Music Recommendation to Groups

Lucas Maystre

June 2012

Supervisor

Prof. Matthias Grossglauser
LCA 4 – EPFL

Abstract

First we present Unison, a conceptual music recommender system for groups of people; the system aims at generating a playlist that takes musical tastes of all the group members into account. We discuss both theoretical and practical concerns related to such a system. We develop a model of user preferences and discuss how we can shift from individual recommendations to group consensus. In constructing the user preferences model we use an intermediary music track model that combines user-generated tags with a dimensionality reduction technique to build a compact spatial embedding of tracks. Secondly we introduce GroupStreamer, a practical implementation of the system that runs on Android devices. We present the technological choices that were made along the way.

Contents

1. Introduction **1**
1.1. Scenario 1

2. Music Recommender Systems **3**
2.1. The recommendation problem 3
2.1.1. Formalization 3
2.1.2. Utility Prediction 4
2.2. The case of music 6
2.2.1. Music Information Retrieval 7
2.2.2. Research Landscape 7
2.2.3. Commercial products 8

3. Recommendations to Groups **11**
3.1. Aggregating Preferences 11
3.1.1. General Impossibility Theorem 11
3.1.2. Rank-based Aggregation Strategies 13
3.1.3. Utility-Based Aggregation Strategies 13
3.1.4. Beyond Preference Aggregation 14
3.2. Group Recommender Systems 14
3.2.1. Music Recommender Systems 15

4. Unison Recommender System **17**
4.1. Challenges 17
4.2. Modelling Tracks 18
4.2.1. Approaches Considered 18
4.2.2. Latent Semantic Model 19
4.2.3. Classification Experiment 21
4.3. Modelling User Preferences 23
4.3.1. Gaussian Mixture Model 24
4.4. Preference Aggregation 27

5. GroupStreamer Application **29**
5.1. Overview 29
5.1.1. Development Process 30
5.2. Android Application 30
5.2.1. Implementation Highlights 31
5.2.2. User Interface 31

5.3. Back-End Services 33
5.3.1. Central Database 33
5.3.2. Requests to Last.fm 35
5.3.3. Playlist Generation 35
5.4. Unison API 36
5.4.1. Back-End 37
5.4.2. Front-End 37

6. Perspectives **39**

A. Latent Semantic Analysis **41**
A.1. Term-Document Matrix 41
A.1.1. TF-IDF Weighting 41
A.1.2. Log-Entropy Weighting 42
A.2. Singular Value Decomposition 42

1. Introduction

As digital collections of all sorts—books, music, Web sites, holiday packages and many others—are growing, it becomes increasingly important to provide intelligent ways of navigating them. From the perspective of a user faced with such a collection the relative number of items that are of interest is often relatively small. Given some initial information about the user's preferences it is desirable to be able to filter the collection such that only *personally relevant* items are shown to her. Systems that allow to filter large collections and provide adaptive, personalized recommendations to end users are called recommender systems. They have attracted a lot of attention and research efforts over the last 15 years, efforts that have been turned into high profile commercial successes. Amazon[1], a large online retailer, uses a recommender system to show related items, driving up sales. Netflix[2], an online movie rental company, uses a recommender system to propose movies that users are likely to enjoy, a conclusive competitive advantage.

Both research and applications of recommender systems have traditionally focused on providing recommendations to single users; the idea can however easily be extended to recommendations to *groups* of people. Such systems would be useful when a group has to take a decision from a large set of possibilities and the decision affects all the members of the group. This arises when people share the same environment, e.g. a family that has to choose which television program to watch (Masthoff 2004). However extending recommender systems to groups is not without difficulties, as we will see.

In this work we explore an application of group recommender systems to music. We investigate issues related to modelling user preferences and choosing a sequence of tracks for the group, as well as more practical considerations. Our main contribution is an end-to-end implementation of a system that automatically generates a music playlist for a group of people.

1.1. Scenario

Let us start by describing the scenario of a typical use case. Imagine that you are organizing a buffet dinner party and invite a small number of people—friends, family, neighbors. You would like to play some music, but you do not know everyone's tastes, so that although you have a sizeable music library spanning several musical genres you are not sure what to play. Fig. 1.1 shows an schematic example of this scenario: three members in a group have somewhat different tastes in music; a single member plugs her media player—nowadays often a smartphone—into

[1] See: `http://www.amazon.com/`.
[2] See: `https://www.netflix.com/`.

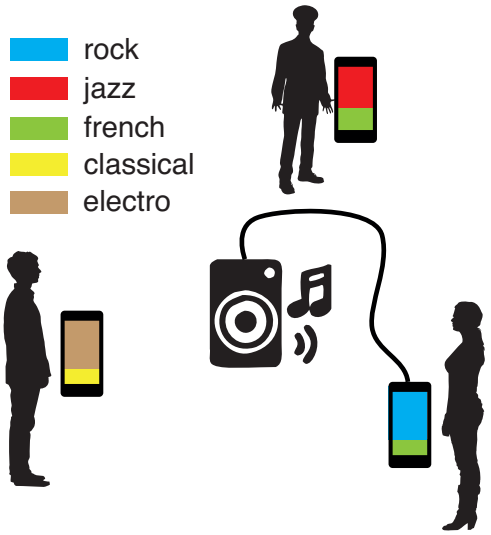

Figure 1.1.: Scenario for a group recommender system applied to music. Members of the group have different musical tastes, yet they have to find a consensus on what music to play.

loudspeakers and plays music for the group. It would be convenient to have a system that would account for each member's musical tastes and automatically select the music to be played. The system would essentially have two tasks: first of all to determine each member's preferences and secondly to aggregate them to generate a music playlist that seeks to maximize the enjoyment of the group as a whole.

In the following we try to build such a system. In order to make it practical, we suppose that every member of the group has a smartphone that is loaded with music. This provides us with music to play, as well as a minimal amount of information about each member's musical profile. At this point many practical questions arise.

- Exactly what information do we get from each user's library? How can we infer preferences based on that limited amount of data?
- How can we compare the preferences of group members? They hopefully share some preferences, however it is possible that their libraries do not intersect, i.e. they do not necessarily have any track in common.
- Once we can predict for each member of the group how much he likes a song, we still need to find a consensus for the group. How do we go from individual preferences to group decision?

We try to address these questions—and many more—in this report. The first two chapters provide a review of selected topics in recommender systems and preference aggregation, as well as an overview of related work. This will provide us with key insights on the challenges we are facing and the techniques used to overcome them. Starting from chapter 4 we turn out attention to Unison, our recommender system. We describe how it works from a conceptual point of view, then we discuss the implementation of GroupStreamer, an instantiation of the system for the Android platform. Finally, we conclude by mentioning possible extensions and promising research directions.

2. Music Recommender Systems

This chapter introduces recommender systems (Konstan et al. 1998; Jannach et al. 2011), in particular systems that recommend music. First of all we formally define the recommendation problem and different approaches that can be taken to tackle it. Secondly we focus on recommender systems applied to music by giving and overview of the research that has been done in the field and presenting some commercial systems.

For the scope of this chapter we restrict ourselves to systems that recommend items (e.g. songs) to a *single* individual—as opposed to a group. We do it for two reasons: firstly, because research on recommender systems has traditionally been focused on single user recommendations and secondly, because generating recommendations for groups introduces a whole new range of surprisingly different challenges. Chapter 3 will explicitly focus on group recommender systems.

2.1. The recommendation problem

We follow the methodology and notation take by Adomavicius and Tuzhilin (2005) whose survey provides an excellent summary of various approaches to recommender systems, in addition to a good formalization of the recommendation problem.

2.1.1. Formalization

Let C be the set of users and S be a (potentially very large) set of items. We define the function $u : C \times S \to R$. $u(c, s)$ to be the *utility* (or pleasure, or any other measure we would like to maximize) provided by item s to user c. We would like to find for each user $c \in C$ the item $s_c^* \in S$ that maximizes the user's utility, which leads us to the following formulation of the recommendation problem[1]:

$$\forall c \in C, s_c^* = \arg\max_{s \in S} u(c, s) \tag{2.1}$$

At first sight, eq. 2.1 seems very simple. The difficulty lies however in that for a given user $c \in C$ we often know the value of $u(c, s)$ only for a small subset $S_c \subset S$. In fact, most of the time we

[1] This is one of many ways to formalize the recommendation problem; we use this particular one here because we think that it gives the essence of the challenges associated to recommender systems.

are interested in recommending items that the user has *not interacted with before*. The problem can then be decomposed into two subproblems.

1. What is the set S_c, and how do we define the utility function over this set? As an example, S_c could be the set of items over which user c has given explicit ratings between 1 and 10 and $u(c,s), s \in S_c$ the value of the ratings, but there are many other alternatives.
2. How do we compute the missing values of u, i.e. $u(c,s), s \in S \setminus S_c$? Intuitively, we would like to assign similar utility values to similar items—however it is not immediately clear how to define this notion of similarity.

The two subproblems are closely dependent; in particular the definition of the utility function can condition how midding values are computed. The utility function depends on the specific nature of the problem at hand and on the data that are available. Beyond explicit user feedback there are a variety of alternatives.

- For an online news recommender system the time spent reading an article can be thought of as an indicator of a user's interest, hence of the article's utility to the user.
- In a music recommender system, the play count (that is, the number of times as particular song has been listened to) is probably a good signal of a user's satisfaction with that song.
- In a physical store, the content of a customer's trolley gives an indication of her preferences. In this case the utility is derived from the mere association of an item with the customer.

2.1.2. Utility Prediction

The second part of the problem—predicting the utility for items that the user has not interacted with—is the one that has traditionally gotten more interest from the research community. The many different approaches can be categorized into two main classes.

Collaborative Filtering

Collaborative filtering (see Su and Khoshgoftaar 2009) is a technique that uses a notion of similarity between *users*—hence the term *collaborative*—to compute the estimate of a missing utility value. A simple example of a collaborative filtering system is:

$$u(c,s) = \frac{1}{\sum_{c' \in \hat{C}_s} sim(c,c')} \sum_{c' \in \hat{C}_s} sim(c,c') \times u(c',s) \tag{2.2}$$

where $\hat{C}_s, s \in S$ is the set of users for which we know the value of utility function for item s, and $sim(c_1,c_2) : C \times C \to \mathbb{R}_+$ is a measure of the similarity between users c_1 and c_2. For example the similarity between users could be derived from the correlation of utility values over the set $S_{c_1} \cap S_{c_2}$. This is one of many forms of collaborative filtering; much more refined models and algorithms have been investigated in the literature.

Collaborative filtering has the advantage of being agnostic to the nature of the items under consideration. It simply leverages the assumption that there will be similar users in the system. Collaborative filtering systems tend to be very successful in practice but they need to overcome several difficulties. We mention two issues below which we think are particularly relevant.

Cold start When a new user enters the system we need to collect (either implicitly or explicitly) sufficiently many utility values in order to compare him to other users. The exact same holds for items: when a new item appears, sufficiently many utility values have to be collected for the item across all users. Generally speaking this problem is related to the *sparsity* of utility values across users and items.

Synonymy While collaborative filtering being agnostic to the nature of the items is one of its strengths it also means that if an item that has two or more entries in the system they will be considered as totally independent. This calls for additional content-aware means of detecting similarities between items.

The computational aspect is also a challenge in collaborative filtering systems. A naive implementation of Eq. 2.2 above runs in time $O(|S| \cdot |C|)$, which becomes problematic as the item space and the user base increases.

It is interesting to note that dimensionality reduction techniques (of both S and C) have been successfully applied to overcome several of these issues. Goldberg et al. (2001) use principal component analysis (PCA) to reduce the computational requirements by two orders of magnitude. In a later version of their recommender system, they use the same technique to overcome the sparsity problem (Nathanson et al. 2007). Earlier work by Sarwar et al. (2000) and work by Hofmann (2004) also show the potential of dimensionality reduction for improving the performance of collaborative filtering systems.

Content-based

Instead of looking at the users, content-based approaches to recommender systems (Pazzani and Billsus 2007) aim at modelling items in such a ways that—in light of a small quantity of information about the user's preferences—we can predict utility values by comparing items. Content-based recommender systems can generally be described through answers to two classes of questions.

1. How are items represented? What are the item features and how many of them are used? The choice of features is crucial as it conditions the performance.
2. How do we model and learn user preferences over the space of items? This can be seen as a classical machine learning problem, opening the door to both general off-the-shelf techniques as well as finely crafter application-specific models.

These two aspects, once again, are closely dependent. We may indeed ask ourselves: what is a good item representation? A meaningful and effective set of features is simply one that allows an algorithm to precisely characterize the relation between the features and the item's utility for a user.

An important distinction between content-based systems and collaborative filtering is the fact that content-based approaches are inherently domain specific: they depend on the particular type of items to be recommended as well as on the data at hand[2].

It is worth noting that content-based recommender systems do not suffer from the issues mentioned above for collaborative filtering systems: synonymy for example is rarely a problem. However they also need to overcome certain difficulties.

Intelligence Without human intervention it is sometimes hard to characterize some essential qualities of the items. Consider the following example in the case of a text recommender system: it is probably very difficult for an algorithm to distinguish (only from the text) between a *good*, seminal research paper and a minor one.

Diversity It might be difficult for content-based recommender systems to go beyond recommending items that are similar, content-wise, to the ones the user already likes. In many recommendation scenarios it is critical to be able to help the user to discover items that are *different* from those he already knows, yet are still expected to be of high utility.

2.2. The case of music

Music has always been a popular application of recommender systems, arguably because of its universality—everyone listens to music—and the sheer number and diversity of artists and musical pieces. Recommender systems can be a great tool to discover new music. In this section we provide an overview of music information retrieval, the science of analyzing and organizing music and of different approaches that to buliding music recommender systems. Let us begin by reviewing a few important characteristics of music.

- The number of distinct musical pieces is very large. The Echo Nest, a leading company in providing and organizing music information reports a catalog of 30 million tracks[3]; This lower bound gives a clear indication of the scale of the problem.
- It is not always clear how to define and separate musical items. Consider a musical piece: do we consider two different (possibly lossy, e.g. when using MP3) encodings to represent the same item? What about a third version which has been recorded on the radio? What about a live performance recording? See Bertin-Mahieux (2012) for a more thorough treatment of this issue. We mention this characteristic because it can become problematic for some recommender systems—especially ones using collaborative filtering.
- Analyzing and describing music is hard in comparison to other types of data such as text. The models and tools developped so far are comparatively less powerful. We do not know whether this is due to the nature of the data or not.

[2] Note that in content-based recommender systems the utility values can be predicted using solely the user's data—to the point where a content-based recommender system could be built for a single user! In certain cases however models are learned over several users, blurring the lines between collaborative filtering and content-based systems.

[3] See: `http://the.echonest.com/company/jobs/`.

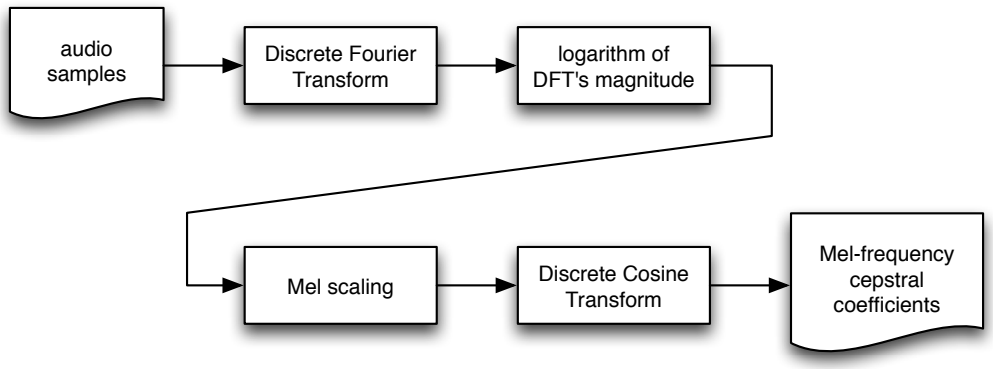

Figure 2.1.: Diagram showing the steps used to compute mel-frequency cepstral coefficients (MFCCs) for a stationary digital sound. MFCCs have been shown to provide a compact representation of a sound's timbre as perceived by humans.

2.2.1. Music Information Retrieval

Music information retrieval (MIR) is the field of research concerned with analyzing, organizing and retrieving information from music (Casey et al. 2008; Li et al. 2012). It has seen tremendous developments over the last 15 years, as the formation of large digital music libraries has created the need for automatic organization and classification capabilities. Part of the research in the MIR community relates to audio processing and how to extract audio features that characterize music tracks.

In a seminal paper Tzanetakis and Cook (2002) showed that a small set of features over a few seconds of audio signal could categorize a track into one of 10 musical genres with high accuracy. Important audio features include tempo, beats, key and timbre; for example the mel-frequency cepstrum (Logan 2000) is a compact representation of our percpetion of the timbre of a sound. Its computation is shown in fig. 2.1.

Also of special interest in the case of recommender systems are *music similarity measures* (Aucouturier and Pachet 2002; Pachet and Aucouturier 2004) which provide a way of musically comparing two tracks based on the audio signal. There are readily available open-source software packages that can be used to extract features and compute the similarity of audio files (Tzanetakis and Cook 1999; Pampalk 2006). We note however that these algorithms often require non-trivial computational resources, if only to decode and process the millions of samples of a digital audio file.

2.2.2. Research Landscape

In the following we briefly present some of the academic literature related to music recommender systems.

Content-Based

There have been several attempts at building systems based on audio features. Among them we mention Cano et al. (2005) who developed a music recommender system working without any metadata. Lloyd (2009) developed a system that automatically generates a playlist based on timbral similarity. Interestingly he also investigated a 2-dimensional spatial embedding of tracks.

The second large class of content-based music recommender systems use additional metadata such as musical genre. Platt et al. (2002) propose a system that generates a playlist based on very few seed songs by estimating a preference function over the genres. Nevertheless simple metadata-based approaches often lack the ability to finely differentiate between tracks.

Collaborative filtering

Despite recent advances, it remains difficult to characterize music based exclusively on intrinsic content. It is therefore natural to turn to collaborative filtering approaches[4]. Early mentions of music recommender systems in the literature—e.g. (Alvira et al. 2001) indeed leverage similar users to find recommendations.

Collaborative filtering systems have to be bootstrapped somehow; In an interesting recent work Aizenberg et al. (2012) investigate the use of radio playlists as providers of human signals about music.

To conclude this brief overview of the research lanscape let us mention Yoshii et al. (2008) who propose a music recommender that combines both a content-based approach and collaborative filtering. This *hybrid* system can overcome issues related to both classes of recommenders: when an item does not yet have enough users content-based features are used, while the collaborative aspect helps tayloring highly accurate and diverse recommendations.

2.2.3. Commercial products

Looking at some of the commercial products related to music recommendation will allow us to get a feel, beyond research prototypes, of what type of technical solutions have been successful in practice. The companies and products we describe below are what we consider to be today's major contenders; as the young age of some of them attests the space is still evolving very quickly.

[4] Remains, of course, the fact that the set of items is very large, which is rather a disadvantage for collaborative filtering systems because of sparsity-related issues.

Last.fm

Last.fm[5] is an online social network around music. At the heart of Last.fm is *Audioscrobbler*[6], a small piece of software that records the artist and title of every track its users listen to. It is available on a wide range on platforms and compatible with many media players. Users can then get statistics about their listening habits and they can listen to virtual radio stations taylored to their tastes. Last.fm is a textbook example of a collaborative filtering recommender system: recommendations are generated by looking at what similar users listen to.

Pandora

On Pandora Internet Radio[7] the playlist is generated by a seed such as a genre, an artist, or the user's listening history. As opposed to Last.fm Pandora uses a content-based approach to recommendation: underlying its recommender technology is the *Music Genome Project*, a music annotation system using several hundreds of attributes[8]. Attributes reflect many characteristics of the music[9] such as structure, rhythm, feel, tonality, lyrical content, instrumentation, etc. It is interesting to point out that tracks are manually annotated, which we believe is the reason why its music database is much smaller than that of its competitors.

The Echo Nest

The Echo Nest[10], a company founded in 2005, is a bit different as it does not properly speaking provide an end-user product; rather, it advertises itself as a *music intelligence* platform. We mention it in our survey because a significant part of their music information seems to be collected from the analysis of the audio signal. Their database powers several music recommendation products such as Nokia Mix Radio[11], a personalized radio station available on the latest Nokia smartphones.

Spotify

Spotify[12] is a music streaming that started in 2008 but only fairly recently became available in a wide range of countries. It is not properly speaking a music recommender system but rather a media player featuring millions of tracks; it is nevertheless clear that music discovery and recommendation play an important role in such a large library. Spotify takes yet another approach by tightly integrating its product with Facebook[13], an online social network. Music is

[5] See: `http://last.fm/`.
[6] See `http://www.audioscrobbler.net/`.
[7] See: `http://www.pandora.com/`. Currently, Pandora is only available in the United States.
[8] See: `http://www.pandora.com/about/mgp`.
[9] For a partial list of attributes, please refer to: `http://en.wikipedia.org/wiki/List_of_Music_Genome_Project_attributes`.
[10] See: `http://the.echonest.com/`.
[11] See: `http://goo.gl/QVeAE`.
[12] See: `http://www.spotify.com/`.
[13] See: `http://www.facebook.com/`.

therefore discovered and shared primarily through the user's own social network. Spotify also allows third-party developers to build extensions that can be embedded in the application. We found several extensions for music recommendation, some of which are based on The Echo Nest's platform.

3. Recommendations to Groups

After presenting recommender systems for individuals we move our attention to *group recommender systems*. As already hinted at before, the difficulty of the additional challenges arising when shifting from individuals to groups is surpisingly big. In this chapter we focus our discussion on the issue of providing a satisfactory way of combining preferences of potentially very different individuals and reach a consensus for the group. After a more theoretical considerations we will turn to actual implementations of group recommender systems.

3.1. Aggregating Preferences

Let us consider the group of users represented by the set G (we do not impose any particular restriction on the size of the group). At this point we assume that for all $c \in G$ and for all $s \in S$ the utility $u(c, s)$ is known, either explicitly from the user, implicitly or by predicting it. For most of our discussion we will not actually consider the utility values directly but only keep the relative order between items (effectively discarding part of the information). Our goal is to find a good way to combine these individual orderings to form an ordering for the group.

Definition (preference aggregation function)**.** *Let R_S be the set of all total orders on the set S of items. A* preference aggregation function *for the group G is a function* $\text{aggr} : R_S^{|G|} \to R_S$ *that maps any combination of user preferences to a single ordering of the items.*

Preference aggregation functions have been studied in economics, political science, multi-agent systems and social choice theory; the latter, which is concerned with the transition from individual interests to societal decisions is especially relevant. In the following we present a well-known result stemming out of this field[1].

3.1.1. General Impossibility Theorem

At first sight we could simply hope to find a good preference aggregation function and use it to form an indisputable ordering of items for the group. We would like this function to satisfy the following properties.

[1] Traditionally, the concepts presented here use a very particular vocabulary due to their roots in social choice theory. For example, items become *social states*, and the preference aggregation function becomes the *social welfare function*. However in this section we try to remain consistent with the terminology introduced in chapter 2.

Universality The function should be defined over *any* set of orderings. This means that users should not be restricted in the way they choose their preferences.

Pareto efficiency If for a given vector of user preferences $\mathbf{r} \in R_S^{|G|}$ we can find items $s_1, s_2 \in S$ such that for all individual preferences r_c, $s_1 \geq_{r_c} s_2$, then $\hat{r} = \text{aggr}(\mathbf{r})$ should also satisfy $s_1 \geq_{\hat{r}} s_2$. This means that if an item s_1 is preferred to another item s_2 *unanimously* by all the users, it should also be preferred in the aggregated preferences.

Monotonicity Let $r = \text{aggr}(\mathbf{r})$. If any user $c \in G$ updates her preferences by *promoting* an item $s \in S$ higher in her preferences resulting in a new vector of preferences $\mathbf{r}'$, then $r' = \text{aggr}(\mathbf{r}')$ should rank s at the same position or higher in r' than in r.

Independence of irrelevant alternatives When looking at a subset of items $S' \subset S$, the relative ordering of S' should not depend on the ordering of $S \setminus S'$ in the users' preferences[2].

Non-dicatorship The preference aggregation function should take into account the preferences of everyone. There should not be a single user that *dictates* the outcome of the aggregation.

Unfortunately even these simple properties make the problem impossible, a result shown by Arrow (1963).

Theorem (general impossibility). *For $|S| > 2$ there exists no preference aggregation function which satisfies the five properties listed above.*

One might wonder what happens in the case where $|S| = 2$. Arrow showed that in this case *majority voting* (i.e. choosing the item which is preferred by a majority of the users) is the only function that satisfies the above properties.[3]. It is also interesting to consider some relaxations.

- Although it is impossible it find a *complete ordering* of the item set, what happens if we just want the top-N items (e.g. with $N = 1$)? It turns out that the situation does not improve by much (Sen 1986).
- The most promising approach is restricting *universality*: by imposing additional structure on the users' individual orderings it is possible to find well-behaved aggregation functions. This is the case with *single-peaked preferences*, i.e. when the items' rank is a concave function of some uni-dimensional embedding of the item set S.

To sum up, the gist of Arrow's theorem is that there is no clearly *good* aggregation function in the general case. Among the many designs we can come up with there will necessarily be some trade-offs involved.

[2] This is arguably the least intuitive property in this list. Without trying to fully justify the inclusion of this property, let us point out that an aggregation function without this property is more easily subject to strategic manipulation.

[3] This also raises the question of whether one might construct an aggregation function based on pairwise majority voting. As it turns out, these types of constructions do not even satisfy Pareto efficiency in general.

3.1.2. Rank-based Aggregation Strategies

Rank-based aggregation functions are based on ordinal measures of utility, meaning that we discard the absolute utility values and consider only the relative ordering of items. There are two main motivations behind that: First of all it might be difficult to compare absolute utility values across several users—utility might not be cardinal to begin with. Secondly we can greatly simplify the analysis of the aggregation function.

Plurality voting

Plurality voting is the extension of majority voting for $|S| > 2$. Users cast a single *vote* and the item with the highest number of votes wins. In order to get a sequence, the winning item is removed from the set and the procedure is applied iteratively until no more items remains. This system is frequently used in political elections. We note that plurality voting does not satisfy the *independence of irrelevant alternatives* property[4].

Borda Count

Each item's Borda count is the sum of its rank over all users' ranking. Items are then ordered by increasing Borda count to form a sequence. It is different from plurality voting in that the item with the highest aggregated ranking does not necessarily have rank 1 in any of the user's ordering. Just like plurality voting Borda count does not satisfy the *independence of irrelevant alternatives* property.

3.1.3. Utility-Based Aggregation Strategies

Considering the cardinal utility values allows more flexibility in the design of an aggregation function. Some care must however be taken to ensure that utility values are indeed comparable across users; in practice it is often necessary to normalize and center the values—especially when they were explicitly given by the users themselves.

Utilitarianism

Utilitarian approaches seek to maximize the *average* utility. For each item we compute its mean utility over all users, and we rank the items in decreasing order. Traditionally, the arithmetic mean is used to compute the average, but an alternative approach is to take the geometric mean. The latter solution is equivalent to taking the arithmetic mean of the logarithm of the utilities, with the effect of penalizing items with higher utility variance across the users.

[4] A spectacular demonstration of this is called the *Spoiler effect* and is well-known by politicians. It happens when a popular candidate has her votes diverted by a similar candidate and loses the election at the expense of a third, very different candidate.

Least Misery

Some people might object that maximizing the average utility is unfair in the sense that there might be users which will be significantly worse off. Indeed, if there is a minority of users whose preferences are different they will likely be ignored in favor of the majority. A *fair* aggregation strategy would try to narrow the gap between the utility across users. One solution is to optimize for the user that is the worst off, which can be achieved by iteratively choosing the items for which the minimum utility is maximal. This method does not scale well with the number of users: a change in a single user's preferences can dramatically change the outcome of the aggregation[5].

There are a number of variants of the above strategies; we mention *average without misery* where the an average utilitarian approach is used but items with very low utility values are excluded from the sequence.

3.1.4. Beyond Preference Aggregation

So far we have discussed how to *aggregate* a set of individual preferences in order to pick the best sequence of items for a group. As there are no definitive solutions emerging from this we might wonder if there are alternative approaches. This is particularly relevant in the context of recommender systems as most of the utility values are not directly available and need to be predicted. A natural question that arises is: can there be a way to model the group's preferences directly, bypassing individual utility prediction? Although we will not explore this approach further we mention that several recommender systems take this alternative path (Jameson and Smyth 2007). Some of them build a group model directly, while others construct intermediate individual or subgroup models before combining them.

3.2. Group Recommender Systems

Group recommender systems have seen relatively little research efforts until recently, especially when compared to the wealth of the literature on individual recommender systems. Both Masthoff (2011) and Jameson and Smyth (2007) provide an excellent survey of the state of the art and the specific challenges related to making recommendations to groups.

Masthoff (2004) looks at several preference aggregation functions in the case of a television recommendation systems that produces a sequence of items TV programs. Two questions she raises are of particular interest:

1. How do we, *humans*, aggregate a set of preferences? Do we consistently follow a specific strategy? If yes, which one?
2. Beyond average utility, how satisfied are we with different aggregation strategies? Does the order of the sequence have an influence?

[5] Informally, it does not always lead to satisfactory outcomes—as perceived by humans—either.

Experimental results tend to show that while we do not all follow a single consistent strategy, we use a mix of average utilitarian strategy and fairness considerations[6]. As for the satisfaction, among other findings we seem to put great importance on having our preferred item in the sequence; also, the first and last elements seem to matter particularly—a finding that is reminiscent of the primacy and recency cognitive biases.

In recent years the interest around group recommender systems appears to have caught up. Baltrunas et al. (2010) evaluate the performance of a group recommender system that aggregates the output of individual recommendations generated by a collaborative filtering system using different rank aggregation strategies. Berkovsky and Freyne (2010) also evaluate the accuracy of different aggregation strategies on a system that recommends recipe to families.

3.2.1. Music Recommender Systems

An interesting line of work is MusicFX (McCarthy and Anagnost 1998; Prasad and McCarthy 1999), which is to the best of our knowledge the first group recommender system related to music. The system is designed for a fitness room, it automatically decides of a musical genre to play a depending on the people currently working out. Users give their preferences explicitly by filling out a form and ratings for each genre are stored on a scale of -2 to 2. In the first version of the system, the aggregation follows a strategy that is roughly equivalent to *average without misery* except that high user ratings have an increased weight. The system enforces some fairness in that it does not play any music genre for which one of the users has a rating of -2. An original aspect of MusicFX is that the aggregation function does not directly output the items to be played; instead it defines a probability distribution over the items so as to provide some variety even when the group does not change. The authors report that the individual preference threshold under which it would not play a given musical genre was soon discovered by some users who found out that it was essentially possible to dictate the genre to be played by manipulating their own preferences. In a second version of the recommender system, they compare the utilitarian strategy to a new algorithm that simulates a market economy. Members of the group are represented by agents that bid on music genres; this gives yet another trade-off between average utility and fairness, but perhaps more importantly avoids the strategic manipulation issues of the first version.

Among other group recommenders related to music, we mention Chao et al. (2005) which highlights the importance of learning what users *don't like* (i.e. their negative preferences) as opposed to what they like.

[6] It is worth noting that the scenario was involving a group of 3 people only.

4. Unison Recommender System

This chapter gives a high-level conceptual presentation of Unison, our group music recommender system. In particular we discuss the practical challenges faced while trying to build a recommender system that works with real music libraries and in a mobile environment. We introduce the different models and techniques that were used to generate the recommendations, as well practical choices that that were made to get the system up and running.

4.1. Challenges

Let us start by mentioning a few considerations that were made during the early stages of out system in order to give a rough idea of the challenges and constraints it will have to overcome. As will be discussed further in chapter 5 we chose to target the Android mobile platform, which has some important implications on the recommender system. We will also draw from a few informal observations we made with respect to *real-world* music libraries.

Accessing the music library Android handles media items (music, but also videos, pictures, etc.) through a centralized service that is part of the operating system. Users typically load their media by connecting the device to their computer and copying the files to the device; by default, there is no synchronization with a desktop media player. When developping an application for Android we can get access to the media through a *content provider*[1] (see Mednieks et al. 2011, chap. 12).

Media metadata Given that the media library is based off of simple files copied over to the device the breadth of metadata available is restricted and their quality rather poor. Through the content provider we get access to the metadata embedded in the audio file (generally through ID3 tags[2]) and to the file path. First of all we observed that the file path often carries little information. Secondly, even though ID3 tags provide ways of embedding a rich variety of metadata in the file we observed that these tags are not used consistently enough to be of any use, with the exception of artist and title information which seem to be the only fields that are more or less consistently maintained by users.

User preferences Just like the rest of the metadata we do not get more information about a user's preferences other than the mere presence of a file in the music library. We intially hoped to leverage the `PCNT` (play count) ID3 tag, but it fell short of our expectations.

[1] In this case the `MediaStore` content provider: `http://developer.android.com/reference/android/provider/MediaStore.html`.

[2] See `http://www.id3.org/`.

Alternatively we could ask the user to give explicit ratings on her music, but it is unrealistic to ask for ratings on thousands of tracks—a typical size for a music library.

Computation & interaction In parallel to challenges related to the scarcity of metadata and other information we have to be particularly considerate with respect to computational requirements, both in terms of processing and in terms of memory. Furthermore the small size of the screen prohibits limits the complexity of interfaces; something to be kept in mind when requiring user interaction.

Opportunities Despite these constraints we mention that a mobile environment also brings positive opportunities. We have for example access to a wide range of sensors (microphone, GPS, camera, etc.), the connectivity (Wi-Fi, Bluetooth, near field communication) is generally rich and most screen nowadays are touch-sensitive. We might find a way to leverage these additional capabilities to improve the system.

4.2. Modelling Tracks

We can now turn out attention to the task of modelling tracks. We first give a brief overview of some of the approaches we investigated before focusing on the latent semantic model we use in the current version of Unison.

4.2.1. Approaches Considered

Before opting for the latent semantic model described below we considered two alternative solutions; we mention them because they both have advantages over our main solution. However we estimated that they would either be too difficult to implement as a first step or that they would not produce significantly better results.

Collaborative Filtering

Putting collaborative filtering under *track modelling* is a bit of a misnomer as collaborative filtering usually sidesteps the need for an explicit model of the item set altogether. We mention it here because it was an obvious contender in the design of our recommender system. The main issue we are facing with collaborative filtering is the *cold start* problem mentioned in chapter 2. It is particularly hitting us because, once again, the space of music tracks is large and we do not want to put any restriction to the music our system can deal with.

One way to alleviate this kind of problem is to use an existing dataset, e.g. of user ratings, which we unfortunately did not find[3]. In light of this we directed our efforts towards a content-based approach, while keeping in ming that we would be able to integrate collaborative aspects later on.

[3] Yahoo! Music provides a dataset (available at `http://webscope.sandbox.yahoo.com/`) with more than 700M ratings from 1.8M users. While very interesting at first sight unfortunately artists and titles of tracks are anonymized making it worthless for our bootstrapping purpose.

Audio Analysis

We also considered borrowing techniques from music information retrieval (section 2.2.1), either to automatically determine the genre of the music or to derive a similarity measure between tracks. This approach has the significant advantage of working on literally any music file, even ones that do not have any associated metadata. It is however delicate to implement on a mobile device as it requires significant computational resources[4]—even though we found some efficient implementations of similarity measures (see e.g. Pampalk 2006). Although we did not investigate this approach further it remains an interesting option as a fallback when there is no metadata.

4.2.2. Latent Semantic Model

As pure collaborative filtering was not an option and metadata very limited, we started looking for external data sources that could be taken advantage of to extract track features. We ended up using an approach that leverages *user-generate tags* to model tracks.

Millison Song Dataset & Last.fm API

The Million Song Dataset (Bertin-Mahieux et al. 2011) is a recent initiative from LabROSA at Columbia University in collaboration with several industrial partners that aims at facilitating large-scale research on music information retrieval. It actually consists of several datasets, all of them referring to the same set of about a million popular music tracks. We list some of the available data.

1. Complete metadata including artist, title, genre, album name, year of release, etc. as well many audio features including tempo, energy, loudness and timbre for each segment of the track. These data are provided by The Echo Nest.

2. The Echo Nest also provides user listening profiles for more than a million users and about 400,000 tracks. We the play count for each *(user, track)* pair[5].

3. User annotations in the form of tags, provided by Last.fm. More exactly, there are 505,216 tracks with at least one tag, 522,366 unique tags and over 8 million associations between a tag and a track.

We are particularly interested by the last dataset; fig. 4.1 gives a few examples of what the data look like. We mention some informal observations we made on the data.

[4] Alternatively we could send the audio file to a third-party server that does the analysis itself, but that would create unreasonable requirements on the bandwidth

[5] Note that this could have helped us in bootstrapping a collaborative recommender system.

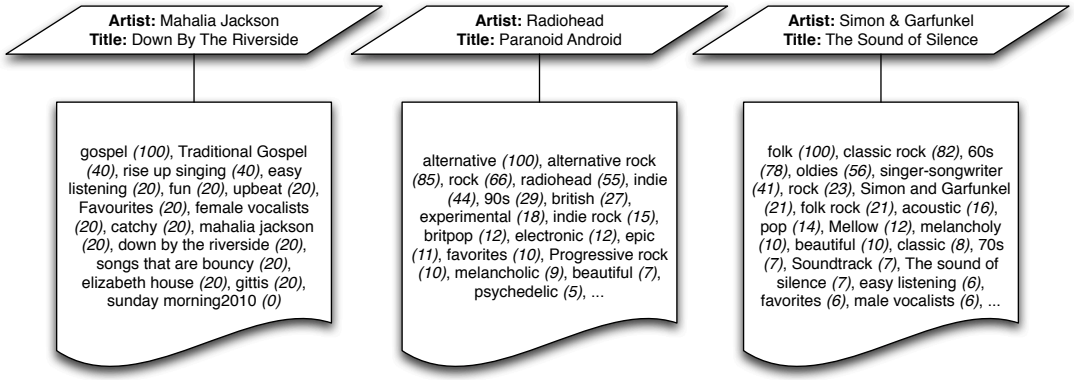

Figure 4.1.: The tags provided by Last.fm for three different tracks. The numbers in parentheses are the tag's *count*, an additional attribute that is available for each tag.

- For each track and each tag we get an additional attribute called *count*. This count is an integer between 0 an 100 and seems to indicate the relative importance of the tag with respect to the track. The dataset does not give any documentation on this attribute but we hypothesize that it might be a normalized indication of how many users used this particular tag to annotate the track.

- At most 100 tags are given for any track (a limit that is shared by the Last.fm API, as will be explained below). With 8 million *(track, tag)* pairs and more than 500,000 tracks the average number of tags is a little more than 16 tags per track. The distribution, however, is not uniform[6].

- We observed that the distribution of the tags themselves follows a power law distribution (the plot of the number of occurences of each tag, not shown here, exhibits a typical heavy tail). The tags occurring most often are respectively *rock*, *pop*, *alternative* and *indie*. Out of the 522,366 tags about 75,000 appear in at least 10 tracks.

- The content of the tags is very diverse and goes far beyond genre information. Even among the tags that are occuring the most often, we find keywords related to emotion (e.g. *beautiful*, *awesome*) time (*80s*, *00s*) and even personal relations to the music (*seen live*).

The exact same data can be queried for tracks that are not part of the Million Song Dataset by using a web service provided by Last.fm[7]. This free, HTTP-based API allows us to leverage Last.fm's huge music catalog and get tags for most of the popular music. Using both the Million Song Dataset and the API, we observed that in practice we could retrieve tags for about 75% of real user libraries. This number is in our opinion quite high: poor metadata (e.g. tracks whose title is *Track 01*) and non-music audio files (podcasts, radio streams, sounds) account for a significant portion of the 25% of untagged music.

[6] Unfortunately, we did not have time to gather more precise statistics about this distribution.
[7] See: `http://www.last.fm/api`.

Latent Semantic Analysis

At this point we can represent a track by a list of tags and for each tag we also have an integer indicating its relative importance. We have in a sense created a *textual* representation of the track: our representation is indeed closely related to the *bag-of-words model*, a popular document model where only the words and their frequencies are kept. We can also think of a track as a huge vector whose components indicate, for each distinct tag, whether the tag is associated with the track or not; or—more interestingly—take the tag's count attribute into account as well. This is also related to a text information retrieval technique known as the *vector space model*. These high-dimensional vectors can however be difficult to deal with and lead sometimes to poor results. Furthermore the dimension of the vectors does not necessarily reflect the *true* dimensionality of the information because synonymy (e.g the tags *disco* and *disco influences*) or noise (e.g. the tags *strangeromanticdeadrockcave*, *v3dd3r*).

We would therefore like to find a smaller space where each dimension corresponds to a *concept* that can encompass many tags; This *latent space* should be carefully built such as to maximize the information we get out of each dimension. A well-known technique stemming out of the text information retrieval community is *latent semantic analysis* whic addresses exactly this kind of problems. It uses a standard matrix factorization technique (singular value decomposition) to uncover the hidden correlations between tracks represented as tag vectors and projects both tags and tracks into a lower-dimensional space. For more details on this technique, please refer to appendix A.

In order to build a latent semantixc space we used data from the Million Song Dataset. We retained all tags that appeared in at least 10 tracks—which amounts to about 75,000 tags. Eliminating tracks that did not have any of these tags reduced the number of tracks to slightly less than 500,000. This resulted in a 74916×471545 tag-track matrix, where each entry in the matrix is copmuted using the Log-Entropy weighting scheme (Dumais 1992). We computed a partial singular value decomposition up to the first 1000 singular values / singular vectors. This could be reasonably fast by using SVDLIBC[8], a program that takes advantage of the sparsity of the matrix (5,427,682 non-zero entries, or a density of about 0.02%). Fig. 4.2 shows the magnitude of the first 1000 singular values; the magnitude of the n^{th} singular value can be thought of as the amount of information the n^{th} dimension carries about the data. We see that the knee of the curve lies around the 70^{th} singular value. We store the singular vectors associated to the tags, i.e. representation of the tags in the latent space in a database. From this database it is trivial to embed any track in the latent space, even one that was not part of the original tag-track matrix. Fig. 4.3 summarizes the process of embedding a track into the latent semantic space starting from its title and artist name.

4.2.3. Classification Experiment

We report on a small experiment that was conducted to get a feel of the quality and relevance of our latent space. In the following we use the 50-dimensional latent space obtained from the rank 50 restriction of the singular value decomposition.

[8] SVDLIBC is developed by Doug Rohde; see `http://tedlab.mit.edu/~dr/SVDLIBC/`.

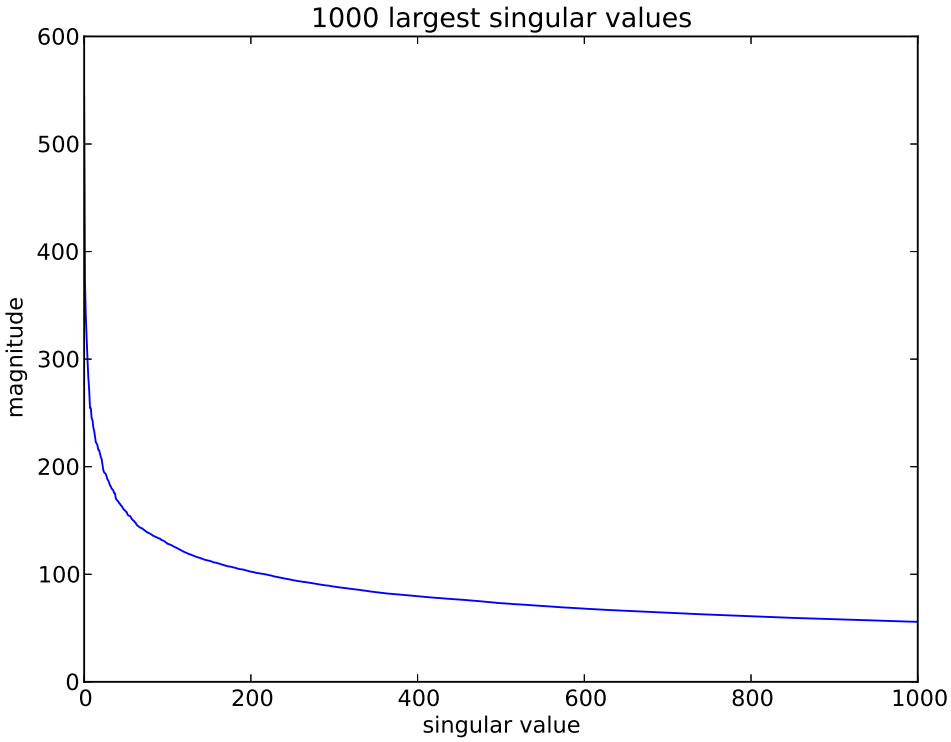

Figure 4.2.: The magnitude of the 1000 first singular values of the tag-track matrix when using the Log-Entropy weighting scheme

We discovered that the Last.fm API could also be used to retrieve information about its users; in particular we can retrieve the list of tracks each user *loved* or *banned* by clicking on a button while listening to one of Last.fm personalized radio streams[9]. We retrieved these two sets for about 30,000 users but only considered a random sample of 11 users having at least 1000 tracks in both list. We kept the same proportion of *loved* and *banned* tracks: exactly 1000 of each.

Our goal was to classify tracks into two classes, *loved* and *banned*, based on their latent space representation. A high classification accuracy would give us confidence in the fact that the latent space is capable of capturing user preferences. Among the different types of classifiers we tried, we report on the one that consistently gave the best performance: a support vector machine classifier with a radial basis function kernel, a simple non-linear kernel that gives good results on many datasets (Hsu et al. 2003). We first used a grid search to find the optimal values for the two model parameters C and γ. Then, for each user, we evaluated the classifier's performance over 100 runs. For each run we picked a random sample of 20% of the user's tracks. Fig. 4.4 shows the average performance for each user.

- The performance varies significantly between users. For some users we have a little more than 80% prediction accuracy, while for others we barely beat a random guess. At first sight there might radically different shapes of preferences—some of which do not seem to be effectively captured by our track model.

[9] See: `http://www.last.fm/api/show/user.getLovedTracks` for example.

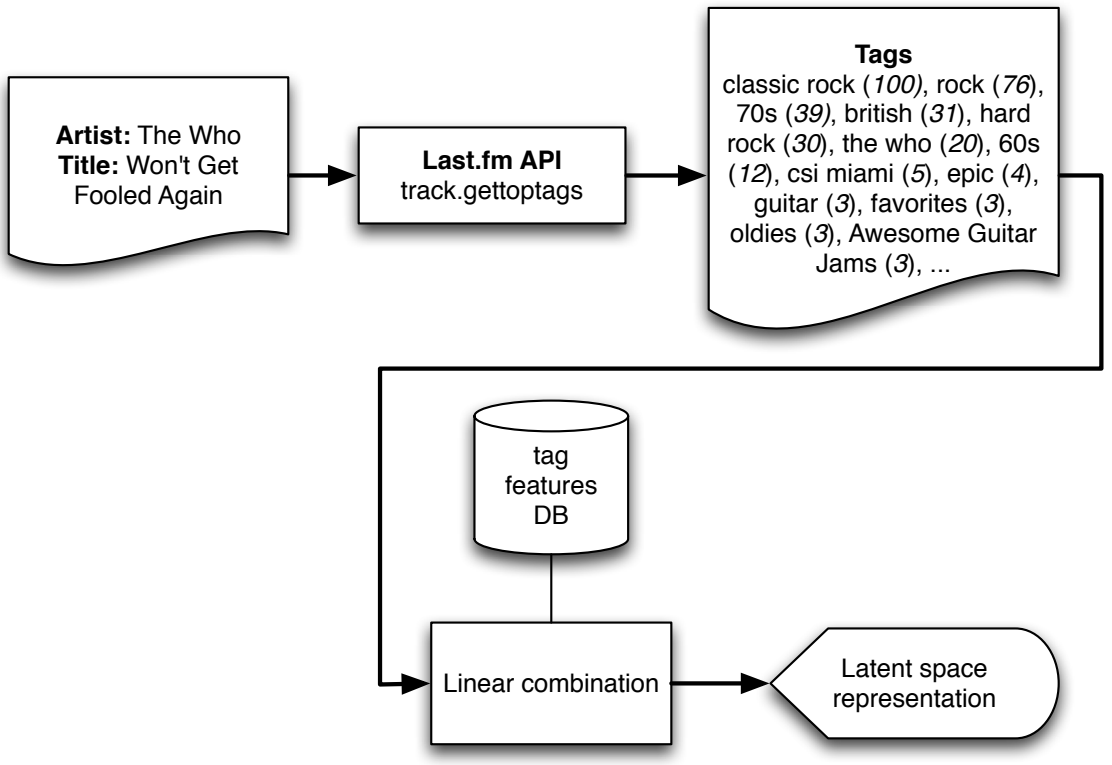

Figure 4.3.: Flow chart showing how a new track is embedded in the latent semantic space. A track is represented as a linear combination of its tags' latent space embedding.

- One has to keep in mind the characteristics of the data; it is not the opinion of 11 users on a random sample of songs. The songs are coming from the Last.fm radio personalized streams which already only play recommended music.
- Finally, it is worth noting that only a very small minority of users had a large number of *banned* tracks: less than 100 users out of our 30,000 sample banned at least 1000 tracks. We might therefore deal with outliers, either music-lovers whose very sharp musical tastes we fail to capture, or users who possibly found an alternative purpose for the love and ban buttons (e.g. as a simple way of bookmarking songs).

Overall this experiment was rather inconclusive. We still take some comfort in observing that we could capture the music preferences of part of the users.

4.3. Modelling User Preferences

We now turn our attention to the next logical step: modelling user preferences. The whole point of modelling tracks was indeed to provide means to the end that is predicting user preferences. Here, the difficulty lies in the fact that we cannot ask too much explicit feedback (e.g. ratings) up front; to make Unison as accessible as possible to a new user we decided to look for an

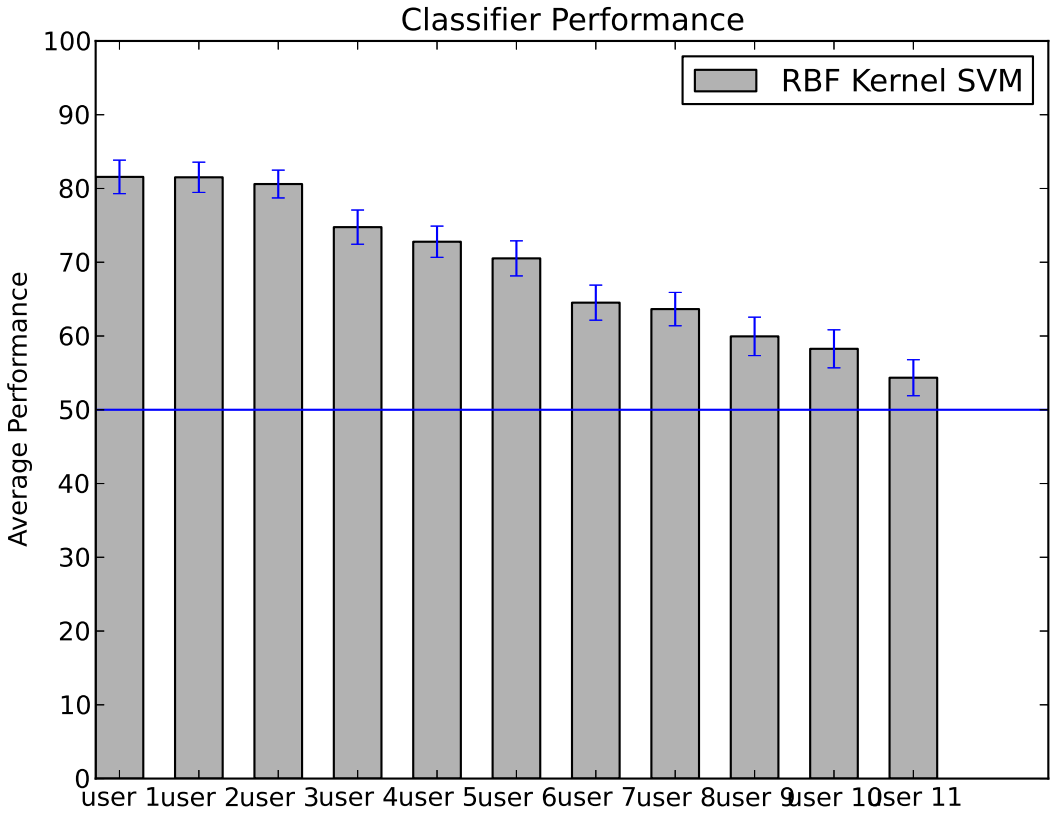

Figure 4.4.: Performance of a non-linear support vector machine (SVM) classifier trained with the tracks' projection onto the 50 first dimensions of the latent space for 11 users. We indicate the average performance and the standard error over 100 runs. The blue line indicates the performance of a random predictor.

implicit way of estimating the user's preferences, one that would not need any user interaction. We make two important assumptions that greatly simplify the problem:

1. Users' utility for a given music track is proportional to the amount of music of the same kind present in their music library, i.e. to the number of tracks that are *close* in some sense. This assumption is arguably not always true, as musical tastes can evolve but users rarely remove music from their library.
2. Users' preferences are static, i.e. they do not depend on external parameters such as location, time and mood. This is clearly a disputable assumption as we do not always listen to the same type of music. However it is extremely helpful in reducing the complexity of the problem.

4.3.1. Gaussian Mixture Model

If we represent tracks in an L-dimensional latent space such as the one presented before, a user's musical library can be considered as a cloud of points. We could hope that these points

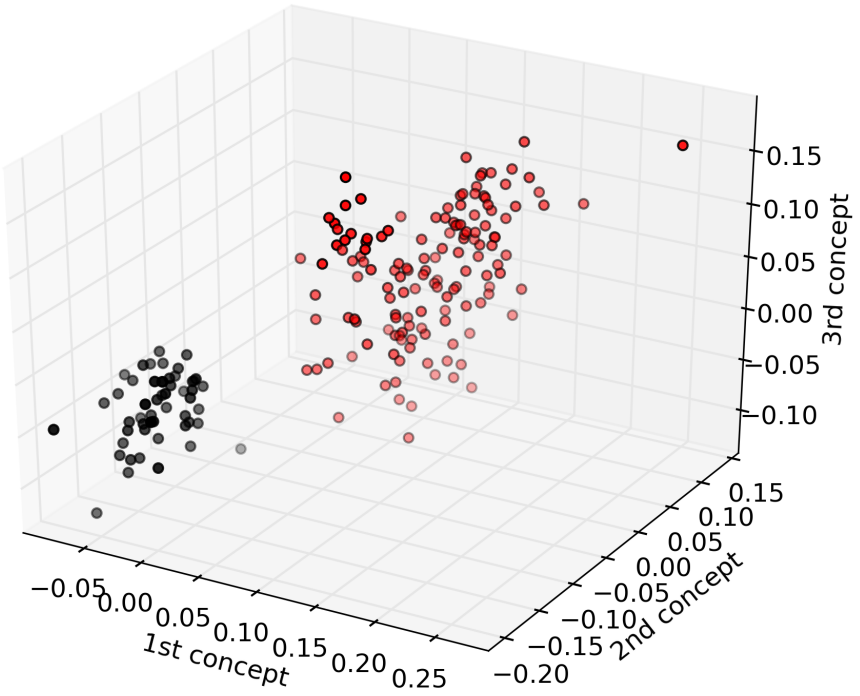

Figure 4.5.: A set of tracks in the 3-dimensional latent semantic space. Black points represent tracks that were manually identified as jazz music while red points were identified as rock music

can be divided into clusters that represent different kinds of music, as shown in fig. 4.5. The first assumption listed above would mean that the relative *density* of points in a certain region of the latent space is proportional to the user's utility for the kind of music it represents. We leverage this last observation in order to build our preference model, which simply amounts to estimating this density from the cloud of points.

We use a Gaussian mixture (Bishop 2006, chap. 9) to model the density; that is, we use a weighted linear combination of K multivariate Gaussian random variables, where K is a model parameter. Gaussian mixtures are known to be good probability density functions approximators, and are very convenient to use. We can think of a library's latent space representation as samples from a gaussian mixture random variable; our task can then simply be thought of as retrofitting a probability distribution onto our cloud of points. In the case of a Gaussian mixture we can search for the maximum likelihood model using the expectation-maximization algorithm (Moon 1996). Fig. 4.6 gives an example of a Gaussian mixture model with 3 components in the 2-dimensional latent space.

Choosing L and K

The dimension L of the latent space as well as the number K of Gaussians in the mixture are two parameters that need to be manually chosen.

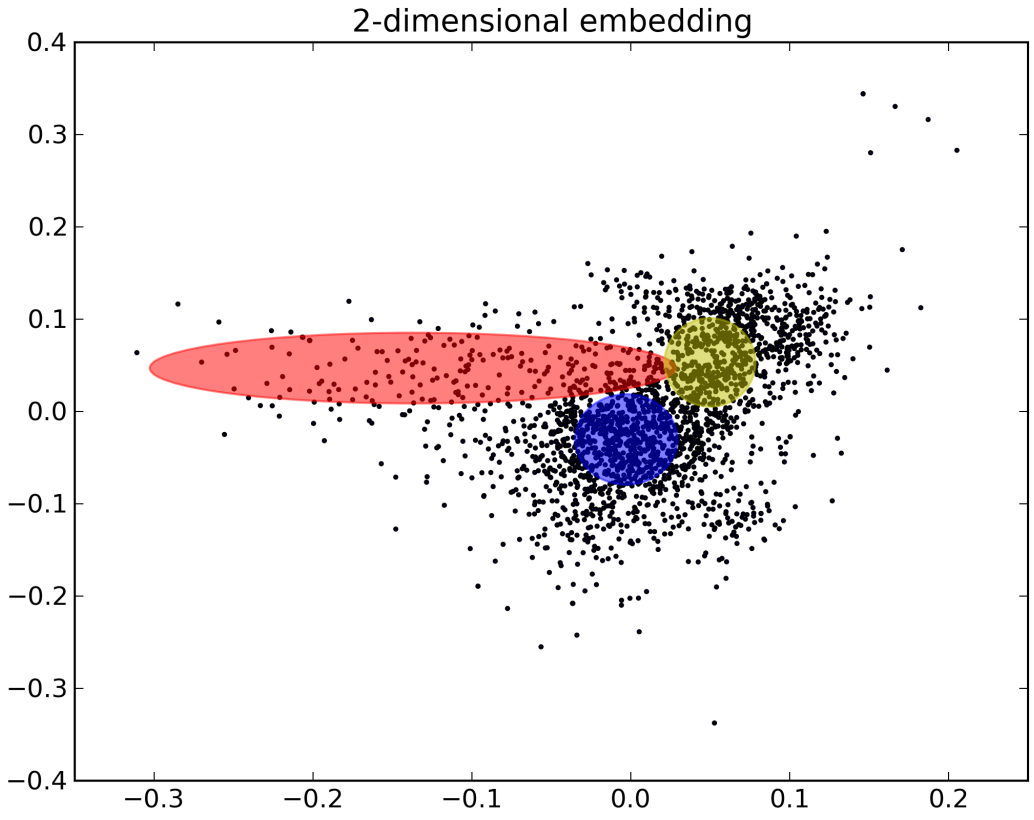

Figure 4.6.: A real music library of about 3500 tracks embedded in the 2-dimensional latent semantic space. We overlay the centers and an indication of the variances of the best gaussian mixture model with 3 components.

The dimension of the latent space is directly linked to the number of *concepts* we think are necessary to model the breadth of user preferences. With a small L, we risk losing some resolution power and not be able to finely differentiate between different preference profiles. In fig. 4.6 we could clearly observe that some regions of the space were corresponding to a certain type of music, but the huge cluster around the origin contains a melting-pot of many different types of music that we would hopefully be able to separate with additional dimensions. However, when increasing the dimensionality we expose ourselves to several problems collectively known under the expression *curse of dimensionality*. Particularly relevant to our clustering problem is the fact that the notion of distance in a high dimension Euclidean space tends to become less meaningful, as most points become equidistant from each other. The choice of the dimensionality is also influenced by the typical size of libraries on which we operate. We decided to choose $L = 5$ as a starting point for several reasons.

1. As we want to be able to easily compare user preferences we need to set L as a fixed parameter shared by all users. We also want to be able to work with small libraries, we thus need a conservative choice.

2. By starting with a smaller dimension it is easier to get an intuitive feeling of the effects at play. We felt that it was still more or less feasible to *understand* a 5-dimensional space

but that additional dimensions would make it very hard.

3. There are probably more than 5 distinct important dimensions in music. However, it is still interesting to see how much we can model with such a limited number of dimensions.

As opposed to L, we can make the number of Gaussian components in the mixture K a *user-dependent* parameter. It makes intuitive sense when considering the broad range of user preferences: some might have very eclectic tastes spanning a wide range of music, while others might prefer one or two narrow genres. We decided to impose an upper-bound of $N/2L$ (where N is the size of the user's library) to force the choice of simpler models when the size of library is limited. We also impose a hard upper bound of 10 to avoid edge-cases[10]. Although these decisions were made without any strong backing we felt that they were necessary to ensure a predictable behavior of the system; we will re-evaluate these decisions when we have more data. To determine K for each user we successively consider the best model for all values of K ranging from 1 to K_{max} as determined by the upper bound described above. We use the Bayesian information criterion (Schwarz 1978), a well-known fitness test to select the final value of K among the candidates. Very broadly speaking, the criterion considers the likelihood of the data under the model but penalizes for the model's complexity. We refer the reader to Bishop (2006, chap. 4) for further information on this model selection method; Burnham and Anderson (2004) also provide insightful explanations on the topic.

Now that we can get for any user and for any track a density value that we interpret as a predicted utility, the last piece missing in our system is a way of aggregating these values across users and form a sequence of tracks for the group.

4.4. Preference Aggregation

In chapter 3 we saw that going from recommendations for single users to recommendations for groups is fundamentally hard, and that there is no correct or incorrect way to aggregate preferences. Therefore we simply present the technique we used to go from the vector of user ratings over a set of track to the order of the sequence in the final playlist. From a practical perspective we would like to achieve two goals:

High average satisfaction intuitively, we want to make users *on average* as happy as possible. This can be seen as a global property: individually, users might be more or less satisfied.

Fairness The counterpart to average satisfaction is trying to ensure that no one is too much penalized, even if for the group's good. This is in some sense a local property—making sure that *every* user is at least somewhat satisfied.

Maximizing average satisfaction while making sure everyone is treated equally is a contradictory goal. At the one end while the solution is optimal on average a minority of users might be bad off. At the other end optimizing for the user that is the worst off does not necessarily lead to acceptable outcomes either. As an initial practical solution for Unison we looked for a middle

[10] In practice on the data we observed so far we did not seem to need this upper bound very often; but it is here just in case.

ground; we chose the multiplicative utilitarian approach described in section 3.1.3, also known as *proportional fairness* in the resource allocation literature[11]. The aggregate utility or aggregate score for item $s \in S$ is computed as follows:

$$u_{group}(s) = \prod_{c \in G} u(c, s) \tag{4.1}$$

This is equivalent to taking the (unnormalized) *geometric mean* of the users' utilities, or maximizing the arithmetic average of the *logarithm* of the utilities. Taking the log has the effect of gradually lessening the weight of high utility values with respect to lower ones, effectively enforcing some level fairness into the choice of the sequence.

We get the final playlist by selecting items iteratively in decreasing order of utility.

[11] A *caveat*: the choice of proportional fairness is usually justified by the law of diminishing marginal returns. We do not have any information that would enable us to claim that the *true* utility (as perceived by the users) based off of our predicted utility values effectively follows this law. Hence we do not have a strong theoretical backing in our choice of using proportional fairness.

5. GroupStreamer Application

Whereas the last chapter was a fairly high-level discussion on how the recommender system works from a conceptual point of view, this chapter will dicuss the *implementation* of the recommender system as seen and manipulated by end users. As such it is much more focused on the technical aspects of the system. We describe the final user-facing product—the GroupStreamer application for Android—and present the technological choices that were made for various components of the system[1].

5.1. Overview

Let us start with some historical context: as the project started, we were considering several directions. One of the earlier motivations for this project was to build a non-trivial application for opportunistic networks; hence we first envisioned a *distributed* recommender system. As these types of networks are not yet well supported by all mobile platforms, we decided to target Android: it is arguably the platform on which it is the easiest to experiment—mostly due to the fact that both hardware and software can be tweaked at will. In particular, we were considering using *AllJoyn*[2], an ad-hoc networking a framework developed by Qualcomm. As the project went forward we decided to slightly narrow the scope and focus on building a *practical* recommender system, and although it did not yield any particular advantage anymore the choice of Android stuck.

The final implementation of the recommender system, GroupStreamer, follows a classical client-server model with a central remote server accessed through the Internet. The system can be decomposed into 3 components, each one of which we will discuss more thoroughly in the subsequent sections.

1. The Android application constitutes the user-facing part of the recommender system. It is relatively lightweight: it listens to changes in the user's music library, plays music and communicates with the remote server, but does not do any other processing.

2. The remote server (publicly accessible through the Internet) does all the computations that generate the recommendations. It stores the information about the libraries of all users. Centralizing these tasks has obvious advantages.

[1] Although in this report we separate the two, concept and implementation are obvisouly deeply intertwined—technological opportunities and limitations were very influential in the design of the Unison recommender system.

[2] See: `https://www.alljoyn.org/`.

3. The communication between client application and remote server is done through the Unison API, a simple API over HTTP.

5.1.1. Development Process

Generally speaking the three components were developped in parallel, which is rather unsurprising. However, the design of the API—which lies at the interface between the Android application and the back-end and forms the narrow waist of GroupStreamer—had to be crystallized quickly. We use Git[3] as version control software. The codebase is split across two repositories; `unison-recsys` contains the code related to the back-end while `unison-android` contains code related to the Android application (names are kept for historical reasons). The two code repositories are publicly available for online consultation[4].

5.2. Android Application

Android is an operating system and application framework primarily targetting mobile devices. It is developped by the Open Handset Alliance, a group of many organizations of which Google is the clear leader. It runs on a wide array of devices ranging from simple mobile phones to tablets and television set-top boxes. It very open and flexible, sometimes at the expense of uniformity. Android applications run on Dalvik, a virtual machine that executes Java bytecode[5]. The primary developement language on Android is Java, although alternatives exist and it is even possible to run native code the Java Native Interface (JNI). Without going into much detail about Android development, let us mention a few basic characteristics.

- Unlike tradition desktop applications, Android applications tend to blend much more into one another and into the system. Applications are loosely coupled and can call each other through *intents*; several functionalities are readily provided by the system, such as a media library, user preferences storage, etc.

- As applications run on devices where computational resources are scarce and battery life is constrained, many low-level events cannot be abstracted away from the application developer. Parts of the application can be destroyed at any moment to free system resources; this results in a particularly defensive programming style.

We refer the reader to Mednieks et al. (2011) for a good overview of the possibilities of the Android platform and some philosophical insights into Android development.

[3] See: `http://git-scm.com/`.

[4] See: `https://github.com/lucasmaystre/`.

[5] Dalvik is one of the core components of the Android platform, and is a mobile-optimized alternative to the traditional Oracle JVM, HotSpot. Actually, Java bytecode is first transformed into a Dalvik executable `.dex` file before being run in the virtual machine.

Version	Codename	API Levels	Distribution
1.5	Cupcake	3	0.3%
1.6	Donut	4	0.6%
2.1	Eclair	7	5.2%
2.2	Froyo	8	19.1%
2.3	Gingerbread	9, 10	65.0%
3.1	Honeycomb	12, 13	2.7%
4.0	Ice Cream Sandwich	14, 15	7.1%

Table 5.1.: Market share of different versions of the Android OS. By only using functionalities introduced at the API level 8 or earlier we reach 93.9% of Android devices.

5.2.1. Implementation Highlights

Since the first release of Android in 2007 many new versions have appeared, each one introducing new features or improvements to existing ones. Table 5.1 gives an overview of the market share of different versions of Android. From an application developer's perspective the choice of a particular version conditions the compatibility with older devices; features available through the bundled libraries are annotated with a so-called *API level* that indicates when they were introduced in the framework[6]. By using features available down to a certain API level we ensure that the application works also on older devices. In the case of GroupStreamer we managed to target API level 8, which makes it compatible with almost 95% of the Android devices in use. Besides the bundled libraries we use three additional third-party libraries.

ActionBarSherlock The latest release of Android, codenamed *Ice Cream Sandwich* introduces important improvements to the user interface. In particular it includes a new design pattern called the *action bar* which provides a much leaner way to navigate in the application. ActionBarSherlock[7] is an open-source third-party library that brings the action bar to earlier versions of Android.

Support Library Android provides additional optional libraries that retrofit some of the newer features into earlier versions of the framework. We use it here mainly because it is a dependency for ActionBarSherlock.

Google Gson The last third-party library that we rely upon is a JSON serialization library. JSON is the format we use for exchanging data through the Unison API, and Gson[8] allows to easily map Java objects to JSON representations and vice-versa.

5.2.2. User Interface

One of the goals of GroupStreamer's design was to make the application intuitive and easy to use, and the user interface as simple and streamlined as possible; as such we tried to ensure that

[6] For example, near field communication (NFC) libraries were introduced in API level 9, and support of Wi-Fi Direct was introduced in API level 14

[7] See: `http://actionbarsherlock.com/`.

[8] See: `http://code.google.com/p/google-gson/`.

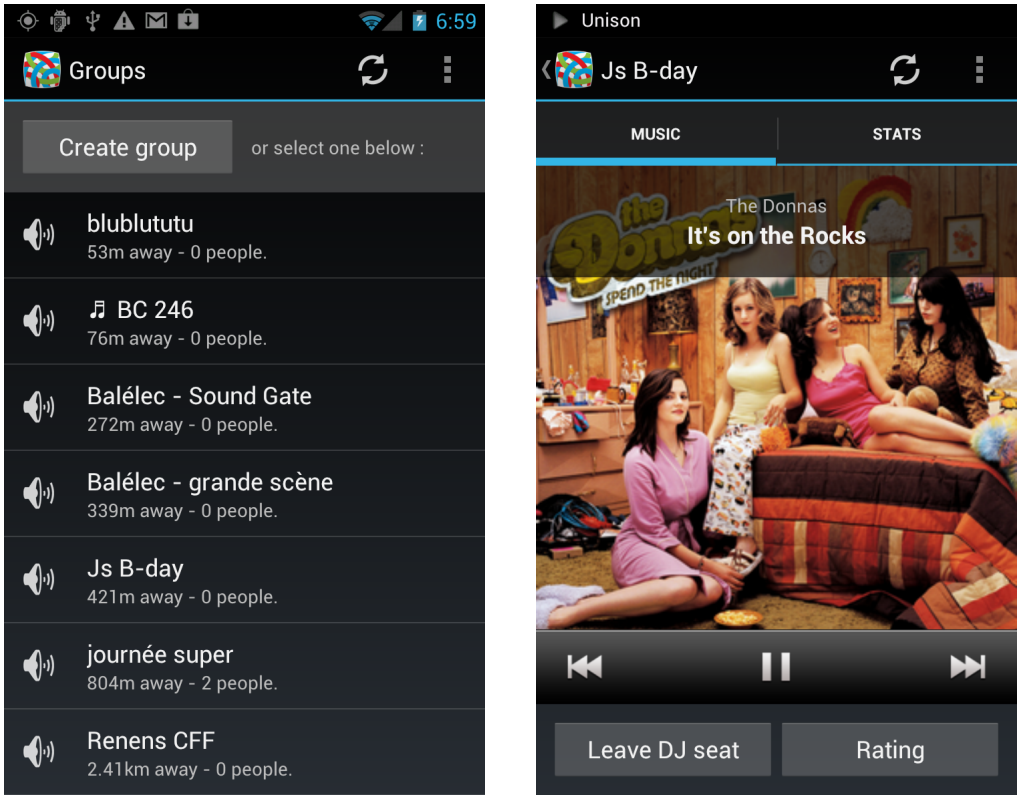

Figure 5.1.: Screenshots of GroupStreamer. On the left, the groups screen lists the nearby groups. On the right, the main screen shows the track currently being played. Non-DJ users do not see the media bar shown here.

there were not too many different screens that the user would be faced with. In this section, we focus on the two most important important screens that the user is interacting with: the list of groups and the music player, both displayed in fig. 5.1.

One important concept to keep in mind is that the GroupStreamer application essentially provides *views* on resources stored on the remote server through HTTP requests. This means that the user has to be connected to the Internet in order to use the application. It also means that the information displayed is in some sense only loosely consistent. Requests for fresh information are sent to the server either automatically every few seconds, or manually by hitting the refresh button.

List of Groups

The first screen users are presented with is a list of groups. In the current version of GroupStreamer users join and leave groups explicitly. Typically, a user creates a new group, gives it a name and asks people around her to join. Users join a group simply by touching the corresponding row in the list of groups. When creating a new group the geographic coordinates (latitude and longitude) of the user who created the group are sent to the server. The user's coordinates are also sent when querying for the list of groups, which allows the server to filter the them according to the distance to the user. The list shows 10 groups that are closest to the

user; in the case where we location information is unavailable[9] we display the most recently created groups as a fallback.

Music Player

The second main screen is the music player. It contains basic metadata (i.e. the name of the artist and the title) about the track that is currently being played, and an image—usually the album cover art. At this point it is important to mention that within a group only a single user (called the DJ) can request a playlist and start playing music. This restriction allows us to broadcast information about the single track that is being played to all the members of the group, such that they all have the title, artist and album cover. Any user can become the group's DJ; requests for the DJ spot are handled in a *first-come, first-served* fashion. Furthermore, the spot has to be explicitly released by the current holder before another user can take it. This limited design has the advantage of keeping things simple. In addition to the information displayed to all users the DJ also sees a control bar with classic media controls. Last but not least let us mention that the playlist request is sent as soon as the user becomes the DJ so that the playback can start the instant the *play* button is pressed (we recall that only music that is locally stored on the DJ's device can be played).

5.3. Back-End Services

The second component of the recommender system is the remote server. It plays a very important role as most of the different tasks required by the system—such as embedding tracks in the latent space, modelling user preferences and generating a playlist for the DJ—are delegated to the server. It is itself composed of several parts, including a relational database, a Web front-end, different maintenance scripts and other services. The configuration is managed in a single YAML[10] file used across the different parts. We use Python[11] for most of the back-end code, essentially as a thin binding layer across many third-party libraries and software components.

5.3.1. Central Database

The database has three roles. First of all it is used to store elementary data needed by the recommender system—that is: information about the users and about the music tracks. Secondly it also stores the information needed for the operation of the system in the short term: which user belongs to which group, who is the DJ, which tracks have been played, etc. Thirdly we want to be able to inspect the history of the recommender system's activity in such a way that we can essentially replay everything that happened. We chose PostgreSQL[12], an open-source

[9] This can happen for a variety of reasons. Users can disable geolocation in their device's settings, or even when geolocation is enabled the location might not have been found yet.

[10] See: `http://yaml.org/`.

[11] See: `http://www.python.org/`.

[12] See: `http://www.postgresql.org/`.

relational database management system.

Table `user` Contains the data relevant to an end-user of the system, notably e-mail address and password hash. It also contains a reference to the group the user is currently in (or `NULL` if he is not using the system). For performance reasons we also store the serialized representation of the user's gaussian mixture model in this table.

Table `group` A group is characterized by its unique ID but also has a name, geographic coordinates (latitude and longitude) and a reference to the user who is currently the DJ. A group can also be set to be *inactive* so that it will not be shown in the application anymore.

Table `track` Contains the information relative to music tracks. A track is uniquely identified by its *(artist, title)* pair. For each track, we store several data obtained from Last.fm such as a URL to the track's image and its tags. For performance reasons, we also persist the latent space representation of the track.

Table `lib_entry` Contains the library entries, i.e. mappings between tracks and users. If the track is present on the user's Android device, we also keep a pointer to its location on the device to be able to play it back. We never remove entries from this table so as to keep a history of the evolution of a user's library, but rows can be invalidated.

Table `group_event` A log of all the important events that happened in a group. Events can be:

- *playing a track* The group's master informs the server every time he starts playing a new track. This information is then relayed to the other members of the group.
- *skipping a track* This event is created when the group's master hits the skip button in the application.
- *joining / leaving a group* These events allow us to record how the group's composition changes over time.
- *change of DJ* Keeps a history of the group's successive DJs.
- *instant rating* Created when users rate a track that is currently being played in the group.

These events do not only leave a precious trail of the the group's history, they could also be used as additional signals by a future version of GroupStreamer that takes the group's dynamics into account.

To access the database from various parts of the back-end we use Storm[13], an object-relational mapper developed by Canonical. It allows easy manipulation of the database through an object-oriented interface.

It is also worth mentioning that PostgreSQL has a high-quality spatial database extension, PostGIS[14], which allows to efficiently store, index and compare geometrical attributes. We

[13] See: `https://storm.canonical.com/`.
[14] See: `http://postgis.refractions.net/`.

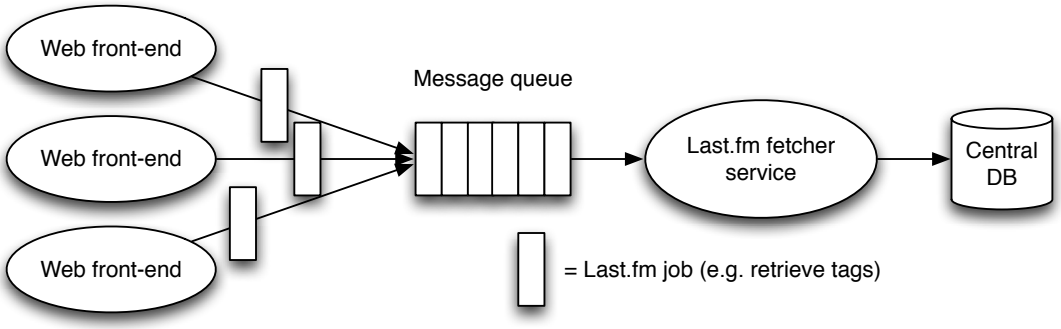

Figure 5.2.: Requests to Last.fm such as fetching the tags for a new track are sent to a queue. A separate process—the fetcher service—then executes the requests asynchronously and sequentially.

could use it to speed up the query of the groups that are nearest to a location—for now, we compute the distance to every single group.

5.3.2. Requests to Last.fm

As discussed in chapter 4, we use the API provided by Last.fm to collect the tags for each new track. We use it for other information as well, such as to get the cover art; currently each new track requires two HTTP requests to Last.fm's servers. Not only do these take a significant amount of time, the rate at which they can be done is also limited by the server to one per second[15]. When a new user arrives and uploads a library with several hundreds to thousands of tracks, it is unthinkable to delay the response for such a long period of time. Therefore we decided to develop a separate, stand-alone service dedicated to the interaction with Last.fm. When a new task involving the Last.fm API arises in any part of the application we send a message to the *Last.fm queue*. The *fetcher* service then processes these messages asynchronously. Fig. 5.2 summarizes this process.

To manage the message queue we use RabbitMQ[16], an open-source implementation of the Advanced Message Queueing Protocol. Note that our use case is extremely basic: we use RabbitMQ as a simple buffer. In the process of building the fetcher service, we also developed our own Python wrapper to facilitate interaction with to the Last.fm API, `liblfm`.

5.3.3. Playlist Generation

For the computations related to modelling and aggregating user preferences as well as for other smaller parts of the system we make heavy use of Python's great numerical analysis and computation libraries: NumPy, SciPy and matplotlib[17]. We also use the excellent Python machine learning library scikit-learn[18] to compute and compare Gaussian mixture models.

[15] In theory and also in practice, as we soon experienced!
[16] See: `http://www.rabbitmq.com/`.
[17] See e.g. `http://scipy.org/Getting_Started` for a short introduction.
[18] See: `http://scikit-learn.org/`.

HTTP Method	Resource	Description
`GET`	`/groups`	List all groups.
`POST`	`/group`	Create a new group.
`GET`	`/groups/18`	Get information about group 18.
`PUT`	`/users/3/group`	Make user 3 join a group.
`DELETE`	`/users/3/group`	Make user 3 leave her group.

Table 5.2.: A subset of the methods provided by the Unison API. Standard HTTP methods are used to create, read, update or delete resources. Additional data in the HTTP request can be used as arguments to the method.

5.4. Unison API

The Unison API is the binding between the user-facing Android application and the remote server. It is for the most part an interface for storing and retrieving information in the central database except for a few methods that also invoke the recommendation engine. We built it as simple Web service; we use therefore HTTP as our transport layer, a solution that has several advantages over using raw network sockets.

- HTTP readily provides many functionalities that would need to be reimplemented if starting from scratch, e.g. user authentication.
- Many platforms have great HTTP client libraries. In particular, Android makes it easy to perform HTTP requests in the application code.
- There are plenty of production ready HTTP servers that we can use on the back-end without much additional effort. It is also arguably easier to scale, as requests can be distributed over many machines.
- Using stateless, client-initiated requests instead of bi-directional persistent connection simplifies the design and the operation of the API. This is especially true in the case of a mobile environment, where network connectivity might be intermittent and persistent connections are fragile.

We considered alternative designs as well, e.g. using a specialized RPC stack such as Apache Thrift[19]. A key point of the Unison API is that it tries to embrace some of the REST principles (Fielding 2000). In practice it means that we try to map URLs to database resources and that we try to stay consistent with the semantics of the HTTP methods we use, especially with respect to the requests' side-effects on the server. In total, there are about 20 different methods; table 5.2 shows a subset of them.

We use JSON[20] to serialize data between client and server. JSON can represent loosely structured data consisting of arrays, dictionnaries and a few primary data types; it is well supported across a wide variety of languages, in particuar in Python and—thanks to Gson—in Java.

We enforce the use of HTTPS—the secure variant of HTTP—in order to provide server authentication and communication secrecy. Client authentication is done with HTTP's basic

[19] See `http://thrift.apache.org/`.
[20] See: `http://www.json.org/`.

```
@app.route('/groups/<int:gid>', methods=['GET'])
@authenticate()  # Require authentication
def group_info(gid):
    # Get the group from the database.
    group = store.get(Group, gid)
    if group is None:
        raise BadRequest('group not found')
    # Return a JSON response: {"name": "..."}.
    return jsonify(name=group.name)
```

Figure 5.3.: Implementation of an API method. The Flask Python package maps URLs to Python functions and provides many helpers (such as serializing data to the JSON format).

access authentication method[21]. The user's e-mail address and password are sent as part of every request's HTTP headers. Although not optimal from a security standpoint, it has the merit of not necessitating any additional round trip.

5.4.1. Back-End

On the back-end we use the Apache HTTP server[22] together with `mod_wsgi`[23], which is more or less the *de facto* standard for calling Python applications through Apache. We also use Flask[24], a minimalistic Python web framework. Fig. 5.3 shows a simplified implementation of the API method which returns information associated with a group. The group is identified through the `gid` parameter; if it is found we return a JSON response with its name.

5.4.2. Front-End

In the front-end application code any interaction with the Unison API goes through an intermediate library providing asynchronous Java bindings. As requests can take up to minutes to complete they cannot be performed on the main application thread, as it would block any kind of user interaction; hence the use of a small wrapper presenting a non-blocking interface. API calls are then sent to a separate thread and upon completion a callback is executed. Fig. 5.4 shows the front-end counterpart to fig. 5.3. The class `UnisonAPI` performs the appropriate HTTP request and takes care of serializing and deserializing the necessary data.

[21] See: `http://tools.ietf.org/html/rfc2617`.
[22] See: `http://httpd.apache.org/`.
[23] See: `http://code.google.com/p/modwsgi/`.
[24] See: `http://flask.pocoo.org/`.

```
UnisonAPI api = new UnisonAPI(email, password);
api.getGroupInfo(groupId, new Handler<GroupInfo>() {
    public void callback(GroupInfo info) {
         // Update the UI with the information we got.
         updateInterface(info);
    }

    public void onError(Error error) {
        // What a Terrible Failure.
        Log.wtf("example", error.toString());
    }
});
```

Figure 5.4.: Typical usage of the Java library that interfaces the Unison API. The calls are non-blocking: we provide asynchronous callbacks that are trigggered upon receiving a response or when an error arises.

6. Perspectives

We have presented Unison, a system that recommends music to groups of people and GroupStreamer, a practical implementation of the system available for the Android platform. To conclude this report we list some ideas left for future work. In addition we identify two promising directions for which our work could a starting point.

First of all, as we gather data from real users of the system, we hope to be able to provide some insights on the performance of our system. Especially, we hope to be able to justify some of our choices and to compare preferences aggregation strategies.

Individual preferences Given enough explicit user ratings, we could start to compare our preference model with *true* user ratings to get a better sense of the predictive power of the preference model we developed in chapter 4.

Aggregation strategies If we start to have a sufficient number of users, we could start testing different preference aggregation strategies. This would require a way to measure the group's satisfaction with the playlist—we could for example measure the average length of a session on GroupStreamer.

The application itself could benefit from several improvements; we list here a selection of enhancements that we feel would greatly benefit the application from a user's perspective.

Missing metadata The quantity of tracks for which we cannot retrieve tags through Last.fm is non-negligible. Currently we simply ignore these tracks when generating the playlist, which is unfortunate. It would be interesting to be able to have a fallback for these situations, e.g. by generating tags from the audio signal (Eck et al. 2008).

Single-user mode At the moment, GroupStreamer is designed to be used only in a group setting. To improve the attractiveness and popularity of the application, we think it would be worthwhile to leverage some of the infrastructure we already have to build a single-user music recommender system.

Automatic group formation Automating the detection of nearby users and the formation of groups would further simplify the user interaction, as well as provide a distinctive *wow effect* to the application.

DJ position As hinted at in chapter 5, we would like to find a more elegant solution for the attribution of the DJ position, currently done in a *first-come, first-served* fashion. We have several ideas on how to improve this, ranging from letting users vote to letting the system decide which user has the best music for the group.

There are also interesting alternatives to be explored for the recommender system at the algorithmic level.

Topic models Beyond latent semantic anaylsis newer dimensionality reduction techniques have been explored recently. Among them are probabilistic topic models (Blei 2011). It would be interesting to replace LSA by a simple topic model such as latent Dirichlet allocation (Blei et al. 2003).

Collaborative filtering As we start collecting user profiles we could start blending collaborative filtering aspects in the rating predictions and in the recommendations.

Exceeding the scope of our music recommender system, we additionally identified two broad directions that we think are of particular interest for further research.

Contextual Recommendations We recall that our recommender system models preferences statically by analyzing the user's library. It seems however fairly reasonable to think that preferences are actually modulated by the context in which the user finds herself in. A simple way to extend our user model to take the context into consideration would to parameterize the weights assigned to the components of the Gaussian mixture (see section 4.3.1). Generally speaking, taking into account contextual information seems to be an very interesting extension of classical recommender systems.

Group Recommenders The topic of generating recommendations for groups is still a widely open research area that could potentially have many applications. For example, automatically adapting the environment goes way beyond music—one could apply the same kind of system for the room temperature or one of the many other decisions affecting groups of people. In this report we mainly saw group recommendation as an additional difficulty on top of individual recommendation. We wonder however if we could not get *benefits* from recommending to groups, e.g. in terms of privacy: a decision for the group might emerge without any of its member giving too much personal information.

A. Latent Semantic Analysis

This appendix gives a short overview of latent semantic analysis (Manning et al. 2008, chap. 18), also known as latent semantic *indexing*. It is a text information retrieval technique originally developped by Deerwester et al. (1990). What seems to be at first simply an optimal way of reducing the dimensionality of a vector space model actually turns out to be have much a deeper meaning: Landauer and Dumais (1997) showed that it could be seen as a general model of how humans learn and represent knowledge. At its core we find *singular value decompisition*, a well-known matrix factorization technique.

A.1. Term-Document Matrix

Consider a collection of N text documents containing words chosen from a dictionary of M distinct terms. Let $T = \{t_i\}_{1 \leq i \leq M}$ be the set of terms and $D = \{d_j\}_{1 \leq j \leq N}$ the set of documents. Using the bag-of-words model we can represent each document as an M-dimensional (column) vector $\mathbf{d_j}$ where the i^{th} component (denoted d_{ij}) corresponds to the *weight* of term t_i for document d_j. A naive way of choosing a value for the weight would be to use the number of occurences of the term, or even just an indicator function. It turns out that slightly more complex weighting schemes work better in practice, although there is no general consensus on which weighting functions works best. These functions can however often be factored into two terms:

$$d_{ij} = gw_i \cdot lw_{ij} \tag{A.1}$$

where gw_i is the *global weight* of term t_i and does not depend on the document (but rather on the term's distribution across documents) and lw_{ij} is a *local* weight that depends on the document.

A.1.1. TF-IDF Weighting

The TF-IDF weighting scheme is probably the most popular one—not only with latent semantic analysis but with vector space models in general. Let tf_{ij}, the *term frequency* simply be the number of occurences of term t_i in document t_j (a local weight). We define the *inverse document frequencency* for the term t_i as:

$$idf_i = \frac{1}{\log \sum_{j=0}^{N} \mathbb{1}\{t_i \in d_j\}} \tag{A.2}$$

where by $t_i \in d_j$ we mean that term t_i appears at least once in document d_j. This means that the weight *decreases* when the term is shared by many documents. Indeed, a term appearing in every document does not bring much information; the inverse document frequency can therefore be seen as indicating the resolving power of the term. The total weight then simply becomes $d_{ij} = tf_{ij} \cdot idf_i$.

A.1.2. Log-Entropy Weighting

Log-Entropy is an alternative weighting scheme empirically shown to work well with latent semantic analysis (Dumais 1992). We first define the (empirical) distribution of the documents conditioned on the term:

$$p_{ij} = \frac{tf_{ij}}{\sum_{j=0}^{N} tf_{ij}} \tag{A.3}$$

The weights are then computed as follows:

$$d_{ij} = \log tf_{ij} \cdot \left(1 - \sum_{j=0}^{N} \frac{p_{ij} \log p_{ij}}{\log N}\right) \tag{A.4}$$

The local weight is just the logarithm of the term frequency. The global weight is much more interesting; upon closer inspection it turns out to be one minus the empirical conditional entropy of the document given the term, normalized between 0 and 1. If the entropy is close to 1, the term tends to be uniformly distributed over all the documents and has thus a limited resolving power. On the other hand if the entropy is low the term is good at reducing the uncertainty on the document.

Once we have chosen our weigthing scheme, we can combine the document vectors $\mathbf{d_j}$ into the $M \times N$ matrix $A = [\mathbf{d_1} \cdots \mathbf{d_N}]$. This matrix is referred to as the *term-document matrix*.

A.2. Singular Value Decomposition

Consider (without loss of generality) the case where $M \leq N$. The singular value decomposition theorem then states that any real $M \times N$ matrix A can be factored in the following way:

$$A = U \cdot \Sigma \cdot V^\top \tag{A.5}$$

where Σ is a $M \times M$ diagonal matrix of decreasing real values, U is an $M \times M$ unitary real matrix and V is an $N \times M$ real matrix such that $V^\top \cdot V = I_M$. the diagonal elements of Σ are

called *singular values* and the columns of U and V are called the left (respectively right) *singular vectors*.

Let $1 \leq k \leq M$. We define the $M \times N$ matrix $A_k = U_k \cdot \Sigma_k \cdot V_k^\top$ where U_k is the $k \times M$ matrix obtained by restricting U to the k first rows, V_k is the $N \times k$ matrix obtained by restricting V to the k first rows, and Σ_k is the $k \times k$ matrix similarly obtained by restricting Σ to the k first rows and columns. It has been shown that the matrix A_k is the best rank-k approximation of A in the least squares sense.

Let us come back to our term-document matrix. We remember that the columns of A are the document vectors of dimension M = the number of distinct terms. When considering the singular value decomposition $A = U \cdot \Sigma \cdot V^\top$ we interpret the matrices as follows[1]:

- The rows of U can be roughly thought of as the representation of the terms in the new basis.
- The elements of S indicate the *importance* of each dimension in accurately reflecting the original matrix.
- The rows of V can be roughly thought of as the representation of the *documents* in the new basis.

By taking the rank-k restriction of the decomposition we embed the terms and documents into a k-dimensional vector space that represents the *best projection* of the data (in the least squares sense) on k dimensions. We call this vector space the *latent semantic space*. Also, we call the elements of its basis the *concepts*; terms and documents can then be simply seen as a mixture (linear combination) of k concepts.

To conclude, we observe that it is very easy to embed a new document in the latent space. Given a new document represented by its term vector $\mathbf{d}_{N+1}$ it is easy to see that the the linear combination of the terms' representation in the latent space:

$$\mathbf{v}_{k,N+1} = \sum_{i=1}^{N} d_{i,N+1} \cdot \mathbf{u}_{k,i} \tag{A.6}$$

where by $\mathbf{u}_{k,i}$ we denote the i^{th} row of matrix U_k, is indeed the latent space embedding of document d_{N+1}.

[1]The interpretations given here are not necessarily very precise, but we think they give the correct intuition.

List of Figures

1.1. Scenario for a group recommender system applied to music. Members of the group have different musical tastes, yet they have to find a consensus on what music to play. 2

2.1. Diagram showing the steps used to compute mel-frequency cepstral coefficients (MFCCs) for a stationary digital sound. MFCCs have been shown to provide a compact representation of a sound's timbre as perceived by humans. 7

4.1. The tags provided by Last.fm for three different tracks. The numbers in parentheses are the tag's *count*, an additional attribute that is available for each tag. 20

4.2. The magnitude of the 1000 first singular values of the tag-track matrix when using the Log-Entropy weighting scheme 22

4.3. Flow chart showing how a new track is embedded in the latent semantic space. A track is represented as a linear combination of its tags' latent space embedding. 23

4.4. Performance of a non-linear support vector machine (SVM) classifier trained with the tracks' projection onto the 50 first dimensions of the latent space for 11 users. We indicate the average performance and the standard error over 100 runs. The blue line indicates the performance of a random predictor. 24

4.5. A set of tracks in the 3-dimensional latent semantic space. Black points represent tracks that were manually identified as jazz music while red points were identified as rock music 25

4.6. A real music library of about 3500 tracks embedded in the 2-dimensional latent semantic space. We overlay the centers and an indication of the variances of the best gaussian mixture model with 3 components. 26

5.1. Screenshots of GroupStreamer. On the left, the groups screen lists the nearby groups. On the right, the main screen shows the track currently being played. Non-DJ users do not see the media bar shown here. 32

5.2. Requests to Last.fm such as fetching the tags for a new track are sent to a queue. A separate process—the fetcher service—then executes the requests asynchronously and sequentially. 35

5.3. Implementation of an API method. The Flask Python package maps URLs to Python functions and provides many helpers (such as serializing data to the JSON format). 37

5.4. Typical usage of the Java library that interfaces the Unison API. The calls are non-blocking: we provide asynchronous callbacks that are trigggered upon receiving a response or when an error arises. 38

Bibliography

Adomavicius and Tuzhilin 2005 Adomavicius, G. ; Tuzhilin, A.: Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. In: *IEEE Transactions on Knowledge and Data Engineering* 17, nr. 6, p. 734–749

Aizenberg et al. 2012 Aizenberg, N. ; Koren, Y. ; Somekh, O.: Build Your Own Music Recommender by Modeling Internet Radio Streams. In: *Proceedings of the 21st International Conference on World Wide Web*, 2012

Alvira et al. 2001 Alvira, M. ; Paris, J. ; Rifkin, R.: *The Audiomomma Music Recommendation System*. 2001. – AI Memo 2001-012

Arrow 1963 Arrow, K.J.: *Social Choice and Individual Values*. 2nd ed. Wiley, 1963

Aucouturier and Pachet 2002 Aucouturier, J.-J. ; Pachet, F.: Music Similarity Measures: What's the Use ? In: *Proceedings of the 3rd International Conference on Music Information Retrieval (ISMIR)*, 2002

Baltrunas et al. 2010 Baltrunas, L. ; Makcinskas, T. ; Ricci, F.: Group Recommendations with Rank Aggregation and Collaborative Filtering. In: *Proceedings of the 4th ACM Conference on Recommender Systems*, 2010, p. 119–126

Berkovsky and Freyne 2010 Berkovsky, S. ; Freyne, J.: Group-Based Recipe Recommendations: Analysis of Data Aggregation Strategies. In: *Proceedings of the 4th ACM Conference on Recommender Systems*, 2010, p. 111–118

Bertin-Mahieux 2012 Bertin-Mahieux, T.: *What is a song, and how many do I have ?* 2012. – available online: `http://labrosa.ee.columbia.edu/millionsong/sites/default/files/cirmmt_workshop_feb2012.pdf`. – retrieved on: June 10th, 2012. – presented at the CIRMMT Workshop 2012

Bertin-Mahieux et al. 2011 Bertin-Mahieux, T. ; Ellis, D.P.W ; Whitman, B. ; Lamere, P.: The Million Song Dataset. In: *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR)*, 2011

Bishop 2006 Bishop, C.M.: *Pattern Recognition and Machine Learning*. Springer, 2006

Blei 2011 Blei, D.M.: Introduction to Probabilistic Topic Models. In: *Communications of the ACM*

Blei et al. 2003 Blei, D.M. ; Ng, A.Y. ; Jordan, M.I.: Latent Dirichlet Allocation. In: *The Journal of Machine Learning Research* 3, p. 993–1022

Burnham and Anderson 2004 Burnham, K.P. ; Anderson, D.R.: Multimodel Inference. In: *Sociological Methods & Research* 33, nr. 2, p. 261–304

Cano et al. 2005 Cano, P. ; Koppenberger, M. ; Wack, N.: Content-based Music Audio Recommendation. In: *Proceedings of the 13th Annual ACM International Conference on Multimedia*, 2005

Casey et al. 2008 Casey, M.A. ; Veltkamp, R. ; Goto, M. ; Leman, M. ; Rhodes, C. ; Slaney, M.: Content-Based Music Information Retrieval: Current Directions and Future Challenges. In: *Proceedings of the IEEE* 96, nr. 4, p. 668–696

Chao et al. 2005 Chao, D.L. ; Balthrop, J. ; Forrest, S.: Adaptive Radio: Achieving Consensus Using Negative Preferences. In: *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work*, 2005

Deerwester et al. 1990 Deerwester, S. ; Dumais, S.T. ; Furnas, G.W. ; Landauer, T.K. ; Harshman, R.: Indexing by Latent Semantic Analysis. In: *Journal of the American Society for Information Science* 41, nr. 6, p. 391–407

Dumais 1992 Dumais, S.T.: Enhancing performance in Latent Semantic Indexing (LSI) Retrieval / Bellcore. 1992 (TM-ARH-017527). – Research Report

Eck et al. 2008 Eck, D. ; Lamere, P. ; Bertin-Mahieux, T. ; Green, S.: Automatic Generation of Social Tags for Music Recommendation. In: *Advances in Neural Information Processing Systems* nr. 20

Fielding 2000 Fielding, R.T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Ph.D. thesis, 2000

Goldberg et al. 2001 Goldberg, K. ; Roeder, T. ; Gupta, D. ; Perkins, C.: Eigentaste: A Constant Time Collaborative Filtering Algorithm. In: *Information Retrieval* 4, nr. 2, p. 133–151

Hofmann 2004 Hofmann, T.: Latent Semantic Models for Collaborative Filtering. In: *ACM Transactions on Information Systems* 22, nr. 1, p. 89–115

Hsu et al. 2003 Hsu, C.W. ; Chang, C.C. ; Lin, C.J.: *A practical guide to support vector classification*. 2003. – available online: `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`. – retrieved on: June 10th, 2012

Jameson and Smyth 2007 Jameson, A. ; Smyth, B.: Recommendation to Groups. In: *The Adaptive Web*

Jannach et al. 2011 Jannach, D. ; Zanker, M. ; Felfernig, A. ; Friedrich, G.: *Recommender Systems: An Introduction*. Cambridge University Press, 2011

Konstan et al. 1998 Konstan, J.A. ; Riedl, J. ; Borchers, A. ; Herlocker, J.L.: Recommender Systems: A GroupLens Perspective / AAAI. 1998 (WS-98-08). – Research Report

Landauer and Dumais 1997 Landauer, T.K. ; Dumais, S.T.: A Solution to Plato's Problem: The Latent Semantic Analysis Theory of Acquisition, Induction, and Representation of Knowledge. In: *Psychological Review* 104, nr. 2, p. 211–240

Li et al. 2012 Li, T. ; Ogihara, M ; Tzanetakis, G.: *Music Data Mining*. CRC Press, 2012

Lloyd 2009 Lloyd, S.: *Automatic Playlist Generation and Music Library Visualisation with Timbral Similarity Measures*, Queen Mary University of London, Master's thesis, 2009

Logan 2000 Logan, B.: Mel Frequency Cepstral Coefficients for Music Modeling. In: *Proceedings of the International Symposium on Music Information Retrieval (ISMIR)*, 2000

Manning et al. 2008 Manning, C.D. ; Raghavan, P. ; Schütze, H.: *Introduction to Information Retrieval*. Cambridge University Press, 2008

Masthoff 2004 Masthoff, J.: Group Modeling: Selecting a Sequence of Television Items to Suit a Group of Viewers. In: *User Modeling and User-Adapted Interaction* 14, nr. 1, p. 37–85

Masthoff 2011 Masthoff, J.: Group Recommender Systems: Combining Individual Models. In: *Recommender Systems Handbook*

McCarthy and Anagnost 1998 McCarthy, J.F. ; Anagnost, T.D.: MusicFX: An Arbiter of Group Preferences for Computer Supported Collaborative Workouts. In: *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, 1998

Mednieks et al. 2011 Mednieks, Z. ; Dornin, L. ; Meike, G.B. ; Nakamura, M.: *Programming Android*. O'Reilly Media, 2011

Moon 1996 Moon, T.K.: The Expectation-Maximization Algorithm. In: *IEEE Signal Processing Magazine* 13, nr. 6, p. 47–60

Nathanson et al. 2007 Nathanson, T. ; Bitton, E. ; Goldberg, K.: Eigentaste 5.0: Constant-Time Adaptability in a Recommender System Using Item Clustering. In: *Proceedings of the 2007 ACM Conference on Recommender systems*, 2007

Pachet and Aucouturier 2004 Pachet, F. ; Aucouturier, J.J.: Improving Timbre Similarity: How high is the sky? In: *Journal of Negative Results in Speech and Audio sciences* 1, nr. 1, p. 1–13

Pampalk 2006 Pampalk, E.: *Computational Models of Music Similarity and their Application in Music Information Retrieval*, Technische Universität Wien, Ph.D. thesis, 2006

Pazzani and Billsus 2007 Pazzani, M. ; Billsus, D.: Content-Based Recommendation Systems. In: *The Adaptive Web*

Platt et al. 2002 Platt, J.C. ; Burges, C.J.C. ; Swenson, S. ; Weare, C. ; Zheng, A.: Learning a Gaussian Process Prior for Automatically Generating Music Playlists. In: *Advances In Neural Information Processing Systems* 2, nr. 14, p. 1425–1432

Popescu and Pu 2011 Popescu, G. ; Pu, P.: Probabilistic Game Theoretic Algorithms for Group Recommender Systems. In: *Proceedings of the 2nd Workshop on Music Recommendation and Discovery (WOMRAD)*, 2011, p. 30–33

Prasad and McCarthy 1999 Prasad, M.V.N. ; McCarthy, J.F.: A Multi-Agent System for Meting Out Influence in an Intelligent Environment. In: *Proceedings of the 11th Conference Innovative Applications of Artificial Intelligence (IAAI'99)*, 1999

Ricci et al. 2011 Ricci, F. ; Rokach, L. ; Shapira, B. ; Kantor, P.B.: *Recommender Systems Handbook*. Springer, 2011

Sarwar et al. 2000 Sarwar, B. ; Karypis, G. ; Konstan, J. ; Riedl, J.: Application of Dimensionality Reduction in Recommender Systems – A Case Study. In: *ACM WebKDD 2000 Web Mining for E-Commerce Workshop*, 2000

Schwarz 1978 Schwarz, G.: Estimating the Dimension of a Model. In: *The Annals of Statistics* 6, nr. 2, p. 461–464

Sen 1986 Sen, A.: Social Choice Theory. In: *Handbook of Mathematical Economics* 3, p. 1073–1181

Su and Khoshgoftaar 2009 Su, X. ; Khoshgoftaar, T.M.: A Survey of Collaborative Filtering Techniques. In: *Advances in Artificial Intelligence* 2009, p. 1–19

Tzanetakis and Cook 1999 Tzanetakis, G. ; Cook, P.: Marsyas: A framework for audio analysis. In: *Organised Sound* 4, nr. 3, p. 169–175

Tzanetakis and Cook 2002 Tzanetakis, G. ; Cook, P.: Musical Genre Classification of Audio Signals. In: *IEEE Transactions on Speech and Audio Processing* 10, nr. 5, p. 293–302

Yoshii et al. 2008 Yoshii, K. ; Goto, M. ; Komatani, K. ; Ogata, T. ; Okuno, H.G.: An Efficient Hybrid Music Recommender System using an Incrementally Trainable Probabilistic Generative Model. In: *IEEE Transactions on Audio, Speech, and Language Processing* 16, nr. 2, p. 435–447