Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

My Music Statistics

Bachelor's Thesis

Sämy Zehnder

zehnders@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Samuel Welten
Prof. Dr. Roger Wattenhofer

Acknowledgements

While accomplishing this bachelor's thesis I got supported by many people which I would like to thank:

Samuel Welten for his great assistance and the many vivid discussions during this semester. Prof. Dr. Roger Wattenhofer for giving me the opportunity to work on this project. Sabine and my Dad for cross-reading and correcting this document. My family and friends for listening patiently to my stories about "Smart-Shuffle" and providing me with interesting inputs helping me finish this work. Luc, Sabine and Tobi to provide me with their music collection.

Abstract

Jukefox is a music player for Android. It provides the user with a Smart–Shuffle play-mode which tries to select songs he could like at the moment.

This thesis discusses the results of providing the user with statistics about his listening behaviour and gives some insights about common patterns appearing in many music collections. Moreover, the attempt to replace the existing Smart-Shuffle algorithm by one based on listening statistics of the user is documented.

Keywords: Jukefox, Statistics, Smart-Shuffle, Agent

Contents

Chapter 1

Introduction

Storing music on the computer and mobile devices got very popular in the last years. Since storage media got cheaper while the capacity increased more and more, the music collections became huge. Deleting out of favour music is not necessary any more. The result is that many people are overstrained with the decision what to listen to. Choosing random songs became popular.

Jukefox is a music player that addresses large music collections. One way the user can listen to music in Jukefox is by using Smart-Shuffle. This play-mode tries to support the user by detecting what songs he currently likes and which not. The next song to be played is then chosen at random from a reduced set of songs which is assumed to be liked by the user. A decrease of song skips is expected to result.

A key concept in Jukefox is that it is defined how close the various songs are to each other. All the music is placed in space – the world of music. Songs which are similar are assigned with coordinates that are nearby in that space. This allows to rub off knowledge of how much one likes a song to its neighbours which are likely to share the same popularity.

This thesis is about bringing statistics of the listening behaviour of the user to Jukefox. They are used to understand the user even more. In Chapter 3 some general properties of the world of music are explained before the concept of ratings is introduced. Chapter 4 describes a new Smart-Shuffle algorithm that is based on statistics.

While working on this thesis, some limitations of the transaction management of SQLite were detected. Chapter 5 talks about these problems and the way they were solved.

Chapter 2

Related work

It has always been a goal of computer science to simulate human behaviour and support users by providing software that 'thinks' like them. But understanding how humans act or defining their taste is very complicated. That is why many different music players which try to improve the users listening experience exist. Most of the available music players can be categorized into one of three different types.

Community-based systems try to provide users with good content created by others. In 8tracks[1] users are able to upload and publish their playlists. The quality of these lists is very high since they are actually created by humans. However, big interaction of users that provide the content is needed.

Content-based systems try to fetch similarity information of a songs type. They analyse the audio content of songs and search for similar songs in the collection. In [2] it is described how such features can be extracted and similarities are computed. Pandora[2] is an online music platform which uses the music genome project – a database about song descriptions that is created by experts – to find good proposals for a users current mood. These lists are typically very expensive to create at a high quality. Although the database that Pandora uses is not publicly accessible, there are projects aiming at creating open versions.[3]

Meta-data based systems use information that does not directly represent the acoustic contents of songs. Possible information can be the artist or genre of a song. There are some more sophisticated attempts such as Genius[4] that makes use of collaborative filtering. This is comparing different users taste. If two users agree about how much they like some songs, their opinion about songs which were not played yet is probably alike as well. Moreover Genius computes similarity info from purchases made in iTunes.[5] If a user purchases song A and B they are considered to be similar. Collaborative filtering systems are able to

[1] http://www.8tracks.com
[2] http://www.pandora.com
[3] See http://osmgp.pbworks.com
[4] http://www.apple.com/itunes/features/#genius
[5] http://www.apple.com/itunes/

return very high-quality results (See [1]). However, they need large data sets to find users that have comparable listening behaviour data. Working with that large datasets is not suitable for hardware constrained devices such as smartphones.
Jukefox also is a meta-data based system since it uses music similarity information from Last.fm.[6]

A lot of research about how to create good playlists is done by big entertainment companies without sharing their results. Moreover, most systems are calculating whole playlists on their servers. An attempt to dynamically extend playlists during listening to music is described in [6].

In [4, 5] it is explained how among others Reddit[7] and StumbleUpon[8] are using sophisticated approaches to determine the popularity of content. We are using a much easier rating function for songs to keep computational costs low. Moreover, our implementation allows to use weights for votes.

[6] http://www.last.fm
[7] http://www.reddit.com
[8] http://www.stumbleupon.com

Chapter 3

Statistics

The first part of this thesis was to collect statistics about the users listening behaviours. They will be used to implement a new Smart-Shuffle algorithm (described in Chapter 4) as well as giving the user an impression about his favourite music.

Ratings express how much a user likes a song. They are represented as numbers. In Jukefox a rating of '−1' describes a song[1] that is not liked at all while '1' stands for beloved ones. There are two types of ratings: *Explicit* ratings are directly assigned by the user. In many music players one is able to award songs with stars or points. Jukefox does not have explicit ratings but uses *implicit* ones. They are computed automatically. Whenever the user skips a song its rating gets slightly decreased while listening a song to its end increases it. Moreover, when changing the rating of a song, similar songs are rated as well.

This chapter describes how ratings are calculated in detail.

3.1 Distances between Songs

In Jukefox, every song is classified by a vector. This vector describes the songs position in the world of music. One can calculate the euclidean distance between the vectors of two songs to know how similar they are. Very similar songs with only very short distances to a fixed song are said to lie in its neighbourhood. We try to find the maximum distance that a neighbour is allowed to have to still count as very similar. We defined that in average every song should have a fixed number of similar songs (we chose 10). However, in different music collections the radius that ensures this property varies. If a collection is very specific (e.g. songs from one genre only) the radius is likely to be smaller than in a diversified one. This comes from the fact, that in the specific collection more songs are positioned

[1] Although the concepts in this chapter are mostly explained for songs only, they apply for songs, albums, artists or genres in the same way - just some grouping of data is done differently. We omit to write all of them whenever possible to make this document more readable.

in a given area in the space of music and therefore the average distance between two songs is much smaller.

However, calculating the exact radius is not possible since its calculation costs are polynomial in the number of songs and an in-time calculation is infeasible for a medium sized music collection. Finding an approximation requires some analysis of the music space which is described below. How the radius gets calculated is described in Section 3.2.2.

3.1.1 Distance Range

To get an impression of what is near and what is far in the space of music we analysed the distances between songs. We see that they have to lie within the range $[0, \sqrt{2}]$. The upper bound follows from the fact, that the euler distance is defined as

$$distance(p_1, p_2) = \sqrt{\sum_{i=1}^{d} (p_1[i] - p_2[i])^2} \tag{3.1}$$

where $d = 32$ is the number of dimensions of the space and the coordinates are defined to sum up to 1 (Due to PLSA [3]. These values can vary in the future.):

$$1 = \sum_{i=1}^{d} p[i] \tag{3.2}$$

We therefore get the maximum distance of two points when considering two unit vectors e_i, e_j with $(i \neq j)$ for which we get $distance(e_i, e_j) = \sqrt{2}$.

3.1.2 Distance Distribution

If we examine how the distances between songs are distributed in an average music collection we get a rather astounding result: They are distributed normally. We have used several real music collections which all show approximately the same curve.[2]

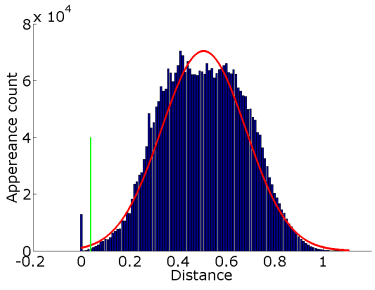


Figure 3.1: The distance distribution between songs of a collection is approximately normal.

In Figure 3.1 the red line is the normal distribution and the green one marks the estimated maximal neighbourhood radius for 10 songs. The outlier at distance 0 is founded by songs which have artist coordinates only. The distance between them is always 0.

[2] See Appendix A for the results.

Problems with this Approach

If a music collection consists of two genres which are rather different - say rock and classical - then we do not get a normal distribution at all. While for each song we have quite some songs from the same genre in a near neighbourhood, we also have many songs of the other genre with a big distance. This leads to a distribution with two hills. Assuming a normal distribution can lead to an error in the maximum neighbourhood radius estimation. In Figure 3.2 we see, that a lot more songs lie within the estimated neighbourhood than desired. The consequence of this error is that the number of songs that a rating gets rubbed off to is large. In an extreme case the rating of every song would be changed if one songs rating is adjusted.[3]

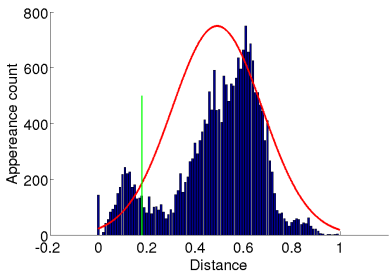


Figure 3.2: If the songs group in two or more different regions of the music space, the distribution plot shows multiple hills.

3.2 Calculation of the Rating

To calculate the implicit rating of a song, we compute several sub-ratings that use different approaches to express the popularity of that song. Each of it returns a value between '−1' and '1'. There are two sub-ratings which currently are in use: The *percental listening rating* which looks at the average skip position of the song and the *listening time rating* which considers the overall listening time for a song. Since we do not consider every sub-rating equally important we get the final rating by computing the weighted average over all of them as follows

$$rating(song) = \frac{\sum_{sr \in \mathcal{SR}} rating_{sr}(song) * weight_{sr}}{\sum_{sr \in \mathcal{SR}} weight_{sr}} \tag{3.3}$$

where $rating_{sr}$ is the rating function of the sub-rating and $weight_{sr}$ defines its assigned weight. We are using the heuristic weights $\frac{2}{3}$ for the percental listening rate and $\frac{1}{3}$ for the listening time rating. This defines the percental listening rating to be twice as important than the listening time rating. We end up with

$$\begin{aligned} rating(song) = &\frac{2}{3} * rating_{percental}(song) \\ &+\frac{1}{3} * rating_{listening\_time}(song) \end{aligned} \tag{3.4}$$

[3]See Section 3.2.2 for more information.

3.2.1 Percental Listening Rating

Whenever the user finishes listening to a song – by listening it to the end or skipping it – a rating is calculated. This is done by using the fraction of how much of the song was played. Figure 3.3 shows how the rating is assigned to the fraction. A linear distribution between a playback fraction of 25% and 75% is used. Outside of this range a rating of '−1', '1' respectively, is taken. We introduced this shift because skipping a song after e.g. ten seconds still means, that we do not like this song at all. On the other hand, skipping the song during the fade out does not mean, that one does like the song less as if listened it to the end.

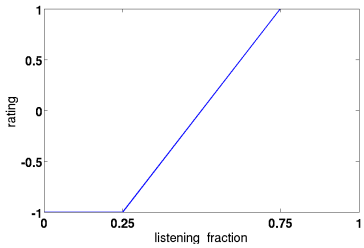


Figure 3.3: Rating by listening fraction.

Every such rating entry also rubs off to the neighbours of the song. To get the overall percental rating of a song, all its listening rating entries ($\mathcal{LR}$) and neighbourhood rating entries ($\mathcal{NR}$) are considered.[4] We want to weight neighbourhood ratings a little less than listening rating entries since they are based on assumptions. We have chosen to weight them 80%. Furthermore, old ratings are weighted weaker than recent ones to be able to 'forget' old rating entries that may not be valid any more. The result is the following rating function:

$$rating_{percental}(song) = \frac{1}{weight\_sum}\left(\sum_{lr\in\mathcal{LR}} r(lr)w_{age}(lr) + 0.8\sum_{nr\in\mathcal{NR}} r(nr)w(nr)w_{age}(nr)\right) \tag{3.5}$$

with

$$weight\_sum = \sum_{lr\in\mathcal{LR}} w_{age}(lr) + 0.8\sum_{nr\in\mathcal{NR}} w(nr)w_{age}(nr) \tag{3.6}$$

$r(x)$ returns the value of a rating entry and $w(nr)$ its neighbourhood weight. The ageing function $w_{age}(x)$ is defined as a linear function that is 1 for entries not older than one month. Then the weight slowly reduces until it ends at 0.25 for entries with ages older than 200 days. Notice, that the older a rating entry is, the less it contributes to the final rating. However, the rating is not reduced for songs that have old rating entries assigned only.

[4] See Section 3.2.2 for more information about neighbourhood ratings.

3.2.2 Neighbourhood Rating

If we like a song, the probability that we like the songs in its neighbourhood as well is high. Therefore every percental listening rating assigned to a song rubs off on the songs close to it in the music space. Since this rating entry should affect nearby songs more than ones far away, we introduced the weight of a rating. The weight has to lie between 0 and 1. The more the weight goes to 1 the more the rating is considered as significant.

Radius of the Neighbourhood

We want, that on average the neighbourhood should contain a fixed number of songs (in our case 10). To ensure this we need to calculate an appropriate radius for the neighbourhood hypersphere for each music collection. We find it using the assumption from Section 3.1, that the distances between two songs are distributed approximately normal within the collection. With this we can calculate the proportion of data which is expected to lie within z standard deviations σ of the mean μ. We get the radius as $r = \mu - z\sigma$.

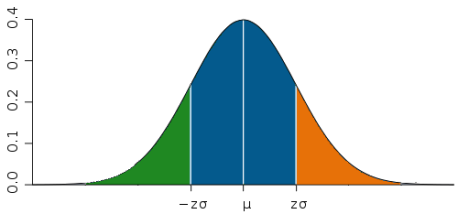


Figure 3.4: Normal distributed song distances. The x-axis represents the distance between two songs while the number of song-pairs with this distance is shown on the y-axis.

$$F(\mu + z\sigma; \mu, \sigma^2) = [\text{blue}] = erf\left(\frac{z}{\sqrt{2}}\right) \tag{3.7}$$

Where erf is the error function. And

$$\begin{aligned}[\text{green}] &= \frac{1}{2}\,[\text{green} \cup \text{orange}] = \frac{1}{2}\,(1 - [\text{blue}]) \\ &= \frac{1}{2}\left(1 - erf\left(\frac{z}{\sqrt{2}}\right)\right)\end{aligned} \tag{3.8}$$

Moreover we want the green portion to be the fraction of ten songs out of our collection:

$$[\text{green}] = \frac{10}{\text{collection size}} \tag{3.9}$$

We end up with

$$1 - 2 * \frac{10}{\text{collection size}} = erf\left(\frac{z}{\sqrt{2}}\right) \tag{3.10}$$

Moving things around, we get the final function for the radius:

$$z = erf^{-1}\left(1 - 2 * \frac{10}{\text{collection size}}\right)\sqrt{2} \tag{3.11}$$

$$r = \mu - z\sigma \tag{3.12}$$

Finding the Neighbours

To efficiently find the neighbours within a given radius around the played song we are using a kd-tree over the song coordinates. If a song does not have valid coordinates we do not search for neighbours at all. Doing so could falsify the statistics since two songs without coordinates lie on the same point in space but don't necessarily need to be similar at all.

Calculating the Weight

We decided to distribute the weight linearly over the neighbourhood. Ratings of songs with the same coordinates as the listened one get a weight of 1 while the weight for ratings of songs whose distance equals the radius of the neighbourhood sphere drops to 0.

3.2.3 Listening Time Rating

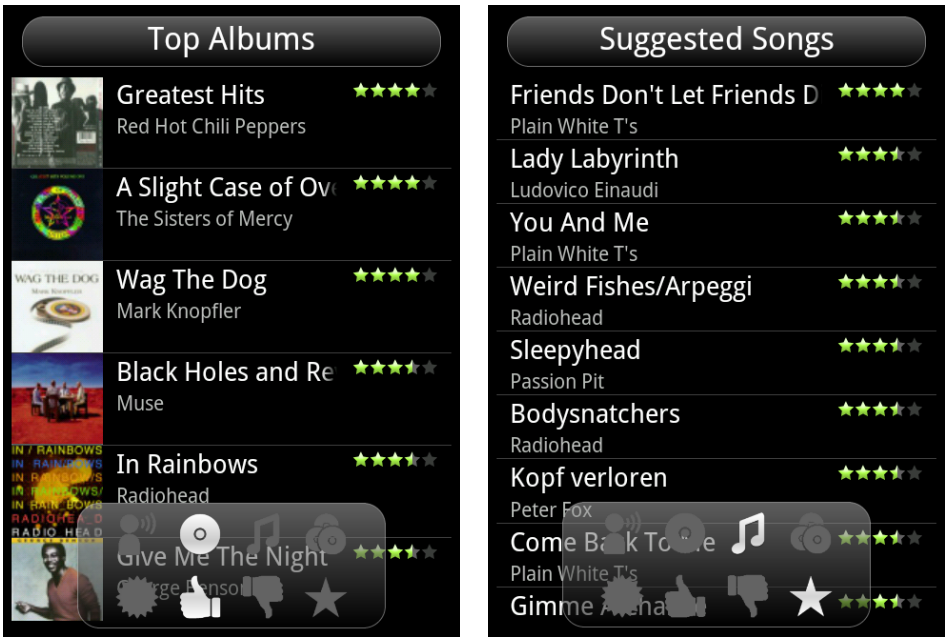
A second part of the final rating is the overall listening time of a song. We consider songs with a high listening time as more important than those which were played for only a short time. We use the rating 1 for the song with the maximum playback duration and are linearly reducing the rating for songs with less listening time. We get the rating function for the listening time as

$$rating_{listening\_time}(song) = \frac{lt(song)}{argmax_{s \in collection} lt(s)} \tag{3.13}$$

where $lt(song)$ returns the listening time for a song. Note that this rating only returns positive values.

3.3 Stabilization of the Rating

It turns out, that the rating calculation of the previous section is good for collections with already some listening statistics available. For collections, where there are only very few rating entries for a song, the rating can change rapidly with a new rating entry. As an example, a song gets skipped by the user, because he is not in the mood for it right now. This song had only one good rating entry



(a) Top albums (b) Song suggestions

Figure 3.5: Statistics that are presented to the user.

so far. Adding the new negative rating entry leads to a big drop of the overall rating into negative and the song gets indicated as not liked at all. We do not want such rapid changes of the rating. We therefore introduced an initial rating with value '0' and a weight of '2' for every song which counts the same as if we listened this song twice to the middle. With this we achieve that extreme ratings settle only after some votes and the change of the overall rating is not that high when adding only one new rating entry.

3.4 User Statistics

We also wanted to give the user access to the statistics that are made about his listening behaviour. In the Android version of Jukefox one is able to get the following lists. How the following statistics are presented to the user is shown in Figure 3.5.

3.4.1 Top-100 & Flop-100

As the name suggests, the Top-100 list consists of the hundred top rated songs, whereas the Flop-100 shows the songs with the hundred lowest rating values. The ratings in these lists are calculated without making use of any neighbourhood

rating entries, since the user is most likely just interested in songs he really listened to.

3.4.2 Recently Imported

In this view the user gets presented with the songs which were imported in the past two weeks.

3.4.3 Suggestions

This list shows songs, the user currently could like listen to. It's a mix of top songs by neighbourhood[5], recently imported and long not listened songs. Songs which were listened recently get filtered out of this list. See Section 4.2.3 for further details, how song suggestions are made.

[5]Only neighbourhood rating data is used.

Chapter 4

Smart-Shuffle v2.0

The second main part of this work was to write a new Smart-Shuffle algorithm for Jukefox based on statistics data. The Smart-Shuffle algorithm should produce suggestions of what song should be played next. The goal is to suggest the music the user currently likes in each round, ending up with long listening time compared to the amount of skips. The key issue is that on the one hand we want to explore new interesting songs to play. For these songs we do not have any popularity data or it is very old. On the other hand we want to exploit the knowledge about songs we believe the user could like. The Smart-Shuffle algorithm needs to find a good balance between exploration that risks that the user will skip the song since he does not like it and exploitation that needs the interesting songs set to get bigger to not be forced to play songs multiple times. Moreover, one weakness of the old Smart-Shuffle implementation was that it was not able to 'forget' outdated information. Since the music taste of the user can change over time, every song should eventually being played again despite its popularity. Our reimplementation should overcome this problem.

In the first section the concept of the Smart-Shuffle algorithm is described. Section 4.2 explains the different agent types. The last section of this chapter describes the Smart-Shuffle algorithm in detail as well as some implementation tricks and tweaks which are used to speed things up.

4.1 Concept

Before we describe the way the new Smart-Shuffle algorithm works, we introduce the terms *agent* and *vote*.

4.1.1 Agents

An agent is an algorithm which proposes and votes for songs which should be played next. Different agents look at the music collection from a different point of view and decide upon their perception.

4.1.2 Votes

Votes represent the opinion of the agents whether the user wants to listen to a song or not. Please do not confuse votes with ratings. Ratings are the result of listening behaviour of the user while a vote can be understood as the prediction of the rating of a song if it was played. Votes are allowed to be in the range $[-1, 1]$. If an agent totally disagrees that a song should be played, it votes '-1'. On the other hand a vote of '1' means that the agent is positive, that this song should be played next. Figure 4.1 shows the way the user gets presented with the most influencing votes that led to choosing that song.

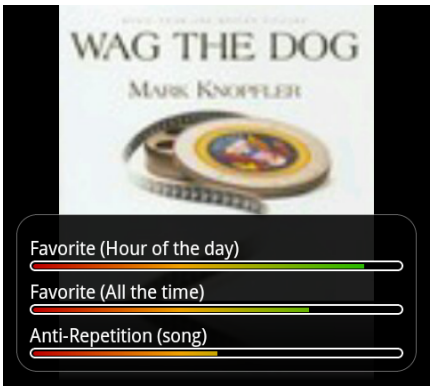


Figure 4.1: The user gets presented with the most influential votes.

4.1.3 Choosing the next Song

Calculating the next song is separated into several steps. They are sketched in Figure 4.2. A central instance coordinates the different steps that are needed to find the song that should be played next. In a first round all agents are asked to propose a list of songs which should be considered for playing next. The agents are allowed to propose only a limited number of songs. From these proposals only few are taken at random. Then they are merged into one proposal list and forwarded to all agents which have to vote for all proposals. All the agent votes are combined into an overall vote for every song proposal. The bigger such a vote turns out, the more probable it is that this song gets played next. Moreover, a long not rated song is added to the list to ensure exploration. Now the final proposal list gets ordered at random with the probability that a song is inserted next into the list as:

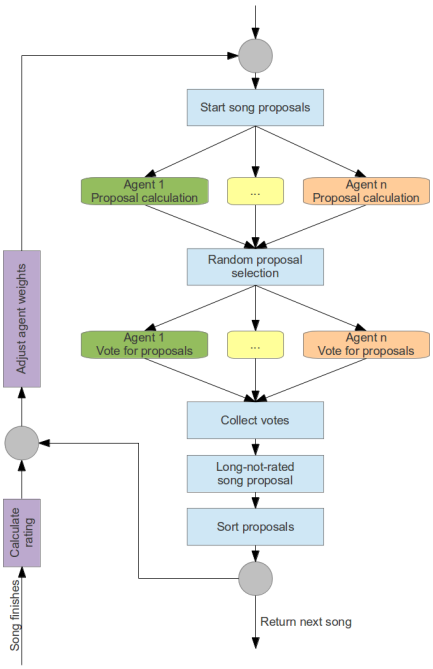


Figure 4.2: The different steps to get the next song.

$$P[\text{song gets chosen}] = \frac{adjustedVote(song)}{votesSum} \tag{4.1}$$

with

$$adjustedVote(song) = (vote(song) + 1)^{stretch\_factor} \tag{4.2}$$

$$votesSum = \sum_{s \in not\_already\_enlisted\_proposals} adjustedVote(s) \tag{4.3}$$

We have chosen $stretch\_factor = 4$ heuristically. This ensures that songs with a high vote appear much more likely at the top of the list than those with bad votes.

The reason why we decided to use this random approach was that proposals which have moderate votes are very unlikely to be played again using pure sorting by vote. If we did not like a song that much once, it then wont ever be played again since it is very unlikely that it reaches the top ranking ever again. With the randomized approach, the probability of such songs being played is less than high voted ones, but it is not impossible.

The first song appearing in the ordered list of proposals is then chosen to be the next song. When the song is finished the agent weights get auto-adjusted and a new calculation round can begin.

Random Proposal Selection Some agents work on a big data set of rating entries and choose e.g. the top song out of it. A new rating entry that gets added to that data pool does not contribute much to the overall decision, since its weight is too weak. This results in very much alike proposals of these agents for several rounds. If a song of those proposals gets played, it probably will be proposed in the next round again (or a similar song of the same agent). We don't want this behaviour.
We therefore introduced random proposal selections. Agents are asked to propose up to 30 songs from which only 7 songs per agent are taken at random. This leads to less static proposal lists and a more balanced listening experience.

Long Not Rated Songs If the random agent is weighted very weak, the song proposals tend to stay approximately the same over several rounds.[1] Moreover most agent implementations are based upon rating entries. To ensure playing songs of the whole collection – even never played ones with no rating entries at all – one song proposal is always chosen to be a song whose vote is forced to be 0.2. That song is chosen u.a.r.[2] from all the songs which were never rated before or the last rating entry is dated at least 200 days back. The vote of 0.2 ensures a sufficient small probability that this random song is played next. This

[1] Since e.g. the overall top agent does not change its proposals very much over successive rounds. This is because new rating entries do not carry that much weight in the large set of rating entries this agent considers.

[2] Uniformly at random.

guarantees an increase of the amount of knowledge independent of the agent weights.

4.1.4 Agent Weights

It has to be considered that not all agents are of the same importance for every user. Some people may not care if two songs of the same artist are played in a row, while others do. The importance of an agent that votes against artist repetitions therefore has to be different for these two users. This is the reason why we introduced the weight of an agent. The bigger the weight of an agent, the more its vote is considered as important. We calculate the total vote for a song as follows:

$$vote(song) = \frac{\sum_{a \in agents} vote_a(song) * weight(a)}{\sum_{a \in agents} weight(a)} \tag{4.4}$$

Automatic Weight Adjustment

Agent weights are tried to be calculated automatically by Jukefox. Whenever a song change takes place, and we therefore know its rating, we look at the agent votes for this song. Agents which voted *right* are rewarded by an increase of their weight. The weight of those who voted *wrong* gets reduced. The new weight is composed of a fraction of the old weight and one of an adjustment:

$$\begin{aligned} weight_{n+1}(agent) = (1 - f) * weight_n(agent) \\ + f * adjustment(agent, voteRating) \end{aligned} \tag{4.5}$$

The adjustment is based upon the vote-rating product:

$$voteRating = vote_{agent}(song) * rating \tag{4.6}$$

This results in big adjustments for extreme differences of the vote compared to the rating and only small adjustments for moderate votes or ratings. This makes sense, since a $|\text{vote}| \approx 1$ means the agent was very certain about its prediction – if the song then gets a very different rating, the expressiveness of this agent is in question and its weight should be reduced.

To make sure, the rewards (reductions) get fulfilled, *adjustment* has to be bigger (smaller) than the current agent weight if the vote-rating product is positive (negative).[3] Moreover, the weight of agents which voted bad multiple times

[3] If we want to reward an agent:

$$\begin{aligned} & weight_n(agent) < weight_{n+1}(agent) \\ \Rightarrow\ & weight_n(agent) < (1 - f) * weight_n(agent) + f * adjustment \\ \Rightarrow\ & weight_n(agent) < adjustment \end{aligned}$$

in a row should be reduced more than others that voted wrong for just once. This results in the following function:

$$\begin{aligned} adjustment(agent, voteRating) = weight_n(agent) \\ + boost_n(agent) * voteRating \end{aligned} \tag{4.7}$$

and

$$boost_n(agent) = \begin{cases} 1 & \text{, if } voteRating \geq 0 \\ boost_{n-1}(agent) + |voteRating| & \text{, otherwise} \end{cases} \tag{4.8}$$

We have chosen $f = 10\%$ in (4.5). This is a heuristic value which appears to be a good mix of keeping weight changes smooth and changing the weight fast for wrong classified agents.

Another attempt was choosing f bigger at the launch of Jukefox and reduce it over time. This leads to faster adjustment of the agent weights at the start since it is more likely one changed its listening behaviour since the last application launch than during listening to music. Since the value of f has to be in balance between not too big jumps and sufficient change of the agent weights if the listening behaviour changes, we were left with only very little freedom for the range of f. We therefore decided to not introduce another dynamic variable into the new Smart-Shuffle algorithm.

A different approach we tried was to use an own f for each agent. It was changed in respect of the sign of $voteRating$. The idea was to reduce f_{agent} when an agent repeatedly votes right and increase it if the votes are wrong all the time. This would keep weights of always correct agents more stable by not reducing their weight significantly when a vote is wrong for one time. But it resulted in very unstable weights for weak agents: When f_{agent} is big, the new weight will mostly be adjusted to $voteRating$. However, that value can change much in each round. The current approach is following the same idea by performing way better.

Normalization of the Weights

As a convenience we decided that the weights of all agents have to sum up to 1. Moreover weights are not allowed to be less than 0.01 to not completely lose an agents vote.

Manual Adjustment of the Agent Weights

The user can manually adjust the weight of the agents via the GUI (See Figure 4.3). There are two modes: The advanced view allows to change the weight of each agent instance separately while the simplified view allows to change the weights for the agent types only. The new weight of an agent type gets distributed to its agents by the ratio of the old agent weights.

Figure 4.3: The agents-weight adjustment menu.

Since the weights are normalized to sum up to 1, small weights are much more probable than weights ≈ 1. With linear progress bars, all the thumbs would lie on approximately the same value on the left and no good weight adjustment could be made. Therefore the progress bar positions are not linear but exponential:

$$pos(weight) = \left(\frac{weight - minWeight}{1 - minWeight}\right)^{\frac{1}{stretch\_factor}} \tag{4.9}$$

Where the expression in the brackets maps the weight from $[minWeight, 1]$ to $[0, 1]$. The $stretch\_factor$ is 4 for the complex view and 3.3 for the simple view. They differ, since bigger weights occur in the simple view, because they are combined from multiple agents. We got these values by trying out what *feels right*.

4.2 Agent Types

There are four agent types which differ greatly in the way they look at the music collection.

4.2.1 Favourite Agent

This agent type votes for songs by the *Top-n* rating of their artist.[4] It composes the song proposals from the Top-7 artists list. The number of songs taken from each artist is proportional to the artists rating:

$$numberOfSongs(artist) = \frac{rating(artist) + 1}{\sum_{a \in top7Artists} rating(a) + 1} * maxSongCount \tag{4.10}$$

The $+1$ is used to shift the ratings from $[-1, 1]$ to $[0, 2]$ to have pure positive values. $maxSongCount$ is the number of songs the agent is allowed to propose. The songs are then chosen u.a.r. from the respective artist.

[4] See Section 3.4.1.

In the voting round the agent just uses the rating of the songs artist as the vote for it.

There are three instances of this agent running in Jukefox. One is considering the Top-n of all the time while the second filters rating information by the current hour of the day. The latter tries to propose songs which the user could like to listen to in the current hour of the day and e.g. propose motivational music during the gym-hour whereas proposing to listen to jazz while drinking a glass of wine in the evening. However, this only works if the user has some listening patterns allotted during the day.
The third instance type runs on recent data. It does not propose any songs but only votes. This instance type is very useful to vote against artists which the user does not like in the moment or boost these he likes currently very much.

There is also a fourth instance type implemented. It filters the Top-n list by the day of the week. However, this instance is currently not used since its results were not that significant while increasing the computational costs. Moreover the implementation of an agent which voted for songs by their Top-n rating[5] has been dropped, since the song proposals were too much alike in every round.

4.2.2 Anti-Repetition Agent

There are two agents of this type. One agents is responsible to avoid repetitions of the same song while the other avoids the playback of songs of the same artist. Both agents are not proposing any songs. The vote for a song is calculated as follows:

$$vote(song) = min\left(\frac{lastPlayed(song)}{minTimeForReplayAcceptable} - 1, 0\right) \tag{4.11}$$

If the song (or a song of the same artist) just got played, the agents vote will be '-1'. Both agents define a time stamp, from when on they consider the song acceptable to be played again and vote '0' thenceforth.

The agent which prevents song repetitions implements $lastPlayed(song)$ as the time span that occurred since the last time the song got played. The constant $minTimeForReplayAcceptable$ is defined as 7 days.

For the anti artist repetition agent $lastPlayed(song)$ returns the time that passed since the last time any song of the artist of *song* was played. $minTimeForReplayAcceptable$ is defined as 20 minutes.

[5] Not the Top-n rating of their artists.

4.2.3 Suggestion Agent

This agent type tries to find out what kind of music the user could like. To find accurate suggestions, this agent considers different, possible interesting song groups:

Top-n by Neighbourhood It is searched for songs with the best rating by only considering neighbourhood rating entries. With this, we get songs which lie in the neighbourhood of liked songs and possibly were not played yet.

Ancient Top-n The result of this filter is the Top-n songs of some time ago. How much we look back at the charts is chosen at random in every round. However, it has to date back to at least one month. All rating data between then and now is omitted while calculating the charts, to get the view of *then*. With this filter it is possible to propose songs which had a good rating once, but were not played much recently.

Recently Imported This group contains all songs which were imported recently. It covers the fact, that the user likely wants to listen to newly imported music. The vote for such songs is calculated as

$$rating_{import}(timeAgo) = max\left(1 - \frac{\lfloor \frac{timeAgo}{bucketSize} \rfloor}{\lfloor \frac{maxImportAge}{bucketSize} \rfloor}, 0\right) \tag{4.12}$$

This is a linear function which gives a rating of 1 if the import just took place ($timeAgo = 0$) and a rating of zero when the import dates back to more than $maxImportAge$ which is defined as 3 weeks. $bucketSize$ is used to threat songs which were imported around the same time as the same[6] and is defined as one hour.

All the songs of the above groups are then merged together into one list, while, when appearing twice, the song with the higher rating is taken. Moreover all songs which were recently played too often are removed from this list. We defined the listening time of suggested songs to be at most 30% of the average song duration[7] while *recently* is defined as the time range between the time stamp for the ancient Top-n filter and now.

Normally such a filter would not be necessary, since the anti-repetition agents are responsible to degrade songs with too much recent listening time. The fact why we still added it, is that otherwise songs would not disappear from the list

[6] Otherwise in one library import call, songs which were imported first would get lower rating than those which were imported last since it takes some time to import them.

[7] The average song duration is currently set as a constant of 4 min 13 s (This is the average song length over all collections we had at hand). This value could also be calculated individually for each music collection during the library import.

even if they get played. This becomes clear, when we look at what happens, when a proposed song gets listened to. The *Top-n by neighbourhood* list only changes, if significantly weighted neighbourhood ratings are added. But the weights of neighbourhood ratings of only one rating entry are weak[8] and only slightly affect the overall ratings. The *recently imported* list changes only slowly over time and is not affected by the playback of songs. The *ancient Top-n* list is the only dynamic factor in the proposals, since its time stamp changes in each round. The result of such a mostly static proposal list is that after songs of it got played, the votes of the anti-repetition agents would take effect and would make this agents proposals useless.

When this agent has to vote for songs it runs all the steps of the proposal process by just regarding the desired songs. It takes the ratings of the songs as the votes for them.

Note that this agent is the only one that is able to make this Smart-Shuffle algorithm outstanding of other music players'. Its use of neighbourhood information in the *Top-n by neighbourhood* and *ancient Top-n* song groups is one thing 'normal' players are not able to do.

Instance Types

There are multiple instance types of this agent. Each of them adds another layer of filters to the song proposals. The following instance types are available:

Current Mood When suggestions and votes are made for the current mood, only rating data from the past hour is used. Suggestions are likely to be very similar to the recently played songs. This instance does not consider any ancient Top-n songs.

Hour of the Day Rating data gets filtered by the hour of the day it was written. When a user has mood patterns along the day, this instance allows to distinguish between them.

Day of the Week Rating data gets filtered by the day of the week when it was recorded. When a user has mood patterns along the week, this instance allows to distinguish between them. This instance type is currently not used in Jukefox since its effect is relatively small compared to the increase of computational costs.

All the Time This instance type does not add an additional filter and uses all rating data available.

[8] There are a lot of other neighbourhood ratings for the same song.

4.2.4 Random Agent

This agent suggests songs u.a.r. from the collection. In the voting round it assigns a vote of '1' for all its proposed songs and '0' for all others.

There were some other attempts with this agent, such as it votes at random for a song. However, it turned out that this behaviour is undesired, since when voting for around 30 songs there is a good chance that at least one song gets a very high vote. This high vote is likely to overrule all votes of the other agents. We assumed the user to be lazy and not skipping pure random songs regularly.[9] The result is that the random agent got the highest weight and we ended up with a mostly pure random play mode.

Another possibility could be to vote '-1' for the not proposed songs. However, if that would improve the weight adjustment or result in more playbacks of random songs is not tested.

4.3 Implementation

One of the critical questions was how and when to calculate the next song. Since the calculation takes some time it cannot be done when the new song is required but has to be carried out before. Otherwise the playback would stop until the next song is calculated. Hence the determination of the next song is done while the current one is being played.

However, with this asynchronous calculation a new problem arises: It is not known whether the user likes or dislikes the current song. But this information is crucial for some agent instances as the current mood suggestion agent since its behaviour is very likely to change with every rating entry.[10] To deal with this problem, we actually run two calculations: One which assumes a negative and one which assumes a positive rating-outcome for the current song. The trick we used, is that a transaction is started on the database. Then a skip of the current song is simulated. The positive assumption calculation uses a playback fraction of 66% while the negative assumes 33%. This writes the rating entries of the currently played song and its neighbourhood into the database. Then the calculations are started. The agents are able to work as usual - they can not differ between the fake and real data since they are not distinguishable in the database. After the computations are finished the transaction gets rolled back. This removes all the simulated rating entries from the database.
Once the next song is requested it only needs to be checked whether the actual rating of the currently played song is positive or negative. The next song can then be chosen to be the outcome of the corresponding calculation.

[9] After all its his music collection. He probably likes many songs in it.
[10] Since it only considers rating entries of the past hour, every entry is quite important.

These two calculations are run sequential. This is on the one hand because SQLite only allows one transaction to be run at a time. On the other hand, at the start of a song the outcome of the negative prediction calculation is probably more urgent needed than the positive one.[11]

4.3.1 Timing & Implementation Tricks

The calculation of the next song takes time. It takes around 25 seconds per calculation for a rating list with 34,000 rating entries whereof 31,000 are neighbourhood rating entries. Some tricks and optimizations were necessary to get to this time and some are needed to work with it.

Finish all next Song Calculations

In an early implementation of the next song calculation, at each start of a song old calculations which were still running got aborted and new ones with regard to the new rating information were started. We replaced this behaviour by letting every calculation finish until a new one gets started. When the new one gets started the decision is made if a negative or positive assumption calculation is due. This is done by looking at the playback position of the currently played song. If it is above 50% a positive attempt is made and a negative one otherwise. This really helps not to waste calculation time. A side-effect of this decision is that two calculations for the same song are done only if the negative calculation for it is finished before the song is changed. If a user is not happy with the current song-proposal he probably does an early skip. If the negative calculation gets restarted after every skip, most likely it wont ever finish. However, letting the negative calculation of the first skipped song finish, ends up with better song proposals eventually.

Reuse of Calculations

If the calculation for the current song and prediction case is not yet at hand, we simply use the most recent calculation. It is possible that a calculation is used multiple times when a song gets skipped very fast. To make this possible, every calculation stores not only the best song proposal but all of them. In each reuse-round the next song on the proposal list is chosen. If the user does more than three negative-skips[12] for songs of the same computation, that computation is considered as too bad and gets dropped. From then on random songs are played until the next computation could finish.

[11] When the user does an early skip, the song gets a negative rating and the negative prediction results are needed.

[12] A skip of a song at a playback fraction which results in a negative rating.

Moreover, songs of the same artist as the currently played one are jumped over when a calculation is reused and the rating of the current song is negative.

Caching of Votes

Most of the agents have the votes for the songs they propose already in the proposal stage at hand. They store these votes and reuse them in the voting round.

Reduction of the Number of Ratings

An other way to reduce the calculation time is to reduce the amount of data it has to process. This is done in two ways: As described in Section 3.2.2 a rating entry rubs off to its neighbours. The radius of the neighbourhood is tried to be chosen that in average ten songs lie within it. However, if there is a region in the music collection where the similarity of songs is very high or the radius is inaccurate[13], we might end up with much more neighbours. For such cases we decided to allow at most 30 neighbourhood ratings to be made for each rating entry.
The second improvement is to remove too weak neighbourhood ratings. If the weight of a neighbourhood rating drops below 0.01 it wont be stored since it is very likely that it gets overruled by an other rating anyway.

Optimization of SQL Statements

Since SQL statements are in text form, they need to be interpreted. SQLite allows the usage of parameters in query statements.[14] This makes it possible that statements look exactly the same while querying for data with different constraints. This query then only needs to be interpreted once. Since some queries we wrote are really long[15], we excessively tried to make them as cacheable as possible.
Furthermore we tried to surrender date functions whenever possible. Information such as in which hour-of-the-day or day-of-the-week a rating entry has been made is explicitly written into the database to avoid expensive calculation time at query time.

[13] See Section 3.1.2.
[14] See http://www.sqlite.org/lang_expr.html#varparam.
[15] To get the top artists a statement of 2100 chars is required to be interpreted.

Remove any Calculations from the GUI-thread

Whenever a calculation is done that accesses the database, we need to ensure it is not running in the main thread. Since write accesses to the database can be blocked for more than 20 seconds,[16] the GUI could be inaccessible for a really long time otherwise.

4.3.2 Backup

When doing a music import from scratch, the whole database gets dropped and recreated. This also removes all rating and playback data. To avoid this, we implemented a backup routine which stores that data to a save place before doing the import and restores it at the end. Entries of songs which are no longer in the collection are dropped.

4.4 Analysis

Let us analyse how good the new Smart-Shuffle implementation performs. The statistical data is based on feedback which is received from users who have the alpha version of Jukefox installed on their device. However, not all implementation tricks mentioned in this chapter made it in the market so far. Therefore the results for the final implementation may vary.

4.4.1 Ratings

Figure 4.4 shows what ratings are given for song proposals.[17] We see that extreme ratings are dominating. Only 5% are floating point values while the rest of the songs are either skipped very early or listened to their end. This enables us to compare these values to the old Smart-Shuffle implementation where only binary ratings were known (skipped or not). Although the rejection rate of the

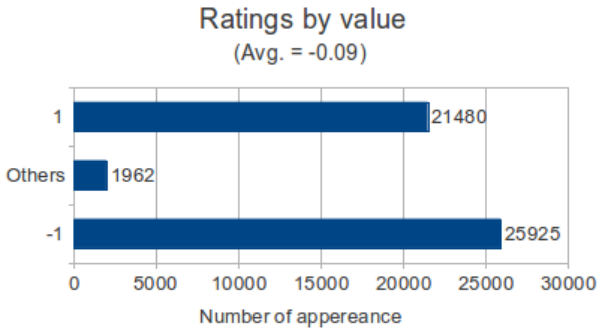


Figure 4.4: Ratings that have been given to song proposals grouped by their value.

[16] The transaction we use in the next song calculation is blocking write accesses to the database. See Chapter 5.

[17] The statistics are based on $\sim 50,000$ rating entries from 361 different users whereof 225 of them sent over 10 rating entries.

proposals (52%) is still bigger than the acceptance rate (44%), the new implementation performs better than the old one with an acceptance rate of 35.25%.

4.4.2 Neighbourhood Size

Figure 4.5 shows how the neighbourhood sizes of a song are distributed. We aimed at having ten songs in the neighbourhood on average. However, we see that most of the songs have one similar song around them only. Around 70% of the songs have less than 10 adjacent songs. This might call for some adjustments of the neighbourhood radius estimation. Yet we rather want to have few than many close-by songs. If every rating entry would affect many other songs, the similarity of them would decrease. This would reduce the gain of neighbourhood ratings. So choosing a slightly bigger radius could result in bigger neighbourhoods in average but might also increase the number of songs that have oversized neighbourhoods.

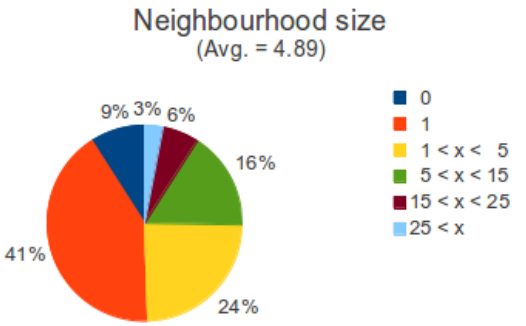


Figure 4.5: How many neighbours lie in average around a song.

4.4.3 Agent Timing

We also included feedback about how much time each of the agents needs to propose and vote for songs. Also the change of the agent weights is logged very fine. However, such data is available from one user only so far. More data is required to be able to make statements about how good they perform.

Chapter 5

Transactions in SQLite

SQLite is a very lightweight DBMS[1] which is used by Jukefox in the Android and CLI[2] version. It is only few kilobytes in size but still able to provide most of the functionality of a SQL database engine. But there is one drawback: Concurrent writes are not allowed.

5.1 What is the Problem?

The calculation of the next proposed song of the new smart shuffle implementation needs to write temporary data into the database. This is done by starting a transaction, inserting that data, working with it and then rolling back the transaction which restores the database state without the temporary data.[3] This is done in the background while a song is being played. However, since a transaction is a write operation, and SQLite only allows one write operation at a time, the whole SQLite database gets locked. In the meanwhile every read- and write-access to the database from an other thread will be blocked. An unresponsive UI is the result during calculating the song proposals.

Android implements a software-side locking mechanism, which results in the waiting described above. The SQLite library used in the CLI version of Jukefox does not provide such a mechanism. As soon as a concurrent access to the database occurs it fails with an exception.

5.2 Some Further Explanations

To get a solution for this problem we had to write our own locking mechanism. To achieve this, we had to take a look at the way how SQLite works. According to [7] there are different states the database can be in:

[1] Database management system.
[2] Command line interface.
[3] See Section 4.3.

UNLOCKED The database has no locks assigned and connections can read or write to it.

SHARED Connections are allowed to read from the database, but writing is prohibited. Any number of simultaneous Shared-locks are allowed.

RESERVED With this lock a connection indicates that it will write to the database eventually. Shared locks are allowed to be created while this lock is held but no Reserved, Pending or Exclusive locks from other connections are allowed.

PENDING A connection wants to acquire an Exclusive lock. All the running Shared and Reserved locks are awaited to be released. No new locks from other connections are allowed to be acquired.

EXCLUSIVE This lock is required to be allowed to write to the database. If an Exclusive lock is held no other locks are allowed to coexist.

Moreover, there are three types of transactions that can be started:

DEFERRED No locks are acquired at the beginning of the transaction. They get acquired when we read or write to the database during the lifetime of the transaction.

IMMEDIATE A Reserved lock is acquired at the beginning of the transaction. This lock is extended to a Pending and then an Exclusive lock as soon as we try to write to the database.

EXCLUSIVE An Exclusive lock is acquired at the beginning of the transaction and held until the transaction gets closed.

Using Deferred transactions can lead to deadlocks: E.g. thread A and B both start a Deferred transaction and read from the database. They hold a Shared lock now. Later A wants to write to the database and therefore acquires a Pending lock. It gets blocked, since B holds a Shared lock. Now B wants to write to the database as well and tries to acquire a Pending lock, too. Since A already holds a Pending lock, B has to wait for A. Both threads are now waiting on each other – a deadlock occurred.

Since Exclusive transactions lock the whole database from the start, even if a write is possibly made only after some time and until then reads would be allowed, we do not want this type either.[4]

Immediate transactions are the type to use. Though only one Immediate transaction is allowed to exist at a time, other connections are allowed to read from the database concurrently until data gets written to the database within

[4]This is the type whose behaviour is described in the introduction of this chapter.

the transaction. When all used transactions are of the type Immediate or Exclusive, no deadlocks can occur, since a concurrent transaction is delayed from the moment when it gets created.[5]

Below Android API 11 the provided locking mechanism only supports Deferred and Exclusive transactions. Since the minimum API version which is supported by Jukefox is 3 we can not use Immediate transactions. Moreover, the CLI version does not implement such a locking mechanism at all, so we had to write it on our own anyway.

5.3 The Solution

To allow as much concurrency as possible, we want to use Immediate transactions all over the application. To get the benefit that a thread waits instead of throwing an exception if it can not acquire a lock immediately, we need a software-side locking mechanism.

To get concurrent access to the database, multiple connections to it are needed. We decided to have one connection open for transactions and one for read and write accesses which are not in a transaction. If an Immediate transaction is running we can still read from the database on the other connection.

To know if a thread is allowed to access the database, the whole locking states as described in the previous section are replicated in the software. The lock manager is therefore always in the same locking state as the database and can decide if the threads lock requirements are granted or if it gets delayed.
Even more convenience is provided by allowing nested transactions which is not supported by SQLite. To get a transaction to commit, all inner transactions need to commit as well. If one is rolled back, the main transaction gets rolled back as well.

Some performance tuning is possible by not implementing the Pending lock as it is described in the SQLite manual. If an Immediate transaction is running and an other thread tries to write to the database, the manual says, that a Pending lock should be acquired. But this leads to the inconvenience that once the request for a Pending lock exists, no other thread is allowed to read from the database. We therefore changed the locking mechanism to not be totally fair and completely delay all write operations until the transaction finishes. Reads that are started later than a write are preferred and executed before the writes. From the moment the transaction finishes, the scheduler gets fair again and enqueues the accesses to the database by their arrival time.

[5] No locks are then held by this connection, only lock requests exist.

5.4 Known Issues

In the current implementation two problems exist:

Multi-Application Access If more than one application accesses the database they do not synchronize their locking states. Exceptions are possible to occur by then. However the ACID-properties are still ensured by SQLite.[6]

In-Memory Cache Overflow All database changes in a transaction are written into a memory cache. This allows other connections not to read the uncommitted changes. If the cache gets full, the SQLite connection automatically acquires an Exclusive lock to write the uncommitted changes into the database. From then on no further accesses from other connections are allowed to be made since they would read uncommitted writes.
If the Exclusive lock cannot be acquired an exception is thrown and the transaction gets aborted. If the lock can be acquired, our locking mechanism does not know about that Exclusive lock and still allows Shared accesses to the database to be made. These accesses will then fail on SQLite-level with an exception.
Therefore, if it is known that big changes are about to be made in the database an Exclusive transaction should be used.

[6] See [8].

Chapter 6

Limitations and Future Work

6.1 Limitations

The time we had for writing the completely revised implementation of the Smart-Shuffle algorithm and the user statistics was relatively short and based on a lot of assumptions about user listening behaviour. Many testing and fine-tuning may still be necessary.

Glut of Parameters One of the biggest problems in the new Smart-Shuffle implementation is choosing the numerous parameters right. Most of them were chosen on a heuristic basis and may not apply for all users. Some of them get regulated dynamically over time. But often their adjustment strategy introduced some new assumptions on e.g. how fast a value should be modified. To get all these parameters right, one needs to do exhaustive testing with a lot of user feedback. We implemented some logging mechanisms which also send reports to the Jukefox server. However, by now only few such usage statistics were uploaded and we were not able to do a meaningful analysis of them.

Silent Decrease of Agent Weights In Section 4.1.4 it is described how the agents weight automatically is adjusted by the skipping feedback of the user. One unsolved problem is, that some agents[1] do not actively vote for songs but prevent others from being played. They do this by voting negatively for songs they consider as inappropriate and '0' for all others. The result is, that mostly songs with no negative vote of these agents get played. However, since the agents voted zero for these songs, they get not rewarded with a weight-increase if the user likes that song. They would only get such an increase if a song they voted against gets played and skipped. The result is, that the agents weight gets smaller and smaller compared to those who actively vote for some songs.
We currently do not have any solution how to solve this problem.

[1] Namely the anti-repetition agents.

6.2 Future Work

Deletion Proposals In the Flop-100 list, deletion proposals could be made for the user to reduce the size of his music collection. As an example, smart suggestions could remove most songs of an album which are not liked but keep the one loved song in it.

Speed Improvements Some agents, such as the overall top agent, propose roughly the same songs for some rounds. This is because the data on which they make their decision does not change much over time.[2] The proposals and votes of these agents could be cached for some rounds which would result in less computational power used.

It also could be checked if some rating data could redundantly be written to the song or artist entries in the database. As an example, when writing the overall rating of a song directly to the database, speed improvements could be achieved for those queries. However, queries are very likely to change between different Smart-Shuffle rounds but only static queries can be represented by a field in the database.[3]

Make Usage of the Old Smart-Shuffle Implementation The old Smart-Shuffle algorithm did not behave very bad but got stuck after a resulting in very similar song proposals in each round. It would be conceivable to reintroduce this algorithm as an agent into the new Smart-Shuffle implementation.

Improve Statistics The current implementation of user statistics are more on a proof-of-concept level than complete, interactive and pleasing. The time for this thesis was too short to provide graphically appealing, versatile statistics. One could think of supplying the user with graphs showing his listening behaviour over the time or highlights in the music maps[4] where songs which the user likes most are located.

6.3 Conclusion

While supplying the user with statistics about his listening behaviour is quite easy, proposing him songs based upon that data is far more demanding. The new Smart-Shuffle algorithm tackles that challenge using agents that divide the

[2] The overall top agent does not get affected very much by one new rating entry since generally it already has quite some rating data around.

[3] As an example, many queries of the suggestion agents are based on ancient data. However, which time range is used is determined at random in each round.

[4] As shown in [3].

big problems into smaller, solvable sub-problems. Ratings are used to express the popularity of songs. Different rating entries for the same song are kept over time. This enables to filter and group data in a broad way. First evaluations of user feedbacks show that the acceptance rate is still below 50% but nevertheless bigger than of the old implementation.[5] This demonstrates the idea of building a playback mode based on listening statistics is realistic and worth it to pursue.

[5]See Section 4.4.

Bibliography

[1] Luke Barrington, Reid Oda, and Gert R. G. Lanckriet. "Smarter than Genius? Human Evaluation of Music Recommender Systems". In: *ISMIR*. Ed. by Keiji Hirata, George Tzanetakis, and Kazuyoshi Yoshii. International Society for Music Information Retrieval, 2009, pp. 357–362. ISBN: 978-0-9813537-0-8.

[2] Luke Barrington et al. *Audio information retrieval using semantic similarity*. Tech. rep. In IEEE ICASSP, 2007.

[3] Michael Kuhn, Roger Wattenhofer, and Samuel Welten. "Social Audio Features for Advanced Music Retrieval interfaces". In: *ACM Multimedia, Florence, Italy*. Oct. 2010.

[4] linkibol.com. *How to Build a Popularity Algorithm You can be Proud of*. URL: http://blog.linkibol.com/2010/05/07/how-to-build-a-popularity-algorithm-you-can-be-proud-of/.

[5] Evan Miller. *How Not To Sort By Average Rating*. URL: http://www.evanmiller.org/how-not-to-sort-by-average-rating.html.

[6] Elias Pampalk, Tim Pohle, and Gerhard Widmer. "Dynamic Playlist Generation Based on Skipping Behaviour". In: *Proc. of the 6th ISMIR Conference*. 2005, pp. 634–637.

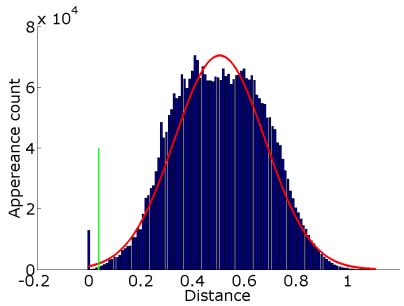
[7] SQLite-Consortium. *File Locking And Concurrency In SQLite Version 3*. URL: http://www.sqlite.org/lockingv3.html.

[8] SQLite-Consortium. *SQLite is Transactional*. URL: http://www.sqlite.org/transactional.html.

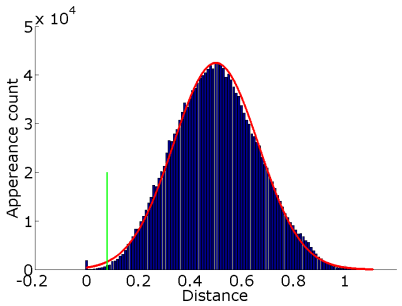
Appendix A

Neighbour Distance Distribution in Different Music Collections

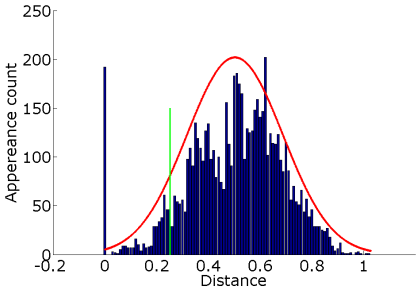
Below are the neighbour distance distributions of some music collections which we have tested. We calculated the distances from every song to every other song. The diagrams below show how many times a distance appeared.



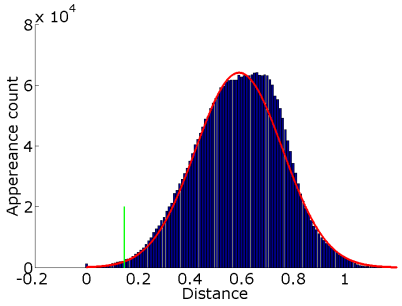
(a) Sabine ($\mu = 0.50, \sigma = 0.18, z = 2.66$)



(b) Tobi ($\mu = 0.50, \sigma = 0.16, z = 2.55$)



(c) Luc ($\mu = 0.50, \sigma = 0.19, z = 1.35$)

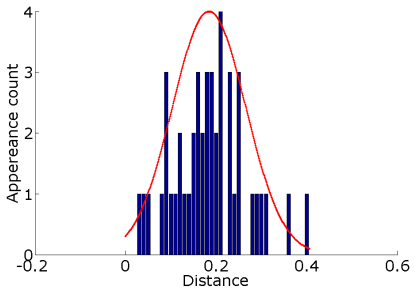


(d) Disco ($\mu = 0.59, \sigma = 0.17, z = 2.64$)

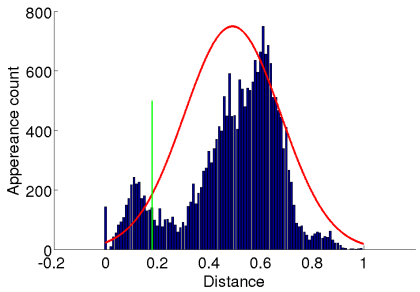
Figure A.1: Distance distribution in usual song collections.

If a music collection has few items only or the songs are from totally different

genres, degenerated distance distributions can appear:



(a) Small ($\mu = 0.18, \sigma = 0.08, z = \infty$)



(b) Two genres ($\mu = 0.49, \sigma = 0.19, z = 1.67$)

Figure A.2: Distance distributions in small or abnormal collections.