Music Similarity Tool for Contemporary Music

Paul Arzelier

Kongens Lyngby 2018

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

Digital music is becoming more and more widespread, with over 600 million tracks sold in the US in 2017. This leads to a need to search through this massive amount of music to find songs people want to listen to, thus making playlist creation difficult even for people with only local music libraries.
This thesis aims at finding ways to make playlists that people will enjoy based on a starting "seed" song and their own music libraries. The novel aspect here is the use of a listening survey a priori to tune a feature-based distance metric: instead of including feedback from the user in an iterative process, such as the songs they skipped, the songs they liked, etc, the feedback will be incorporated to the system beforehand.
If the training of the tool using data from the listening survey didn't lead to much improvement when trying to predict results from the same survey when splitting it into test/training sets, the training metric proved to generate more enjoyable playlists than the non-trained metric on the final listening tests.

Preface

This thesis was prepared at DTU Compute in fulfillment of the requirements for acquiring an M.Sc. in Computer Science and Engineering. This thesis was done by Paul Arzelier during the period from January 2018 to June 2018, at the department of Computer Science and Engineering of Danmarks Tekniske Universitet (DTU).

This work was supervised by both Professors Jan Larsen and Tobias May, in a collaboration between DTU Compute and the Hearing Systems Group.

Lyngby, 02-June-2018

Paul Arzelier

Acknowledgements

I would first like to thank my two supervisors: Jan Larsen, for his valuable input on all machine-learning related issues, and Tobias May for helping me on the audio extraction/analysis side of this project. I would like to thank both of them as well for their guidance and constant advice on the general path to follow.
Thanks to Anders U. as well, and to all the other people I frantically e-mailed whenever I had a question on their paper for their very useful and patient pieces of advice.

I would also like to thank Chloé and Auriane for their (hopefully) thorough proofreading; all the people that took time to answer both surveys -which were quite long, to be honest; and in a more general note, a particular Freenode IRC canal for their friendly support during the entire duration of the thesis.

Contents

Summary i

Preface iii

Acknowledgements v

1 Introduction 1

2 Feature Selection 7
- 2.1 State of the art review . . . 7
- 2.2 Features description . . . 9
  - 2.2.1 Timbral Features . . . 10
  - 2.2.2 Temporal features . . . 14
  - 2.2.3 Tonal features . . . 15
- 2.3 Feature summarization and storage . . . 16

3 Web Survey 19
- 3.1 Survey Description . . . 19
  - 3.1.1 Survey Objectives . . . 19
  - 3.1.2 Survey characteristics . . . 21
  - 3.1.3 Survey dataset . . . 22
  - 3.1.4 Incomplete Random Design . . . 23
- 3.2 Survey Implementation and Technical details . . . 26
  - 3.2.1 Survey workflow . . . 26
  - 3.2.2 Survey technical details . . . 27
- 3.3 Survey results summary . . . 31

4 Use of survey: metric learning **33**
4.1 Survey of the state of the art 33
4.2 Minimization in practice 36

5 Playlist generation **39**
5.1 Playlist generation objectives 39
5.2 Playlist generation algorithm 40

6 Evaluation **43**
6.1 Preliminary figures 44
6.2 Objective Evaluation 44
6.3 Subjective Evaluation 48

7 Conclusion **51**

A Matlab source code **55**

B Python source code **61**

C Differentiation details **77**

Bibliography **79**

Chapter 1

Introduction

Thanks to mass digitization of tracks, the music industry is growing, and growing fast: between 2016 and 2017, the total audio consumption in the US, e.g. the total of albums being physically bought, downloaded, and streamed has increased by 10.2% [Mus18], going from 566.1 million units obtained in 2016 to 636.6 million in 2017. In fact, an extremely large number of people consume music, would it be physical or numeric, leading to an extremely diverse music consumption landscape.
Even if there are many different ways to enjoy music, one issue that often arises when listening to music is track organization. In most cases, the listener has the choice of either setting all the tracks manually, which can be an inconvenience as they might have to create hours-long playlists themselves, or using the shuffle mode of their audio player, which usually doesn't lead to smooth transitions, and can even be troublesome when the context is not suitable for such transitions.

Because so many people have their own habit habit when listening to music, there are, as said above, lots of different ways and reasons to listen to it. These reasons can be, but are not limited to[LN11]:

- Mood management: the listener wants to be put in a certain mood
- Having a background noise: the listener dislikes silence, wants to make

other activies more enjoyable, or finds it easier to focus with a background noise

- Enjoying the music itself as a distraction: the listener wants to take his mind off things, dance...
- Using it as a social interaction: the listener likes to talk/interact about music with people

Most of these reasons have a common denominator, which is to keep the listener in a certain state, would it be a certain mood, a precise mindset, or a focused state. Thus, many people have expressed the need for non-random, logically ordered playlists [DBSK12a]. Indeed, in many mundane cases such as when putting on some music while working, studying, during a party, or even to ease sleeping, having "rough" music transitions can cause a great inconvenience to the user. Respectively, here, prevent the listener from focusing on his work/studies, ruin the atmosphere of a party, or prevent him from falling asleep and waking him up in the worst cases.

Even if 30% of these people, at least in America, use commercial audio streaming, where playlist order control is usually left to a third-party algorithm controlled by the streaming platform, 20% of the global music market revenue is earned by digital download, and 30% by physical format sales [ifp]: there are still many users with downloaded music libraries that could benefit from a tool allowing playlist generation from local files.

This report will focus on building such a tool, i.e. a tool that can generate playlists with transitions that are smooth to the ear. Even if this term is loosely defined, and ideal playlists depend on the user's profile [CFS15], here "smooth" means that the user will not interrupt what he is currently doing because of the change of songs, i.e. the transition is not strange enough to bother his mind.
This means that some kind of music similarity will need to be evaluated between songs, and that Music Information Retrieval (MIR) will need to be performed. This discipline is the science of extracting meaningful information from songs [BFD15]; in the case of the present report, it will be used to characterize songs in some way, as explained below.

Two main approaches exist to extract descriptors that reflect or characterize music [KS13]: context-based information retrieval, and content-based information retrieval. Context-based information retrieval usually doesn't require having access to audio files, but only to metadata, for instance, the artist of a song, its genre, but also user-generated data, such as who liked what on a streaming platform. Conversely, content-based information retrieval works on the audio input alone, no matter how/if there are metadata associated to it; for example,

tempo, rhythm, etc for a song is considered as content-based information as it only uses the raw audio signal to perform the information retrieval.

Choosing between context-based information retrieval and content-based information retrieval can be a difficult choice and depends heavily on which data is available on the platform running the information retrieval algorithm, thus making it important to know the scope of the system and the user target.
In the case of this report, the target will be people having human-sized music libraries, i.e. from 10 to 10 000 songs, which might not be tagged properly, i.e. metadata can be missing on most/all tracks, and that are listening to music indifferently with or without internet, i.e. who are not impacted in their listening experience by the loss of their internet connection.
The system aiming at working on any person's computer, there is no available centralized user database on which it could extract relevant context data to do collaborative filtering, as in these papers [Sha17][KD]. All of that has been said above makes the use of **content-based information retrieval** much more relevant than context-based information retrieval since almost no metadata is available, so that is what the present report will use.

Content-based MIR relies on feature extraction directly from the audio signal [KS13][Ler12]. The usual workflow of content-based MIR is shown fig. 1.1.

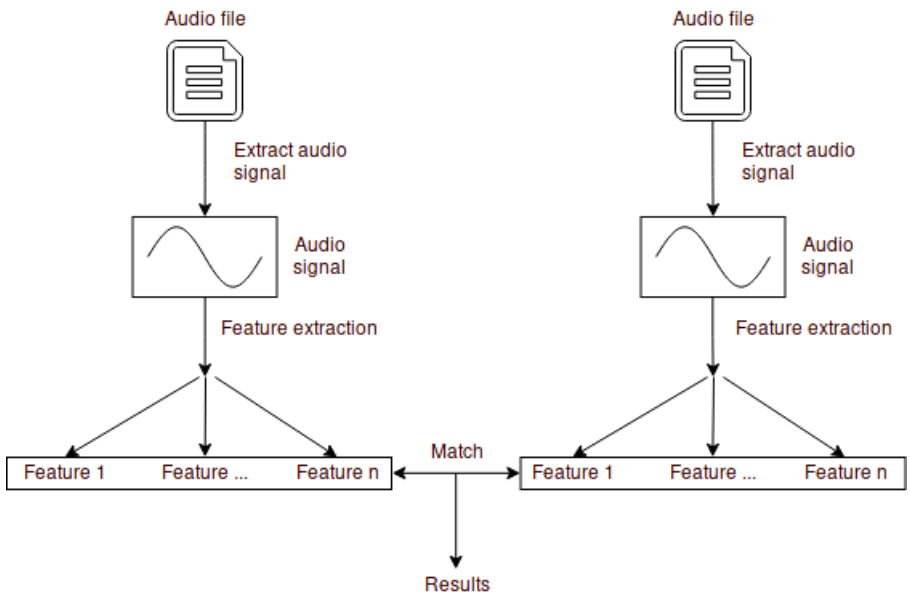

Figure 1.1: Usual content-based MIR workflow

Historic approaches used only timbral features, e.g. spectral-related features [TFS+12] [EzLSW15]. But music is more than only timbral components [Ler12]. For example, level has a great influence on similarity ratings[TFS+12]. The broad categories of relevant features for music similarity are[Ler12]:

- Timbre [TC02] [EzLSW15]
- Temporal components (rhythm, tempo, etc) [YC18]
- Tonal components (harmony) [FMD14]
- Intensity [Ler12]
- Structure [CVG+08]

Once the features are chosen, in order to avoid a time-consuming recomputing of them for each playlist creation, there is need for storing them efficiently. Indeed, storing gigabyte-sized (as many features are block-level features, e.g. songs are segmented in blocks and features are computed for each block) feature data is not practical for the targeted user. This is done in the literature by summing them up using summarizing functions, such as (but not limited to) mean and standard deviation[BSW+11] [TC02].

To make playlists using these processed features, a similarity/distance metric has to be chosen. It can range from a simple Euclidean distance to Earth Mover's Distance, or a more complex distance metric involving Gaussian Mixture Model for feature summarization[LS01][SWP10][DBSK12b]. The simple distances take the same weight for each feature, but to obtain better results, they can be weighted differently. It can be done *a posteriori* [DBSK12b], using user feedback after each playlist's iteration, but has never been done *a priori*. Taking user input into account even *before* letting the listener use the system might deliver a better "out-of-the-box" experience. Computing a set of default optimal parameters and integrating them directly into the system, ready to use for the user could potentially lead to better direct results. Thus, the present report will take user input into in order to increase system's performance, through a web survey.

This study is to be conducted in the present report, through the design and use of a web survey on which users input the odd song out for three songs, through different rounds. Once this user data is available, the parameters of a distance metric are tuned using minimization of a proper cost function [Uhr15] using metric learning.

Finally, once the parameters are properly tuned, the playlist has to be generated, using a seed song. Several playlists generation methods are available, ranging

from a simple chain-like algorithm to more complicated network flow models [Her08] [LS01] [AT01].

The accuracy of the generated playlists will be evaluated by users, which will have to chose the best playlist between:

- A randomly generated playlist
- A playlist using this report's algorithm and a regular euclidean distance
- A playlist using this report's algorithm and a trained distance metric.

Genre classification tests will also be performed to further evaluate the algorithm, inspired by MIREX evaluation tasks [mir].
The final goal of this present work is then to report whether or not including user input *a priori* to tune an open-source content-based music similarity system's parameters will achieve better performances to make smooth playlists than its equivalent without user input.

Chapter 2

Feature Selection

2.1 State of the art review

Feature selection in music information retrieval can be a daunting task, because an overwhelming number of papers already did some kind of music similarity system, and the features used are very diverse. Indeed, some papers tackle this issue in ways that would not seem common at all, such as doing music similarity based solely on onsets[YC18], or, instead of using feature values for computing music similarity, dropping these values completely and using sequential complexity as a descriptor for music similarity[FMD14].
Nevertheless, there is still a thread in music similarity, started by G. Tzanetakis' paper "Automatic Musical Genre Classification of Audio Signals"[TC02], that uses a mix of summarized spectral and temporal features, and then clusters music pieces using a simple distance metric.
Since a survey of the most influential papers has already been done in the introduction, a table with the most used features for music similarity and their corresponding references will be most useful and display fig. 2.1. All of these features will be used for this report's similarity tool.

Timbral features are historically the most used features, since they are based on the music spectrum, something easily obtainable from an audio file, and since they give good results when it comes down to music similarity[FLTZ11]. Still,

better results are obtained when timbral features are used along with other kind of features, such as temporal (tempo, BPMs) or tonal ones (HPCP)[FLTZ11], which is why features listed fig. 2.1 are not only timbral features.
They have been chosen for several reasons: first, as said above, they have to be from various domains to yield the best result; papers such as [TC02] back this assertion up. Second, since a survey will be then used to tune the algorithm's parameters, the number of features must be negligible compared to the number of observations in the survey. Since the expected number of observations is between 300 and 1000 - the detailed computation is carried out in 3.1.2 - a number of up to 10 features is reasonable, since there would then be a factor of up to 10^2 between the number of observation and the number of features. Finally, the features were chosen because they were the ones unanimously used among a panel of 15 well-known and widely cited papers.

Table 2.1: Summary of used features and related publications

Feature name	Feature type	Associated publications
Zero-crossing rate	Timbral	[BSWH] [FMD14] [BSW+11] [Pam06]
Spectral centroid	Timbral	[KS13] [BSWH] [TC02] [FMD14] [BSW+11]
Spectral roll-off	Timbral	[BSWH] [TC02] [FMD14] [FLTZ11] [BSW+11]
Spectral flatness	Timbral	[BSWH][FMD14] [FLTZ11] [BSW+11]
MFCC	Timbral	[KS13] [BSWH] [TFS+12] [MPWE07] [TC02] [FMD14] [FLTZ11] [LS01] [BSW+11] [Pam06]
Beats per minute (BPM)	Temporal	[BSWH] [FLTZ11] [BSW+11]
Onset loudness evaluation	Temporal	[BSWH] [TC02] [FMD14] [FLTZ11] [BSW+11]
Harmonic Pitch Class Profile (HPCP)	Tonal	[Gó06] [TC02] [FMD14] [FLTZ11] [BSW+11]

With that set of features, there are two problems that are still not addressed. First, the algorithm does not take level into account, which is an important parameter that has an influence on music similarity [TFS+12]; second, it doesn't necessarily track the evolution of songs, depending on how the features are summarized.
The first issue will be dealt with by using loudness normalization over the user's song library before computing feature values. The second issue will be addressed only partially, through the use of different means of summary for features. This is detailed in 2.3, but the idea is to use, instead of only the features' mean and

the variance, which is a crude way to summarize the song, percentiles, to obtain a little more fine-grained feature values. Still, this is an incomplete solution, because time frames don't necessarily capture the semantic divisions of the song, such as choruses, verses, etc. Some papers, such as [OH05], give ways to achieve this semantic segmentation; using this, though, would yield to more problems, such as the handling of instrumental songs without a clear semantic distinction. Thus, percentiles will be used, while keeping in mind that more elaborate methods could be used for song summary in the future.

2.2 Features description

In this section will be described the features that will be used for the tool, along with their main parameters value, usually the FFT window size, the windows overlap, etc.
Even though the rest of the report (survey website, learning algorithm...) will use Python, an open-source language with many modules available for a wide range of purposes, the feature extraction process will be done in MATLAB®, a proprietary programming language widely used for prototyping in music information retrieval. One of the main reasons for this choice is that many papers use algorithms from the MIR toolbox as almost standards from which results are directly usable without any further modification[FMD14]; the only problem being its poor memory management due to MATLAB®'s own poor memory management.[MRR].

Therefore, below are described features used in this report's MATLAB® tool, and their corresponding MIR toolbox functions. Most features will be computed along frames, with different framerates from features to features. The way to extract a meaningful value from this list of frame-based values will be discussed in section 2.3.
The parameters' values will be discussed in the light of both the current literature and evaluations that were made on a very small subset of 10 songs. These evaluations were ran to make sure that changing parameters would preserve the logical order of values. Songs that should have close feature values together, indeed have closer values than songs that shouldn't. This evaluation is of course informal and subjective and is just used as a mean to have a first insight of the features: the final parameter decision is also taken using relevant literature.

2.2.1 Timbral Features

2.2.1.1 Zero-crossing rate

The zero-crossing rate is one of the most simple metric available for music information retrieval; it is the number of time the audio waveform changes signs, or the number of time the waveform is equal to zero, as seen fig. 2.1.

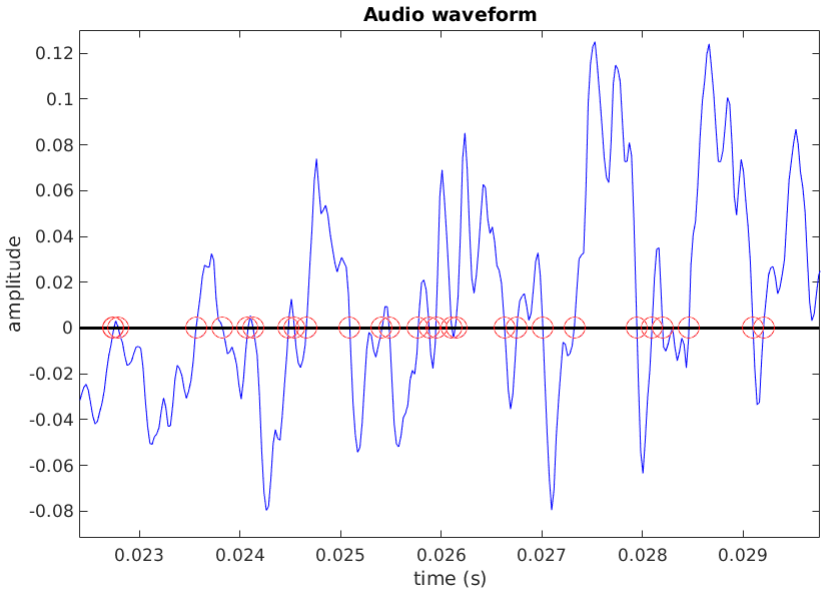

Figure 2.1: Red dots are points taken into account for zero-crossing rate computation

The zero-crossing rate is used as a rough method to detect human speech[BKAB10] and classify percussive sounds[GPD00].

Parameters: The default MIRToolbox framing parameters are changed to half of the default size, like in [FMD14]: a frame size of 25ms is used, with half-overlapping frames. These parameters will be kept among almost all the main spectral features for the same reasons.

2.2.1.2 Spectral Centroid

The spectral centroid indicates the "mean" of a signal's frequency distribution, sometimes also called the "center of mass". The spectral centroid S_c of a signal x given its discrete Fourier transform X on N samples is computed as follows:

$$Sc = \frac{\sum_{n=1}^{N} nX[n]}{\sum_{n=1}^{N} X[n]}$$

This is the mean of the normalized frequency distribution. For a normal distribution, the spectral centroid is shown fig. 2.2

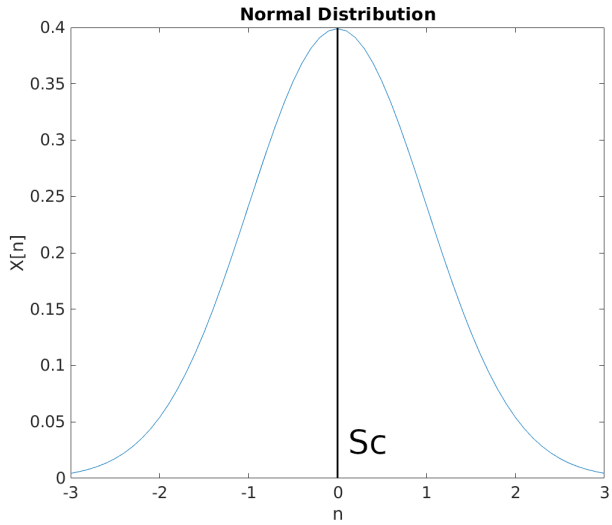

Figure 2.2: Spectral centroid of a normal distribution

It is a timbral feature that accounts for the "brightness" of a sound[Mcl07], along with the explosiveness of the attacks[GG78].

Parameters: 25ms frame size, with 50% overlap.

2.2.1.3 Spectral roll-off

The spectral roll-off of a signal corresponds to the frequency such that a percentage of this signal's energy is located below this frequency. For music information

retrieval, a percentage of 85% is considered as a standard[TC02] value. An example is shown fig. 2.3

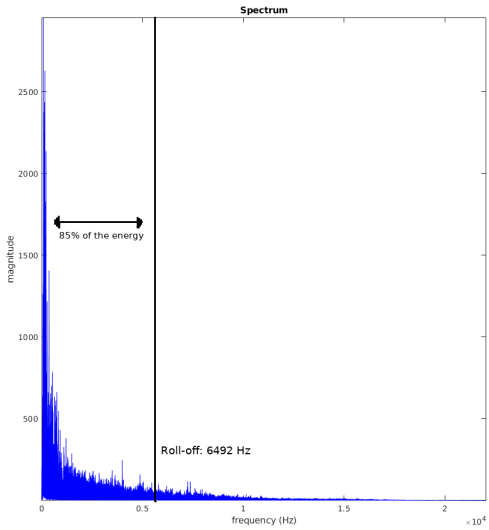

Figure 2.3: Spectral roll-off of an audio signal

Spectral roll-off is a timbral feature that accounts for the amount of high frequency in the signal.

Parameters: Roll-off ratio of 85%[TC02], 25ms frame size, with 50%.

2.2.1.4 Spectral Flatness

The spectral flatness corresponds to the geometric mean of the power spectrum divided by its arithmetic mean. If the power spectrum is divided in N bins, let, $\forall n \in [1, N], x(n)$ be the power spectrum magnitude for the bin n, the spectral flatness is:

$$\frac{\sqrt[N]{\prod_{n=0}^{N-1} x(n)}}{\frac{\sum_{n=0}^{N-1} x(n)}{N}}$$

It is measured in decibels, and is used to measure how noise-like the signal is, compared to being tone-like: high spectral flatness indicates that the signal

sounds like white noise (same power for each frequency band), while low spectral flatness indicates that the signal is tonal, in the sense that spectral power is concentrated in a small number of frequency bins.

Parameters: 25ms frame size, with 50% overlap.

2.2.1.5 Mel-frequency cepstral coefficients

The Mel-frequency cepstrum is a description of the *spectral shape* of a sound, on a non-linear mel scale. A mel scale is a scale that takes into account human sound perception: it is supposed to approximate the human auditory system better than a simple linear scale.
The cepstrum coefficients are usually obtained by performing the following steps:

- Compute the Fourier transform of the signal
- Transpose the corresponding power spectrum on a mel scale
- Compute the discrete cosine transform of mel log powers
- Take the amplitude of the resulting spectrum as the MFCCs

Using the MFCCs usually yields to a better representation of a sound for music information retrieval purposes than simple spectrum coefficients, since it takes into account the human auditory system.

Parameters: 40 bands used in the mel-band spectrum decomposition (MIRtoolbox default), 13 coefficients computed (excluding the "0^{th}" coefficient that just corresponds to the total spectral power of the signal, not its shape). The music similarity literature uses a very different numbers of coefficients among different paper: it ranges from 5 coefficients used[TC02] to 13[BSW+11] up to 20[MME05] and more. The choice of using 13 coefficients has been made after comparing the performances of the MFCC alone on a k-nn genre classification evaluation (see section 6.2 for more details). Even though genre classification is not the purpose of the tool, it can still be used to approximate the tool's performance; the comparison can be seen fig. 2.4, with the classification score ranging from 0 (no genre was guessed correctly) to 1 (every genre was guessed correctly).
It is clear that adding more coefficients will not drastically improve the tool's accuracy, and that 13 coefficients offer a good performance trade-off.

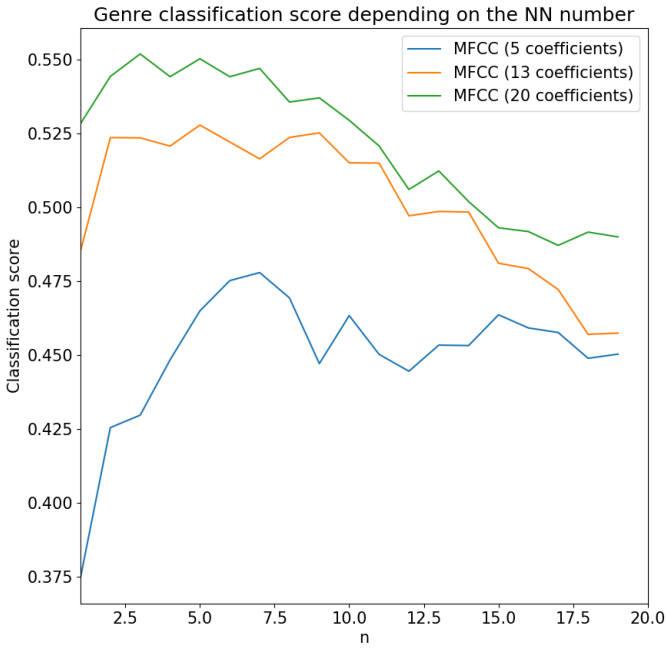

Figure 2.4: Performance of genre classification for different numbers of MFCC coefficients

2.2.2 Temporal features

2.2.2.1 Tempo (Beats per Minute)

Tempo is a broad topic in musicology in general, but a rough definition could be that tempo is the "speed" or the pace of a song.
One way to estimate the tempo of a track is to compute its number of beats per minute (BPM). It would be the number of times per minute a listener would intuitively tap his feet due to the rythm of the song.
The entire description of the default MIRtoolbox tempo extraction algorithm is too complicated for the scope of this study, but a small overview is still needed to understand the main parameters.
The audio signal is separated along different frequencies bands using a filter bank. These filtered signals, combined with the envelope of the original audio signal are then (after some pre-processing steps) used to compute event curves,

on which periodicity is detected using standard autocorrelation. Framing can optionally be performed on this event curve before autocorrelation computation. The autocorrelation bands are then "summarized", and peak-picking is then performed to compute the final BPM score. (The bands can also be summed before the autocorrelation computation).
The MIRToolbox has another, more precise, metre-based algorithm for computing BPM, using a hierarchical metrical structure. It was not used here because it required 5 to 15 seconds for computing the tempo of a single song, compared to between 0.5 and 2 seconds for the default event curve/autocorrelation algorithm.

Parameters: The default MIRToolbox parameters give a good BPM estimation and are used on music similarity papers[FMD14]: the default filter bank is a Gammatone filter bank decomposition in 10 bands with a lowest frequency of 50 Hz. The frame decomposition (used for feature summarization in section 2.3) of the detection curve has a frame length of 3 s and a hop factor of 10%. The tempo can range between 40 BPM and 200 BPM.

2.2.2.2 Beat loudness[BSW+11]

Beat loudness is used in order to have more information than just the beat periodicity computed above 2.2.2.1.
One way to compute beat loudness is to compute the event curve of a song (using `mirevents`), and then gather the peak values of the framed onsets. These values are then summarized using the standard method in section 2.3.

Parameters: Like tempo, a long enough frame size of 3 seconds is needed to have time to capture onsets, with 10% overlap.

2.2.3 Tonal features

2.2.3.1 Chromagram

The chromagram, also called HPCP, captures some melodic characteristics of music by showing the distribution of energy long pitch classes. A pitch class is the set of frequency/pitches that are an octave apart; for example, the pitch class B is the set B_0, B_1, B_2, etc.
The output of the `michromagram` function on an audio file is the distribution of the magnitude along the 12 pitch classes. It is shown in fig 2.5

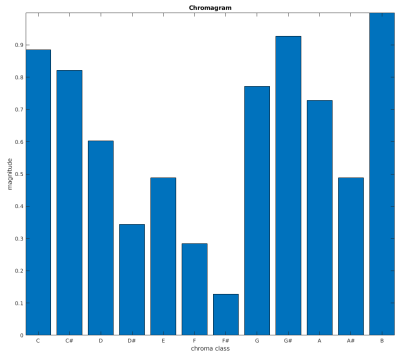

Figure 2.5: Chromagram of an audio signal

It is computed by first taking the spectrum of the audio signal on a logarithmic scale selecting only the 20 highest dB, restricting the frequency range to one that covers an integer number of octaves. The audio waveform is normalized before computing the FFT. The chromagram is then obtained by outputting the distribution of the spectrum energy along the different pitches.

Parameters: Defaults options are kept, with the minimum frequency range being 100Hz and the maximum 5000Hz (the maximum can be extended to get an integer number of octaves, though). The frequency associated to chroma C is 261.6z.

2.3 Feature summarization and storage

Features are computed using sliding windows, outputting a numerical value for each step of the window; thus, there is a need to summarize them, to avoid having to compare a too great number of values for each song, and store them efficiently as well.
Several methods exist to summarize feature sets over a song. The most common ones, used by [TC02] [FMD14], use statistical tools such as the mean and the standard deviation. Other papers also suggest using percentiles, such as the median [SWP10], or even Gaussian Mixture Models.
To keep the feature summarization simple, though, for each feature, only 5 values will be extracted from the block-level values: the mean, the standard deviation, the median, the 25^{th} percentile and the 75^{th} percentile, except for the MFCC, for which only the 13 first coefficients will be used without any other

form of summarization.
Once computed, the features will then be stored on a simple Python shelve, associating for each filename a 53-values feature vector, to avoid having to re-compute the features for each audio file - which would be quite long, since it took almost 3 days to compute the features for the 14k songs dataset.

Chapter 3

Web Survey

3.1 Survey Description

3.1.1 Survey Objectives

As previously said in chapters 1 and 2, one of the ways to tune the features weights is needed in order to maximize the performances of the algorithm. Though it would be perfectly possible to tune the tool using for example genre classification, which would avoid having to gather user input, such training is limited due to the fact that music separation in different genres is a very blurry concept; some even say that this concept is outdated and suggest to use other, new classifications[GKS+16].
Thus, and because this tool is most particularly aimed at the listener's comfort, asking for user input and using it to tweak the tool's parameters is a better fit than learning through genres or other "objective" concept.

Lots of methods exist to gather user feedback in music information retrieval; the most common ones are people rating the level of similarity of two songs. People can also make playlists that they feel are smooth out of different songs, or they can finally point the odd song out of several tracks. Each method has its pros and cons.

First, rating the similarity level between two songs can be done using either a Likert scale (discrete scale with labeled choices, such as "not similar at all", "a bit similar", "somehow similar", etc) or a percentage scale, on which the user rates from 0 to 100 the similarity between the songs (0 being not similar at all and 100 meaning the same song). But both of these scales have their flaws: the percentage scale, because it takes some time for the user to get used to it, as they need a "calibration" time to output coherent results (as at first they don't know how similar the songs are going to be). It also yields a *consistency* problem, as two people taking the survey are not likely to put the exact same result, thus no order relationship can be used with the filled data. The Likert scale solves this problem, but has the disadvantage of needing a lot of results from the survey if there are many songs to rate: to somehow be able to rate all pairs, a single user would have to answer $\binom{2}{1000} = 499500$ rounds.
Second, making the user create playlists that feel smooth from a finite set of songs is an interesting idea, but it has some issues as well. First, the user doesn't necessarily know all the songs, so they have to listen to all of them before being able to make any choice. If the user has to make a playlist of 5 songs out of 20 songs available and a seed song, given that the songs are 30-seconds excerpts, they have to listen to $25 \times 30s = 12$ minutes 30 seconds before even starting to sort them. But the real problem of this survey method is that the user has to more or less remember all the songs in order to be able to *select* and *sort* them in a somehow "smooth" order. Since the brain can hold about seven items at the same time[Mil56], this playlist-making task seems indeed difficult, and even more if you factor in the fact that a music piece is complex, and might be considered as more than one item to remember.
Finally, the user can also be asked to point the odd song out of a 3-songs selection, and repeat that task for some rounds, with different songs each time. This test is known as the *triangle* discrimination test. It is about as quick to do as the paired comparison above (the user just has to listen to one more song), and offers more statistical power[BS 04], i.e. less tests are required to roughly obtain the same amount of information as the paired comparison. It is though more prone to problems if the survey is poorly designed, as the candidates might tend to point the odd-one-out because an item has a property that the two others don't have, but the property is completely unrelated to the survey's purpose. It can range from simple unconscious things such as the labelling of the examples (candidates might be more tempted to chose the C answer because it is associated to bad results in their minds) as more audio-related issues, such as the loudness of songs that are not supposed to be considered in this study, but can heavily influence the candidate if they are strongly hesitating between two songs. These issues, though, can easily be solved, by, for example, randomizing the song orders and perform audio loudness normalization beforehand.

For all of these reasons, the triangle discrimination test will be used, but the survey will be designed with extra care to avoid unintentional bias from the

users' side.

3.1.2 Survey characteristics

The goal of the survey, as mentioned above, is to collect user input on music similarity, to tune the music similarity algorithm through the use of learning algorithms. Estimating first the number of people that will eventually answer the survey is a good way of scaling it and its features. Given the fact that this report is a master's thesis and is very limited in time, the survey has maximum three months to run, which lets little time to spread the word, limiting the number of people answering the survey to around 40. We can then compute the number of songs ratings that are possible: let us consider that for a round, the user has to listen to 3 thirty-seconds excerpts, and that it takes him thirty seconds to make up their mind. That makes $3 \times 30 + 30 = 120s$, two minutes per round. Let's limit the time per person to 30 minutes, around the maximum time someone is willing to spare for a non-critical survey. That makes 1800 seconds available, which is $\frac{1800}{120} = 15$ rounds per person. 15 rounds per person is $15 \times 3 = 45$ songs evaluated. If the objective of 40 people answering the survey is reached, that makes $45 \times 40 = 1800$ songs rated, i.e. almost 2000, which in turn makes around 700 total observations (rounds). This number of observations is also used in some papers on genre classification[HHK], so that will be roughly the number of song ratings considered needed for this report. With the number of observations available, the approximated number of features needed can be approximated.
Even if there is no rule of thumb for the number of features versus the number of observations, it can be safe to assume that `number of observation` $= 10^2 \times$ `number of features` will yield decent performances. Having around 700 observations for a total of almost 2000 songs evaluated makes around 8 to 10 features needed in total for the tool to have good performances. Following the survey methodology above, a rough estimation of the number of people needed to rate every song at least once can be done.

But having to find, handle and eventually reward a minimum of 20 to 40 people for conducting the actual study in a room needs too much organization and constraints for the scope of this paper. Hence, a web survey will be conducted instead, making it easier for people to participate, thus maximizing the potential of reaching more users and having more output.

The web survey also allows to ask users for demographics questions, link these results to the users' answer and cluster the answers depending on that more

easily than asking them face-to-face. Demographics are useful, because they allows post-processing of the obtained information in a more subtle fashion than just taking the raw results. An (exaggerated) example could be that asking for the person's age can explain why some people have strange results compared to the others: it might just be that they are old and their hearing is not so great. The demographics questions asked in this survey will be as follow:

- Age group:
  People in different age groups won't necessarily have the same musical sensibility, and some older people might even not know some musical genres (like Rap, which was created around the seventies)
- Music background:
  Ranging from Indifferent to Casual to Enthusiast to Savant[Jen06]
- Current time of the day:
  Grouped between Morning/Noon/Afternoon/Evening/Night, this captures some information that could be difficult for the person to formulate. It can indicate the mood, but more importantly the activity the user was most likely to be doing before answering the survey
- Music genres the user most listens to:
  People listening to, for example, rock, might not have the same sensibility towards rock tracks than people listening to classical music; they might focus on some very precise elements that the classical music listener would not pick up, and the opposite is true for classical music pieces.

3.1.3 Survey dataset

Now that the approximate number of needed songs is known, as well as the platform and the survey methodology, an actual song dataset is needed.
Lots of datasets specifically designed for music information retrieval exist, such as the GZTAN genre classification dataset, or the Million Song dataset[BmEWL11]. One of the problems with these datasets is that some of them (like the Million Song dataset) only contain already pre-extracted features metadata and not the actual song, which is not usable in the scope of this study, aiming above all at extracting features from the raw audio signal. The other main issue is that these datasets contain well-known copyrighted songs, and streaming them for a web survey without the proper rights is extremely difficult.
Thus, the dataset chosen for this study is the Free Music Archive dataset[DBVB16], a curated set of songs extracted from the Free Music Archive. All the songs have the Creative Commons Attribution 4.0 International license (CC BY 4.0) which

means that anybody is free to share them and adapt them freely (i.e. reprocess them), under the attribution condition (credit has to be given). Because this is an actual dataset and not a simple collection of disparate songs, a pre-processing has been applied to make this collection uniform: metadata has been cleaned and processed to be well-formatted, and all songs are mp3-encoded, with a sampling rate of 44.1kHz and an average bit-rate of 263kbits/s. Four downloadable datasets are available, with the characteristics highlighted in the table 3.1.

Table 3.1: FMA Dataset summary

Dataset type	Features	Number of songs
Full	Full songs, unbalanced genres	∼106000
Large	30-seconds excerpts, unbalanced genres	∼106000
Medium	30-seconds excerpts, unbalanced genres	25000
Small	30-seconds excerpts, balanced genres	1000

The "Small" dataset seems perfect for this project, but the evaluation method will then lack a set of songs that was not used for the survey. Hence, the "Medium" FMA dataset will be used for both the survey and the evaluation.
There are 16 different top-level genres in the "Medium" dataset, but they are not balanced: indeed, "Easy-listening", for example, only appears in 23 songs. To have a genre-balanced survey dataset, and because this genre is the least standard of the 16, "Easy-listening" songs have not been included in the survey database, as well as "Old-Timer/Historic" and "Experimental", which are very broad, "Classical" and "Instrumental", which are out of the scope of this study, and "Spoken" songs, as only music will be considered in this report. That makes 10 genres left: Rock, Electronic, Hip-Hop, Folk, Pop, International, Jazz, Country, Soul-RnB, and Blues. Because only sub-level genres associated to each song are written in the files' tags, and not top-level genres, pre-processing is performed. It consists of performing a lookup on the dataset's genre list, and re-tag songs that only have sub-genres tags by their original root genre. This pre-processing is applied to the 25 000 songs, so that the evaluation can be performed consistently with the data obtained by the survey.

3.1.4 Incomplete Random Design

After the pre-processing is done, the song triplets could theoretically be used "as is", that is, each user having random triplets across rounds. Since the number of people answering the survey is very small, though, it would be better to have an "optimal" survey design, i.e. a design that maximizes the statistical power

of each round, as well as a design that minimizes user bias.
The survey was first started with, for each user, the same 10 first rounds of curated songs, and 5 rounds of random songs. The decision of making a thought-through incomplete random design was made after realizing that this first quick design attempt did not maximize at all statistical power, and the tuning algorithm seemed be very efficient when it came down to compute distances between songs that were the same as the songs in the 10 rounds, but was likely to lack efficiency for new songs (i.e. to overfit).
Still, this first collected input (around 10 people answered the survey when the rounds were designed like this) can be used in two ways: first, it can give a good insight on how people answer similarly for the same rounds, and second, it highlighted the need of a better design, described below.
Several options can be considered for rounds:

- Same rounds for every user
- Completely different rounds for every user
- A proportion of constant rounds for every user, along with a proportion of non-constant rounds across users

The first option has the advantage of being very useful to test user's coherence, so that it is very easy to see if there is a consensus on a rating, and in that case, to pick up outliers. Still, it has the huge drawback of only rating a very short amount of songs - here, only 45, which is too little.
The second option has the drawback of having no redundancy between users, leading to somehow trust each user input as a ground truth, while it is not necessarily the case, as sometimes people can answer surveys randomly without thinking; nevertheless, it also offers the best song coverage. Finally, the third option combines the best of both worlds, as it offers redundancy to pick-up outliers, but also gives a better song coverage. Still, a user could have coherent results for the constant rounds, and results that don't make sense for rounds that are non-constant across people that answer the survey. Moreover, after many users answer the constant part, having more people answering the corresponding rounds doesn't offer much statistical power.
For these reasons, since the number of people answering the survey is very limited, and since solutions exist and will be introduced below to mitigate the lack of redundancy, **the rounds will be different for every user**.

Similarly, the songs can be picked in different ways:

- Rounds are made of completely random songs

- Rounds are made of curated (picked by the survey's author) songs
- Songs are neither curated nor chosen at random, but in some other way

Taking completely random songs is interesting in that that all of the dataset's songs could be potentially hit. It is probably a very good idea if the number of people answering the survey was very large, since it would mean that at some point all of the songs would be hit, and probably not only once. Nevertheless, since this present report doesn't aim at having millions of people answering the survey, this option will be discarded.
The second option was the one used in the first iteration of the survey. Having a curated set of songs per rounds is very enticing, since it seems to allow a logical and controlled set of songs, that would maximize the statistical power of the survey. For example, the surveyor can try to find songs that seem very similar to them and put them in the same rounds, to test if there is a consensus among the surveyees, and/or very dissimilar songs, to again find if there is consensus or not. The problem with this method is that it is very subjective, and the final tuned algorithm will be very much tailored to the person that hand-made the survey and their notion of "music similarity". For this reason, this method will not be used either.
What will be used instead is a sort of mix between these two methods. For each round, a random genre will be picked among the dataset's, with equal probability for each genre. A random song from this genre will then be selected. The two other songs will respectively be a song from a similar and a song from a dissimilar genre, as the learning from each round is very limited if the user perceives that the songs are very similar or very dissimilar, and can't make up his mind on the odd-one-out.
Of course, there is no certitude that the result from this design will be an universal user-driven similarity function, but the amount of information each round yields is thereby optimized.
The "similar genre" and "dissimilar genre" concepts have been explained in some papers, such as [SCBG08] and [OL15]. These works also give a genre distance/similarity matrix. Since what is needed is not a precise distance matrix result but only a binary 10x10 genre matrix, where 1 meant similar and 0 dissimilar, [SCBG08] was used as a baseline for a precise similarity matrix, and these results where completed by [OL15] since two genres were merged as one in the first paper.
From this similarity matrix, the binary matrix was generated by putting ones on columns where the similarity figure was higher than a precise figure, and zeros otherwise. This figure has to be positive so that similarity is meaningful, and is the first positive figure that makes each genre have at least one other similar genre and one other dissimilar genre.
The resulting matrix is displayed fig. 3.2.
This approach has to be taken with a grain of salt though: the concept of

"genres" is not clearly defined, and genres are usually not as complete intrinsic property that songs have, but rather as a concept to facilitate communication between humans. Still, given that the use of genres here is only to select songs that are roughly similar and roughly dissimilar, a binary genre similarity matrix is more than enough.

Table 3.2: Binary genre similarity matrix.
"Elec." stands for "Electronic", "Inter." for "International" and "Soul" for "Soul-RnB".

Genres	Folk	Country	Elec.	Rock	Pop	Jazz	Hip-Hop	Soul	Blues	Inter.
Folk	0	1	0	0	0	0	0	0	0	1
Country	1	0	0	0	0	0	0	0	0	0
Elec.	0	0	0	0	0	1	0	1	0	0
Rock	0	1	0	0	0	0	1	0	1	0
Pop	0	0	0	0	0	0	1	0	1	0
Jazz	0	0	1	0	0	0	0	1	1	1
Hip-Hop	0	0	0	0	0	0	0	0	0	1
Soul	0	0	1	0	0	1	0	0	0	0
Blues	0	0	0	0	0	1	0	0	0	0
Inter.	1	0	0	0	0	1	1	0	0	0

3.2 Survey Implementation and Technical details

3.2.1 Survey workflow

The web survey is made of three general parts:

- Presentation (introduction fig. 3.1 and end fig. 3.2)
- Demographics fig. 3.3
- Audio discrimination test itself fig. 3.4

The workflow is pretty simple: when somebody connects to `https://survey.lelele.io`, they are prompted with the introduction page displayed fig. 3.1. This page describes the task that they will have to complete (which is summarized again on the main survey page), along with licensing information about the dataset. To go to the next page, the user has to tick a box, there for legal

reasons[ano], specifying that they agree on the anonymous storage of their answers.
Once the user clicks "Begin!" they are required to fill the demographics questions defined in 3.1.2 and shown in fig. 3.3. All of these questions are mandatory, and if the person forgets to fill one of the item and submits, he is redirected to the same demographics page with a message telling him to fill the incriminated field(s).
If all is correct, the user is then directed to the actual audio survey page displayed fig. 3.4. They are shown a short text explaining that they will have to listen to three musical excerpts, and rate the one that feels the less similar to the other ones. Below that explanatory text, the three audio fields are displayed, and the user can play the 30-seconds songs at will, until they make up their mind. Once they do, they select the radio button corresponding to his choice, and click "next". This procedure is then repeated 15 times, for a total of 15 steps (the user has the number of remaining steps shown on the screen as well). Finally, after the 15 steps, the survey is completed. The user sees the end screen, fig. 3.2, and has the possibility to either close the tab and end the survey for good, or proceed to the "unlimited mode". This mode looks exactly like the one in fig. 3.4, except that the steps are not shown (because the user can do as many ratings as they like), and all the songs are selected at random.

3.2.2 Survey technical details

The survey is hosted at `https://survey.lelele.io`, is running on a dedicated server with an Intel Atom N2800 processor with 2 cores @ 1.86 GHz, 4GB of RAM and a 100 Mbps up connection. The machine runs Archlinux, a Linux distribution.
It uses Flask, a Python micro-framework, to render the web pages. Gunicorn is used as WSGI HTTP server, and passes the rendered Python code to nginx, an HTTP server and reverse proxy. This is summed up fig. 3.5.
In order to store user input, the survey uses PostgreSQL, a relational database management system, as it has a very good integration with Flask through the use of SQLAlchemy, more specifically Flask-SQLAlchemy that allows easy manipulation of SQL through the use of a more "pythonic" syntax.
The classes defined in the schema file that will further create the corresponding SQL database tables are defined in fig 3.6.
There are two tables: first, a "person" table, that represents a person that answered the survey through the fields "id", "age", "music_background", "time_of_day" and "genres". This column is created each time someone answers the "demographics" part, and is populated with the corresponding answer.
The second table stores the answers to the survey. The fields "song1", "song2", and "song3" store the file names of the three songs that were displayed to the

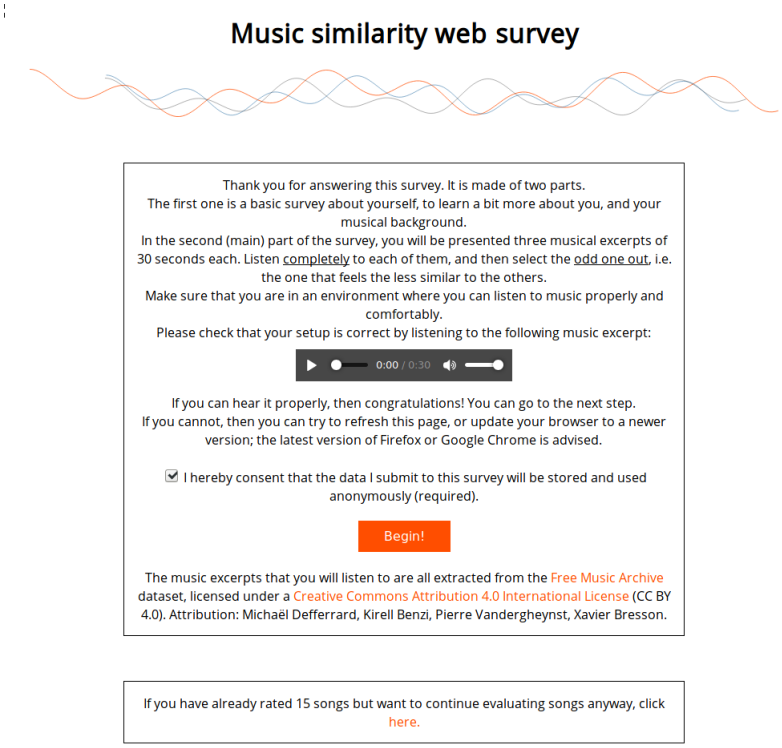

Figure 3.1: Introduction screen

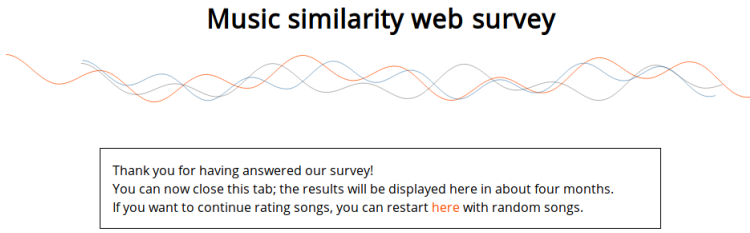

Figure 3.2: End screen

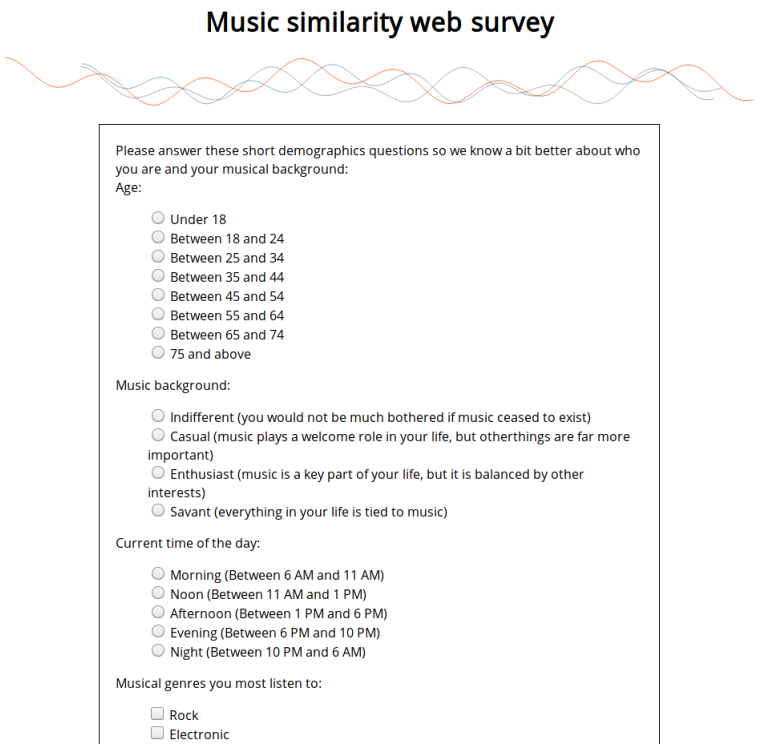

Figure 3.3: Demographics screen

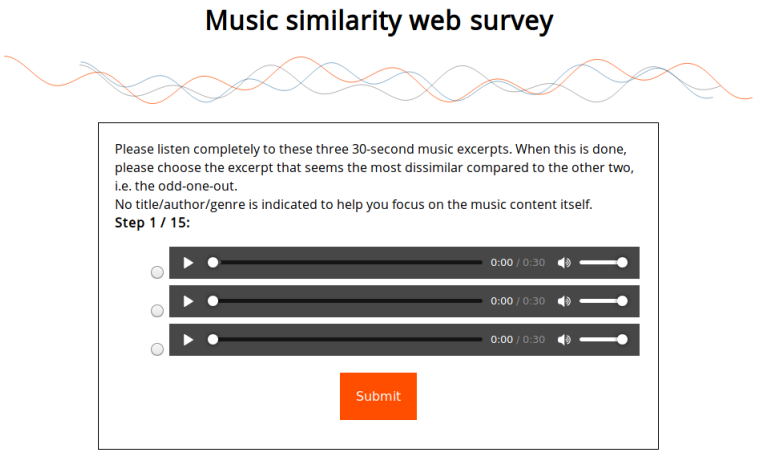

Figure 3.4: Global survey screen

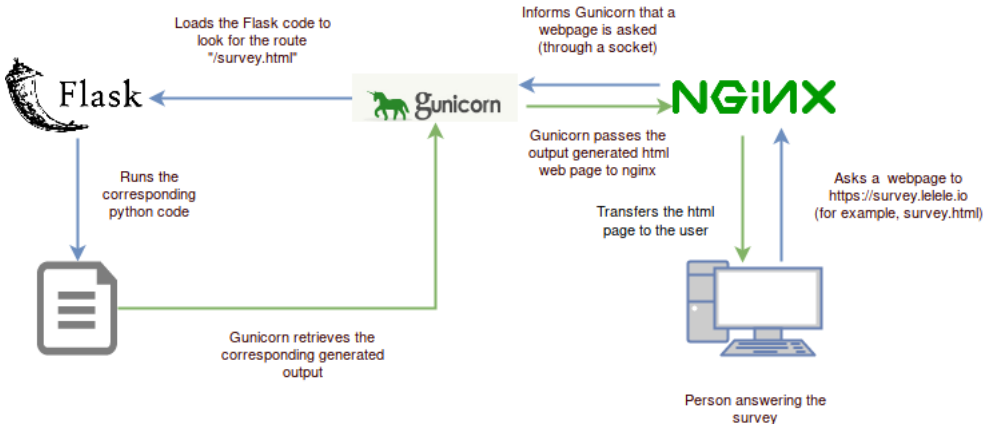

Figure 3.5: nginx/Gunicorn/Flask workflow

user during a round, "picked_song" is the song the user chose as the odd-one-out (it can either be song1, song2 or song3) and "person_id" is a foreign key that stores the ID of the user that answered the question.

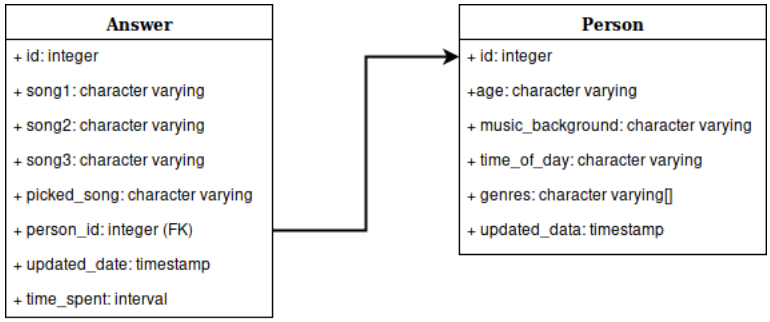

Figure 3.6: PostgreSQL tables layout

Timestamps are also stored in the fields "updated_date" for each anwser; they are set to `datetime.utcnow()` each time the corresponding column is updated, to be able to further process the answers when the survey will be done. Furthermore, the "answer" table also contains a field "time_spent" that captures the time the user spent answering the corresponding answer; it is the interval computed between the timestamp at which the person answered the question and the other most recent answer's timestamp created by this user.
The code made for creating the database is available in appendix B.1.

Flask is used to produce the HTML/CSS to give the user. Each url (`/demographis`,

`/survey`, `/index`, etc) is associated to a *route*, i.e. a python function, that will generate the corresponding web page through the use of the `render_template` function. The `render_template` function uses a Jinja2 html template as a mandatory argument (the web page skeleton), and takes objects as arguments that can be included in the template and be shown to the user.
For example, the main survey page's `render_template` takes the three evaluated songs as arguments, as well as the form used to prompt user input and the number of steps already done to show the user how many steps are left.
Flask comes with a `session` module, that is used in our case to determine which user answered which question, and is extremely useful in case several users are answering the survey at the same time. In the Python code, each user has a private `session` dictionary, that can hold custom values. It is used to store the user's id, which is the same as the one used in the database, so that when a person comes back and answers again the survey, their answers are edited and not added on top of the others. It also detects if the user already filled the survey, and if it is so, it shows them the link to the limitless version of the survey, on which they can rate as many triplets as they like. The session `dict` also holds other helper values, such as forms throughout the request.
These values are stored encrypted in the server, and an encrypted cookie, unique to each person's web browser is used to determine session attribution. It means that two different users under the same IP address but not the same computer can answer the survey and their answers will be stored independently.

3.3 Survey results summary

The survey results are extracted from the server using `pg_dump` and further analyzed through SQLAlchemy. The results are as follows:
A total of 55 people answered the survey. Since some people didn't answer all of the survey, a total of 601 rounds has been done; a little more than 500 have been done with the second incomplete random design.
The average time per round is 1 minute 15 second, and it has been computed by discarding answers that took more than 10 minutes (since it's likely that the users did something else in the meantime and came back to the survey afterwards - the highest time interval is for example a day!)

The first users answered the first survey design, where 10 rounds were common. This allowed to compute the consistency between users, which was computed as $\frac{\text{number of answers that are the same}}{\text{total number of answers}}$. The result obtained was 0.40, i.e. 40%, on 80 rounds. This shows little coherence between users, but can also be accounted for by the difficulty of answering the survey reported by the users with this design, and the small sample size.

A summary of the results of the demographics part of the survey are shown fig. 3.7.

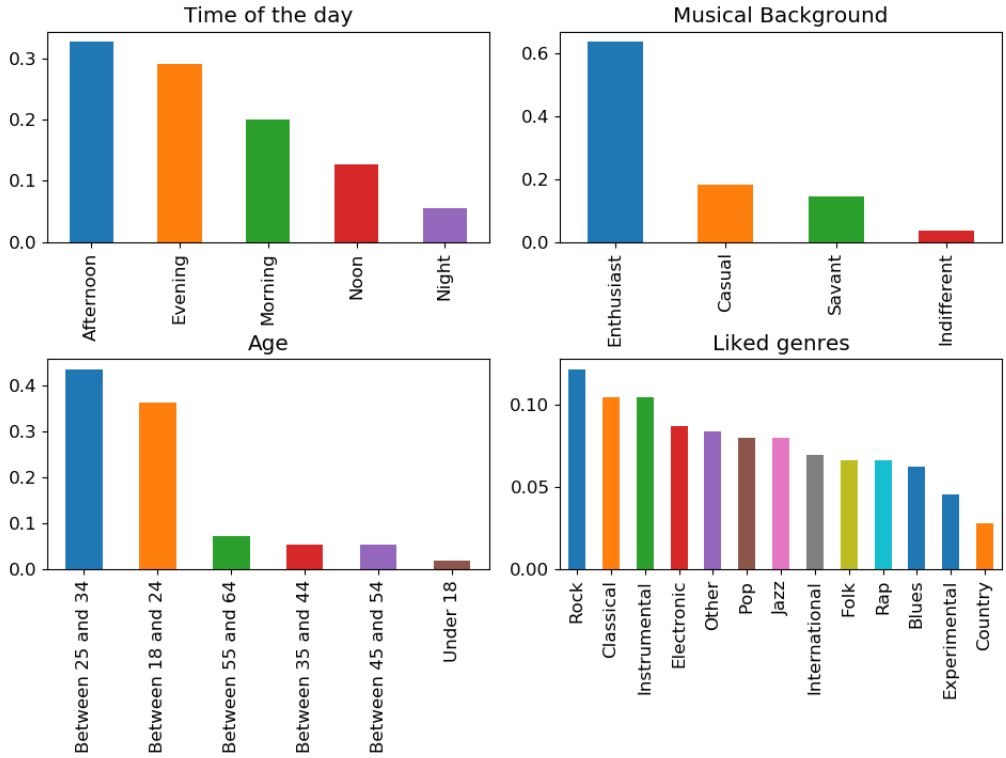

Figure 3.7: Demographics results

Briefly, what can be seen is that out of around fifty people, more than half of them answered during the day (mostly during the afternoon or the evening), probably people that came back from work and/or studies, that more than half of the people consider themselves as music enthusiasts, which makes sense for people answering a 30-minutes survey on music similarity, that half of them are between 18 and 34, and that people listen to a wide array of genres - the least listened to genre being Country.

The actual usable survey results will be further discussed in part 4.

Chapter 4

Use of survey: metric learning

4.1 Survey of the state of the art

Before working on the actual learning algorithm, it is good to remind the reader of the data format that the survey ran in chapter 3 provides.
People who answered the survey had to choose the odd song out from a list of three songs. The collected data is then training triplets of songs (x_i, x_j, x_k) on which a distance relationship is known. Let x_k be the picked-out song, then it means that, with d being an arbitrary distance metric, the relationships $d(x_i, x_j) < d(xi, xk)$ and $d(x_i, x_j) < d(xj, xk)$ ought to be preserved.

This problem - finding a distance metric that fulfills a certain set of conditions - is called *metric learning*. This topic has received some attention, and many methods are available to do what is called *weakly-supervised* metric-learning[BHS13].
The distance metric used in most metric-learning problems is the *Mahalanobis distance* (or generalized quadratic distance):
Let M be a symmetric positive semi-definite matrix of size $n \times n$, x_i and x_j two vectors of $\mathbb{R}^n$, the Mahalanobis distance between x_i and x_j is defined as:

$$d_M(x_i, x_j) = \sqrt{(x_i - x_j)^T M (x_i - x_j)}$$

Since this is a distance metric, it obeys the following properties, with $(x_i, x_j, x_k) \in \mathbb{R}^n$:

$$d(x_i, x_i) = 0, d(x_i, x_j) = d(x_j, x_i), d(x_i, x_j) \geq 0, d(x_i, x_j) + d(x_j, x_k) \geq d(x_i, x_k)$$

This metric is going to be used for the metric-learning part of this report: the objective is to find a matrix M that satisfies all of the problem's relative constraints, expressed as follows[BHS13]:

$$\mathcal{R} = \{(x_i, x_j, x_k) : x_i \text{ should be more similar to } x_j \text{ than to } x_k\}$$

If the problem has must-link/cannot-link constraints (e.g. if $\mathcal{X}$ is the examples space, it is known that $\forall (i, j) \in \mathcal{X}$, x_i and x_j should be similar or x_i and x_j should be dissimilar), then many algorithms exist to find a suitable M matrix, such as the Large Margin Nearest Neighbors (LMNN)[WBS06].

Unfortunately, the only training data available for this report are the training triplets, which only encapsulates relative constraints, and not must-link/cannot-link constraints, nor classes/labels. Thus, most methods used for finding the optimal M matrix cannot be used for this problem, making the possibilities scarce.

Therefore, two possibilities were then considered for obtaining the metric's best parameters.
The first one comes from [SJ03], that offers to solve the problem with the M matrix being diagonal (it also offers to use kernels, but for simplicity's sake, only diagonal matrices will be considered).
The paper shows that finding an optimal diagonal $n \times n$ matrix (where n is the number of features) W so that $\forall (i, j, k) \in \mathcal{R}, d_W(x_i, x_k) - d_W(x_i, x_j) > 0$ is the same as solving the following optimization problem, with w being W's diagonal coefficients, and $\forall (i, j, k) \in \mathcal{R}, \xi_{i,j,k}$ are slack parameters that account for constraints that cannot be satisfied:

$$\min_w \quad \frac{1}{2} \sum_{i=1}^{n} w_i^2 + \sum_{i,j,k \in \mathcal{R}} \xi_{i,j,k}$$

$$s.t. \quad \forall (i, j, k) \in \mathcal{R} : w^T (\Delta^{x_i - x_k} - \Delta^{x_i - x_j}) \geq 1 - \xi_{i,j,k}$$

$$\forall i \in [\![1, n]\!] \quad w_i \geq 0$$

$$\forall (i, j, k) \in \mathcal{R}, \quad \xi_{i,j,k} \geq 0$$

Where $\Delta^{x_i - x_k} = (x_i - x_k) \circ (x_i - x_k)$ and $\circ$ denotes the Hadamar product, or element-wise product.

This can be seen as a constrained convex quadratic program (as the function to be minimized is convex, and quadratic), that falls under the Karush–Kuhn–Tucker regularity conditions. Indeed, the constraint function $g_{i,j,k}(w) = -w^T(\Delta^{x_i - x_k} - \Delta^{x_i - x_j} + 1 - \xi_{i,j,k} \leq 0$ is an affine function of w, and all the functions are continuously differentiable, thus the function $f(w) = \frac{1}{2}\sum_{i=1}^{n} w_i^2 + \sum_{i,j,k \in \mathcal{R}} \xi_{i,j,k}$ to minimize has an optimal solution.
Though, after some testing using the Python solver CVXOPT, it showed that, using the survey's data, the system's matrix was singular and thus it could not be solved. This might be due to the fact that some constraints are contradictory, and since that even with the slack parameters, the aim is a global optimum satisfying all the constraints, it is not possible to find it. Indeed for example, the equation $\exists x \in \mathbb{R} f(x) = 0$ s.t. $x > 0$ and $x < 0$ doesn't have any solution, no matter f.

The second possibility for solving this optimization problem is to use a result from [Uhr15], and transform the constrained minimization problem in a simple minimization problem without constraints.
To do so, it is important to notice that, since the matrix M has to be positive semi-definite, it can be rewritten as LL^T. Finding M then means finding a suitable L matrix.
To find L, for each triplets (x_i, x_j, x_k), where x_k is the rejected element, there are two constrained relationships that appear: $d_L(x_i, x_j) < d_L(x_i, x_k)$ and $d_L(x_i, x_j) < d_L(x_j, x_k)$. Instead of just applying these constraints to the global minimization problem, they will be included in it, through the use of a cost function that will penalize distances that are not coherent with the answers. The cumulative Gaussian distribution Φ will be used as the cost function, that gives the probability of observing a given answer by a user (with σ being a fixed value):

$$C(L, x_i, x_j, \sigma) = \Phi\left(\frac{d_L(x_k, x_i) - d_L(x_i, x_j)}{\sigma^2}\right)$$

Indeed, if $d_L(x_k, x_i) > d_L(x_i, x_j)$, which it should be, then there difference will be positive and Φ will get close to one. Instead, if $d_L(x_k, x_i) < d_L(x_i, x_j)$, which should not happen with a well-computed L, then Φ will get close to zero probability of seeing this answer.
The final function to minimize is then the negative log likelihood function:

$$f(L, x_i, x_j, x_k, \sigma) = -\log\left(\prod_{i,j,k \in \mathcal{R}} C(L, x_i, x_j, x_k, \sigma)\right)$$

Which is, computationally:

$$f(L, x_i, x_j, x_k, \sigma) = -\sum_{i,j,k \in \mathcal{R}} \log\left(C(L, x_i, x_j, x_k, \sigma)\right)$$

This minimization on L can then simply be carried out by a solver, for example using Scipy.

4.2 Minimization in practice

Before starting the minimization, features are first normalized with zero-mean and unity variance, by the source code given in appendix B.2.
This minimization can then be carried out using Scipy's `minimize` function. This method needs the function to minimize as an argument, along with an initial value and the derivative of the function with respects to every variable, and outputs the found minimum vector (the L matrix is converted to a vector using numpy's `.ravel()` method). The initial L matrix is chosen as the identity matrix, as the distance metric then consists of the Euclidean distance. The differentiation details are available in appendix C.
The differentiation results are tested using Scipy's `check_grad` method, which takes a function and its gradient as an input, and outputs the error between the finite difference approximation of the gradient function and the gradient function given as input: the error has to be close to zero, otherwise the gradient function is most likely to be incorrect. In the present case, every tested sub-function had a gradient error of about 10^{-08}, except the global optimization function f, which had an error between the estimated gradient and the actual gradient of about 10^{-03}, which might seem too much at the first glance, but it is explained by the fact that f is made of sums, which means that all the gradient's error would add up to make a slightly higher error score, but that still means that the differentiation formula is the right one.
The source code for the optimization helpers is available in appendix B.3 and B.4.

The optimization is carried out using data from the survey, filtered so that only answers that were given in more than 20 seconds are taken into account, since it is roughly the minimum time to actually be able to listen to parts of the excerpts and give a judgment about it; that leaves 472 answers to use.
A first method to check whether the optimization leads to usable results is to run it on the full set of answers, and see the number of distances that were successfully preserved. And indeed, this first test performs well, with around 98% of distances preserved.

This is only a first rough evaluation, though. After evaluating it on testing/-training subsets of answers, the mean percentage of answers preserved was disappointing, with around less than 40% of distances preserved.
This is probably due to overfitting; to avoid this problem, regularization is per-

formed, using L2 norm.
Regularization consists in adding a *regularization term* to the global optimization function: the global minimization function then becomes:

$$g(L, x_i, x_j, x_k, \sigma, \lambda) = f(L, x_i, x_j, x_k, \sigma) + \lambda \, ||L||$$

Cross-validation is then used to find the λ parameter. The set of answers is divided into a design set and a test set, the test set being 20% of the global set. The goal of the design set is to find the best λ possible. To do so, a list of possible λ is made, and tests are run as follows on the design set: the design set is split into 5 folds, and for each λ, the optimization is ran on the 4 folds, and the number of distance constraints respected is computed for the last fold.
To make sure luck was not involved into a particularly good test result, the accuracy for each lambda is set to be the mean of all the accuracy results found by running the process on each fold.

Since the optimization function outputs values around 1000, the regression parameter will be set between 10^{-02} and 5000, and since optimization runs can take a long time (up to five hours for high lambda parameters), the following lambda parameters will be tested: $[10^{-02}, 10^{-01}, 1, 10, 50, 100, 500, 1000, 5000]$.
The source code for regularization is available in appendix B.5.
After running the cross-validation process, the following mean accuracies were found, and shown fig. 4.1

Table 4.1: Accuracy as a function of λ

λ	Mean accuracy percentage
10^{-02}	37.6%
10^{-01}	37.2%
1	36.6%
10	39.6%
50	38.8%
100	39.2%
500	38.8%
1000	41.0%
5000	39.7%

The mean accuracy for the euclidean method (no training) is 40.2%. The best mean accuracy is 41.0%, for $\lambda = 1000$: this parameter will thus be chosen for regularization.

After training the metric on the whole design set using $\lambda = 1000$, the number

of distances preserved is 44.0%, against 42.0% for a non-trained metric. The accuracy is slightly better using a metric trained after the survey rather than a simple euclidean distance, but not by far.
This is probably due to the lack of coherence between people's answers. After asking some people who answered the survey what they thought of it, many of them found it very difficult due to the fact some tracks sounded like almost pure noise and not music, and it was not an easy task to compare these tracks. This could not be easily avoided though, due to the nature of the dataset and the huge number of tracks that made manual checking a very difficult task (and would introduce a bias that is not wanted, as described in part 3.1.4).

Now that the trained music similarity system is built, the next logical step is to actually generate playlists with the new distance workflow shown fig. 4.1.

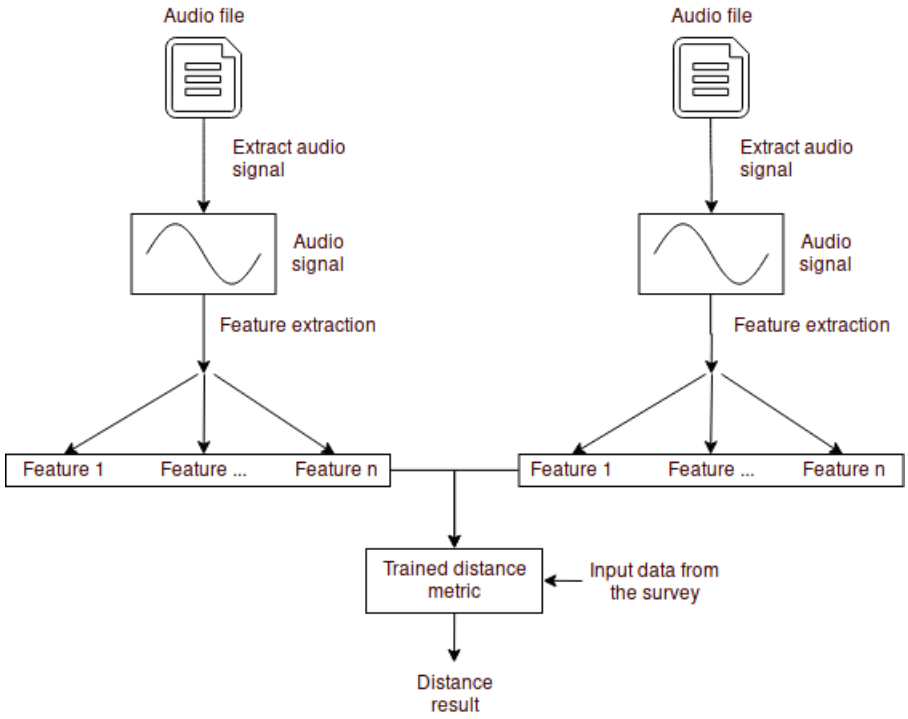

Figure 4.1: Distance workflow using the trained metric

Chapter 5

Playlist generation

In this report, a playlist will have the accepted "music playlist" meaning: simply an ordered list of songs that the user can play.
There are several, more or less advanced ways to generate playlists: the user can make the playlist themselves (manual playlist generation), they can make a random playlist out of songs in their music library (shuffle mode) - this way of generating playlist is mainly only useful if the user have their own song library, since it is very impractical on streaming services -, they can pick a seed song or artist, and let an algorithm give them songs close to the seed song or the artist's songs, or, in the case of an online system/social media, an algorithm can make a playlist out of their neighbour's musical tastes[Her08].
Since no third-party data is available, this part will focus on playlist generation made out of the user's own music library.

5.1 Playlist generation objectives

There are many different - and sometimes opposite - objectives for playlist generation. Some are:

- Allow the user discover music

- Make a "smooth" playlist (as defined in chapter 1) the user will enjoy
- Both goals - make a smooth playlist in which the user will discover music

The first goal implies that the user has access to music they do not know yet, which is not in the scope of this study, which focuses on offline, user-owned music libraries.
For that reason, allowing the user to discover music is a task that can hardly be achieved by this playlist generation algorithm, which will focus instead of making coherent playlists from a seed song (that can be input by the user or the system itself), that aims at maximizing user happiness.
Since "user happiness" is a too general goal that can hardly be easily fulfilled via programming, the playlist generation will instead focus on song coherence, as described in 1: a good playlist is a playlist thankks which transitions are not too rough, which does not necessarily mean that all the playlist's songs have to be similar to the seed song, but rather that, if there is a shift of music style, it has to be done smoothly, without the user noticing too harshly.
This way of considering playlist generation is supposed to make playlists the user will be happy with; this statement will be evaluated in chapter 6.

5.2 Playlist generation algorithm

Many different playlists generation algorithms exist in the literature, ranging from very simple song-to-song hoping methods to more complicated feedback-related or constrained models.
Indeed, some papers describe playlist generation systems that take into account user feedback: skipped songs, songs the user liked, etc. For instance, using user input as a "feedback loop" has been tested in [Pam06].
Another method is proposed in [DBSK12a], in which a playlist is created dynamically using a "pheromone" system: each time a song is played, a pheromone (a simple value) is associated to all the songs from the same artist, making these songs less likely to be played since the music similarity computation takes this pheromone into account; the pheromone disappears after some iterations. This is only applied to artists, but this could also factor in other user feedback and other metadata.

In the case of this paper, though, user feedback is considered to be taken into account through the use of the trained metric. Thus, the playlist generation will be much simpler than those described above, and no scalability methods such as the ones used in [jAP02] (since the tool should work on human-sized libraries, as described in chapter 1) nor constrained playlists generation methods (allowing

for example to choose the first, the last and some songs in the middle) such as the ones presented in [AT01] and [Vos] will be used.

Two methods have been envisioned for the playlist generation algorithm, both relying on seed songs, i.e. a song picked by the user to begin with and from which the playlist will be derived. This song could also be automatically chosen by an algorithm given the time of day/current user's mood guess, but for simplicity's sake, this will be let as a future potential improvement.
The first method simply consists in taking the first N closest (in respect to some metric) songs to the seed song. It has the advantages of being very efficient computationally, as the only thing to be computed is the pairwise distance matrix, then sort the seed song's column by ascending distances and take the first N corresponding indices. The drawback is that this method doesn't guarantee the smoothness of the playlist, as shown on the two-dimensional example fig. 5.1, where each song is represented by its plan coordinates. The playlist, with $N = 3$, made of the closest songs to the seed song will be [Song 1; Song 2; Song 3], but just because they are close to the seed song doesn't necessarily mean they are close together, and indeed, fig. 5.1, they are as far from one another as possible, meaning the user might experience some rough transitions.

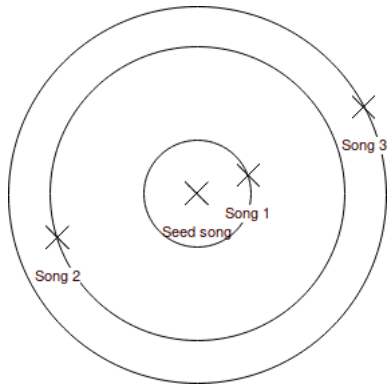

Figure 5.1: Worst-case scenario for method 1

The second method can make smoother playlists: to make a N songs playlist from a seed song S, instead of taking the N closest songs to the seed song like in the first method, the closest song $S1$ to S is taken as the first song. Then the closest song to $S1$ is taken as the second song, and all the playlist is generated that way.
More formally, let S_n be the n^{th} song in the playlist, $\mathcal{S}$ the pool of songs that haven't been chosen yet (updated by removing S_n at each iteration), and $\forall j \in$

$\mathcal{S}, s_j$ a song in the song pool, the playlist $\mathcal{P}$ is made like this:

$$\mathcal{P} = \begin{cases} S_0 = \text{Seed song (user input)} \\ n \in [\![1, N]\!]\, S_n = \arg\max_{j \in \mathcal{S}} d(S_n, s_j) \end{cases}$$

The pros of this method is that it will give smoother playlists than the first one; the cons are that it might give *too smooth* results, and lead to a playlist the user might consider as "boring".
To avoid this, what can be done is generating an $m \times N$ playlist, with m an integer, and then only take a song every m songs. This is shown fig. 5.2 for $m = 2$ and $N = 3$:

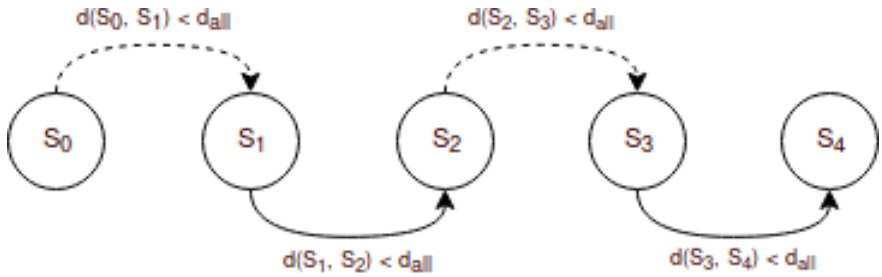

Figure 5.2: Playlist making example for method 2 with $m = 2$ and $N = 3$

The obtained playlist is then $\mathcal{P} = \{S_0, S_2, S_4\}$.
The m number could be deduced in relation to the user's library size and/or with the user's library distance standard deviation. For simplicity purposes for this report, though, playlists will be made only with $m = 1$ (the default case).

Chapter 6

Evaluation

The current section will evaluate both the untrained and trained distance metric's accuracy.
The fact that "music similarity" is a vague concept also has an impact on the evaluation: to better capture the different means "music similarity accuracy" can convey, the evaluation will be divided in three parts.
The first one, "Preliminary figures" will give some first figures about the evaluation: the number of songs in the dataset used, some computed statistics with the different metrics, etc. The second part, "objective evaluation", will give insight on how the dataset performs on genre classification, a task that is always associated with music similarity as a mean to evaluate "objectively" how well a music retrieval algorithm performs: it is for example used as an evaluation method for the well-known Music Information Retrieval Evaluation eXchange, or MIREX. Some papers prefer to predict a song's release year for the objective evaluation[FMD14], but in any case, an objective evaluation is a first rough step to see how well the algorithm is performing.
The last part, "subjective evaluation", is based on user feedback on playlists; it is the last and most valuable piece of feedback of this chapter.

6.1 Preliminary figures

Two disjoints subsets of the Free Music Archive are going to be used for the evaluation.
The first one, that will be called the "survey dataset" is simply made of all the songs that were used for the web survey, i.e. 1162 songs. The other one, made of the FMA complement of these songs, will be called "the complementary dataset" and is made of 14787 songs (only the songs that had the tags considered during the survey have been kept).
A good first way to evaluate how well a music similarity system performs is to measure the mean distance between all songs of a dataset, and the mean distance between songs in a cluster in which they are supposed to be closer[LS01]. Usually this is done by computing the mean distance between songs in the dataset, and songs from the same album, since songs from the same album come from the same recording and are usually the same style, so they are supposedly closer than songs from the whole dataset.
Since the notion of "album" is not really embedded into the FMA dataset, instead the mean distance between every song and songs from the same genres was compared on the complementary dataset. The results are shown fig. 6.1.

Table 6.1: Statistics of the distance measure

Distance/Method	Average distance between all the songs	Average distance between songs of the same genre
Computed features Euclidean Distance	9.89	9.05
Computed features Trained Mahalanobis distance	1.21×10^{-07}	1.06×10^{-07}

The average distance between all the songs is indeed greater than the average distance between songs of the same genre. Still, the difference between the two is not so great. That can be explained by the fact that, as said in section 6.2, the tagging in the FMA dataset is sometimes not so thoroughly executed, and some songs are tagged on genres they don't seem to belong to, even for really large genre definitions: thus, this can lead to some shifts in the mean distance in respect of what could be expected.

6.2 Objective Evaluation

To further evaluate the algorithm, genre classification on the complementary dataset is performed.

Since the final goal of this study is to learn a metric that would lead to making playlists people would like, nothing guarantees that because the algorithm performs well on a genre classification task, it will perform well at making playlists people like, and vice-versa. Still, even if genre classification is an often-contested task, it is widely used to roughly evaluate how a music similarity algorithm performs [SWP10].

The genre classification task will be done on the 14787 songs of the complementary dataset, among 10 genres: Folk, Country, Electronic, Rock, Pop, Jazz, Hip-Hop, Soul-RnB, Blues, International, the genres chosen for the survey in chapter 3. Since the FMA dataset doesn't have balanced genres, the genres will be unbalanced, instead of restricting the evaluation to the 100 songs or so the "Soul-RnB" genre have.
The k-nearest neighbors algorithm will be used to cluster the genres, with k ranging from 1 to 20. The genre classification will be performed on:

- Features generated at random for each song, and Euclidean distance
- Features computed as in 2 and Euclidean distance
- Features computed as in 2 and Mahalanobis distance fully trained in 4
- Features computed by a third-party software, Musly, with Euclidean distance

Musly is an open-source music similarity software that will be used to evaluate how well the system performs against real-world systems. For simplicity and temporal reasons, only a small part of Musly's features will be used for the genre classification task. Musly's performance are greatly enhanced by the use of the Kullback-Leibler divergence on the full set of features, so the performances measured on this test are not the full performances of the tool.
The source code for this evaluation is available in appendix B.7.

The results of performing the genre classification task with the k-NN algorithm are shown fig. 6.1, with n being the number of nearest neighbors for the k-NN algorithm, and the accuracy score ranging from 0 (no genre correctly classified) to 1 (everything correctly classified).

While the results are undoubtedly better than random features, the general accuracy increase still seems pretty low compared to what has been achieved before [TC02], with little to no difference between the trained distance and the euclidean distance, and between the trained distance and Musly. There can be two explanations to this.
First, both tools are not genre classification tools, but music similarity tools,

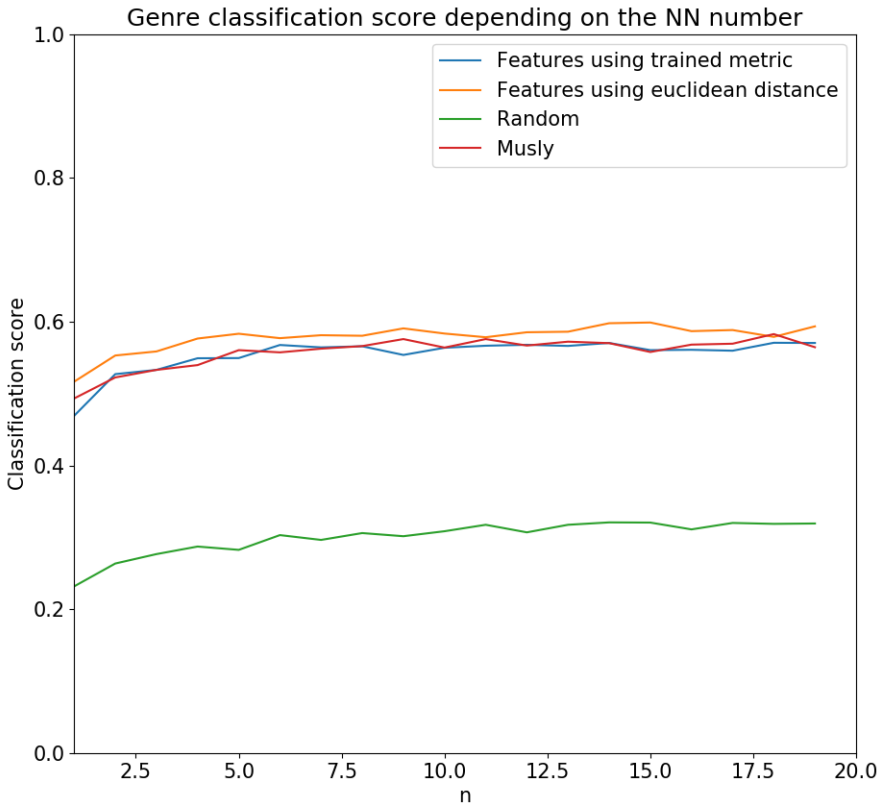

Figure 6.1: Genre classification on 14k songs

and the distance metric hasn't been trained to specifically achieve good performances on genre classification, as said above.
Second, the dataset's tagging seems to be sometimes off; for example, this song is considered as jazz, even though most people would consider that as electronic/experimental music.

To try and see if that was really the reason, a second (informal) genre classification task was done on a subset of this report's author's hand-tagged music library, made of a thousand commercial songs, and a hundred songs per genre. Even if there is improvement, it would not necessarily mean anything, since the second dataset's songs from the same genre could just sound more similar than

the first dataset's. The results are shown fig. 6.2, with the score still ranging from 0 to 1.

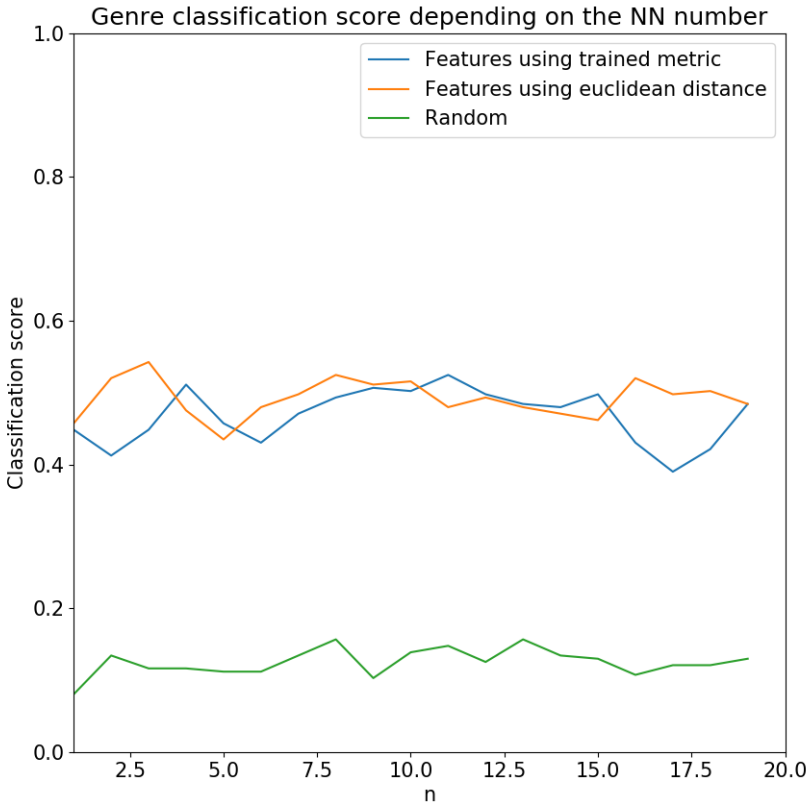

Figure 6.2: Genre classification on 1k songs

The accuracy is slightly better for the trained metric with this dataset, but nothing spectacular. This result would tend to rule out the second explanation above: the relatively low accuracy is probably just because the system built is not specifically aimed at evaluating genre similarity.

6.3 Subjective Evaluation

Since the entire system is built to generate playlists that people would qualify as "smooth", actual people have to evaluate it at some point. This section describes this "subjective" evaluation process.

The distance metric trained by the survey in chapter 4 is compared between a simple euclidean distance metric and random song picking, by making playlists out of the same seed songs chosen at random from the complementary dataset made of 14k songs. The playlist is made using the chain method described in section 5.2. Like in [BSW+11], there will be 4 common seed songs among users that evaluate the survey and 4 seed songs that will be different among users.

To compare these methods, people are asked to answer a survey. Each person listens to 5-songs playlist pairs from the same seed song, and rate the best one - each playlist being generated by a different method. The survey layout is shown fig. 6.3.

Figure 6.3: Evaluation Survey Round

As said above, 4 seed songs are common throughout the users, and 4 seed songs will be picked at random from the dataset, which makes 12 rounds of common seed songs and 12 rounds of random seed songs, so that every method permutation is evaluated.
Everything is of course randomized for each user: the orders in which the playlist pair (('method1', songs1), ('method2', songs2)) is shown is shuffled, and the playlist apparitions between each method are shuffled as well.
The source code for the evaluation's playlist generation is available in appendix B.6.

Once the evaluation is done, for each people and for each seed song, the winning method is extracted; if there are no winning method (because the results are incoherent, for example somebody picking playlist 1 over playlist 2, then playlist 3 over playlist 1, and playlist 2 over playlist 3) for a seed song, the result is simply ignored.
The experiment is run on 10 people. The percentage of time each method has been chosen as the best method is displayed fig. 6.2. The common seed songs statistics are also displayed fig. 6.2: the winning method is simply the one who was considered best by the most people.

Table 6.2: Subjective Evaluation Results

Method	% of times preferred (overall)	% of times elected winning method (for common seed songs)
Random	3.4%	0%
Euclidean (non-trained) distance	45.8%	50%
Trained distance	50.8%	50%

As can be seen in the evaluation results, the two engineered metrics performed better than the random playlist selection. The trained metric was preferred to the non-trained metric, even if both of them are on an equal ground for common songs.
After listening to the created playlists and to feedback from the users, though, the playlists seemed a bit off, and choosing between two playlists could sometimes be a daunting task, because both were really strange. This (relatively) poor performance can be explained in several ways.

First, the small number of examples (about 500 usable training triplets) makes it difficult to derive an "universal" distance metric between songs, since people don't have a lot of coherence together (40% of coherence for people rating same triplets). A higher number of examples would have allowed opposite answers to average out together a bit more, leading to better results (since the "true"

result for the majority of people would've stood out): this can be seen as early as 4.2, where the distance preservation for euclidean distance was roughly the same as the trained distance metric.
Second, the quality of the survey itself might have been a problem, due to the FMA's sometimes erratic tagging. This is already explained in 6.2, and could have cancelled a bit the advantages of the survey design by genre similarity explained in 3.1.4, leading to an even less important information gain from the survey.
Then, and this adds up to what has been said above and even after filtering out quick answers, maybe some people answering the first survey were speeders or straight-liners, making it even more difficult to derive the said universal metric with answers that didn't make sense.
Also, and this comment was made by people who answered the first survey as well (see 4.2), sometimes "music tracks" don't sound like music at all, but more like noise. It is especially true for music coming with the "Electronic" tag, but also for some very poorly recorded rock tracks. This might have confused the feature extraction algorithm presented in chapter 2, as these songs were sometimes found in the middle of a very coherent playlist.
Finally, the optimization algorithm only found local solutions, and maybe a better trained metric could be found by training the model with different initial conditions, a different regularization norm, etc.

Chapter 7

Conclusion

This thesis was aimed at creating a playlist building tool based on people feedback *a priori* to make playlist generation more universal, i.e. making playlists that as many people as possible will enjoy.
Achieving this was made through several steps:
First, feature selection and extraction were performed. A various range of audio features were used, such as timbral features, temporal features, and even tonal features. This, along with the use of feature summarization via mean, standard deviation, and percentiles, allowed to represent each song by its feature values, making them easy to store. Indeed, features have only to be computed once and are very small, offering a very good space-time compromise.
Simultaneously, a survey was ran during about three months, to collect data from people about music similarity, based on the Free Music Archive (FMA) dataset. This survey was answered by 55 people, and led to around 500 exploitable answers.
This data was then used for training the distance metric that derives the distance between two songs using the feature computation mentioned above. The metric learning process itself relied on minimizing a global regularized optimization function. It was then shown that using the trained metric resulted in slightly better accuracy at predicting people's music similarity choices than regular euclidean distance.
Playlists were then created using this trained metric. They were generated using a simple algorithm that took a seed song, and then built the playlist by

taking the closest song from the current song in the playlist. This relied on the assumption that the simpleness of the algorithm would be compensated by the fact that the distance metric used was trained.
Finally, several evaluations were conducted. The first evaluation, based on genre classification gave better results for the euclidean (non-trained) distance, due to the fact that genre is not necessarily correlated with music similarity, and that the song tagging tends to be a bit off for the FMA. The second evaluation was made by people that answered a survey on which they had to pick the best playlist, generated by two different methods from the same seed song. This evaluation method tended to favor the trained metric against the non-trained one, but not by an extremely large margin.

This lack of real difference between trained and non-trained metrics leads to some potential improvements.
First, a thing that has often been criticized during both surveys was the quality of the dataset's music, which has been qualified by almost all the people that took the surveys as "not even music half of the time". Instead of doing an online survey, perhaps it would have been better to gather some people and run the survey locally with commercial music, since it seemed that music quality had a very high impact on the answer of some people. Since the second survey was not conducted offline, it would perhaps have been better to use commercial music for this one instead of the FMA dataset.
The small amount of data made available by the survey was also problematic: the trained metric would probably become more and more efficient with as many examples as possible. If more training examples are to be obtained, the running time of the optimization algorithm to find the right metric would become problematic.
Thus, a second improvement would be to vectorize the optimization code a bit more, in order to run the optimization quicker. This would also help finding a better optimum for the optimization problem and tune the regularization parameter more finely, since it would allow for many more optimization runs.
Since obtaining more survey answers might be problematic, another way to tackle the issue would be to instead ask the user to answer the same survey but on their own music library, and train the metric with these answers. This would allow the training metric to be even more personalized, and probably more accurate for each user. Designing an audio player with this idea in mind, the default distance metric used could be the one trained with the global survey, and the user could replace it by "their own" metric by answering music similarity questions about their own music.
The Matlab feature extraction code could also be simplified and optimized, but instead of correcting the Matlab code, porting it to Python or C would probably be the way to go. Finally, another interesting path to take would be to use some meta-data/user data to choose the first seed song of the playlist, instead of asking the user to chose it. It could for example be based on the hour of the

day, and guess/ask the user their mood to find the most suitable starting song.

Appendix A

Matlab source code

Listing A.1: 'Matlab code for feature extraction'

```
songlist = 'songlist.txt';
framerate = 1/40; % 40 Hz of framerate
centroids = mircentroid(songlist, 'Frame', framerate);
centroids_data = get(centroids, 'Data');
centroids_mean = mean_cells(centroids_data);
centroids_std = sqrt(variance_cells(centroids_data));
centroids_Q1 = Q1_cells(centroids_data);
centroids_Q2 = Q2_cells(centroids_data);
centroids_Q3 = Q3_cells(centroids_data);

zcrs = mirzerocross(songlist, 'Frame', framerate);
zcrs_data = get(zcrs, 'Data');
zcrs_mean = mean_cells(zcrs_data);
zcrs_variance = sqrt(variance_cells(zcrs_data));
zcrs_Q1 = Q1_cells(zcrs_data);
zcrs_Q2 = Q2_cells(zcrs_data);
zcrs_Q3 = Q3_cells(zcrs_data);

rolloffs = mirrolloff(songlist, 'Frame', framerate);
rolloffs_data = get(rolloffs, 'Data');
rolloffs_mean = mean_cells(rolloffs_data);
```

```
rolloffs_std = sqrt(variance_cells(rolloffs_data));
rolloffs_Q1 = Q1_cells(rolloffs_data);
rolloffs_Q2 = Q2_cells(rolloffs_data);
rolloffs_Q3 = Q3_cells(rolloffs_data);

flatnesses = mirflatness(songlist, 'Frame', framerate);
flatnesses_data = get(flatnesses, 'Data');
flatnesses_mean = mean_cells(flatnesses_data);
flatnesses_std = sqrt(variance_cells(flatnesses_data));
flatness_Q1 = Q1_cells(flatnesses_data);
flatness_Q2 = Q2_cells(flatnesses_data);
flatness_Q3 = Q3_cells(flatnesses_data);

tempo = mirtempo(songlist, 'Frame');
tempos_data = get(tempo, 'Data');
tempos = tempo_mat(tempos_data);
tempos_mean = mean_cells(tempos_data);
tempos_std = sqrt(variance_cells(tempos_data));
tempos_Q1 = Q1_cells(tempos_data);
tempos_Q2 = Q2_cells(tempos_data);
tempos_Q3 = Q3_cells(tempos_data);

mfccs = mirmfcc(songlist, 'Rank', 1:13);
mfcc_data = get(mfccs, 'Data');
mfcc_val = [];
for k = 1:length(mfcc_data)
    mfcc_val = [mfcc_val; mfcc_data{k}{1}'];
end
mfcc_val = mfcc_val';

framed_onsets = mirevents(songlist, 'Frame', 3, 0.9);
peaks = get(framed_onsets, 'PeakVal');
onsets = create_cells(peaks);
mean_onsets = mean_onset_cells(onsets);
var_onsets = var_onset_cells(onsets);

chromagrams = mirchromagram(songlist);
chromagrams_data = get(chromagrams, 'Data');
chromagrams_val = [];

for k = 1:length(chromagrams_data)
    chromagrams_val = [chromagrams_val; chromagrams_data{k}{1}'];
end
chromagrams_val = chromagrams_val';
```

```
final_vector = [
    centroids_mean; centroids_std; centroids_Q1;
    centroids_Q2; centroids_Q3; zcrs_mean; zcrs_variance;
    zcrs_Q1; zcrs_Q2; zcrs_Q3; rolloffs_mean; rolloffs_std;
    rolloffs_Q1; rolloffs_Q2; rolloffs_Q3; flatnesses_mean;
    flatnesses_std; flatness_Q1; flatness_Q2; flatness_Q3;
    tempos; tempos_mean; tempos_std; tempos_Q1; tempos_Q2;
    tempos_Q3; mfcc_val; mean_onsets; var_onsets; chromagrams_val
];

function A = tempo_mat(cells)
    A = [];
    for k = 1:length(cells)
        A = [A, cells{k}{1}{1}];
    end
end

function A = var_onset_cells(cells)
    A = [];
    for k = 1:length(cells)
        A = [A, nanvar(cells{k})];
    end
end

function A = mean_onset_cells(cells)
    A = [];
    for k = 1:length(cells)
        A = [A, nanmean(cells{k})];
    end
end

function A = create_cells(cells)
    A = {};
    for k = 1:length(cells)
        A{end+1} = mean_frames_onsets(cells{k});
    end
end

function B = mean_frames_onsets(frames)
    B = [];
    for k = 1:length(frames{1})
        B = [B, nanmean(frames{1}{k})];
    end
```

```
end

function A = mfcc_mean(cells)
    A = [];
    for k = 1:length(cells)
        A = [A; nanmean(cells{k}{1}')];
    end
    A = A';
end

function A = mfcc_var(cells)
    A = [];
    for k = 1:length(cells)
        A = [A; nanvar(cells{k}{1}')];
    end
    A = A';
end

function A = Q1_cells(cells)
    A = [];
    for k = 1:length(cells)
        if iscell(cells{k}{1})
            A = [A, prctile(cell2mat(cells{k}{1}), 25)];
        else
            A = [A, prctile(cells{k}{1}, 25)];
        end
    end
end

function A = Q2_cells(cells)
    A = [];
    for k = 1:length(cells)
        if iscell(cells{k}{1})
            A = [A, prctile(cell2mat(cells{k}{1}), 50)];
        else
            A = [A, prctile(cells{k}{1}, 50)];
        end
    end
end

function A = Q3_cells(cells)
    A = [];
    for k = 1:length(cells)
        if iscell(cells{k}{1})
```

```
            A = [A, prctile(cell2mat(cells{k}{1}), 75)];
        else
            A = [A, prctile(cells{k}{1}, 75)];
        end
    end
end

function A = mean_cells(cells)
    A = [];
    for k = 1:length(cells)
        if iscell(cells{k}{1})
            A = [A, nanmean(cell2mat(cells{k}{1}))];
        else
            A = [A, nanmean(cells{k}{1})];
        end
    end
end

function A = variance_cells(cells)
    A = [];
    for k = 1:length(cells)
        if iscell(cells{k}{1})
            A = [A, nanvar(cell2mat(cells{k}{1}))];
        else
            A = [A, nanvar(cells{k}{1})];
        end
    end
end
```

Appendix B

Python source code

Listing B.1: 'PostgreSQL schema file'

```
import datetime
import os
import sys
from sqlalchemy import (
    Column, DateTime, ForeignKey, Integer, String, Interval, ARRAY,
)
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship
from sqlalchemy import create_engine

Base = declarative_base()

class Person(Base):
    __tablename__ = 'person'
    id = Column(Integer, primary_key=True)

    # Demographics
    age = Column(String)
    music_background = Column(String)
    time_of_day = Column(String)
    genres = Column(ARRAY(String))
```

```
    updated_date = Column(DateTime, default=datetime.datetime.utcnow)


class Answer(Base):
    __tablename__ = 'answer'
    id = Column(Integer, primary_key=True)

    song1 = Column(String)
    song2 = Column(String)
    song3 = Column(String)

    picked_song = Column(String)

    person_id = Column(Integer, ForeignKey('person.id'))
    person = relationship(Person)


    updated_date = Column(DateTime, default=datetime.datetime.utcnow)
    time_spent = Column(Interval)

engine = create_engine('postgresql://postgres@localhost/prod_db')
Base.metadata.create_all(engine)
```

Listing B.2: 'Feature normalization code'

```
import numpy as np
import shelve

features = np.genfromtxt('features_full.mat', delimiter=',')
features = np.nan_to_num(features)
features = np.transpose(features)
mu = features.mean(axis=0)
sigma = features.std(axis=0)
# Zero-mean + unity variance
features = (features - mu) / sigma

f = open('songlist_full.txt')
songs = f.readlines()
with shelve.open('features_normalized_songs_full.db') as sh:
    for feature, song in zip(features, songs):
        sh[song[2:-1]] = feature
```

Listing B.3: 'Optimization functions (distances)'

```
import numpy as np
from scipy.stats import norm
from scipy.spatial.distance import norm as L2_norm


def d_metric(x1, x2, L=None):
    return d(L, x1, x2)

def d(L, x1, x2):
    L = L.reshape(len(x1),len(x1))
    sqrd = ((x1 - x2).dot(L.dot(np.transpose(L)))).dot(x1 - x2)
    if (0 > sqrd) and (sqrd > -1e-10):
        print('Got negative value: {}'.format(sqrd))
        sqrd = np.abs(sqrd)
    ret = np.sqrt(sqrd)
    return ret


def grad_d(L, x1, x2):
    L = L.reshape(len(x1), len(x2))
    ret = grad_d_squared(L, x1, x2) / (2 * d(L, x1, x2))
    return ret


def grad_d_squared(L, x1, x2):
    L = L.reshape(len(x1), len(x1))
    grad = 2*np.outer(x1-x2, x1-x2).dot(L)
    return grad.ravel()


# x3 here is the odd thing
def delta(L, x1, x2, x3, sigma, second_batch=False):
    ret = (d(L, x2, x3) - d(L, x1, x2)) / sigma
    if second_batch:
        ret = (d(L, x1, x3) - d(L, x1, x2)) / sigma
    return ret


def grad_delta(L, x1, x2, x3, sigma, second_batch=False):
    ret = (grad_d(L, x2, x3) - grad_d(L, x1, x2)) / sigma
    if second_batch:
        ret = (grad_d(L, x1, x3) - grad_d(L, x1, x2)) / sigma
    return ret
```

```
def p(L, x1, x2, x3, sigma, second_batch=False):
    cdf = norm.cdf(delta(L, x1, x2, x3, sigma, second_batch))
    if cdf == 0:
        print(delta(L, x1, x2, x3, sigma, second_batch))
    return norm.cdf(delta(L, x1, x2, x3, sigma, second_batch))


def grad_p(L, x1, x2, x3, sigma, second_batch=False):
    return (
        norm.pdf(delta(L, x1, x2, x3, sigma, second_batch)) *
        grad_delta(L, x1, x2, x3, sigma, second_batch)
    )


def log_p(L, x1, x2, x3, sigma, second_batch=False):
    return np.log(p(L, x1, x2, x3, sigma, second_batch))


def grad_log_p(L, x1, x2, x3, sigma, second_batch=False):
    return (
        grad_p(L, x1, x2, x3, sigma, second_batch) /
        p(L, x1, x2, x3, sigma, second_batch)
    )


def opti_fun(L, X, sigma, l):
    batch_1 = -sum (
        np.array([
            log_p(L, x1, x2, x3, sigma)
            for x1, x2, x3 in X
        ])
    )
    batch_2 = -sum(
        np.array([
            log_p(L, x1, x2, x3, sigma, True)
            for x1, x2, x3 in X
        ])
    )
    return batch_1 + batch_2 + l * L2_norm(L)


def grad_opti_fun(L, X, sigma, l):
    batch_1 = (
```

```
        -np.sum(
            np.array([
                grad_log_p(L, x1, x2, x3, sigma)
                for x1, x2, x3 in X
            ]),
            0,
        )
    )
    batch_2 = (
        -np.sum(
            np.array([
                grad_log_p(L, x1, x2, x3, sigma, True)
                for x1, x2, x3 in X
            ]),
            0,
        )
    )
    return batch_1 + batch_2 + l * L / L2_norm(L)
```

Listing B.4: 'Optimization helpers'

```
import sys
sys.path.append('..')

from common.distances import (
    d,
    grad_d,
    grad_d_squared,
    delta,
    grad_delta,
    p,
    grad_p,
    log_p,
    grad_log_p,
    opti_fun,
    opti_fun_no_lambda,
    grad_opti_fun,
    grad_opti_fun_no_lambda,
)
import numpy as np
from scipy.optimize import approx_fprime, check_grad, minimize


def check_gradients(X):
```

```
    L0 = np.identity(len(X[0][0])).ravel()
    sigma2 = 2
    x1 = X[0][0]
    x2 = X[0][1]
    x3 = X[0][2]
    l = 1

    print(
        'Error in the distance gradient: {}'
        .format(check_grad(d, grad_d, L0, x1, x2))
    )
    print(
        'Error in the delta gradient: {}'
        .format(check_grad(delta, grad_delta, L0, x1, x2, x3, sigma2))
    )
    print(
        'Error in the cdf gradient: {}'.
        format(check_grad(p, grad_p, L0, x1, x2, x3, sigma2))
    )
    print(
        'Error in the log cdf gradient: {}'
        .format(check_grad(log_p, grad_log_p, L0, x1, x2, x3, sigma2))
    )
    print(
        'Error in the optimization function gradient for '
        'one example: {}'
        .format(check_grad(
            opti_fun, grad_opti_fun, L0, [[x1, x2, x3]], sigma2, 1,
        ))
    )
    print(
        'Sum of errors in the global optimization function gradient: {}'
        .format(check_grad(opti_fun, grad_opti_fun, L0, X, sigma2, 1))
    )


def percentage_preserved_distances(L, X):
    count = 0
    for x1, x2, x3 in X:
        d1 = d(L.ravel(), x1, x2) # short distance
        d2 = d(L.ravel(), x2, x3) # long distance
        d3 = d(L.ravel(), x1, x3) # long distance
        if (d1 < d2) and (d1 < d3):
            count = count + 1
```

```
    return count / len(X)

def optimize(L0, X, sigma2, l):
    l_dim = len(X[0][0])

    res = minimize(
        opti_fun,
        L0,
        args=(X, sigma2, l),
        jac=grad_opti_fun
    )
    L = np.reshape(res.x, [l_dim, l_dim])
    return (res.success, L)
```

Listing B.5: 'Regularization code'

```
import sys
sys.path.append('..')

from common.features_dict import song_features
from common.optimize import (
    optimize,
    percentage_preserved_distances,
)
from common.query import make_feature_triplets
from datetime import datetime
import numpy as np
from sklearn.model_selection import KFold, train_test_split


X = np.array(make_feature_triplets(song_features))
l_dim = len(X[0][0])
sigma2 = 2
L0 = np.identity(len(X[0][0])).ravel()

design, test = train_test_split(X, test_size=0.2)

lambdas = [0.01, 0.1, 1, 10, 100, 50, 100, 500, 1000, 5000]

accuracies = [[] for _ in lambdas]
accuracies_euclidean = []

print('Started {}'.format(datetime.now()))
```

```
kf = KFold(n_splits=5)
rounds = 0
for train_index, test_index in kf.split(X):
    rounds = rounds + 1
    print('Doing {}th fold ...'.format(rounds))
    X_train, X_test = X[train_index], X[test_index]
    for i, l in enumerate(lambdas):
        L = optimize(L0, X_train, sigma2, l)
        accuracy = percentage_preserved_distances(L, X_test)
        accuracies[i].append(accuracy)
        print(
            'Done for lambda = {}, accuracy is {}'
            .format(l, accuracy)
        )
    accuracies_euclidean.append(
        percentage_preserved_distances(L0, X_test)
    )
    print(
        'Euclidean accuracy is {}'
        .format(percentage_preserved_distances(L0, X_test))
    )


mean_accuracies = np.array(
    [
        np.mean(local_accuracies)
        for local_accuracies in accuracies
    ]
)

idx = mean_accuracies.argmax()
max_accuracy = mean_accuracies[idx]
l = lambdas[idx]
print(
    'Mean accuracy for euclidean is: {}'
    .format(np.mean(accuracies_euclidean))
)
print(
    'Best accuracy is {} for lambda = {}\n'
    .format(max_accuracy, l)
)

L = optimize(L0, design, sigma2, l)
```

```
print('At the end of the day:')
print(
    'Accuracy for non-trained metric on the test test: {}'
    .format(percentage_preserved_distances(L0, design))
)
print(
    'Accuracy for trained metric on the test test: {}'
    .format(percentage_preserved_distances(L, design))
)

L_total = optimize(L0, X, sigma2, 1)
np.save('L_total', L_total)
print('Done {}'.format(datetime.now()))
```

Listing B.6: 'Playlist making file for evaluation'

```
import sys
sys.path.append('..')

from common.features_dict import song_features_full_dataset, L
from common.distances import d_metric
from itertools import combinations
import numpy as np
import pickle
from random import sample, shuffle
from scipy.spatial.distance import pdist, squareform


def make_playlist(Y, seed_idx, playlist_length, ordered_songs):
    song_list = [ordered_songs[seed_idx]]
    for _ in range(playlist_length):
        next_idx = np.nanargmin(Y[seed_idx])
        Y[seed_idx] = np.nan*np.ones(len(Y[seed_idx]))
        np.transpose(Y)[seed_idx] = np.nan*np.ones(len(Y[seed_idx]))
        song_list.append(ordered_songs[next_idx])
        seed_idx = next_idx
    return song_list


playlist_length = 5
ordered_X = []
ordered_songs = []
for song in song_features_full_dataset.keys():
    ordered_songs.append(song)
```

```
    ordered_X.append(song_features_full_dataset[song])

M = np.dot(L, L.transpose())
Y_euclidean = squareform(pdist(ordered_X))
Y_trained = squareform(pdist(ordered_X, 'mahalanobis', VI=M))
np.fill_diagonal(Y_euclidean, np.nan)
np.fill_diagonal(Y_trained, np.nan)

nb_people = 20
block_rounds = []

seeds = [
    '150267.mp3',
    '140602.mp3',
    '149609.mp3',
    '021800.mp3',
]

blocks = []

del(song_features_full_dataset['150267.mp3'])
del(song_features_full_dataset['140602.mp3'])
del(song_features_full_dataset['149609.mp3'])
del(song_features_full_dataset['021800.mp3'])

for _ in range(nb_people):
    rounds = []
    temp_Y_trained = Y_trained.copy()
    temp_Y_euclidean = Y_euclidean.copy()
    round_seeds = (
        seeds +
        sample(list(song_features_full_dataset.keys()), 4)
    )
    shuffle(round_seeds)
    for seed in round_seeds:
        seed_idx = ordered_songs.index(seed)
        list_trained = make_playlist(
            temp_Y_trained,
            seed_idx,
            5,
            ordered_songs,
        )
        list_euclidean = make_playlist(
            temp_Y_euclidean,
```

```
                seed_idx,
                5,
                ordered_songs,
            )
            list_random = [seed, *np.array(ordered_songs)[
                np.random.randint(len(ordered_X), size=playlist_length)
            ]]
            combined_list = [
                [('euclidean'), list_euclidean],
                [('trained'), list_trained],
                [('random'), list_random],
            ]
            shuffle(combined_list)
            rounds = rounds + list(combinations(combined_list, 2))
        del(temp_Y_trained)
        del(temp_Y_euclidean)
        shuffle(rounds)
        blocks.append(rounds)

with open('blocks_eval_survey', 'wb') as f:
    pickle.dump(blocks, f)
```

Listing B.7: 'Genre evaluation'

```
#!/usr/bin/python3

import sys
sys.path.append('..')
from common.features_dict import (
    song_features,
    song_features_full_dataset,
    song_features_yolo,
    L,
)

import matplotlib
from matplotlib import pyplot as plt
import mutagen
import numpy as np
import os
import shelve
from sklearn import preprocessing
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
```

```
from sklearn.neighbors import KNeighborsClassifier


def genres_classification_evaluation(url, features_dicts, M):
    matplotlib.rcParams.update({'font.size': 15})
    genres = [
        'Folk',
        'Country',
        'Electronic',
        'Rock',
        'Pop',
        'Jazz',
        'Hip-Hop',
        'Soul-RnB',
        'Blues',
        'International',
    ]

    genres_files = {}
    file_list = np.sort([
        os.path.join(dp, f)
        for dp, dn, fn in os.walk(os.path.expanduser(url))
        for f in fn
    ])

    for song in file_list:
        try:
            tags = mutagen.File(song)
        except mutagen.MutagenError:
            continue
        genre = tags['TCON'].genres[0]
        if genre in genres:
            genres_files[song] = genre

    le_genres = preprocessing.LabelEncoder()
    le_genres.fit(list(genres))

    files = np.array(list(genres_files.keys()))
    le_files = preprocessing.LabelEncoder()
    le_files.fit(files)
    genres_corresp = np.array([genres_files[x] for x in files])
    y = le_genres.transform(genres_corresp)
    X = le_files.transform(files)
```

```
X_values_methods = []
for i, feature_dict in enumerate(features_dicts):
    if i != 3:
        X_values_methods.append(
            np.array(
                [
                    feature_dict[os.path.basename(song)]
                    for song in files
                ]
            )
        )
    else:
        X_values_methods.append(
            np.array(
                [
                    feature_dict[url+os.path.basename(song)]
                    for song in files
                ]
            )
        )

scores = []

axes = plt.gca()
axes.set_ylim([0, 1])
axes.set_xlim([1, 20])
axes.set_xlabel('n')
axes.set_ylabel('Classification score')
axes.set_title(
    'Genre classification score depending'
    ' on the NN number'
)

for i,X_values in enumerate(X_values_methods):
    print('Doing it for i = {}'.format(i))
    scores = []
    confusion_matrices = []
    for n in range(1,21):
        print('Doing it for n = {}'.format(n))
        local_scores = []
        local_confusion = []
        X_train, X_test, y_train, y_test = train_test_split(
            X_values,
            y,
```

```
                test_size=0.33,
            )

            neigh = KNeighborsClassifier(n_neighbors=n)
            if i == 0:
                neigh = KNeighborsClassifier(
                    n_neighbors=n,
                    metric='mahalanobis',
                    metric_params={'VI':M}
                )
            neigh.fit(X_train, y_train)

            local_scores.append(neigh.score(X_test, y_test))
            y_predict = neigh.predict(X_test)
            local_confusion.append(confusion_matrix(y_test, y_predict))
            scores.append(np.mean(local_scores))
            confusion_matrices.append(np.mean(local_confusion, axis=0))
        print(scores)
        plt.plot(scores)
        plt.legend([
            'Features using trained metric',
            'Features using euclidean distance',
            'Random',
            'Musly'
        ])
        max_accuracy_i = scores.index(max(scores))
        confusion_mat = confusion_matrices[max_accuracy_i]
    plt.show()


url = 'static/dataset/'
url = 'songs_without_dataset/'
musly = shelve.open('musly_all.db')
X0 = list(song_features_full_dataset.values())[0]
random_features = {
    song: np.random.rand(len(X0))
    for song in song_features_full_dataset.keys()
}
M = np.dot(L, L.transpose())

genres_classification_evaluation(
    url,
    [
        song_features_full_dataset,
```

```
        song_features_full_dataset,
        random_features,
        musly
    ],
    M,
)
```

Appendix C

Differentiation details

In order to minimize a function with multiple variables computationally, its jacobian has to be computed, i.e. the expression of all its partial derivatives. Here, with x_i, x_j, x_k being n-dimensional vectors, x_k being the odd song out's vector, sigma being fixed and L an n by n matrix, the objective function to minimize is:

$$f(x_i, x_j, x_k, L, \sigma) = -\sum_{i,j,k \in \mathcal{R}} \log\left(\Phi\left(\frac{d_L(x_k, x_i) - d(x_i, x_j)}{\sigma^2}\right)\right)$$

Instead of differentiating f for each one of its n^2 variables, the differentiation will be matrix-wise. With $d_L(x_i, x_j) = \sqrt{(x_i - x_j)^T L L^T (x_i - x_j)}$ and the nabla (∇) being the gradient operator, it yields, by [PP12] eq. 77:

$$\nabla d_L(x_i, x_j) = \frac{L^T (x_i - x_j)(x_i - x_j)^T}{d_L(x_i, x_j)}$$

Let Δ be the name of the distance difference:

$$\Delta(x_i, x_j, x_k, L, \sigma) = \frac{d_L(x_k, x_j) - d_L(x_i, x_j)}{\sigma^2}$$

Its jacobian is:

$$\nabla\Delta(x_i, x_j, x_k, L, \sigma) = \frac{\nabla d_L(x_k, x_j, L) - \nabla d_L(x_i, x_j)}{\sigma^2}$$

The cumulative distribution of this is then:

$$p(L, x_i, x_j, x_k, L, \sigma) = \Phi(\Delta(x_i, x_j, x_k, L))$$

The derivative of the cumulative distribution is, with ϕ being:

$$\nabla p(L, x_i, x_j, x_k, L, \sigma) = \phi(\Delta(x_i, x_j, x_k, L)) \times \nabla \Delta(x_i, x_j, x_k, L, sigma)$$

The derivative of the log is:

$$\nabla log p(x_i, x_j, x_k, L, \sigma) = \frac{\nabla p(x_i, x_j, x_k, L, \sigma)}{p(x_i, x_j, x_k, L, \sigma)}$$

So, the final derivative of the optimization function is:

$$\nabla f(x_i, x_j, x_k, L, \sigma) = -\sum_{i,j,k \in (R)} \frac{\nabla p(x_i, x_j, x_k, L, \sigma)}{p(x_i, x_j, x_k, L, \sigma)}$$

With regularization, the optimization function becomes, with $\|.\|$ the L2-norm:

$$g(x_i, x_j, x_k, L, \sigma, \lambda) = f(x_i, x_j, x_k, L, \sigma) + \lambda \|L\|$$

The derivative is then, by [PP12] eq. 129:

$$\nabla g(x_i, x_j, x_k, L, \sigma, \lambda) = \nabla f(x_i, x_j, x_k, L, \sigma) + \lambda \frac{L}{\|L\|}$$

Bibliography

[ano] Data storage and data security. `http://www.ethicsguidebook.ac.uk/Data-storage-and-data-security-308`.

[AT01] Masoud Alghoniemy and Ahmed H. Tewfik. A network flow model for playlist generation. In *In Proc IEEE Intl Conf Multimedia and Expo*, 2001.

[BFD15] John Ashley Burgoyne, Ichiro Fujinaga, and Stephen Downie. *A New Companion To Digital Humanities*. 2015.

[BHS13] Aurélien Bellet, Amaury Habrard, and Marc Sebban. A survey on metric learning for feature vectors and structured data. *CoRR*, abs/1306.6709, 2013.

[BKAB10] R.G. Bachu, S. Kopparthi, B. Adapa, and B.D. Barkana. Voiced/unvoiced decision for speech signals based on zero-crossing rate and energy. In Khaled Elleithy, editor, *Advanced Techniques in Computing Sciences and Software Engineering*, pages 279–282, Dordrecht, 2010. Springer Netherlands.

[BmEWL11] Thierry Bertin-mahieux, Daniel P. W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *In Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR*, 2011.

[BS 04] Sensory analysis – methodology – triangle test. Standard, British Standards Institution, June 2004.

[BSW+11] D. Bogdanov, J. Serra, N. Wack, P. Herrera, and X. Serra. Unifying low-level and high-level music similarity measures. *IEEE Transactions on Multimedia*, 13(4):687–701, Aug 2011.

[BSWH] Dmitry Bogdanov, Joan Serrà, Nicolas Wack, and Perfecto Herrera. Hybrid music similarity measure.

[CFS15] Keunwoo Choi, George Fazekas, and Mark B. Sandler. Understanding music playlists. *CoRR*, abs/1511.07004, 2015.

[CVG+08] M. A. Casey, R. Veltkamp, M. Goto, M. Leman, C. Rhodes, and M. Slaney. Content-based music information retrieval: Current directions and future challenges. *Proceedings of the IEEE*, 96(4):668–696, April 2008.

[DBSK12a] G. Dubey, K. K. Budhraja, A. Singh, and A. Khosla. User customized playlist generation based on music similarity. In *2012 NATIONAL CONFERENCE ON COMPUTING AND COMMUNICATION SYSTEMS*, pages 1–5, Nov 2012.

[DBSK12b] G. Dubey, K. K. Budhraja, A. Singh, and A. Khosla. User customized playlist generation based on music similarity. In *2012 NATIONAL CONFERENCE ON COMPUTING AND COMMUNICATION SYSTEMS*, pages 1–5, Nov 2012.

[DBVB16] M. Defferrard, K. Benzi, P. Vandergheynst, and X. Bresson. FMA: A Dataset For Music Analysis. *ArXiv e-prints*, December 2016.

[EzLSW15] Hamid Eghbal-zadeh, Bernhard Lehner, Markus Schedl, and Gerhard Widmer. I-vectors for timbre-based music similarity and music artist classification. In *ISMIR*, 2015.

[FLTZ11] Z. Fu, G. Lu, K. M. Ting, and D. Zhang. A survey of audio-based music classification and annotation. *IEEE Transactions on Multimedia*, 13(2):303–319, April 2011.

[FMD14] Peter Foster, Matthias Mauch, and Simon Dixon. Sequential complexity as a descriptor for musical similarity. *CoRR*, abs/1402.6926, 2014.

[GG78] John M. Grey and John W. Gordon. Perceptual effects of spectral modifications on musical timbres. *The Journal of the Acoustical Society of America*, 63(5):1493–1500, 1978.

[GKS+16] David M. Greenberg, Michal Kosinski, David J. Stillwell, Brian L. Monteiro, Daniel J. Levitin, and Peter J. Rentfrow. The song is you: Preferences for musical attribute dimensions reflect personality. *Social Psychological and Personality Science*, 7(6):597–605, 2016.

[GPD00] Fabien Gouyon, Francois Pachet, and Olivier Delerue. On the use of zero-crossing rate for an application of classification of percussive sounds. In *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00*, 2000.

[Gó06] E. Gómez. *Tonal Description of Music Audio Signals*. PhD thesis, Universitat Pompeu Fabra, 2006.

[Her08] Oscar Celma Herrada. *Music recommendation and discovery in the long tail*. PhD thesis, University Pompeu Fabra, 2008.

[HHK] Michael Haggblade, Yang Hong, and Kenny Kao. Music genre classification.

[ifp] An explosion in global music consumption supported by multiple platforms. `http://www.ifpi.org/facts-and-stats.php`. Accessed: 2018-02-08.

[jAP02] Jean julien Aucouturier and Francois Pachet. Scaling up music playlist generation. In *In Proceedings of the IEEE international conference on multimedia and expo (ICME 2002*, pages 105–108, 2002.

[Jen06] David Jennings. Groups and behaviour patterns among music listeners. `http://alchemi.co.uk/archives/mus/groups_and_beha.html`, 2006.

[KD] U. Kuzelewska and R. Ducki. Collaborative filtering recommender system in music recommendation.

[KS13] Peter Knees and Markus Schedl. Music similarity and retrieval, 2013.

[Ler12] Alexander Lerch. *An Introduction to Audio Content Analysis*. John Wiley and Sons, 2012.

[LN11] Adam J. Lonsdale and Adrian C. North. Why do we listen to music? a uses and gratifications analysis. *British Journal of Psychology*, 102(1):108–134, 2011.

[LS01] Beth Logan and Ariel Salomon. A music similarity function based on signal analysis, 2001.

[Mcl07] Mcld. Spectral centroid — Wikipedia, the free encyclopedia, 2007. [Online; accessed 28-March-2018].

[Mil56] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956.

[mir] 2016: Audio music similarity and retrieval. `http://www.music-ir.org/mirex/wiki/2016:Audio_Music_Similarity_and_Retrieval`. Accessed: 2018-02-11.

[MME05] Michael Mandel, Michael I. M, and Daniel Ellis. Song-level features and support vector machines for music classification, 2005.

[MPWE07] Michael Mandel and Daniel P W Ellis. Labrosa's audio music similarity and classification submissions. 01 2007.

[MRR] David Moffat, David Ronan, and Joshua D. Reiss. An evaluation of audio feature extraction toolboxes.

[Mus18] Nielsen Music. *2017 Year-End Music Report U.S.* 2018.

[OH05] B. Ong and Perfecto Herrera. Semantic segmentation of music audio contents. In *International Computer Music Conference*, 2005.

[OL15] Cian O'Brien and Alexander Lerch. Genre-specific key profiles. *41st International Computer Music Conference, Icmc 2015: Looking Back, Looking Forward - Proceedings*, pages 70–73, 2015.

[Pam06] Elias Pampalk. *Computational Models of Music Similarity and their Application in Music Information Retrieval*. PhD thesis, Technischen Universität Wien Fakultät für Informatik, 2006.

[PP12] K. B. Petersen and M. S. Pedersen. The matrix cookbook, nov 2012. Version 20121115.

[SCBG08] Mohamed Sordo, OÌ<81>scar Celma, MartiÌ<81>n Blech, and Enric Guaus. The quest for musical genres: Do the experts and the wisdom of crowds agree? *Ismir 2008 - 9th International Conference on Music Information Retrieval*, pages 255–260, 2008.

[Sha17] E. Shakirova. Collaborative filtering for music recommender system. In *2017 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 548–550, Feb 2017.

[SJ03] Matthew Schultz and Thorsten Joachims. Learning a distance metric from relative comparisons. 2003.

[SWP10] Klaus Seyerlehner, Gerhard Widmer, and Tim1 Pohle. Fusing block-level features for music similarity estimation. *13th International Conference on Digital Audio Effects, Dafx 2010 Proceedings*, September 2010.

[TC02] G. Tzanetakis and P. Cook. Musical genre classification of audio signals. *IEEE Transactions on Speech and Audio Processing*, 10(5):293–302, Jul 2002.

[TFS+12] Michael J. Terrell, György Fazekas, Andrew J. R. Simpson, Jordan Smith, and Simon Dixon. Listening level changes music similarity, 2012.

[Uhr15] Anders Kirk Uhrenholt. Recommendation system for sound libraries, 2015.

[Vos] M. P. H. Vossen. Master's thesis local search for automatic playlist generation.

[WBS06] Kilian Q. Weinberger, John Blitzer, and Lawrence K. Saul. Distance metric learning for large margin nearest neighbor classification. In *In NIPS*. MIT Press, 2006.

[YC18] Shingchern D. You and Ro Wei1 Chao. Music similarity evaluation based on onsets. *Lecture Notes in Electrical Engineering*, 422:153–163, 1 2018.