0.1$. For mel-spectrograms it is also necessary to choose mel-frequency bands which were set to 320 because in the findings from [29], the performance of neural networks did not increase for values higher than 320. For MFCCs we decided to set the number of coefficients to 320 which is the same as the number of Mel- frequency bands.

With these parameters, the shape of an extracted spectrogram was (408 x 2206) which is a vector of lenght 900,048 when flattened. A mel-spectrogram matrix had the shape of (408 x 320) which when flattened is 130,560 and the MFCC matrix had the shape of (646 x 128) which when flattened is a vector of length 82,688.

We used Python's `librosa` library [31] to cut songs into the 15 second excerpts which we then stored the in .wav files. We used methods `librosa.core.stft` to generate spectrograms `librosa.feature.mel_spectrogram` to generate mel-spectrograms and for the MFCC coefficients we used `librosa.feature.mfcc` all of which received the 15 second audio excerpt from the .wav file loaded using `librosa.core.load` as a parameter. These functions return a matrix of shape (m x n) where m is the number of timestamps and n the number of features. The reason why the MFCC matrix has a different number of timestamps than spectrograms and mel-spectrograms for the same length of audio is, that we only set the number of MFCCs in the `librosa.core.mfcc` function but left the window width and window overlap default.

4.4.2 Raw Mel-spectrograms

The mel-spectrograms were created from the 15 second long audios described in 4.4.1. Extracting spectrograms respectively mel-spectrograms does not require any training. It a is mathematical procedure explained in 3.1.3.

As mentioned in the previous section, the mel-spectrograms we got after transforming a 15 second long audio with 320 mel-frequencies bands were matrices of size (408 x 320) which when flattened turned into vectors of size 130,560. This turned out to be too long to implement in our application, however, we still tested this method and the results are in Subsection 4.9.1.

4.4.3 Raw MFCCs

The 15 second audios of all songs from SD were given as parameters into the `librosa.feature.mfcc` method one by one with parameters defined in 4.4.1. Training was not necessary since acquiring mel-frequency cepstral coefficient is a matter of Fourier Transformations and is explained in Subsection 3.1.4. The resulting MFCC matrix for one song from SD had the shape of (646 x 128). When flattened we got a vector of length 82,866. That turned out to be too long for practical use in the application. Nevertheless we tested this method and the results can be found in Subsection 4.9.2.

4.4.4 PCA with spectrograms

Input

We used spectrograms acquired as described in Section 3.1.2 as input for the PCA. Because the output of the `librosa.core.stft` method is a complex matrix, we computed the absolute value of each matrix entry. Afterwards, we flattened the matrix into a single vector and normalized it using a `MinMaxScaler` from the `sklearn` Python package. Normalization was very important, without it, the resulting rankings were almost random.

Training

Training was a little bit challenging with input vectors of length 900,048. As the whole (16594 x 900048) matrix did not fit into memory at once instead of using `sklearn.decomposition.PCA` we turned to a PCA with the possibility to be trained in batches. This kind of PCA is also provided by `skearn` in the `sklearn.decomposition.incrementalPCA` module.
Our batch size was 1,106. When we tried to increase it, we got a memory error. This was a little inconvenient because the PCA's the maximum number of components is $min(n_samples, n_features)$. In our case, $n_samples$ was only 1,106 meaning we could get a maximum of 1,106 components which explained about 57% of the dataset's variance. We saved this incrementalPCA instance with 57% varince explained as our first model and the method was denoted as PCA_spec_1106.

We then tried to decrease the number of components even more to get vectors of length 320. The lenght was inspired by the number of Mel-frequency bands used when extracting mel-spectrograms. The training was the same as for the PCA_spec_1106 but we kept only 320 components and saved the incrementalPCA instance as the model for this method called PCA_spec_320.

Output

The output vectors for PCA_spec_1106 were of length 1,106 and for PCA_spec_320 they were of length 320.

4.4.5 PCA with mel-spectrograms

Input

The input for the PCA_mel method were mel-spectrograms from 4.4.1 which were flattened and normalized using a `MinMaxScaler` from the `sklearn` Python package.

Training

Unlike the spectrograms, mel-spectrograms did fit into memory so we were able to train the PCA on the whole dataset at once. This allowed us determine what number of components explains 90% of the variance ratio and use it. We found that 90% of variance is explained by 5,715 components which is what we used to train one model.

The results were good, therefore we decided to try reducing the dimension even more to 320 as we did with PCA having spectrograms as input. Like this, we again created two models, the first one is called the PCA_mel_5715 the other the PCA_mel_320.

Output

The output for the bigger model was a vector of length 5715, for the smaller model, it was a vector of length 320.

4.5 Deep audio experiments

4.5.1 Architecture

Before analysing each of the deep audio methods independently, we will describe the two neural network architectures we used.

As stated multiple times before, we decided to build the networks based on the [29] paper. We designed the architecture in a way they did with some slight adjustments and extensions. The audio representation learning they performed is displayed in Figure 4.1. The first most notable thing is, that their network was designed to classify sounds, not encode songs. However, it consisted of two parts. The first part was an autoencoder and the second part a multi-layer perceptron. The autoencoder was trained in an unsupervised manner and the outputs were then fed into the MLP which did the classification.

We therefore figured, that we can take advantage of the first part of their neural network and discard the MLP. The portion of their procedure that we also performed is marked with the red rectangle which we added to their illustration. The reason why they performed feature fusion in (d) is because they worked with stereo wav files and were putting together the features of the different channels. In our case this was not done as we only had a mono wav files for each song.

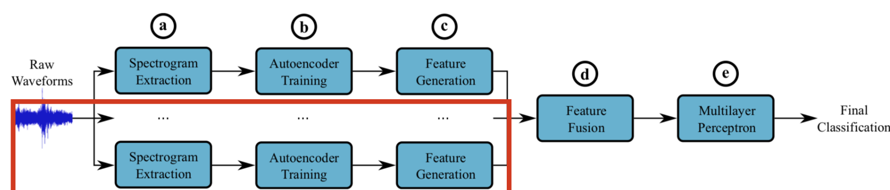

Figure 4.1: The steps in feature learning from [29] where we also got this diagram from. We added the red rectangle which represents the portion of their procedure that was also (with adjustments) performed by us.

This means, we only used the autoencoder part. Unlike them, instead of using the `auDeep` library[2] we decided to build the networks with the `Keras` library [32] as it has a convenient model-creation API for Python. We had also access to GPU computers and `Keras` (with `Tensorflow` backend) makes it easy to take advantage of faster training on GPUs.

[2]https://github.com/auDeep/auDeep

We created two architectures. One with two GRU layers for the encoder and one Bidirectional layer for the decoder. This follows the paper. We also decided to create another architecture with LSTM layers instead of GRU layers even though [29] found in their work that the additional complexity did not yield better results. Both architectures can be seen in Figure 4.5.1. We decided to use sequence-to-sequence RNNs to encode the vectors.

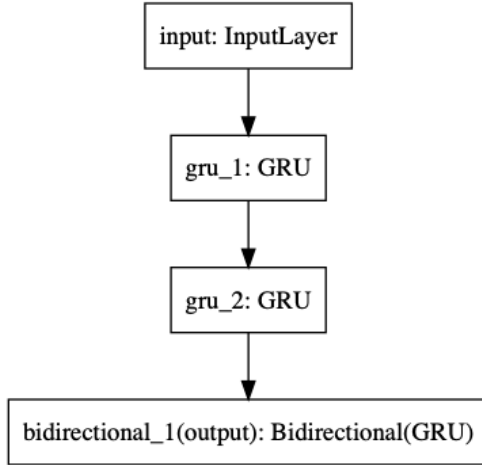

Figure 4.2: The general architecture of GRU neural networks

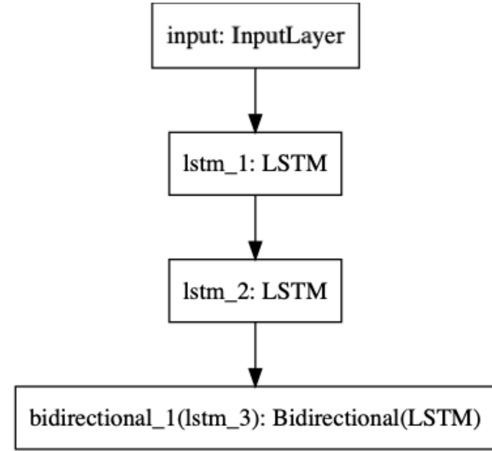

Figure 4.3: The general architecture of LSTM neural networks

The main motivation behind using LSTMs as well as GRU layers in this thesis was that LSTM layers are specifically suited for sequential data such as audio. GRU layers are a newer, simplified version of LSTMs. We were hoping that the more complex layers could encode audio data into the same dimensions as the GRU networks but help to make the similarities more accurate.

We used the *mean squared error* as the loss function which is the standard for autoencoder networks. We used the *adam* optimiser, the same as in the [29] paper. We had to decrease the learning rate from 0.001 to 0.0001. Before we did that, the resulting predicted vectors often consisted of just *NaN*s.

4.5.2 Inputs and outputs

Both the GRU and the LSTM network architectures were used to create models with all three kinds of inputs — the spectrograms, mel-spectrograms and the MFCCs. The inputs were all passed in the form of matrices containing ($n_features$ x $n_timestamps$). GRU as well as LSTM networks take matrices, not just vectors as input. Before training, the input matrices were normalized using the `MinMaxScaler`.

One important thing to note here is that we used sequence-to-sequence RNNs so the dimensionality of only the $n_features$ and not the $n_timestamps$ was reduced. The 15 second audios yielded 408 time stamps and 2206 features for spectrograms which when flattened is a vector of length 900,048 and 408 time stamps and 320 features for mel-spectrograms which is a vector of length 130,560.

With MFCCs the number of time stamps was 646 and the number of features 128 which gives us a vector of length 82,688 when flattened. Therefore, we did not attempt a dimensionality reduction as big as with PCA which does not care if a feature is a time stamp or a sample and the output vectors had to be of length at least 408 for autoencoders with spectrograms and mel-spectrograms as input and 646 for autoencoders with mfccs as input.

The output lengths for the different autoencoders were different depending on the kind of input. The specific values are described in specific method sections along with the reasons for choosing them. Their length is specified as the length of the flattened matrix.

4.5.3 GRU network with spectrogram input

Training

We trained two GRU spectrogram networks with variable output vector lengths. We decided to base the output vector's length on the PCA's output vectors that explained 90% of the variance ratio. For spectrograms however, we only found out that 1,106 explains 57% of variance. Therefore, we took the information from the mel-spectrogram PCA where 5,715 compnents explain 90% of variance and did a simple calculation:

$$l(mel_spec_{transformed})/l(mel_spec) = l(spec_{transformed})/l(spec)$$

to keep the proportional reduction of spectrograms same as for mel-spectrograms.

This would mean an output vector length of almost 40,000 which is too much for any practical use in the proposed web application. Because of that we reduced it to 20,400 (408 x 50) which at that point we thought could be potentially used. The fist GRU layer cut the number of features to 100 and the second then to 50.

The second model produced shorter vectors as encodings of songs. The first GRU layer cut the number of features to 28 and the second GRU layer cut it to 14. They were of length 5,712 (we wanted them to be a multiple of 408 so they are not 5,715 as the output vectors of PCA_mel) when flattened which means, that the output matrix had the shape (408 x 14). This is inspired by the mel-spectrogram PCA reduction as we thought that the neural network could mimic mel-scaling of the spectrogram as well as additional reduction.

In [29], they trained their autoencoders for 50 epochs using batch sizes of 64. We found this to be insufficient, especially with the learning rate reduction. For GRU networks with spectrograms we set the number of epochs to 100 and the batch size to 295 (bigger batches did not fit into memory).

The training losses of these two methods and all other neural network methods are illustrated in Figure 4.4.

4.5.4 LSTM network with spectrogram input

Training

We chose the same training strategy for LSTMs with spectrogram input as we did for the GRU_spec networks. We created two versions of LSTM models, one is LSMT_spec_20400 and the shorter version is LSTM_spec_5712. The output lengths are also the same as with GRU_specs.

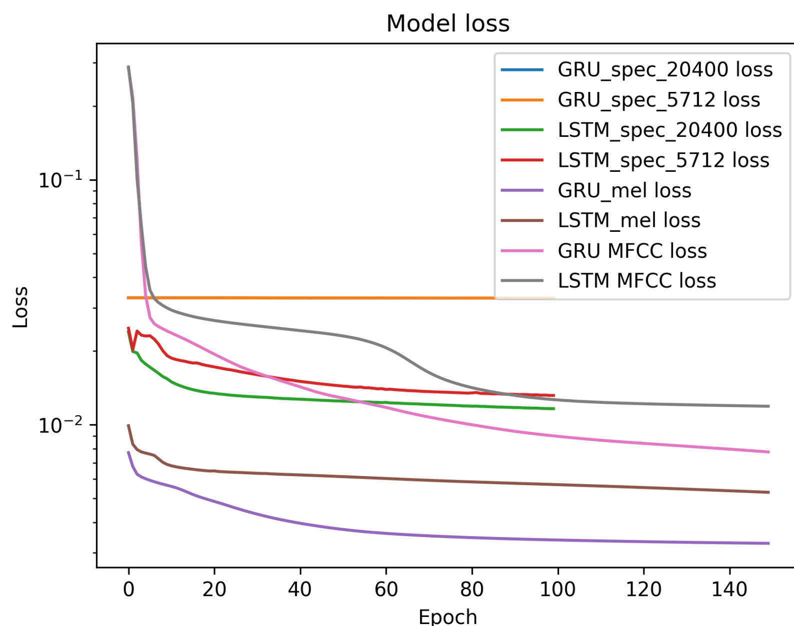

Figure 4.4: The training mean squared error loss values for all the neural network methods that were trained.
The blue curve for the GRU_spec_20400 is hidden behind the orange curve.

4.5.5 GRU and LSTM networks with Mel-spectrogram input

Training

The GRU_mel and LSTM_mel networks were both trained under the same conditions. We trained them on mel-spectrograms for 150 epochs with a batch size of 256. The output length of the encoded song vector was based on keeping 90% of the variance ratio of the PCA_mel which was 5,715. The final output length however was not 5,715 but 5,712 (408 x 14) to be divisible by 408. The first GRU/LSTM layer reduced the number of featrues to 28 and the second layer to 14.

We did not attempt any further reductions here partly because as stated before, we were reducing only the number of features with the sequence-to-sequence RNN, so that the minimum length for the encoded vector had to be 408 (which is the number of timestamps). Partly also because vectors of length 5,712 are of acceptable length to be implemented in the proposed web application.

4.5.6 GRU and LSTM networks with MFCC input

Training

At first we did not plan on using MFCCs as input into neural networks. However because it turned out that raw MFCCs are too long to be used in the proposed application directly we decided to try to reduce their dimension with both the GRU and LSTM network architectures.

We trained both architectures for 150 epochs and a batch size of 256 which is the same number as for neural networks having mel-spectrograms as input.

The output vectors were of length 5,168 when flattened which means the output matrix had the shape of (646 x 8), where the first GRU/LSTM layer cut the number of features to 18 and the second to 8. We were again trying to come close to the number of dimensions that explains 90% of the variance ratio when using PCA which is 5,715. We chose $646 * 8$ which is 5,168 rather than $646 * 9$ which is 5,814 because we favored bigger reduction over proximity to 5,712.

4.6 Similarity metrics

As the reader might have noticed, we presented several possible methods which encode a song into a vector. But we still need to define similarity between these vectors. There are many ways of asserting how similar two vectors are, however, we do not consider studying these to be main focus of this thesis. After the initial exploration of this topic, we decided to use the **cosine similarity metrics** for all the representation methods.

Although throughout the proposed web application the metric is referred to as distance, it in fact is a similarity measure, meaning, the greater value the more similar two songs are. We use the `metrics.pairwise.cosine_similarity` method from Python's `sklearn` package for all the similarity calculations.

4.7 Evaluation

4.7.1 Desired recommender-system features

We already touched this in the introduction but it is useful to revise what we want from a good recommendation system and what its most important features are. Let's skip the software part for now as it is further discussed in Chapter 5 and focus purely on the quality of recommendations. Probably the most crucial property a recommender system should have is that it should include the relevant items between the first 10 to maybe 50 recommendations. Because it does not really matter if an item the user would like ends up on the 500th or 5,000th position. People rarely go that deep.

Another thing we want is for the system to be able to improve recommendations when it has more data about a user. In the context of this thesis it means, that we would expect the predictions to be better for users with longer playlists.

Even though the whole idea of this thesis is to provide variable recommendations, we still want our methods to posses these features to a reasonable extend. Also, since these are the properties other recommendation systems are being evaluated on, we can gain a better understanding of the features our methods share with other recommendation techniques as well as where they differ.

4.7.2 Evaluation measures

To test the wanted features described in the section above we performed evaluation as follows.

The UD dataset contains 11,123 playlists that were used for evaluation. We chose to evaluate the tested methods only on playlists of length at least four.

For each method, we did a 5-cross-validation where in a validation epoch, every playlists p_i was divided into two parts a training part $p_{i_{train}}$ and a testing part $p_{i_{test}}$ and the values of the evaluation measures below were determined. The playlists were split in an approximately 80:20 ratio. A higher priority was set on the fact that the test part always had to contain at least one entry (meaning that for playlists of length 2, the ratio would be 50:50).

The processing of one playlists was performed as follows:

For each song s_k from the song dataset SD, the similarity of the whole $p_{i_{train}}$ to s_k which we denote as $sim(p_i, s_k)$ was calculated as

$$sim(p_i, s_k) = \sum_{s_j \in p_{itrain}} cos_sim(s_k, s_j)$$

where $k \neq j$ and cos_sim is the cosine similarity. These similarities were then sorted in a descending order and it was determined at what position the songs from $p_{i_{test}}$ came as those are the ones that actually belong to the playlist.

These positions were then used to calculate multiple evaluation measures to assess how well each algorithm predicts the missing part of a user's playlist. We chose the following evaluation measures:

- Recall at 10 (= **R@10**) defined as the proportion of songs from $p_{i_{test}}$ that placed between the top ten most similar songs.

- Recall at 50 (= **R@50**) defined as the proportion of songs from $p_{i_{test}}$ that placed between the top fifty most similar songs.

- Recall at 100 (= **R@100**) defined as the proportion of songs from $p_{i_{test}}$ that placed between the top hundred most similar songs.

- Normalized cumulative discounted gain (= **nCDG**) defined as

  $$nDCG_r = \frac{DCG_r}{IDCGr}$$

  where

  $$DCG_r = \sum_{k=1}^{r} \frac{rel_k}{\log_2(k+1)}$$

  is the discounted cumulative gain at position r and

  $$IDCG_r = \sum_{k=1}^{|REL|} \frac{2^{rel_k} - 1}{\log_2(k+1)}$$

  is the ideal discounted cumulative gain at r where r is the number of songs that had to be predicted, the rel_k, meaning relevance, is the same for all songs because all songs in one playlists have the same relevance and $|REL|$ is the length of the list of relevant items (in this case the songs from $p_{i_{test}}$).

- Average rank of a song from the $p_{i_{test}}$ set $\mathbf{(= \overline{\textit{rank}})}$ which we included to have also an indicator of the overall behaviour, not only the first 100 ranks.

The overall evaluation results for a tested method were then taken as its average evaluation-measure values over all the playlists over the 5 cross-validations. In addition we also retrieved the evaluation values for only certain playlist's lengths again by averaging over the 5 cross-validations but selecting only values of playlists of desired lengths.

Moreover, we created one more evaluation technique whose purpose was to visualize the results of a method. It is a graph plotting the distribution of rankings of songs from the test part of each playlists. It is denoted as the **RDG** (=*rank distribution graph*). The number of songs from p_{test} (which is a union of all songs from all $p_{i_{test}}$) for each individual rank from 1 to 16,594 is summed and divided it by the number of all songs in p_{test}. So for example if we had two playlists both with two songs in p_{test} and a method assigned ranks 30 and 2,900 to the p_{test} songs from the first playlist and 2,900 and 4,872 to those from the second playlists, the graph would plot the values 0.25 for rank 30, 0.5 for rank 2,900 and 0.25 for rank 4,872 and 0 for the rest of the ranks. We did this with all playlist lengths included but we also plotted these distributions for chosen playlist lengths separately to see if the predictions improve for longer playlists which we expect from a good recommender system. The x-axis of the *RDG* was log-scaled as we are much more interested in what is going on among the first 100 positions than in what is going on in the middle or towards the end.

After running the evaluation, we noticed from the RDGs that for most of the similarity methods, the positions for songs from $p_{i_{test}}$ where p_i was a short playlist were higher up, than those from long playlists, especially for the first three ranks. We concluded that the reason for such behaviour could be, that inside all playlists, there are groups of very similar songs but the groups are rather dissimilar to each other. Songs which are somewhat similar to all groups then cloud the recommendations and take place of songs that are very similar to one group but dissimilar to the other groups.

Because of this, we changed the recommendation method a little bit. We set a threshold for each similarity metrics, and if the similarity between two songs was smaller than this threshold, we set the similarity to 0. This means, that the new similarity function $cos_sim_t(s_i, s_j)$ is defined as follows:

$$cos_sim_t(s_i, s_j) = \begin{cases} cos_sim(s_i, s_j) & \text{if } cos_sim(s_i, s_j) \geq \text{threshold} \\ 0 & \text{if } cos_sim(s_i, s_j) < threshold \end{cases}$$

The threshold for a method was chosen as the value of the 846,294th biggest element from its D_m. 846,294 is $51 * |SD|$. The most similar song is always the song itself, so it leaves us with approximately 50 most similar songs for each song. 50 is approximately 0.03% of 16,594 so we also refer to the threshold as the 0.03%-threshold throughout this thesis.

The results presented in Sections 4.8, 4.9 and 4.10 are all acquired by evaluation of recommendations using the similarity with threshold. If the evaluation of similarity without threshold had better results for a method, it is mentioned in the method's section.

Another reason to use the threshold-similarity is the fact, that we use the threshold to calculate similarity in the proposed application. Not only because of the better results as we shall see, but also because of the fact, that it dramatically

reduces the number of similarities that have to be stored in the database. The 0.03%-threshold values for various methods are in Table 4.1.

Tf-idf	W2V	PCA_Tf-idf	SOM_W2V
0.2817861171	0.956714893	0.189907224	0.999997393
PCA_mel_320	PCA_mel_5715	PCA_spec_1106	PCA_spec_320
0.383122074	0.189912771	0.3368717729	0.442181335
GRU_spec_20400	GRU_spec_5712	LSTM_spec_20400	LSTM_spec_5712
0.999742671	0.99999024	0.975920383	0.981116184
GRU_mel	LSTM_mel	GRU_MFCC	LSTM_MFCC
0.3634592744	0.994544642	0.953069695	0.997860599

Table 4.1: Table containing the value of the similarity threshold we used. The threshold for a particular method is always below the method's name.

To give an idea about how the results changed with the threshold, lets take a look at the plot in Figure 4.5. We plotted the change in the maximum, average and minimum values of our evaluation measures for all playlists and also for different playlist lengths with and without using the threshold for similarity definition. Short playlists in this graph are defined as playlists of lengths 4 to 7 and their evaluation values have an ”_S” appended at the end. Medium playlists are of lengths 8 to 15 and have ”_M” appended. Long playlists are playlists 16 and longer with and ”_L” appended. Although, the minimum values did not improve much, the difference for the maximum and most notably the average values is appreciable. The most crucial remark here is the behaviour of short and long playlists. The results for short playlists did not improve much, especially the maximum values, whereas the results for long playlists improved quite a lot. This is exactly what we anticipated when applying the threshold and also better performance for longer playlists is the desired behaviour for recommender systems.

Even with the threshold, we can observe in the *RDG* graphs later on in specific method result sections, that for short playlists, the first one to four ranks are very numerous. For the following ranks however, there is a sharp drop, which does not happen so noticeably with longer playlists. This behaviour inspired us to use the threshold and even though it is less significant after applying it, it is still observable in almost every method.

4.8 Text method results

As mentioned in Section 4.7 we calculated each of the five measures (*R@10, R@50, R@100,* $\overline{rank}$ and *nGDC* for each playlist and then averaged the values over the whole playlist dataset. Every method section (not only for text methods but also for simple and deep audio methods) contains a table with the five measure averages and a RDG graph. Both are accompanied with a short summary of the most prominent observations.

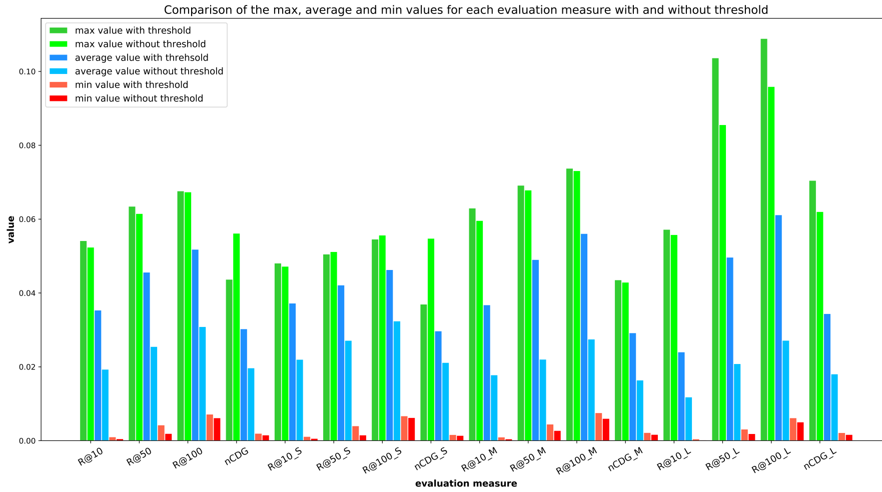

Figure 4.5: The comparison of absolute max, average and min values for the evaluation measures for recommendation with and without threshold

4.8.1 Tf-idf results

The results of the Tf-idf method place well above average between our methods. This was a bit of a surprise as we did not expect this ”simple” method to perform so well compared to others.

method	**R@10**	**R@50**	**R@100**	**nGDC**	$\overline{rank}$
Tf-idf	0.05045	0.06187	0.06648	0.04214	7795
PCA on Tf-idf	0.05417	0.0635	0.06649	0.04371	7838

Table 4.2: Table summarizing average Tf-idf and Tf-idf with PCA evaluation measure values averaged over the 5 cross validation that were performed.

When looking at the numbers in Table 4.2 we can see that 5% of songs that were in our p_{test} set ranked in the top ten, 6.2% in the top fifty and 6.6% in the top hundred. The average rank of a song from the p_{test} was 7795 which is quite close to the middle. We also created an *RDG* as one can see in Figure 4.6. It appears that according to the distribution, a song is more likely to end up in the first 10-100 songs than it is at the end.

Another thing to notice in 4.6 is that there is a general trend not only for the Tf-idf to rank a lot of songs between the first few but then drop sharply for further ranks in short playlists. Longer playlists seem to drop more steadily but do not start so well.

4.8.2 PCA_Tf-idf results

The results of the PCA-reduced Tf-idf vectors turned out to be better then full Tf-idf vectors. As we can see in Table 4.2 the numbers are better for the *R10* and *R50*. The *R@100* the values are almost the same. Figure 4.27 illustrates the

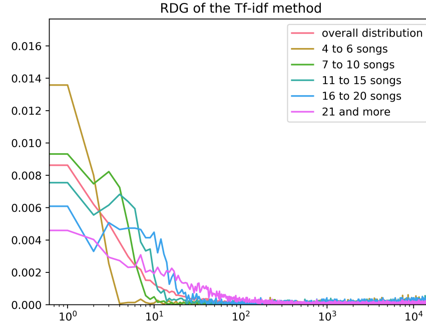

Figure 4.6: RDG of the TF-idf method

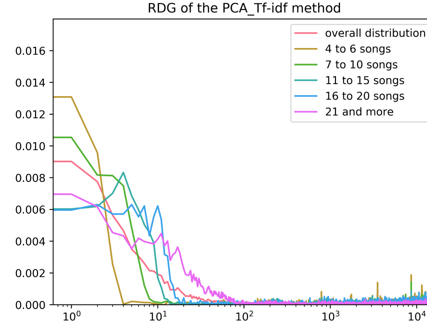

Figure 4.7: RDG of the PCA_Tf-idf method

fact, that after the application of the threshold, this method became the best one for long playlists and also the best method overall overtaking the PCA_mel_5715 which was the best method when defining similarity without threshold as can be seen in Figure 4.26.

4.8.3 W2V results

Results

The small size of the vectors being produced by the W2V method are a significant advantage of this method. However the evaluation-measure values it yielded make it average to below average compared to methods. It ought to be mentioned that methods producing vectors of similar length such as PCA_mel_320 outperform the W2V.

method	**R@10**	**R@50**	**R@100**	**nGDC**	$\overline{rank}$
W2V	0.03519	0.04780	0.05544	0.030313	7804

Table 4.3: Table summarizing average W2V evaluation values averaged over the 5 cross validation that were performed

Table 4.3 shows lower numbers than for the Tf-idf method. 3.5% of songs from the p_{test} set ranked in the top 10, 4.8% in top 50 and 5.5% in top 100. The average rank was 7804. When looking at the distribution graph in Figure 4.8, it is very clear that the gap between the number of predicted ranks within the first 5 ranks for short and longer playlists is big and Figures 4.26 and 4.27 show us, that the threshold helped this method especially for ranking of songs from longer playlists, where it even outmatched the PCA_mel_320.

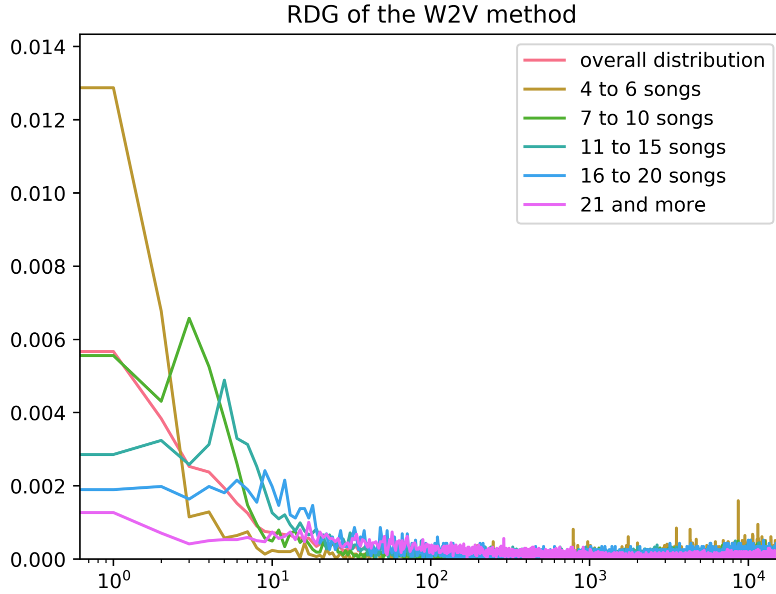

Figure 4.8: Distribution of ranks of songs from the test set the w2v method assigned them.

4.8.4 SOM results

Results

When we then tried to display resulting SOM map with all the songs, the size of the image would have to be immense for all 16,594 songs to be recognizable. The problem was that there are many songs to display on the map and the titles and artists overlap. Because of that, we decided to randomly select 20 playlists and show where the different songs that belong to each playlist are placed on the map. Each playlist has its own color. The playlist map for the SOM with W2V input is depicted in Figure 4.9. The playlists do not really form any visible clusters which suggests that songs that should be similar because they are in the same playlist are not close to each other in the space created by the SOM_W2V.

This observation supports the results of the self organizing map algorithm which are quite poor. Actually, it is the worse method that we implemented and our hope to enhance the results of W2V were not fulfilled. The threshold did not make it better either. The results improved but it still stayed at the bottom of the method rankings as the dark red color in Figure 4.27 suggests.

method	R@10	R@50	R@100	nGDC	$\overline{rank}$
SOM with W2V	0.00103	0.00427	0.00720	0.00200	8034
SOM wiht PCA_Tf-idf	0.00044	0.00208	0.00462	0.00125	8243

Table 4.4: Table summarizing average SOM evaluation values averaged over the 5 cross validations

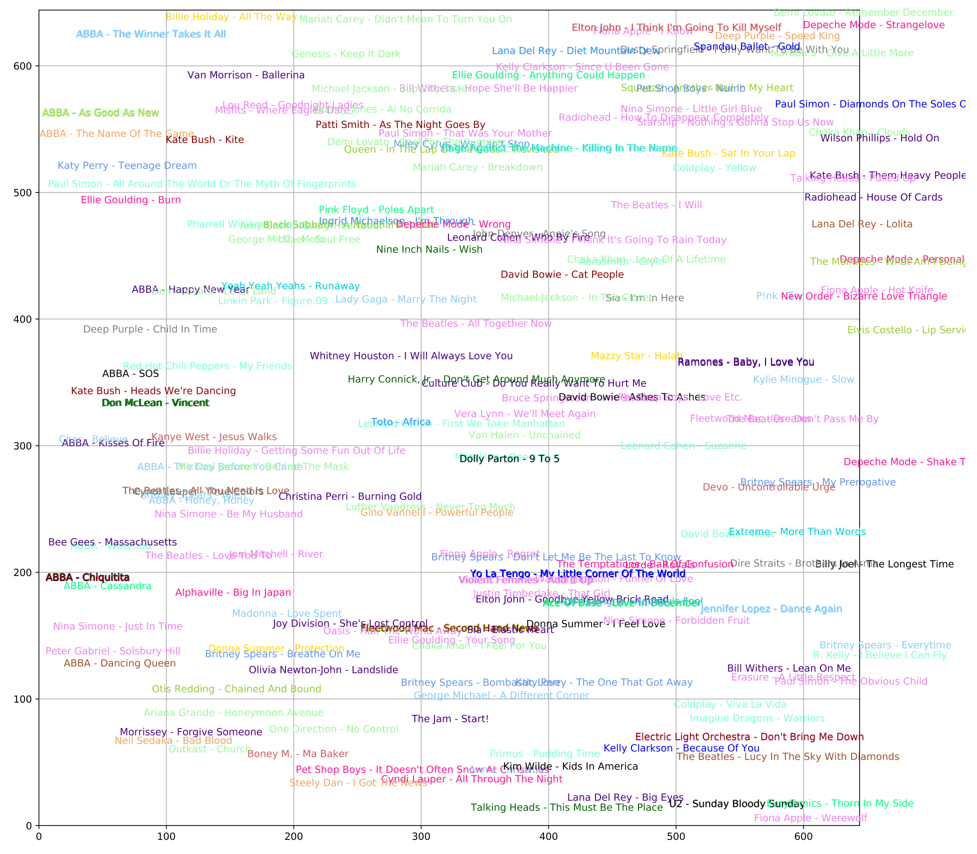

Figure 4.9: The location of different songs from 20 randomly selected playlists on the map created by SOM. Each playlist has its own colour.

The RDG of the SOM_W2V depicted in Figure 4.10 method clearly shows, that the distribution of ranks is random or worse. The main reason for the failure of this method is unclear but it is possible that the data from the W2V vectors compress the information so much, that the SOM network is not able to cluster data based on it. But since we received even worse results for the SOM_Tf-id as can be seen in Figure 4.11 and Table 4.4 we are more inclined to another possibility which is, that the SOM network is not able to provide satisfying results because two dimensions are simply too little to represent input data of such complexity.

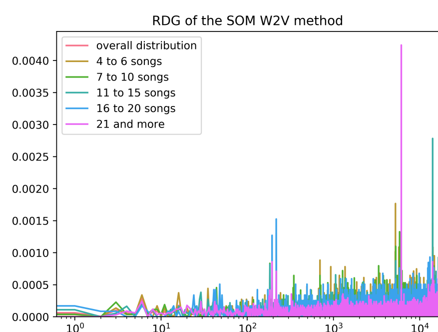

Figure 4.10: RDG of the SOM_W2V method

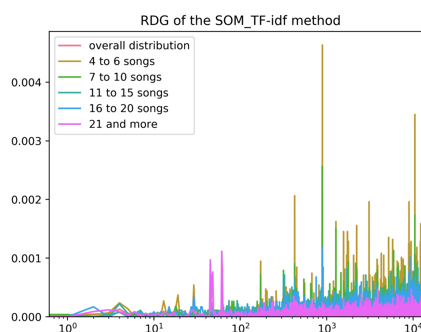

Figure 4.11: RDG of the SOM_Tf-idf method

4.9 Simple audio representation results

4.9.1 Raw mel-spectrogram results

As can be seen in Table 4.5 with the raw mel-spectrogram method, 3.7% of songs ended up between top ten predictions, 4.3% between the top 50 and 4.7% in between the top 100. This method was actually better than any other method using mel-spectrograms as input was before applying the threshold, except of the two PCA_mel methods.

Results

method	R@10	R@50	R@100	nGDC	$\overline{rank}$
Raw mel-spectrograms	0.03696	0.04275	0.0473	0.03063	7604
PCA_mel_5715	0.05287	0.06298	0.06765	0.04317	7803
PCA_mel_320	0.04716	0.05928	0.06550	0.03989	8357
GRU_mel	0.04628	0.05715	0.06285	0.03856	7601
LSTM_mel	0.03197	0.04291	0.04978	0.02750	7776

Table 4.5: Table summarizing average evaluation values for all methods with mel-spectrogram input averaged over the 5 cross validations.

The patterns observed in Figure 4.12 display the same tendencies as all other methods we are studying (except of those that appear to be random such as SOM). The short playlists go through a sharp drop at the beginning. Longer playlists are worse in the beginning as the top ranks for them are a bit less numerous, especially for playlists of length 21 and more but they drop more steadily.

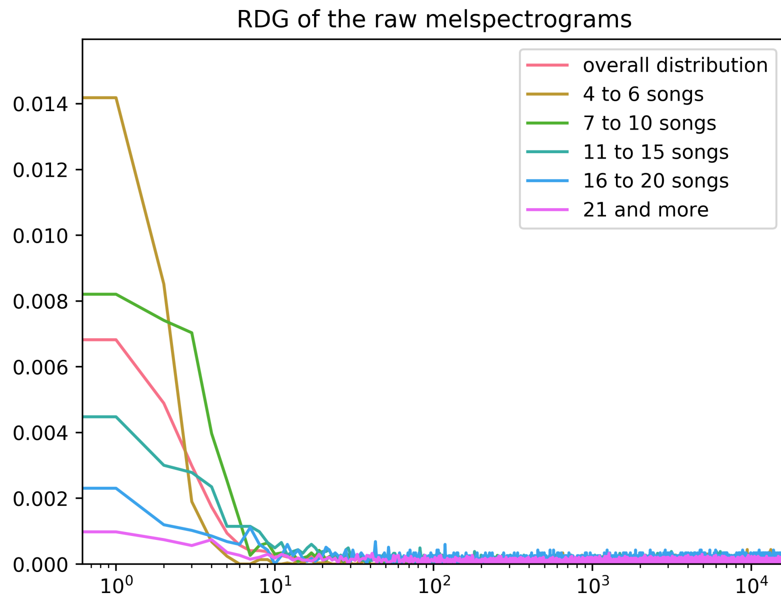

Figure 4.12: RDG of raw Mel-spectrograms.

4.9.2 Raw MFCCs results

The results of raw MFCCs are quite poor compared to other methods. Only 0.4% of songs rank between the top 10, 0.9% in the top 50 and 1.4% in the top 100 as we can see in Table 4.6. It was better than the LSTM_MFCC method before we applied the threshold to our evaluation but not very good overall.

method	R@10	R@50	R@100	nGDC	$\overline{rank}$
Raw MFCC	0.00415	0.00919	0.01423	0.00607	7552
LSTM_MFCC	0.03887	0.04935	0.05670	0.033058	7774
GRU_MFCC	0.03769	0.04737	0.05438	0.032151	7774

Table 4.6: Table summarizing average evaluation values for all methods with MFCC input averaged over 5 cross validations with threshold.

When looking at the traditional RDG, we again observe a drop for short playlists, not as sharp as for other methods though. Nevertheless, the rankings for longer playlists, do not seem to be very stable and this method does not behave as a good recommendation method should.

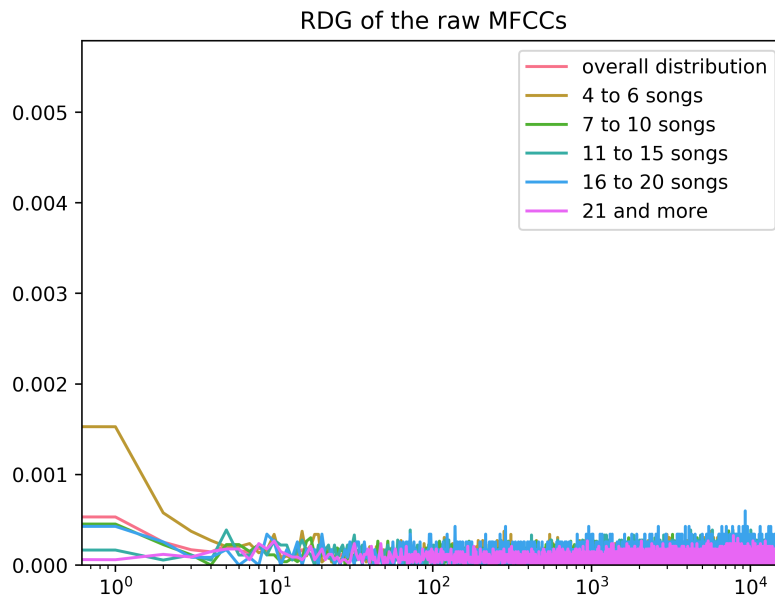

Figure 4.13: RDG of raw MFCCs.

4.9.3 PCA on spectrograms

The dimensionality reduction in this method was the biggest of all methods. Our song representation went from 900,048 to 1,106 respectively 320 dimensions. Even with only 57% of explained variance for the PCA_spec_1106, this method's results were above average compared to other methods. So it seemed reasonable to go for an even more radical reduction to vectors of length 320. The results of the method encoding songs into shorter vectors were only slightly worse than those of the one yielding longer vectors with similarity defined without threshold.

One interesting thing is, that the PCA_spec_320 method is the only one whose results worsened with the application of the threshold and the gap between the performance of these two methods widened. It can be observed in Figure 4.26 where method ranking without the use of similarity with threshold is depicted

and both methods are green, and Figure 4.27 where methods are ranked with using the threshold in the similarity definition and the PCA_spec_320 is in the orange area and the PCA_spec_1106 stays green.

method	**R@10**	**R@50**	**R@100**	**nGDC**	$\overline{rank}$
PCA_spec_1106	0.04747	0.05915	0.06472	0.03982	7797
PCA_spec_320	0.03166	0.04289	0.05094	0.02890	7496
GRU_spec_20400	0.04287	0.05196	0.05723	0.03563	7824
GRU_spec_5712	0.00248	0.00628	0.01076	0.003757	7761
LSTM_spec_20400	0.03641	0.05096	0.05921	0.03126	7786
LSTM_spec_5712	0.01892	0.03263	0.04252	0.01899	7643

Table 4.7: Table summarizing average evaluation values for methods with spectrogram input averaged over 5 cross validations

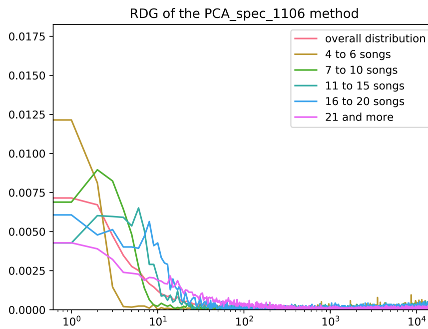

Figure 4.14: RDG of the PCA_spec_1106 method.

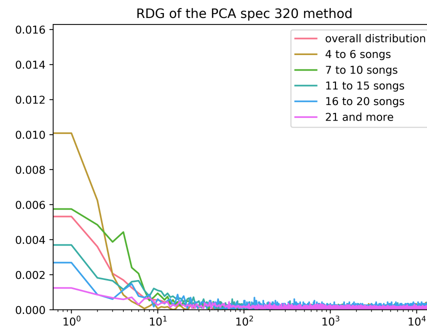

Figure 4.15: RDG of the PCA_spec_320 method

4.9.4 PCA on Mel-spectrograms

We had two PCA methods taking mel-spectrograms as input. The more radical dimension reduction did not lead to better performance. It actually worsened the results significantly and the PCA_mel_320 was our worse method. But only until we applied the threshold. For this method, the threshold improved the results over ten times. It is interesting because it was the PCA_spec_320 for which the results worsened after applying the threshold and one might think, that PCA methods with audio inputs will behave similarly.

Figure 4.16 and Figure 4.17 illustrate the distributions for PCA_mel_5715 and PCA_mel_320. A thing to notice in the PCA_mel_5715 RDG graph is that the number of the top 1-5 rankings for longer playlists is not quite as low compared to the other methods we tested. We still observe the trend of a considerably higher

number of the top 1-5 ranks for shorter playlists, however, it is not as significant as for example for raw mel-spectrograms.

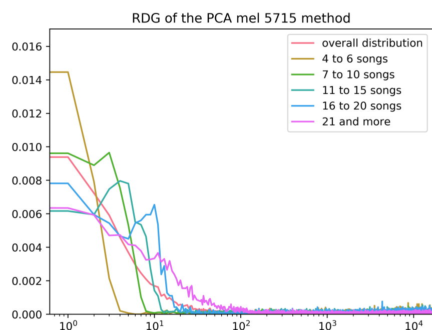

Figure 4.16: RDG of the PCA_mel_5712.

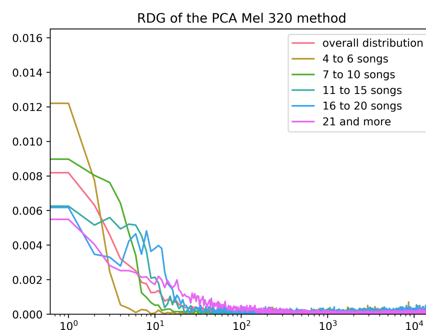

Figure 4.17: RDG of the PCA_mel_320 method.

4.10 Deep audio representation results

4.10.1 GRU network with spectrogram input

As can be observed in Figure 4.4 the training loss is basically constant for both GRU networks with spectrogram input (the GRU_spec_20400 is hidden behind the orange line of GRU_spec_5712 as their progress or rather the lack of it is the same). This means, that the network did not really learn. Therefore it is not suprising that the GRU_spec_5712 does not rank among the best. And it is quite surprising, that the GRU_spec_20400, even though it appeared to be quite bad before the application of the threshold, improved with defining similarity with the threshold and became the second best method with spectrogram input (right behind the PCA_spec_1106) as can be seen in Table 4.7. The table shows that 4.3% of songs were ranked in the top 10, 5.2% in the top 50 and 5.7% in the top 100 for the longer GRU spectrogram model. For the short GRU spectrogram model, the results are significantly worse with 0.2% for the top ten songs 0.6% ranked in the top 50 and a little over 1% ranked in the top 100.

The RDGs depicted in Figures 4.18 and 4.19 of both GRU_specs is are dissimilar too. The overall trend for the short playlist drop and stability of long playlists is visible in the graph for GRU_spec_20400 but not so much in the graph for GRU_spec_5712 where there is a trace of it but the values seem to be quite random so it fades.

4.10.2 LSTM network with spectrogram input

Unlike the GRU model which showed almost no improvement even after 100 epochs of training, the LSTM_specs as one can observe in Figure 4.4, where the LSTM_spec_20400 is rendered in green and the LSTM_spec_5712 in red, went

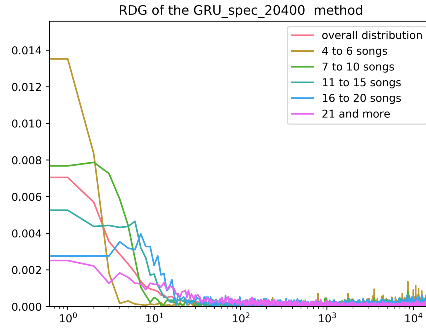

Figure 4.18: RDG of the GRU_spec_20400 method.

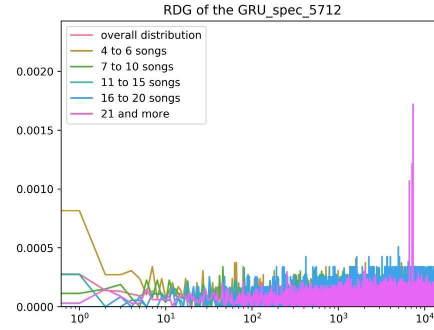

Figure 4.19: RDG of the GRU_spec_5712 method.

through progress. The decrease of training loss slowed down significantly towards the end of the 100 epochs.

The bigger improvement of the training loss correlated with better results for the LSTM_spec methods. But only until we applied the threshold. It raised performance of all methods but the boost for GRU_spec_20400 was so big that it vaulted over both LSTM methods. In Table 4.7 are the results of both LSTM_spec models.

A thing worth noting is that the LSTM_spec method with bigger dimension reduction had worse results even before applying the threshold. And it stayed that way after the threshold similarity as we can see in Table 4.7 and in Figures 4.20 and 4.21 which visualize the difference between the two LSTM_spec methods.

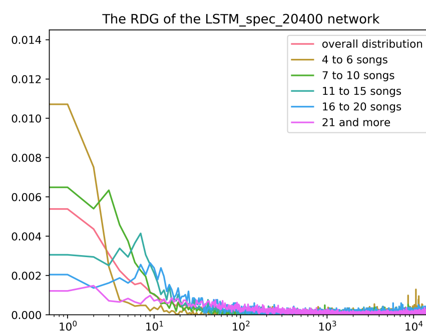

Figure 4.20: RDG of the LSTM_spec_20400 method.

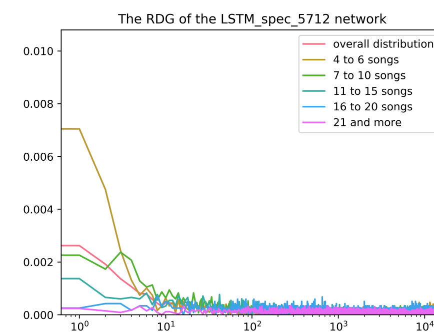

Figure 4.21: RDG of the LSTM_spec_5712 method

4.10.3 GRU and LSTM networks with Mel-spectrogram input

Methods with mel-spectrograms as input seem to yield better results than methods with spectrogram inputs and GRU_mel network confirms it as it has the

best results within the neural network method group. The GRU_mel network performed better than the one with LSTM layers and also showed the smallest training loss. Figures 4.22 and 4.23 display typical tendencies most of the methods have, with the sharp drop for short playlists and more stability for longer playlists. This is more apparent with the GRU_mel method. For the LSTM_mel, longer playlists drop unconventionally early.

Table 4.5 puts recalls and nDGC of "mel" neural networks into perspective with other "mel" methods. As we can see, the GRU_mel network placed 4.6% of songs into the top 10, 5.7% into the top 50 and 6.3% into the top 100. The LSTM_mel network is worse. It puts 3.2% of songs in the top 10. For *R@50* and *R@100* it outperforms raw mel-spectrograms with 4.3% of songs in the top 50 and 5% of songs with assigned ranks in the top 100.

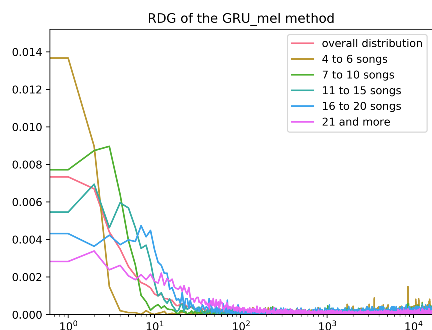

Figure 4.22: RDG of the GRU_mel method

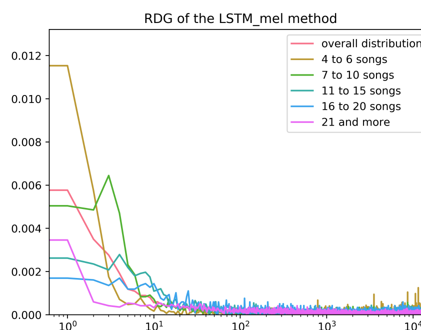

Figure 4.23: RDG of the LSTM_mel method

4.10.4 GRU and LSTM networks with MFCC input

The MFCC networks seem to have the biggest potential of improving if they were to be trained for a longer period of time as their losses in Figure 4.4 do not stagnate as much towards the end of training as the other networks. This is especially true for the GRU_MFCC network. They would, however, probably never achieve a loss as small as the "mel" networks have.

Compared to the raw MFCCs the GRU_MFCC and LSTM_MFCC were an improvement as the values in Table 4.6 suggest. The Figures 4.24 and 4.25 containing both method's *RDGs* indicate the superiority of the LSTM_MFCC method over the GRU_MFCC. Both methods made a huge jump upwards with the threshold similarity. Without it, they were at the bottom of the tested methods as the orange color suggests in Figure 4.26, but the threshold helped them to perform around average as they are in the green-yellow zone in Figure 4.27.

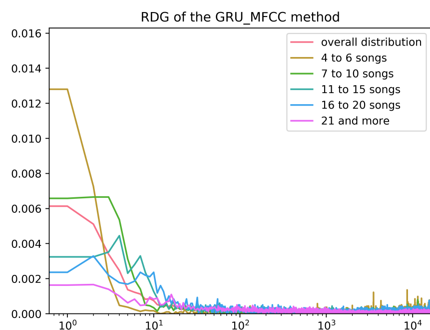

Figure 4.24: RDG of the GRU_MFCC method

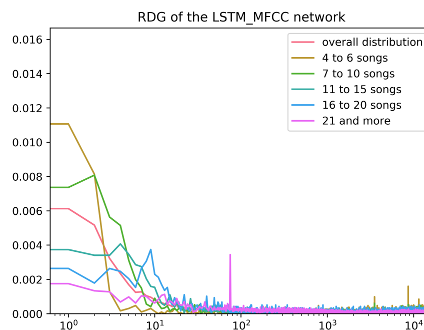

Figure 4.25: RDG of the LSMT_MFCC method

4.11 Discussion

In this section, we first state, in Subsection 4.11.1, if the results match the expectations from Section 4.1. Then to summarize the results and put all the methods into perspective we did two things. First in Subsection 4.11.2 we compared the methods relatively to each other. Secondly in Subsection 4.11.3 we focused on the values of the evaluation measures compared to values of random recommendations and we also considered what the absolute numbers of the evaluation measures suggest. This was to estimate the recommendation qualities of the tested methods. In subsection 4.11.4 we introduce multiple graphs of performance depending on properties which we found during the evaluation could correlate with it.

4.11.1 Expectations vs. reality

Let's recapitulate what we expected. First we expected, that audio-based methods will perform better than lyrics-based methods. Because the best method – PCA_Tf-idf – is a lyrics based method, we cannot say that this prediction was confirmed. Raw Tf-idf also performed very well compared to others and outperformed most audio-based methods. The W2V was a little behind and the SOM network was a failure, however we definitely cannot declare, that when a method is audio-based it means it will have better results than a lyrics-based method.

There are various theories that could explain this. It could be connected to the user dataset we did the evaluations on. Therefore, we tested the playlists on diversity. To calculate diversity of a playlist, we divided the number of unique artists it contained with its length. When we averaged over all the playlists of length at least four, we got 0.83 which means, that an artist rarely repeats in a playlist.

This variety could have suited the Tf-idf and PCA_Tf-idf methods as the presumed diversity of their recommendations poses an advantage for such heterogeneous playlists. To test this hypothesis, we took 1000 random songs and found 100 most similar songs for each of the songs using each of the tested methods. We

than computed the average diversity of the 101 song playlists for each method as we did for the playlists in the dataset. The results are in Table 4.8.

method	**playlist diversity**
PCA_mel_5715	0.739
PCA_mel_320	0.746
PCA_spec_1106	0.700
PCA_spec_320	0.702
LSTM_mel	0.749
GRU_mel	0.725
LSTM_spec_20400	0.721
GRU_spec_20400	0.752
LSTM_spec_5712	0.729
GRU_spec_5712	0.751
LSTM_mfcc	0.794
GRU_mfcc	0.776
TF-idf	0.800
SOM_W2V	0.861
PCA_tf_idf	0.812
W2V	0.71186

Table 4.8: Table containing the value of the diversity index that was also calculated for the UD we have.

As one can see, the most diverse method is SOM_W2V which is not very surprising as it yields basically random results. Nonetheless, the best lyrics methods, PCA_Tf-idf and Tf-idf are those with the second and third biggest diversity which confirms, that is probably where lays their advantage.

Another reason for the evenness of the audio and lyrics-based methods might be, that 15 seconds from each song's audio was not enough and if we made the excerpts longer, the results for audio-based methods would have been better.

We also expected that more advanced methods will outperform simpler methods. This did not happen at all. The Tf-idf which we though of as a simple baseline outperformed almost all of the other methods. Overall, PCA methods were the most successful ones (with the exception of PCA_spec_320) and left the neural networks behind.

This could be caused by the fact that we did not tune our neural networks. There is not much to improve about the PCA but a lot of parameter optimisation and also architecture re-creation can be done on all of the neural networks we implemented. Moreover, the training conditions can be also altered. The batch size as well as the number of epochs can be lowered or increased.

4.11.2 Relative results

We created two heat maps to compare the methods mutually. One is depicted in Figure 4.26 for results without threshold and one in Figure 4.27 for results with threshold. We ranked the methods on how well they performed in each evaluation measure and we took playlist lengths into account.

We can see, that the PCA_Tf-idf was the most successful method winning in 8 out of the 20 measures. It performed best on long playlists. The average rank, where its performance is in the red area, is not that important for recommendation. We included it to have some understanding of what is going on on throughout the ranks, not only at the first 100 places which are, however, most important for recommendation.

Second comes PCA_mel_5715, then Tf-idf with PCA_spec_1106 right behind it. Then we have the GRU_mel method which is orange for short playlists, nevertheless, otherwise in the green area and it is the best neural network method, lyrics or audio-based.

Overall we can see, that the PCA methods were very successful compared to other methods. Except of the PCA_spec_320, they are all green and the only method that comes between them on the top places 4 is the Tf-idf.

Within text-based methods, the PCA_Tf-idf and Tf-idf show strong dominance, the W2V is below average and the SOM has the worst results of all methods. For audio-based methods, the winner is PCA_mel_5715.

4.11.3 Result interpretation

Because we did not perform classification on a standardised set and no one has used the datasets we are using to evaluate their algorithms, it is a little challenging to find something outside of this thesis what we could compare the results to.

We decided to compare our methods to random suggestions. If songs were assigned ranks from 1 to 16,594 in a uniform distribution, meaning each rank had the probability of 1/16594 to be assigned to a song the values of the recall evaluation measures would be following:

- R@10 $= 10 * (1/16594) = 0.06\%$ of songs
- R@50 $= 50 * (1/16594) = 0.3\%$ of songs
- R@100 $= 100 * (1/16594) = 0.6\%$ of songs

Knowing this, we created Figure 4.28 showing the multiples of how many times a method is better than random recommendation would be. We divided it into categories based on playlist lengths. Measures with _L" appended contain results of long playlists (length 16 and more), methods with "_M" are the results for medium playlists (length 8-15) and methods with "_S" are evaluation measures of short playlists (length 4-7). When there is nothing appended to the evaluation measure name, it is the value for all playlists. We can see, that there is major improvement especially for values of R@10 where the best method, the PCA_Tf-idf is 90.3 times better than random recommendation. This means, that when random recommendation places 0.06% of songs into the top 10, the PCA_Tf-idf places $90.3 * 0.06\%$ of songs into the top 10.

If we disregard the SOM_W2V and GRU_spec_5712 methods which appear to be basically random, we can see, that the rest of the methods are significantly better than random suggestions. Especially for the first ten ranks, it seems that with the method we implemented, we can provide relevant recommendations.

However, if we look at it from another perspective and realize that if using the best method — PCA_Tf-idf we placed 6.6% of songs into the top 100, and 93.4% percent of songs somewhere further, we probably want a recommendation system to perform better. Nevertheless, some of the methods have shown great potential of providing good recommendations and with for example more parameter optimisation for the neural networks they could improve even more. An incorporation into a collaborative-filtering recommender system, could also be a good idea.

4.11.4 Additional findings

Additionally we also created some interesting graphs. We tried to find a correlation between the training loss of neural networks and their recommendation performance. The training curves are displayed in Figure 4.4.

Figure 4.29 visualizes the dependency of the values of the evaluation measure of neural networks (y-axis) on the training loss the neural networks had when they finished training (x-axis). We can see that, except of the four dots in the upper right corner that belong to the GRU_spec_20400 method, there is some correlation that suggests that a smaller training loss indicates better recommendations. This would be logical as an that is able to more accurately recreate its input can learn the features better and so provide better recommendations.

The 0.03%-threshold inspired us to look at the relative similarities between songs for each method because the value of the threshold varied a lot for different methods. We again created a correlation graph between the performance of methods and the value of the 0.03%threshold. It is depicted in Figure 4.30. We can see, that for methods with threshold values smaller than 0.5 there is a visible trend of better performance being dependant on smaller threshold value. Then there is a big gap as no method had its threshold value between 0.5 and 0.95. For the methods with threshold close to 1 correlation is not present. Some are very low which would follow the trend of thresholds between 0 and 0.5 but there is a significant number of methods that have good results and do not follow the decrease in performance.

The reason for the correlation for methods with smaller threshold values could be, that they are better at distinguishing songs. They place them more variably in the feature space they create, whereas methods with bigger threshold values stuff the songs nearby each other which results in the unclarity of which songs are similar to which as all are very close.

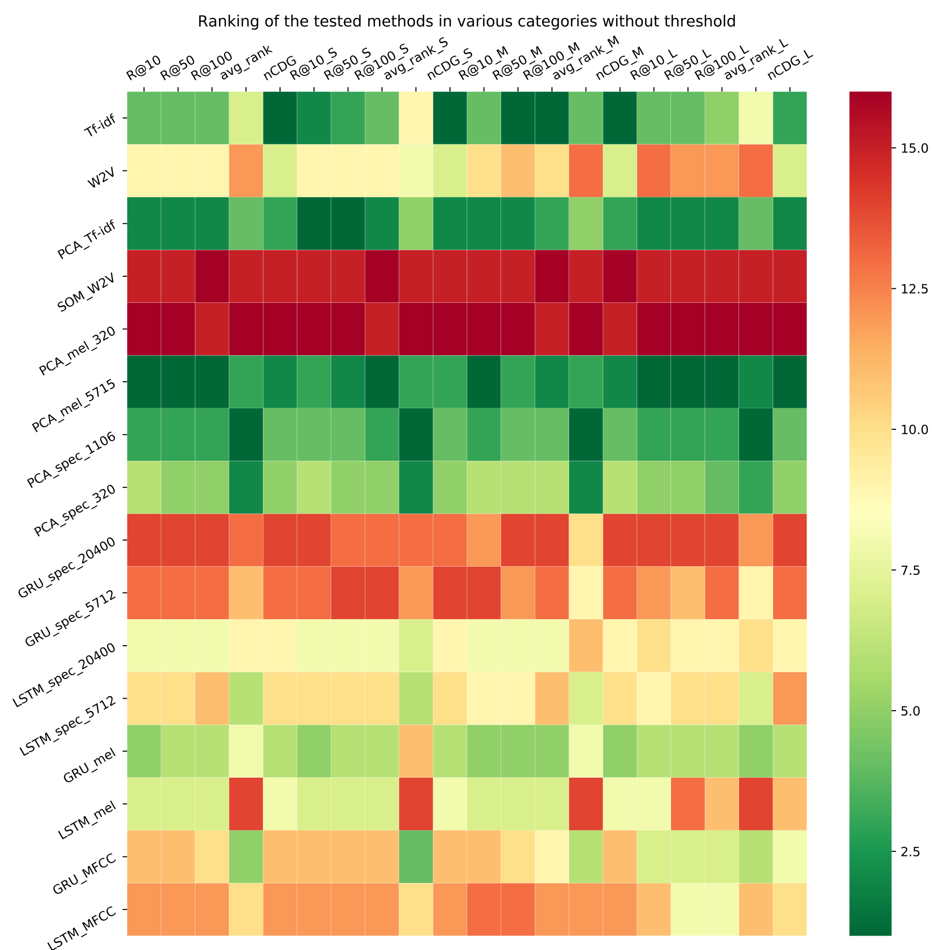

Figure 4.26: Method performance comparison heat map without threshold applied.
The relative performance of methods based on how they performed for various evaluation measures when similarity was defined without threshold. Each method was assigned a rank from 1 to 16 (16 is the number of the methods in this figure) for each measure based on how it performed for the particular measure compared to the other methods. The more darker green the better was the performance, yellow are the average ones and red are the for the worst performances. The evaluation measures are displayed on the x-axis. Measures with "_L" appended contain results of long playlists (length 16 and longer), measures with "_M" are the results for medium playlists (length 8-15) and measures with "_S" are evaluation measures of short playlists (length 4-7). When there is nothing appended to the evaluation measure name, it is the value for all playlists.

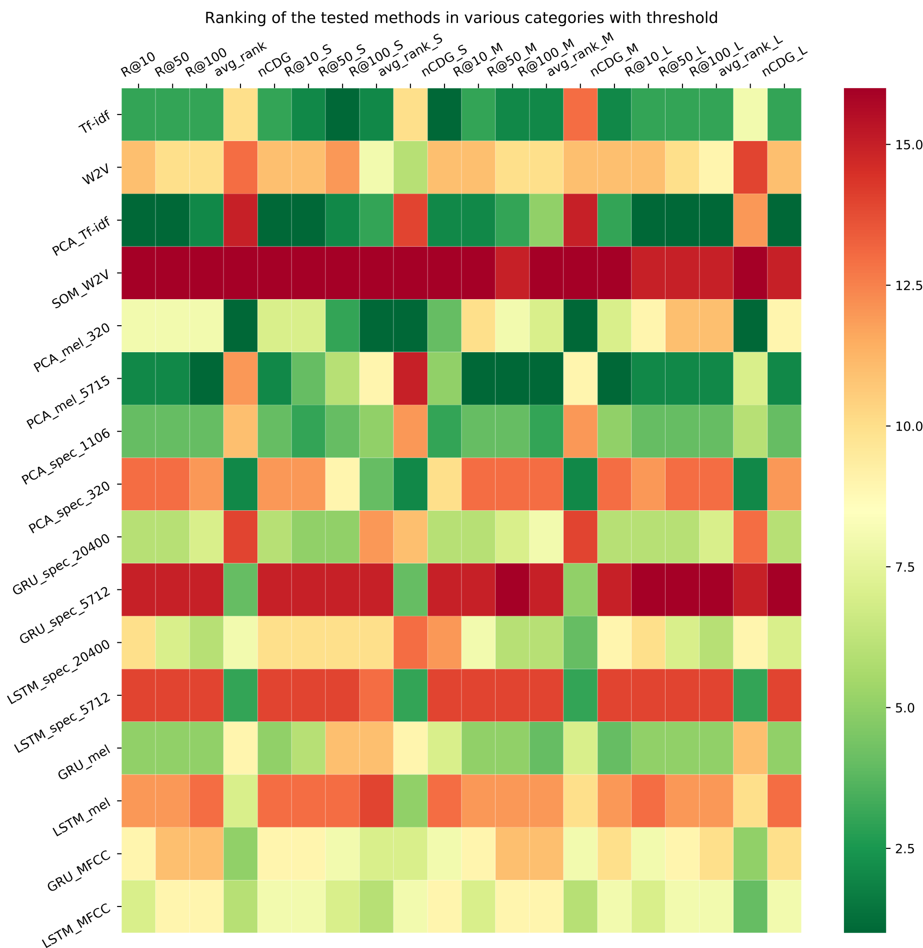

Figure 4.27: Method performance comparison heat map with threshold.
The relative performance of methods based on how they performed for various evaluation measures when similarity was defined with threshold. Each method was assigned a rank from 1 to 16 (16 is the number of the methods in this figure) for each measure based on how it performed for the particular measure compared to the other methods. The more darker green the better was the performance, yellow are the average ones and red are the for the worst performances. The evaluation measures are displayed on the x-axis. Measures with "_L" appended contain results of long playlists (length 16 and more), measures with "_M" are the results for medium playlists (length 8-15) and measures with "_S" are evaluation measures of short playlists (length 4-7). When there is nothing appended to the evaluation measure name, it is the value for all playlists.

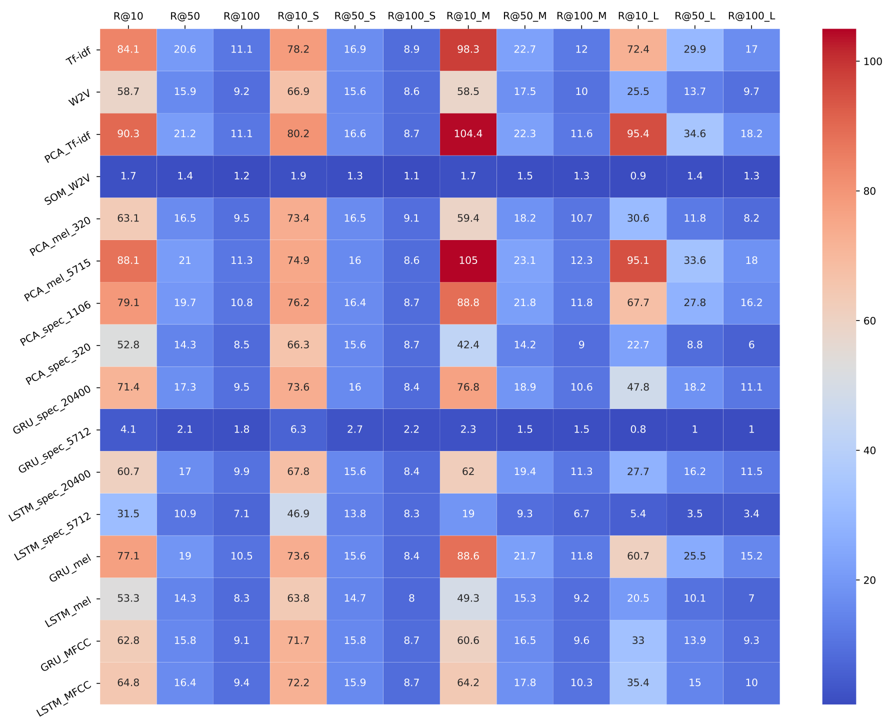

Figure 4.28: Method comparison to random suggestions heat map.
Each rectangle contains the multiple of how many times the method on its corresponding y coordinate is better than random suggestions of the evaluation measure on the corresponding x coordinate. For example if random recommendation would place 0.5% of songs into the top 10 meaning, that its value for R@10 was 0.005 the Tf-idf method would place $84.1 * 0.05\%$ of songs into the top ten so the value of R@10 for Tf-idf would be $84.1 * 0.005$ because 84.1 is the value in the rectangle that has Tf-idf on the y coordinate and R@10 at the x coordinate.

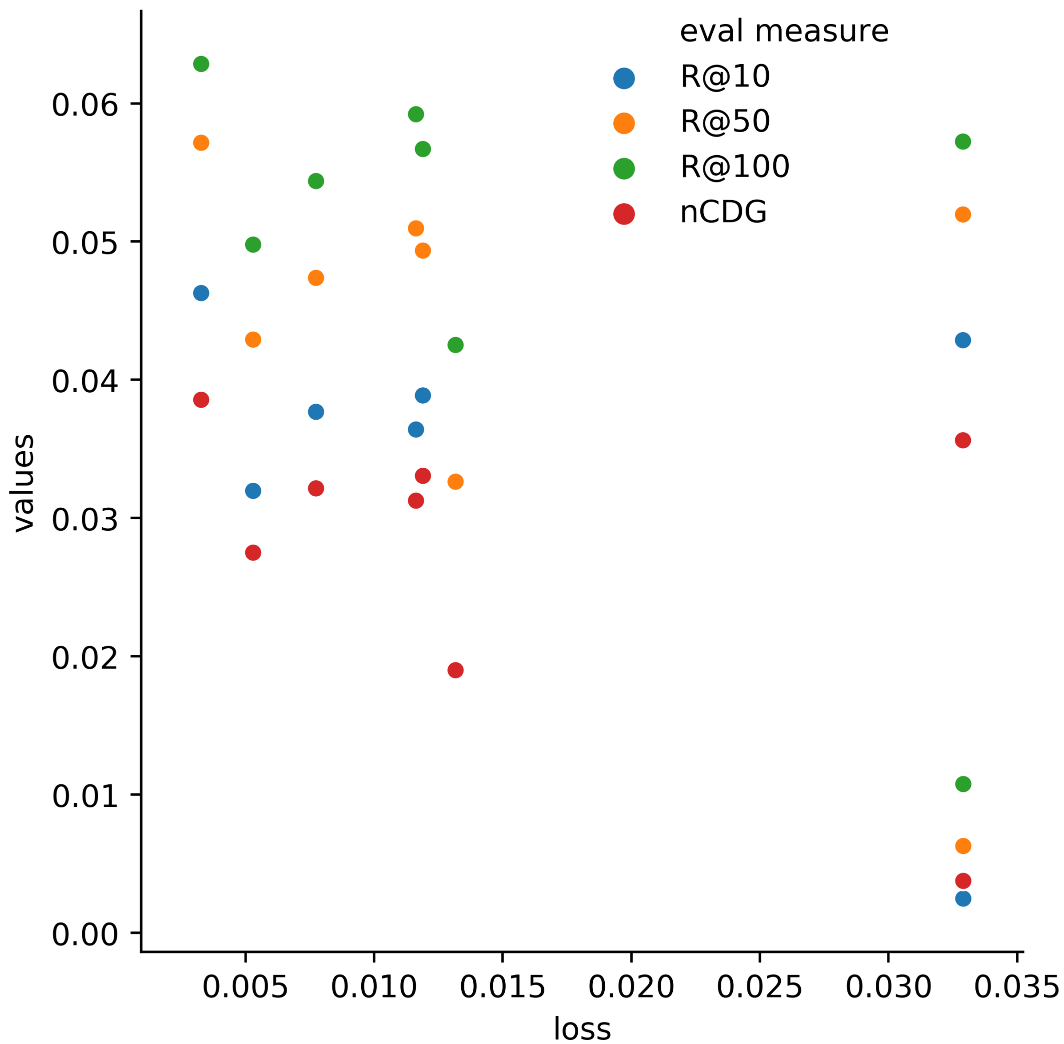

Figure 4.29: Training loss and performance correlation graph.
This graph visualized the correlation between the final training loss displayed on the x axis and the performance of neural networks on various evaluation measures. The values of the evaluation measures are displayed on the y axis. Each evaluation measure has a different color which is illustrated in the legend.

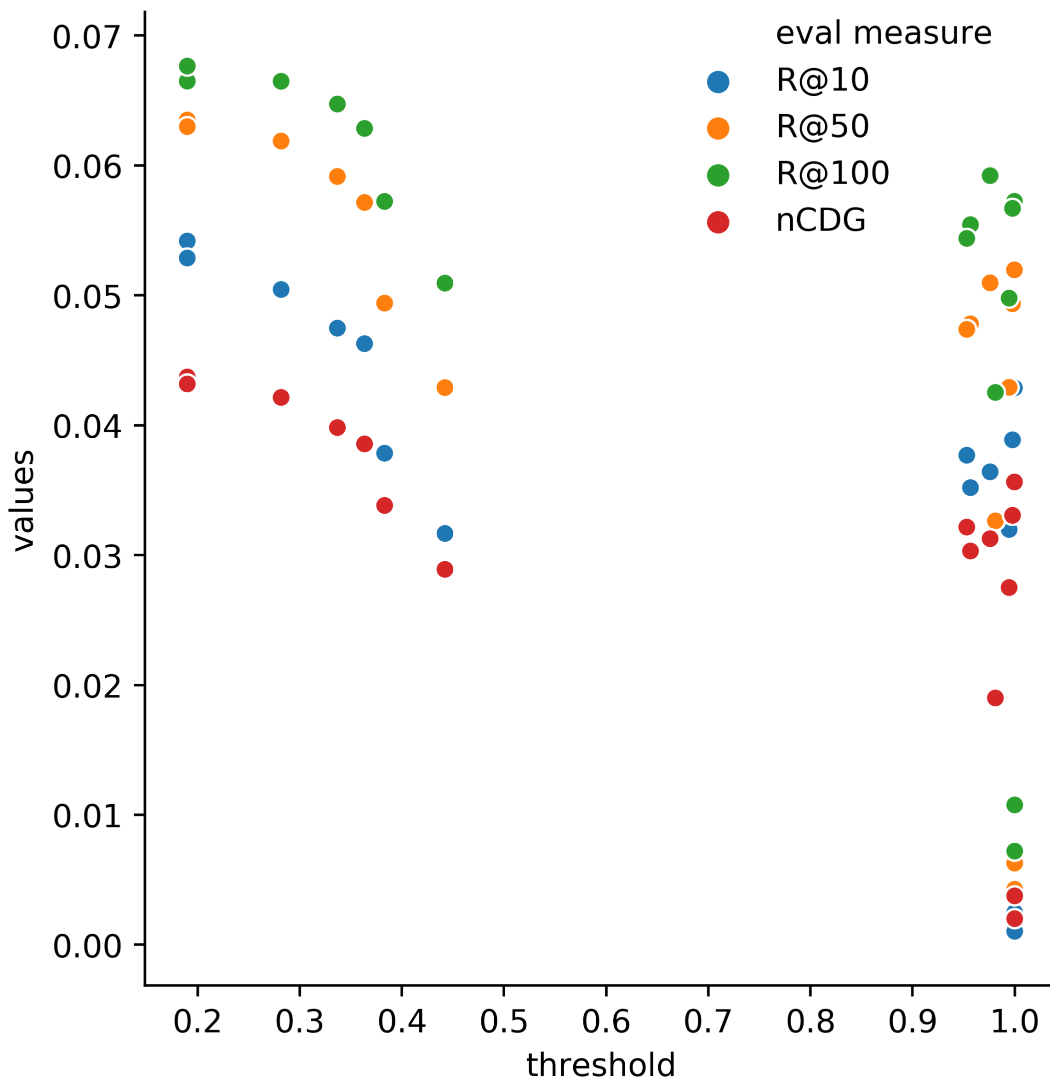

Figure 4.30: 0.03%-threshold value and performance correlation graph.
This graph visualized the correlation between the value of the threshold at position 846,700 in the D_m of every method and the method's performances on various evaluation measures.

5. Web Application

In this section, we will describe the proposed web application for novel song recommendation which we called the *SongRecommender*. The section is structured as follows:

- In Section 5.1 we analyze our application goals and describe what the user can expect from our application.
- In Section 5.2 We briefly introduce the building blocks of our application with focus on the individual similarity measures implementation and calculation of recommendations.
- In Section 5.3 We present the possible configurations of our application.

The source code of this project is included in Attachment A.1 together with user documentation which we decided not to include in this thesis. The modules we are referring to in this chapter can also be found there. To navigate the various modules and directories, see `README.md` which can be found in the attachment in the `songRecommender_project` directory.

5.1 Analysis

There exist many music recommendation apps on the web such as YouTube[1], Spotify[2] etc. They have a lot of data about users, user activity, a lot of songs, a lot of tags. Our application does not aspire on growing to such extend. We want to provide our users with an inspiration for new songs which they can then play on another musical platform (for example on YouTube or Spotify).

We aim to provide common web app functionalities such as creating accounts, logging in and out, browsing individual songs, etc. Besides that, since it is a song recommender application, we also want to propose recommendations to users, let them create playlists, search for songs and like and dislike songs to improve the suggestions. Moreover, we want the users to be able to add songs they already know and that are missing in the application in order to explore which songs are similar to it based on various recommendation methods. We are also aiming on providing information about the application and its similarity measures, so even if the recommendations do not seem relevant to the user, it might be interesting for him to see, which songs are similar to which based on, for example, lyrics. It is an opportunity for a more hands on experience and it makes it not only about music but also a bit about the theory behind this application.

5.2 Implementation

In this section, we present the overall architecture of our application with focus on the recommendation functionalities which are described in more detail in Subsection 5.2.3.

[1] www.youtube.com
[2] www.spotify.com

5.2.1 Technologies

We build our web application in the Django[3] framework using Python 3.6. We chose Python[4] because it is well suited for machine learning and Django because it is a Python based framework. To ensure a smooth user experience while performing complex computational tasks, we included Celery[5], an asynchronous task queue, to run expensive tasks in the background. We used RabbitMQ[6] as Celery's message broker. We chose the PostgreSQL[7] database to store the data instead of Django's default — SQLite — because it provides support for `ArrayFields` which are an efficient and convenient way of storing the calculated song's feature vectors.

5.2.2 Design

The application follows the Model-Template-View (MTV) pattern which is a variation of the standard Model-View-Controller (MVC) pattern with the difference that the Django framework handles the controller part itself. The controller part would in this case be the processes that send the requests to the appropriate view according to the url configuration. This pattern is pre-wired in Django.

The *model* part handles the data representation as it does in the MVC pattern. The main things we need to store are the songs with their attributes such as title, artist, etc. and the representations of the implemented methods. We also need to store the users, lists that users create and similarities which we use to calculate recommendations.

The *template* part is an analogy to the view part in MVC and handles how the data is displayed to the user in the form of HTML5 templates.

The *view* handles what data is displayed to the user so it does a part of the job of the view from MVC. It also contains the business logic of the application so one could say that it is also a variation of the controller or rather a bridge between the models and templates and the Django controller. In the songRecommender application, the views call functions from the "logic" part of the application which takes care of calculating recommendations.

Models and Database

For every table in the database, there is a class-based model in the module `models.py` specifying its features. There is a class of the same name for the `Song` and the `List` tables. The `Profile` model which is an extension of the build-in Django `User` model is included to enable storing the similarities of songs to the user and also checking if the user has confirmed his email.

The class `Song_in_list` keeps track of songs and lists that belong to each other, an instance of the `Played_song` model specifies a user and a song he has played. One cannot un-play a song but it can be disliked and it will not appear

[3]https://www.djangoproject.com
[4]https://www.python.org
[5]http://www.celeryproject.org
[6]https://www.rabbitmq.com
[7]https://www.postgresql.org

anymore in recommendations and it will also not be used to calculate recommendations of other songs.

There are three different models for storing similarities. As mentioned in Chapter 4 even though we calculate and store similarities between songs they are named as if it were distances. The models are called `Distance` whose instances store basic similarities between two songs, `Distance_to_list` whose instance stores a similarity of a song to a particular list and `Distance_to_user` whose instance stores the similarity of a song to a particular user. The database

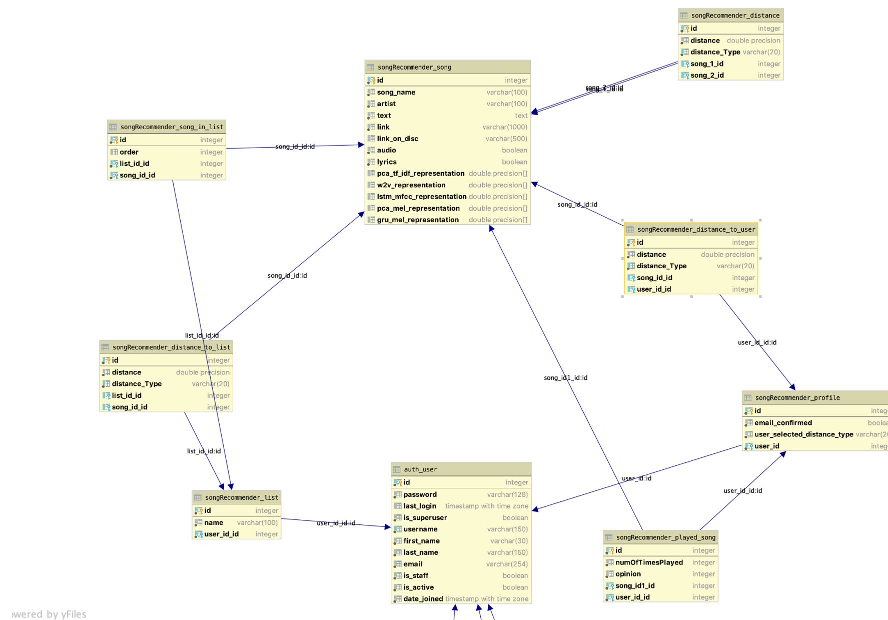

Figure 5.1: Diagram of the applications database

is structured as Figure 5.1 illustrates. Build-in Django tables are omitted for clarity.

Views

Views handle the requests users send. Each request from some HTML page is processed by a function from `views.py`. It takes it request's parameters, calls a functions from the "logic" part of the application if necessary, collects the context for the next HTML page based on that and displays that page to the user.

There are two main kinds of views in this application. First are build-in *class-based views* which are structured around a model class from `models.py`. For exapmle the `SongDetailView` displaying a detail page of a song is a class-based view structured around the `Song` class. The second kind are *function-based views* which are not tied to a model. In the application, these are used for example for handling, liking and disliking songs.

We used both kinds of views as we could make use of the abstraction and code simplification *class-based views* offer for most of the pages that revolved around models. The *function based views* on the other hand provided a more flexible choice for more operational views, not only liking and disliking songs, but also changing the distance metrics etc.

Server

All the logic of the application is running on the server. Most expensive tasks are sent to Celery to be handled asynchronously, so users do not have to wait. The expensive tasks are those that include calculating and recalculating song similarities. They can be triggered by four main events:

First demanding event is adding a new song. The song's mp3 is downloaded from the link the user provides, then the 15 second audio excerpt is created and turned into a mel-spectrogram and MFCC which are then input for corresponding audio method models. The songs lyrics are stripped of punctuation characters and prepared as input for the lyrics methods. After obtaining the various encodings of the newly added song, the application then calculates and stores the similarity of this song to all the other songs that are already in the database for each implemented method separately.

Afterwards, the similarity of the song to other users and to all the lists in the database is calculated and stored. It is again, for each of the implemented methods. Adding a song takes about 15 seconds if there are no other songs in the database, there is 1 user, 1 list and the five similarity methods we chose in Section 5.2.3 are implemented. It takes about 555 seconds if the full song dataset is loaded into the database and there is 1 user with 1 list and the same 5 methods are implemented.

Secondly, the user can play a song for the first time. The overall recommendations are based on the songs the user has played so it is necessary recalculate similarities of all songs to the user when adding something to his *played songs*.

The third event is adding a song to a list. In that case, the similarities of all songs to this list are recalculated.

And the fourth event is liking or disliking a song, which results in recalculating similarities of all songs to the user.

These are quite time consuming tasks especially with a growing song, list and user counts. The addition of a song is by far the most complex one.

Client

The client only receives pre-computed HTML5 pages with some CSS for better design. We used the Bootstrap[8] library which provides nice page layouts even on smaller screens and phone screens. There is no computation on the client side.

5.2.3 Similarity measure implementation

We did not chose methods for the application in Chapter 4 because in order to implement them, we cannot look only at their performance but also at their computational properties — the time and space complexities. Therefore, we first describe the time/space complexities of the various tested methods and then we finalize our choice of methods for implementation.

[8] https://getbootstrap.com

Time and space complexity

There are four main events described in 5.2.2 that trigger the similarity calculations and recalculations in the application. However, only the duration of an addition of a new song into the database is dependant on the length of the vector representation. All the other events only use similarities already stored in the database which are represented by a single floating point number.

When a new song is added, it is transformed into the respective vector-encoding for each implemented method. Afterwards, its similarity to all the songs inside the database has to be calculated. This step is the one which takes significantly longer for longer vector representations.

A *for cycle* over all songs in the database is inefficient for Python. Therefore, we take advantage of the `pairwise.metrics.cosine_similarity` method from Python's `sklearn` package. It is the same function as we used for calculating D_m matrices for evaluation. We insert the new vector as the first parameter and all the vector representations in the database as the second parameter of this method. The similarities are calculated for each implemented method.

Because the server has limited RAM and because we do not want to limit the number of songs inside the database, we split the songs in the database into chunks and the similarities are calculated for one chunk, then saved and then the next chunk is processed. This helps us avoid a memory error even with a growing song count.

The chunk-size differs for different methods. With an increasing chunk-size the similarity calculations are faster so we made them as big as possible. The size is smaller for longer vectors and bigger for shorter vectors as it is there merely to avoid a memory error and more short vectors fit into memory at once.

The reason we care about how fast the calculations are is not really to make the next page load faster as the user does not have to wait in any case because of the asynchronous task queue. We however want our application to be responsive and recommend relevant songs as soon as possible.

The biggest issue with space complexity for methods is the size of their trained model. The models are needed when a new song is added to predict the respective song-representation for the implemented methods. To avoid re-loading the models every time a new song is added, they are loaded into memory once at the beginning when the server starts. For some methods, the models are quite small (tens to hundreds of kilobytes) but for some, their size goes up to Gigabytes.

This poses a problem. It takes time to load big models which is the smaller issue as it happens only once in theory. However, some of them are as big as the RAM of the server so it has to put them aside while performing other tasks, which makes predictions much slower when it loads them again.

Method selection

With keeping the above in mind, let's look at Table 5.1 and Figure 4.27. We put our final choices together based on these values but there is also a third thing to consider. We are setting a higher priority on lyrics-based methods as they are more unique and their recommendations maybe a bit more interesting for the user. We also take the diversity of our methods into account as we want to offer

an insight on how recommendtaions for various methods with various inputs look like.

First we rule out the methods that seem to perform badly/are ranked last in most categories. It is the SOM_W2V, the GRU_Spec_5712, the LSTM_Spec_5712, and the LSTM_mel, and the PCA_spec_320.

We also will not implement the GRU_spec_20400 and LSTM_spec_20400 and Tf-idf, raw mel-spectrograms and MFCCs because of the length of their vectors. The PCA_spec_1106 and PCA_spec_5712 disqualify because of the size of their models.

This leaves us with two text based methods — the W2V, and the PCA_Tf-idf — and four audio-based methods — the PCA_mel_320,GRU_mel, LSTM_MFCC and the GRU_MFCC. As we can see, none of these has spectrograms as input so we will not use any spectrogram method. Out of these, the worst one is W2V, however, it is a lyric method and it is very different from all other methods, so we decided to implement it. We left out the GRU_MFCC as we still have have the LSTM_MFCC network method with the same input and GRU_MFCC is the second worst after W2V. We could keep all the six methods but we want to reduce the number of methods because the distance calculations are not calculated in parallel but one after another. More methods are therefore more complex.

Let's recapitulate our method choices:

- **PCA_Tf-idf** is a lyrics-based method and also the best method we presented and it has reasonable vector length.
- **W2V** is also a lyrics-based method, it is the worst from the ones we decided to implement but it has short vectors and a small model and helps with method diversity.
- **PCA_mel_320** is a audio based method. It appears to be good in the average rank of a song, however, that is not so important for recommendation. Its overall results are average and its time and space complexities very convenient.
- **GRU_mel** is also deep neural network method which performed best between neural networks.
- **LSTM_MFCC** is another deep neural network method which has two components we have not used in any of our implemented methods, the LSTM layers in its architecture and it takes the MFCCs as input. It is unique with good results relative to other methods.

Recommendation calculation

The similarity of two songs is calculated using the cosine similarity with threshold cos_sim_t as described in Subsection 4.7.2. The similarity of a song to a user is an addition of similarities. To be specific, the similarity of a song s_i to a user U is calculated as:

$$\sum_{k=0}^{k=n} c * cos_sim_t(s_i, s_k)$$

method	vector length	model size in KB
raw mel-spectrograms	130,560	no model
raw MFCCs	82,688	no model
Tf-idf	40,165	2,600
PCA on Tf-idf	4,457	1,430,000
W2V	300	251,100
SOM_W2V	2	358,792
PCA_spec_1106	1,106	7,980,000
PCA_spec_320	320	2,320,000
PCA_mel_5715	5,715	5,970,000
PCA_mel_320	320	336,300
GRU_spec_20400	20,400	67,000
GRU_spec_5712	5,712	59,000
LSTM_spec_20400	20,400	3700
LSTM_spec_5712	5,712	1003
GRU_mel	5,712	146
LSTM_mel	5,712	185
GRU_MFCC	5,168	48
LSTM_MFCC	5,168	471

Table 5.1: The vector length and model size for different methods

where $i \neq k$ and s_k is a song from the user's played songs and n is the number of songs the user has played. c is a constant which is either 1 when the song s_k was played, 2 when s_k was also liked or 0 when it was disliked. The similarity of a song s_i to a list L is calculated as:

$$\sum_{l=0}^{l=m} c * cos_sim_t(s_i, s_l)$$

where $i \neq l$, m is the number of songs in list L, s_l is a song from list L and c is the same constant as before.

The displayed recommendations do not include songs the user has already played. It is also possible to see always only the top 10 recommendations, overall and for a each list. For the detail page of a song there is an audio area where it is possible to play the song. Its 10 most similar song are displayed on the right. There is also a possibility to like it or dislike the song.

5.2.4 Base data import

To provide songs to users of the application from the beginning, we uploaded the data from the SD dataset and the data we acquired during experimentation to the database. This made them available in the application from the moment of its launching.

The methods that were used to transfer the data into the database are in the `data/load_distances.py` module. The songs from SD with corresponding titles, artists, lyrics, YouTube links, and relative paths to .mp3 files in the file system were stored using the `load_song_to_database` method as instances of the class `Song`. Afterwards, we added all five vector representations for the five implemented methods to each of the 16,594 `Song` instances using the `load_all_representations` method. The representations were extracted from the R_m matrices. Finally, we called the `load_all_distances` method to create and store instances of the `Distance` class for the similarities from D_m matrices that were above the 0.03%-threshold, meaning, there are about 829,700 instances of `Distance` for each of the five implemented similarity methods. We did not store the similarity of the song to itself.

5.3 Configuration options

In this section we will describe the configuration options the application provides. There are some options that influence recommendation as well as options to change the server settings. All of the configurations are in the module `setting.py`.

5.3.1 Similarity method configurations

The things that can be set for similarity methods are the threshold values for the implemented methods. Right now, the threshold values correspond the to keeping 827,900 biggest similarities for each similarity method, meaning, there are about 50*16594 instances of `Distance` stored in the database for each of the implemented methods. Changing the threshold does not influence the songs that are already in the database. Nevertheless, when a new song is added, potentially more distances will be created, when the threshold is lowered, or less, when its raised. Another thing that can be configured in the `setting.py` module is the default similarity method that is assigned to a new user account. The user can then change it once he is logged in.

5.3.2 Email

There is a boolean variable called `EMAIL_DISABLED` which is set on True by default as there is no email service set up with our application. However, if an email service and a domain is provided, it is only necessary to change the `EMAIL_DISABLED` variable to False, delete the `EMAIL_BACKEND` and replace it with an email service configuration. If the variable is set to False without configuring an email service, the email is printed to the console and the user has no way of authenticating his account.

5.3.3 Server

The application has not been completely deployed. It runs on `http://acheron.ms.mff.cuni.cz:42009/index/` in debug mode. Since the launching, there were songs added to it and multiple users have created accounts and lists. To actually deploy the application, it is necessary go to `settings.py` and change the `Debug` variable to `False`. Then it is necessary to collect all static files (including all the mp3 files) and tell Django where they are as it stops handling static files itself without the debug mode on. It is also necessary to choose some wsgi, for example `gunicorn`[9] with `nginx`[10] or `apache`[11] as web servers and configure those.

[9]https://gunicorn.org
[10]https://nginx.org/en/
[11]https://httpd.apache.org

Conclusion

We tested multiple content-based recommendation techniques and the results show that both lyrics-based and audio-based methods are able to provide relevant recommendations. The numbers suggest that for some methods the recommendations are more than hundred times better than random recommendation.

A closer examination of the results also indicates that lyrics-based methods – specifically the Tf-idf and PCA with Tf-idf as input — profit from the variability of items they recommend which appears to suit real user playlists well. The deep audio-based methods, even though they stayed a bit behind in performance, seem to be prospective methods especially with additional hyper-parameter and architecture adjustments.

We also successfully introduced a running web application with recommendation methods that provide users with novel, relevant suggestions without being dependant on song popularity. The fact that our dataset, which was loaded into the application, probably contains mostly at least somewhat popular songs is compensated by the fact that each user can add his own songs. Like this less popular songs can become part of the database and have an equal chance of being recommended as popular songs which was the goal of this work.

Future work

Recommendation methods

Further study of lyrics-based methods is one way to go in the future. We could use for example the Tf-idf vectors not only as input for the PCA or SOM but also for other types of neural networks. They might not be suited for RNN networks as there is no kind of sequential information stored in the Tf-idf vector but it would be interesting to try different architectures.

As we tested our audio methods, some proved to be more perspective than others. Mel-spectrograms and MFCCs seem to be a better input for similarity methods than raw spectrograms. Also the PCA showed to have great potential with both lyrics and audio based methods so further research into this and other dimensionality reduction methods seems to be a good idea.

When it comes to neural networks, networks with the "GRU" layers seem to perform better than "LSTM" layers. Also many different layer combinations and architectures can be tested. Using RNNs without returning whole sequences also seems as a reasonable thing to do because it appears that the fact that the sequence-to-sequence RNN autoencoders only reduce the number of features and not the number of timestamps is limiting for these networks. The PCA which reduced the features and the timestamps might have benefited mainly from that. Also other types of neural network layers can be used to build the autoencoders which then can be tested and possibly implemented.

Moreover, besides making changes in the methods that encode songs into vectors, research could be also done on the aggregation of vectors by some similarity measure. We used only cosine similarity throughout the thesis but apart from other simple distance metrics such as the *Euclidean distance* or *Manhat-*

tan distance more advanced aggregation methods can be tested for example the GRU4rec [33] method might be a suitable metric especially if we had data about songs that were played most recently and would like to introduce session-based recommendation.

Web application

The web application can be further extended in multiple ways. One thing would be creating a system, where the users could rate the recommendations so we would have feedback about method performance not only from the evaluation we did in this thesis but also from real time users.

Also, more advanced recommendation metrics could be applied in the web application. We could keep track of the users lastly played songs or take into account how many times he played a song and then use this in the final similarity calculation. For example have something like *The most similar songs to the last 10 songs you played* or *The most similar songs to your 10 most played songs* etc.

There is obviously also the possibility of including more similarity methods into the application. However, this now involves a non-trivial amount of changes to the source code. A simplification and a better design for the logic of the application could be a step to take in the future.

Then there are some application features that could be implemented which follow the functionalities of traditional music-applications. This includes playing whole playlists or creating an endless playlists from the recommendations as well as adding videos and tags to songs, allowing searching based on genres etc.

Bibliography

[1] Ayush Singhal, Pradeep Sinha, and Rakesh Pant. Use of deep learning in modern recommendation system: A summary of recent works. *CoRR*, abs/1712.07525, 2017.

[2] Aaron Van den Oord, Sander Dieleman, and Benjamin Schrauwen. Deep content-based music recommendation. In *Advances in neural information processing systems*, pages 2643–2651, 2013.

[3] Alexandros Tsaptsinos. Lyrics-based music genre classification using a hierarchical attention network. *CoRR*, abs/1707.04678, 2017.

[4] Derek Gossi and Mehmet Hadi Gunes. Lyric-based music recommendation. In *CompleNet*, 2016.

[5] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[6] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[7] Erion Çano. *Text-based Sentiment Analysis and Music Emotion Recognition*. PhD thesis, 06 2018.

[8] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.

[9] Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, Jan 1982.

[10] Teuvo Kohonen. The self organizing map som. `http://www.cis.hut.fi/research/reports/quinquennial/ch1.ps`. Accessed: 2019-03-16.

[11] Braja Patra, Dipankar Das, and Sivaji Bandyopadhyay. Retrieving similar lyrics for music recommendation system. 12 2017.

[12] Jan Schlüter. *Deep Learning for Event Detection, Sequence Labelling and Similarity Estimation in Music Signals*. PhD thesis, Johannes Kepler University Linz, 2017.

[13] S. S. Stevens. A Scale for the Measurement of the Psychological Magnitude Pitch. *Acoustical Society of America Journal*, 8:185, 1937.

[14] S Umesh, Leon Cohen, and Douglas Nelson. Fitting the mel scale. volume 1, pages 217 – 220 vol.1, 04 1999.

[15] Karl Pearson F.R.S. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[17] Cs231n: Convolutional neural networks for visual recognition. `http://cs231n.stanford.edu`. Accessed: 2019-04-26.

[18] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[19] Paul Smolensky. Information processing in dynamical systems: Foundations of harmony theory ; cu-cs-321-86. 1986.

[20] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[21] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[22] Mike Schuster and Kuldip K. Paliwal. Bidirectional recurrent neural networks. *IEEE Trans. Signal Processing*, 45:2673–2681, 1997.

[23] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013.

[24] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, and Puneet Agarwal. Long short term memory networks for anomaly detection in time series. 04 2015.

[25] S. Dieleman and B. Schrauwen. End-to-end learning for music audio. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6964–6968, May 2014.

[26] Xinxi Wang and Ye Wang. Improving content-based and hybrid music recommendation using deep learning. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 627–636. ACM, 2014.

[27] Prasanna Ramakrishnan. song 2 vec : Determining song similarity using deep unsupervised learning. 2017.

[28] Honglak Lee, Peter Pham, Yan Largman, and Andrew Y. Ng. Unsupervised feature learning for audio classification using convolutional deep belief networks. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1096–1104. Curran Associates, Inc., 2009.

[29] Shahin Amiriparian, Michael Freitag, Nicholas Cummins, and Björn Schuller. Sequence to sequence autoencoders for unsupervised representation learning from audio. 11 2017.

[30] G. Vettigli. Minisom. https://github.com/JustGlowing/minisom, 2019.

[31] Brian McFee, Matt McVicar, Stefan Balke, Vincent Lostanlen, Carl Thomé, Colin Raffel, Dana Lee, Kyungyun Lee, Oriol Nieto, Frank Zalkow, Dan Ellis, Eric Battenberg, Ryuichi Yamamoto, Josh Moore, Ziyao Wei, Rachel Bittner, Keunwoo Choi, nullmightybofo, Pius Friesch, Fabian-Robert Stöter, Thassilo, Matt Vollrath, Siddhartha Kumar Golu, nehz, Simon Waloschek, Seth, Rimvydas Naktinis, Douglas Repetto, Curtis "Fjord" Hawthorne, and CJ Carr. librosa/librosa: 0.6.3, February 2019.

[32] François Chollet et al. Keras. https://keras.io, 2015.

[33] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. Session-based recommendations with recurrent neural networks. 11 2015.

List of Figures

1.1 First two entries of the 55000+ Lyrics Dataset 9
1.2 Playlists' lengths histogram 11
1.3 The histogram of playlist counts per individual songs 11

2.1 The CBOW and Skip-gram Word2Vec architectures from [7] . . . 14
2.2 The Doc2Vec DM and DBOW architecture taken from [8] 15
2.3 A visualization of the training used for SOM networks 16

3.1 A diagram displaying the steps taken in audio extraction and feature learning. ML stands for machine learning and DL for deep learning. 19
3.2 Spectrogram of the song 'Someone Like You' by 'Adele'. The intensity of different frequencies over time is converted to decibels. . 20
3.3 Mel-spectrogram of the song 'Someone Like You' by 'Adele'. The intensity of different frequencies over time is converted to decibels. 21
3.4 MFCCs of the song 'Someone Like You' by 'Adele'. The intensity of different frequencies over time is converted to decibels. 22

4.1 The steps in feature learning from [29] where we also got this diagram from. We added the red rectangle which represents the portion of their procedure that was also (with adjustments) performed by us. 34
4.2 The general architecture of GRU neural networks 35
4.3 The general architecture of LSTM neural networks 35
4.5 The comparison of absolute max, average and min values for the evaluation measures for recommendation with and without threshold 42
4.6 RDG of the TF-idf method 43
4.7 RDG of the PCA_Tf-idf method 43
4.8 Distribution of ranks of songs from the test set the w2v method assigned them. 44
4.9 The location of different songs from 20 randomly selected playlists on the map created by SOM. Each playlist has its own colour. . . 45
4.10 RDG of the SOM_W2V method 46
4.11 RDG of the SOM_Tf-idf method 46
4.12 RDG of raw Mel-spectrograms. 47
4.13 RDG of raw MFCCs. 48
4.14 RDG of the PCA_spec_1106 method. 49
4.15 RDG of the PCA_spec_320 method 49
4.16 RDG of the PCA_mel_5712 50
4.17 RDG of the PCA_mel_320 method 50
4.18 RDG of the GRU_spec_20400 method 51
4.19 RDG of the GRU_spec_5712 method 51
4.20 RDG of the LSTM_spec_20400 method 51
4.21 RDG of the LSTM_spec_5712 method 51
4.22 RDG of the GRU_mel method 52
4.23 RDG of the LSTM_mel method 52

4.24 RDG of the GRU_MFCC method 53
4.25 RDG of the LSMT_MFCC method 53
4.26 Method performance comparison heat map without threshold applied 57
4.27 Method performance comparison heat map with threshold applied 58
4.28 Method comparison to random suggestions heat map 59
4.29 Training loss and performance correlation graph 60
4.30 0.03%-threshold value and performance correlation graph 61

5.1 Diagram of the applications database 65

List of Tables

4.1 Table containing the value of the similarity threshold we used. The threshold for a particular method is always below the method's name. 41
4.2 Table summarizing average Tf-idf and Tf-idf with PCA evaluation measure values averaged over the 5 cross validation that were performed. 42
4.3 Table summarizing average W2V evaluation values averaged over the 5 cross validation that were performed 43
4.4 Table summarizing average SOM evaluation values averaged over the 5 cross validations 44
4.5 Table summarizing average evaluation values for all methods with mel-spectrogram input averaged over the 5 cross validations. . . . 47
4.6 Table summarizing average evaluation values for all methods with MFCC input averaged over 5 cross validations with threshold. . . 48
4.7 Table summarizing average evaluation values for methods with spectrogram input averaged over 5 cross validations 49
4.8 Table containing the value of the diversity index that was also calculated for the UD we have. 54

5.1 The vector length and model size for different methods 69

A. Attachments

A.1 First Attachment

The zipped attachment has the following contents:

- A folder called `songRecommender_project` containing the source code of the project. It includes a `README.md` file that describes the project's directories and modules.
- User documentation in pdf format under the name `srUserDocs.pdf` also included in the `songRecommender_project` directory