Personalized Content Creation using Recommendation Systems

Einar Søreide Johansen

Master's Thesis in Information Science

Supervisor: Marjia Slavkovik

Department of Information Science and Media Studies

University of Bergen

31.05.2018

TABLE OF CONTENTS

ABSTRACT

This thesis explores whether recommender systems can be used to create personalized content. To this end a prototype was created that generates music based on a user's preferences. The prototype, named RecOrder, is therefore a song composer. The song creation process of RecOrder combines small audio clips into a song. 37 audio clips were created specifically for the prototype to use. These clips are short recordings of a singular instrument and are designed to be modular and fit together in any order.

The selecting of what audio clips to combine, is based on the preference of the users interacting with the prototype. This is done by the implemented recommendation system, which is a *weighted hybrid system*. Hybrid recommendation systems consist of more than one recommendation algorithm. For the prototype, a collaborative item-based recommendation algorithm called *Slope One* was selected. Additionally, a custom knowledge-based algorithm was created as the secondary recommendation algorithm. Once the song has been created, the prototype moves on to a feedback phase. With the feedback phase, the user rates different sections of the generated song. This feedback, which is given on a scale from one to five, will help guide the recommendation system to more accurate recommendations.

An empirical evaluation was conducted, which aimed to establish whether the generated songs were successfully tailored to the tests subjects' personal preferences. 15 test subjects partook in the experiment, selected without any preference to musical background. In the experiment, the test subjects used the prototype to create multiple songs. They also answered questions in a survey, regarding the prototype and the song creation process. The results from the empirical evaluation show a positive trend in terms of user satisfaction. As the prototype creates more songs, the user felt that the songs were becoming gradually more personalized. Using recommendation systems for personalized content creation is also possible. However, there are some limitations that recommendation systems pose towards what domains are suitable. The limitations are related to the cold-start problem. If the recommendation system cannot learn the preferences of the user, it will not be able to yield recommendations. Similarly, the number of available items is also a critical factor for a suitable domain. Too few available items, and the domain is not fitting for a recommendation system approach.

ACKNOWLEDGEMENTS

I want to thank everyone who has supported me through the writing of this thesis; friends and family alike. Especially thanks to the great people I shared room 644, and later room 634 with. You all have provided me with many great laughs and have pushed me to do my best.

A huge thank you to Marjia, my supervisor. You have steered me in the right direction from the start. Your great humour, comfortable chair and unrelenting honesty has been a great motivator.

Additionally, thank you to the people who participated as test subjects for this thesis.

1 INTRODUCTION

1.1 MOTIVATION

Today's digital environment is a crowded place. The internet and its users create and upload more data, video and text than any one person can consume in a lifetime. Video libraries like YouTube[1] have several hundreds of hours of uploaded content each minute (Statistic Brain, 2016). Navigating such vast amounts of information and content is next to impossible if there was no way of filtering the displayed content. There are many ways to filter content. A common method is through personalization. Typically, websites select content that they estimate the user is interested in.

Large websites like YouTube, Amazon[2] and Facebook[3] often tailor the information found on their sites towards each user's individual taste. The sites try to predict the user's taste by finding similarities between content that the user previously has interacted with. For example, in terms of YouTube, if a user likes a series of videos that all are in a certain topic, YouTube will recommend more videos on that topic.

YouTube and Amazon find specific content that their recommendation algorithms predict the user will like. This predicted content is then prioritized over other content when displaying the website to the user. Facebook will offer advertisement based on the users likes and activity. Facebook also creates special posts thanking the user for their continued use of their services and videos celebrating a friendship or an anniversary. These examples are how sites like Facebook try to create a personal relationship between the user and the site. Spotify[4] have another approach to personalization of their content. In the Spotify app the user can find generated playlists, called *Your Daily Mix*[5]. The Spotify app registers what songs the user has listened to lately. Then it generates a series of playlists based on the collected user data. Since the actual creation of the playlist is automated, the user does not have to make any decisions during the process. However, the user can still influence what songs appear in the playlist, by voting "like" or "dislike" on the songs in the playlist. Then Spotify will adjust the playlist according to the votes. In 2009 Google[6] introduced Personal Search for everyone that use their site[7]. Google rearranges the results on the result page according to what they perceive to be the preferences of the user.

These examples highlight a trend towards a personalized internet. The sites create personalized products and services to make people feel welcome at their sites and express themselves in ways they cannot elsewhere. As a result, the user might use the site for a longer time.

With the rise of personalization, the technology to facilitate user generated content also became evident. Content creation is one of the venues of personalization. Content creation is the act of creating something new for a specific individual. What categorises as content is different from what domain the creation is in. For example, in terms of music, content might be a song, lyrics or just a section of a song. It is often domain experts that create content; such as musicians, directors, etc. The content

[1] https://www.youtube.com
[2] https://www.amazon.com
[3] https://www.facebook.com
[4] https://www.spotify.com
[5] https://support.spotify.com/us/using_spotify/features/daily-mix/
[6] https://www.google.com
[7] https://googleblog.blogspot.no/2009/12/personalized-search-for-everyone.html

creators sell the content from time to time, for example music or films. It is possible to combine personalization and content creation in many domains, given that the content creation process includes the user's preferences.

1.2 Objectives

This thesis explores the area of computer assisted content creation, with a special focus on personalization. The thesis uses music to this end. The content creation process uses recommendation system algorithms to personalize the generated content. The recommendation algorithms leverage the preferences of users, and therefore serve as an interesting solution to personalization in content creation.

Two research questions are posed:

- **RQ 1**: How can computer generated music be personalized?
- **RQ 2**: How can a recommendation system be used to create new content?

1.3 Contributions

A prototype was created for this thesis to explore the two research questions. The prototype is a song creation system, that creates songs by combining a series of smaller audio clips. A recommendation system selects which items (audio clips) to include in the song, and therefore acts as the main decision-agent in the prototype. Figure 1 is an overview of the steps the prototype takes in order to create a song. Recommendation systems find content the user might like and help in providing a user-preference oriented approach to content creation. By including a recommendation engine into the song creation process, the user can influence the generated songs by providing feedback (ratings) on the generated song. Although the prototype created for this thesis is in an artistic domain, this content creation approach is certainly applicable in other fields.

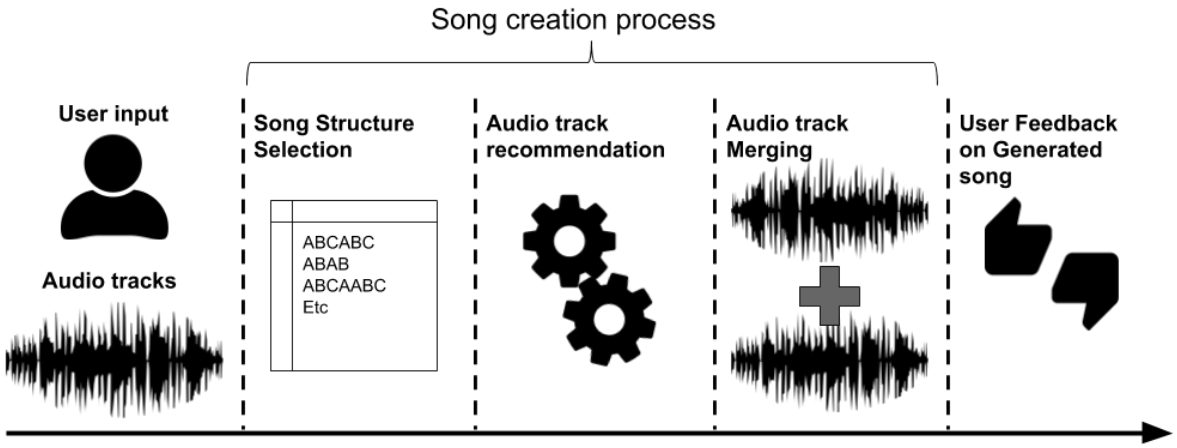

Figure 1 - Schematic overview of the song creation process

The main reason for choosing music is that music is both formulaic and modular in structure. Therefore, music is easy to split into smaller segments. This modular structure lends itself to combining small pieces of music into a song. Rearranging and shuffling around well-constructed sections of music will still make sense when combined into a song. A series of short audio clips was created for this prototype. These clips contain the recordings of a single instrument. By using audio tracks with predefined melodies and rhythms, it is easier to create a more controlled and listenable result. Therefore, most of the audio clips are created to fit together. However, some clips are created to be distinct from the rest of the clips, to check if the user and the algorithm can filter those out.

To validate the quality of the generated songs, and whether the prototype managed to include the user's preferences, we ran a series of tests with multiple test subjects. The testing consists of interactions with the prototype which creates multiple songs, in addition to answering a survey. The survey contains questions that ask the tester to compare several generated songs, and questions where the user will elaborate on their thoughts about the song creation process. We evaluate the success of the prototype based on whether the test subjects that participated in the experiment felt that their preferences were affecting the generated songs. The included discussion will illuminate the degree of success of the prototype.

The use of recommendation systems for personalized content creation, like done in this thesis, is new. We have not been able to find previous work that used recommendation systems to this end. It is therefore interesting to see if recommendation systems are viable as an aid in content creation, and if the recommendations truly make the generated content personalized.

1.4 Thesis Structure

The rest of the thesis is structured as follows. The *Preliminaries* define and describe terminology that is central to the thesis. The definitions cover recommendation systems, music theory and content creation. Additionally, this section includes needed knowledge to understand recommendation systems. Next, the *Related Work* section highlights studies previously done. The selected studies are on recommendation systems, music generating and the combination of music and recommendation systems.

Two sections describe the prototype and its functions. The first highlights how the prototype creates music, and how the song creation process includes the recommendation system. The second section consist of the technical details of the prototype, including a detailed description of the various subprocesses of the prototype. The sections are called *Music Generating with Recommendation Systems* and *RecOrder – The Implementation* respectively.

The *Evaluation Method* section describes the experiment conducted for the thesis. The section includes a detailed description of all the questions asked during the experiment. In addition, the section includes an explanation of the details of the experiment and why there was a need for empirical evaluation.

The second to last section, *Results and Discussion* describe the results of the empirical evaluation. It also includes a discussion of the findings from the evaluation. Lastly, the *Conclusion* section summarizes the thesis and outlines directions for future work.

2 PRELIMINARIES

The following section contains an overview of theory and concept related to the research questions. It contains a description of recommendation systems, various musical terms and content creation. In addition, the recommendation system section describes five related topics, three of which is the most common sources to error found in recommendation systems. These five topics are: *feedback, the cold-start problem, active learning, the long-tail effect* and *over-specialization*.

2.1 RECOMMENDATION SYSTEMS AND RELATED TERMS

Many of today's computer systems and websites provide the user with a vast library of content. Most of those systems contain a recommendation system component. Recommendation systems (RS) are a type of information retrieval algorithm, designed to find and highlight content to the user. The goal of RS systems is to provide the user with a relevant and desirable recommendations, based on their preferences. To achieve this, the recommendation engine will analyse user and item data, looking for patterns and correlations between the data. The recommendation engine then uses these patterns to predict what the user will like (Aggarwal, 2016, p. 2). The result is a personalized set of recommendations, that are unique to each person.

Examples of recommendation systems are not hard to find. Both YouTube and Netflix[8] suggest content to their users by utilizing recommendation systems. These recommendation systems scan the user's activity, and recommend content based on patterns they find in the user data. With those two mentioned examples in mind, it is not hard to visualize the sites without their suggestions. It would be hard to find new content and much of their catalogue would go un-discovered. In fact, most websites that utilize recommendation systems are sites that find it preferable that the user explores as much of their catalogue as possible and that often have large catalogues which makes this hard. Amazon was one of the early adopters that utilized this technology to recommend content to the users (Aggarwal, 2016, p. 5).

By recommending content that the user potentially would not find on their own, the sites try to counter a common problem within any website with a large catalogue: how to make sure that the user sees as much of the content as possible. By exposing the user to a larger set of content, the user is much more likely to purchase items from the site (in the case of mercantile sites), or simply stay longer on the site (in the case of entertainment sites). Because of this, recommendation systems are beneficial for both the user and the system-owner.

Aggarwal, McNee, *et al.* highlights four important goals that maximise the potential of a successful recommendation: *relevance, novelty, serendipity* and *increasing diversity* (2016, pp. 3-4; 2006). A definition of *Relevance* is how fitting the recommendation is to what the user is currently looking for. Aggarwal argues that having relevant recommendations is the most important part of a recommendation system (2016). However, relevance is no good on its own. The recommendation at hand can be relevant, but the user might have seen it before. This highlights the importance of *novelty*; whether a user has seen an item before. If the user only gets recommendations for items they already have seen, the recommendations will be of a lesser quality. In parallel to novelty, Aggarwal explains that the users surprise of seeing a new item is also important. This is called *serendipity*. The main difference

[8] www.netflix.com

from novelty is that serendipity it not only a new item, it can also be an item from a different (but similar) category (Aggarwal, 2016). If the user expects the recommendation, the serendipity is low. Thus, considering serendipity as one of the important goals is valid. These three goals should ensure good recommendations; however, they can result in a recommendation that the user does not like. Aggarwal explains that an *increasing diversity* in the recommended items is also necessary, in order to maximise the probability that the user likes one of the suggested items. A varied, side by side presentation of items from different categories is something to strive for (Aggarwal, 2016).

2.1.1 Different Types of Recommendation Systems

There are several different types of recommendation systems. The main difference between these types are on what they base their recommendation. These are the three broad categories of recommendation system: *content-based, collaborative* and *knowledge-based filtering*. In addition to these three main categories, there are several minor sub-categories of recommendation types (Aggarwal, 2016).

As the name suggest content-based recommendation are basing the decisions on the properties of the content. These properties are also called item-descriptors or attributes. To find similar content to what the user already like the recommendation system search for similar content using these attributes. Collaborative filtering on the other hand, uses the actions (ratings) of other users to determine fitting recommendations. This type of filtering relies on an assumption that there is a connection between users that rate items similarly (O'Donovan & Smyth, 2005). The assumption is that if person A and person B both like item 1 and person A likes item 2, person B would also probably like item 2. The third type, knowledge-based filtering, use specific knowledge found in each domain to make predictions. Expressing the domain knowledge is done through formalized rules, which in some cases are rules that narrow the results.

Content-based filtering

Content-based filtering use the content of the items to determine what is a fitting recommendation. As a result, it is dependent on well described items. In this context, a well described item is an item that has attributes that describe the item fully. One should be able to examine the attributes and conclude both what the item is and how it differs from other items. Attributes can be a description text, tags, metadata, etc. It is not feasible to create well described items in all systems that require recommendation systems. In the cases where the attributes cannot describe the content fully, content-based filtering is the wrong type of filtering. An example of a good use case for content-based algorithms is recommending news articles similar to the one viewed. The algorithm scan for words that appear in both articles, for shared authors or descriptor tags (Aggarwal, 2016, p. 140).

A content-based system normally consists of two data sources. The first source is a description of the items in the database, as described above. The second data source is the generated feedback from the user. A *user profile* stores this feedback, creating an overview of the user's preferences. When a user makes a choice related to an item, the user profile saves the rating. Each user profile is kept separate, and the ratings of other users does not affect the recommendations of the current user. This in turn means that there is no need for large comparisons between users (Aggarwal, 2016, p. 140). This type of recommendation algorithm is good at recommending items, despite the addition of new items. However, when adding a new user to the system, content-based filtering struggles (Aggarwal, 2016, p. 15). In that case, the system has no notion of what the user prefers, and therefore cannot

make a prediction. Since the comparison is between the similarity of items, rather than user's activity, recommendations for content-based systems might appear obvious or not surprising. In other words, content-based filtering often lacks in serendipity. The system will only recommend the most similar items, and not something completely "new". This might not be preferable to what the system at hand needs. To keep a product from becoming monotonous and too repetitive there needs to be a certain amount of surprise (McNee, *et al.*, 2006).

Collaborative filtering

As the name suggests collaborative filtering bases its recommendations on the similarity of a group of users. By leveraging the combination of the user's ratings, the system predicts a new rating or recommends an item to the user (Herlocker, et al., 2000). There are two approaches to collaborative filtering: memory-based (also called neighbourhood-based) approach and model-based approach (Lemire & Maclachlan, 2005; Aggarwal, 2016). This text will be primarily referring to the memory-based approach. The other approach is model-based collaborative filtering, which is closer to machine learning techniques (Aggarwal, 2016, pp. 8-9). This text will not feature the model-based approach. Therefore, in the rest of the thesis we refer to memory-based (neighbourhood-based) approach as collaborative filtering.

Within memory-based collaborative filtering, there are two main groups: *item-based* and *user-based*. Their difference is in what aspect of the dataset they compare. When determining how well a user (user A) will like an item (item 1), *item-based filtering* will find items with similar ratings to item 1. The similarity of ratings is based on a comparison between user A and other users. By similar it is meant similar in terms of user A's preferences. For example, if several other users have rated an item with a similar score to user A's rating of said item, they are considered similar. Then, by comparing what other users have rated item 1, with items that both user A and the other users have rated, one could predict how well user A will like item 1. *User-based filtering* compares what other users with a similar rating profile as the current user have rated the item at hand. The average of those ratings will determine if the user will like the item (Aggarwal, 2016, p. 9).

Since the comparisons are between user to user or item to item, most collaborative filtering methods are resilient to adding new items. Meaning, the addition of new items does not impact other ratings or items directly. Adding more items does not affect the score of previous items. There are, naturally, some exceptions. For example, item-based filtering cannot provide a user with a recommendation for an item that no-one has rated. This touches on the cold-start problem (defined in Section 2.1.4). Likewise, the user-based approach cannot recommend content to a completely new user, but neither can item-based.

Item-based RS also differentiate itself from user-based RS in accuracy versus serendipity. Item-based systems tend to be more accurate than user-based systems (Aggarwal, 2016). This is because the item-based recommendation bases the prediction on the user's own ratings, contrary to user-based RS which primarily bases the recommendation on the similarity of other user's ratings. This weight on the user's own preferences has a negative aspect: it might lead to the recommendations being boring or obvious. When the only input the system gets is the user's own preferences, the system will recommend no new types of content (Aggarwal, 2016). This is because the item-based approach will recommend similar items, while the user-based approach will recommend items that similar users also liked.

It is the difference between the items (or the users) that determine the recommendation. To find that recommendation the system must make a series of calculation. For example, computing the rating-difference between each item to item (Aggarwal, 2016, pp. 41-42). In systems with a large set of items, this will demand both a lot of processing power and quite a long time. In addition, this process must run regularly, since a user's ratings might change with the addition or removal of new users or items. As a result, running these calculations prior to giving new recommendations is advantageous, or else the system risks being inaccurate. Aggarwal calls these series of calculations the *offline phase*. As he highlights, there is a tendency to make the offline phase more demanding than the *online phase*[9] (Aggarwal, 2016, p. 41). This is helpful, since it leverages the larger available timespan of an offline phase, in contrast to the online phase. This in turn, diminish the online recommendation task, resulting in a more efficient recommendation. However, since the offline phase potentially must compute the difference between every item, it can be very demanding in terms of processing power. In fact, with the addition of more items and users the offline phase gets increasingly more demanding.

Collaborative filtering is often easy to implement, due to the relative simple math behind them (Aggarwal, 2016, p. 44; Lemire & Maclachlan, 2005). Most systems that need a recommendation system already have a catalogue of items and a set of user profiles. Very little alterations to those existing systems are necessary. It is only necessary to add a system for rating items, along with an offline phase. Of course, this will be slightly different from system to system, but it is a smaller change compared to for example content-based filtering. Since content-based filtering depends on well described items, adding it to an existing system could potentially lead to a change in every item in the content library. On the other hand, implementing a collaborative filtering system only adds to an existing system. The addition of collaborative filtering to an existing system does not need the developers to change the user-profiles nor the items.

Knowledge-based filtering

Not all systems have the luxury of having a steady stream of user activity. Domains such as finance or vehicle-purchases do not have particularly active users. Users interact with the system rarely, and they are very specific on what they want. The typical use is an item that costs a lot, and the user rarely needs to buy. For example, a house or a car. Since there is little information on the user's preferences, predictions can be hard to make. Some of these domains do not have a uniform set of properties for items, and some items might contain certain properties, while others might not. Housing is a good example: some houses have an included garage, while other houses do not. There might be multiple floors in some houses, while other houses only have one. In addition, the recommendation of certain items might be sensitive to time. For example, it is preferable to recommend a new computer model instead of an old one, even though the old one might be more in line with user's specifications. In these situations, a knowledge-based filtering (KBF) algorithm is the preferred choice. Knowledge-based systems take the domains specifics into account when constructing a recommendation.

KBF-systems are heavily dependent on the data being structured using relational attributes, rather than in text (keywords, text-parsing, etc). KBF-systems are highly customized systems, specifically tailored to both their domain and to their specific implementation. It is therefore hard to use them across different use-cases (Aggarwal, 2016).

[9] The online phase is where the actual predictions are made. The offline phase therefore prepares the data for the online phase to use (Aggarwal, 2016, p. 41).

There is a unique challenge with knowledge-based systems: they are often case based. This means that they do not have access to a user library, or that using the system will create a new "user". As a result, the knowledge-based filtering does not learn from previous iterations (Burke, 2007). KBF-systems are basing their predictions on the "now", instead of historical data. As a result, there must be a way for the system to circumvent the learning and arrive at a recommendation (Aggarwal, 2016). There are two main types of KBF-systems: *Constraint-based* and *Case-based.*

Constraint-based systems is, as the name suggests, when the user defines constraints (or requirements) to the recommendation. These constraints are typically upper or lower limits on certain item properties, for example price or number of bedrooms when buying houses. In addition, it is possible to make constraints between user properties (like age or sex) and item properties (number of bedrooms or price). That way, one can make recommendations that reflect the nature of the domain. For example, people that are looking to buy a house when they are between 30 and 40 years old, might want more bedrooms (since they might have or want kids), than both older and younger buyers. This can be summarised into a rule (Aggarwal, 2016, pp. 172-174). Case-based systems is when you use the result of the previous query as a basis for the next one. The user guides the query by specifying alterations to different parameters. Aggarwal highlights an example: "*Give me more items like X, but they are different in attribute(s) Y according to guidance Z."* (Aggarwal, 2016). The possible results of the search are narrowed for each iteration.

Constraint-based systems are the most relevant for the prototype created for this thesis. One of the known downsides to constraint-based systems is that if you repeat the same steps you would end up with the same result. The repeat of the same results is normally not the case in recommendation systems, as the system would register that the user has seen the results. However, in terms of KBF-systems, it is mostly due to the session-by-session nature, but the domain features implemented in the constraints is also a factor (Aggarwal, 2016). In a normal use-case the repeated results could be considered a negative aspect, since you want to have the user see new content. However, in the case of music, repetition is not necessarily a bad thing. Most songs have repetitious elements, for example in the chorus.

2.1.2 Hybrid Recommendation Systems

All recommendation techniques have situations where providing a recommendation is impossible or where the recommendations are too uncertain to be valid (Burke, 2007). The most known of these situations, is the cold-start issue (Section 2.1.4 defines the cold-start issue). There are several ways to address this issue. One of the most common ways of guaranteeing valid recommendations is to create a hybrid recommendation system. As the name suggests, a hybrid system is the combination of several recommendation algorithms into one system. Hybrid systems are designed to counter the shortcomings of the selected recommendation algorithms with the strengths of the other(s) algorithms. Thus, a selection of algorithms with different strengths and weaknesses is necessary. Burke highlights one such combination:

> *[…] a collaborative system and a knowledge-based system might be combined so that the knowledge-based component can compensate for the cold-start problem, providing recommendations to new users whose profiles are too small to give the collaborative technique any traction, and the collaborative component can work its statistical magic by finding peer users who share unexpected niches in the preference space that no knowledge engineer could have predicted. (Burke, 2007)*

The prototype created for this thesis use the two recommendation system types that Burke mention in the citation above. However, there are different approaches to the combination of algorithms. Four of the most common approaches are: *Weighted*, *Mixed, Switching* and *Cascade* (Burke, 2007; Aggarwal, 2016).

A *weighted hybrid* system combines the results of two recommendation algorithms, then sorts the result by the highest scored items. There are several ways to combine the result set from the two algorithms. The way that the two result sets are combined will affect the end-result. In some settings, it is preferable to use an intersection of the two sets. The intersection will result in a smaller number of items, since the intersection only includes the items from both result-sets. The other option is to merge the two result-sets. Then the result might be a larger set, but in turn more generous with its recommendations. The algorithm uses the combination of the scores from the two lists to sort the resulting list (Burke, 2007).

A *mixed hybrid* system is similar to the *weighted hybrid* approach, in that it also has two recommendation systems producing lists of recommended items. The lists of recommendations are then combined (*i.e.* mixed) into one list of recommendations. The combination of the lists can be based on different criteria, often dependent on the setting of the recommendation. In some settings, it is preferable to use one of the recommendations engines over other, thus giving that engine's results a priority in the merging of the lists (Burke, 2007).

Switching hybrid is a system that chooses the most fitting algorithm based on which recommendation algorithm will give the best recommendation at the time. It is possible for a switching hybrid system to have more than two recommendation algorithms, where most of the algorithms work as a fail-safe or backup from the primary algorithm (Burke, 2007). In other words, in the cases where the primary recommendation algorithm cannot give a good recommendation, one of the other algorithms will take over. Using the two algorithm-types from the quote above as an example, we can highlight how a switching system operates. The primary engine will be the collaborative filtering approach, and this algorithm will therefore do most of the recommendations. However, in the cases where making a recommendation is impossible (for example with the addition of a new user), the system switches to the knowledge-based algorithm.

A *cascade hybrid* system on the other hand, runs several recommendation algorithms in a sequence. Each of the algorithms in the sequence refines the results of the previous one. As highlighted by Burke, in a cascade hybrid system the secondary recommendation algorithm is often used to break ties in the ratings from the previous algorithm(s) (2007).

2.1.3 Feedback

Feedback is essential to recommendation systems. It the way the user can express if they like or dislike the recommended content, and it is through feedback that the recommendation systems learn and improves. Feedback can be express in many ways, but one of the most common ways is a scale of number, for example the five-star rating scale (Aggarwal, 2016, p. 10). When a recommendation system gets the ratings for an item from the user, it will use the rating to predict if the user will like similar items.

There are two types of feedback: *explicit* and *implicit* (Herlocker, et al., 2000; Herlocker, 2000, p. 27; Aggarwal, 2016, pp. 10-11). A rating given by explicit feedback can for example be that the user rates an item with a thumbs-up or down. It is explicit because the user is knowingly and intentionally rating the item. An implicit rating, on the other hand, is when the user is showing preferences in the form of

their actions. For example, the user is going to buy some new item, and the store presents her with two recommendations of relevant items. Both items are recommended based on her preferences. If the user clicks on one of the items, a system that listens for implicit feedback will store that click and interpret it as a sign of interest for that item. Explicit ratings are often more accurate than implicit ratings, since the system needs to infer the user's preferences from implicit ratings. However, implicit ratings are easier to collect (Schafer, et al., 2007). All recommendation systems can use both explicit and implicit ratings.

2.1.4 The Cold-Start Problem

The cold-start problem is a well-known problem in the field of recommendation systems. The problem occurs in systems where there is a user-base, a library of items and where the users can rate the items. As the name suggests, it is related to starting a new process, when there are not enough ratings available to the recommendation algorithm (Bobadilla, *et al.*, 2012). The cold-start problem is most apparent in collaborative filtering methods, where there is a larger reliance on historical data (Aggarwal, 2016). There are three main causes to the cold-start problem, all related to the lack of ratings (Lika, *et al*., 2014; Bobadilla, *et al.*, 2012).

The first cause is with the addition of a new user to the system. The recommendation engine has no knowledge of the new user's preferences. Therefore, the system has no historical data to base the recommendations on. For example, if a new user joins Netflix, the system has no knowledge of what movies the user likes and dislikes. As a result, it is impossible for the system to give a personalized recommendation. Both collaborative filtering and content-based algorithms are particularly bad at handling the addition of new users, since they both leverage the user's preferences (Aggarwal, 2016).

The addition of a new item highlights the second cause. When there is a new item added to the content-library, there is no ratings to connect the item to an existing set of user preferences. Since no users have rated the item, collaborative filtering algorithms cannot compare the item to other items. Content-based algorithms however, do not encounter this problem, since they use the item description as a basis for the recommendation (Aggarwal, 2016).

The third cause of the cold-start problem is the combination of both the previous causes. There are no historical data to base decisions on with both new items and users. Bobadilla, *et al*. define this as the *new community problem* (2012). The new community problem can occur with the first implementation of a recommendation algorithm in a system. Since no user data exists, neither collaborative or content-based filtering can make valid recommendations. This cause of the cold-start problem is the rarest of the three. Two ways to counter this problem is to have the users answer queries about their preferences, or to wait until there are enough users and ratings (Bobadilla, et al., 2012).

Systems that do not rely on user ratings typically do not encounter the cold-start issue. An example of such systems are the knowledge-based filtering algorithms. It is possible to use such algorithms in unison with systems that are weak to the cold-start problem. By combining the strengths of two systems, one can try to negate the individual systems' weaknesses. This union of recommendation algorithms is referred to as hybrid recommendation systems (Lam, et al., 2008). Other systems have been proposed to counteract the cold-start problem *(Feil, et al., 2016; Lika, et al., 2014; Bobadilla, et al.*, 2012).

2.1.5 Active Learning

The process of mapping a user's preferences can be demanding, both in terms of time and in resources. This is tied to the cold-start problem. Active learning is a way of quickening the process of finding the user's preferences in areas of the item-catalogue that the user has not seen. Aggarwal (2016, pp. 25, 430) explains that active learning methods attempt to give the recommendation system better accuracy in their predictions by asking the user to rate specific user-item combinations. By asking the user to rate specific items, a broader model of the user's preferences can be made.

> *"The simplest approach to active learning is to query for items that have been rated sparsely by the users. This can naturally help in the cold-start setting. However, such an approach is useful only in the initial stages of the recommender system setup." (Aggarwal, 2016, p. 431)*

The challenge with active learning arise if a user has no relation to the item that they are asked to rate. Therefore, a smart selection of item is necessary. In addition, the active learning process gets more complicated once the initial stages of the recommender system setup, as mentioned by Aggarwal in the quote above. The selection of these user-item combinations should be constructed to map a sufficiently large model of the user's preferences, while still keeping the selection limited. If the selecting is too broad, the user might get tired of rating items before getting to the "point", and there will be a smaller selection of items for the recommendation system to recommend from (Rubens, *et al., 2011, p. 736)*.

2.1.6 Long-Tail Effect

The long-tail effect appears when there is a skewed distribution of rated data. The data is skewed towards a small subset of items. Few of the items get most of the votes and most of the items have few ratings (Schafer, *et al.*, 2007). The name is derived from the look of a sorted graph containing data with this kind of distribution. Figure 2 is an example that illustrates the distribution of artist by popularity. The figure is from Òscar Celma's PHD paper, which is about music recommendation and discovering items in the long tail. The figure is numbered 4.2 in the paper (Celma, 2008).

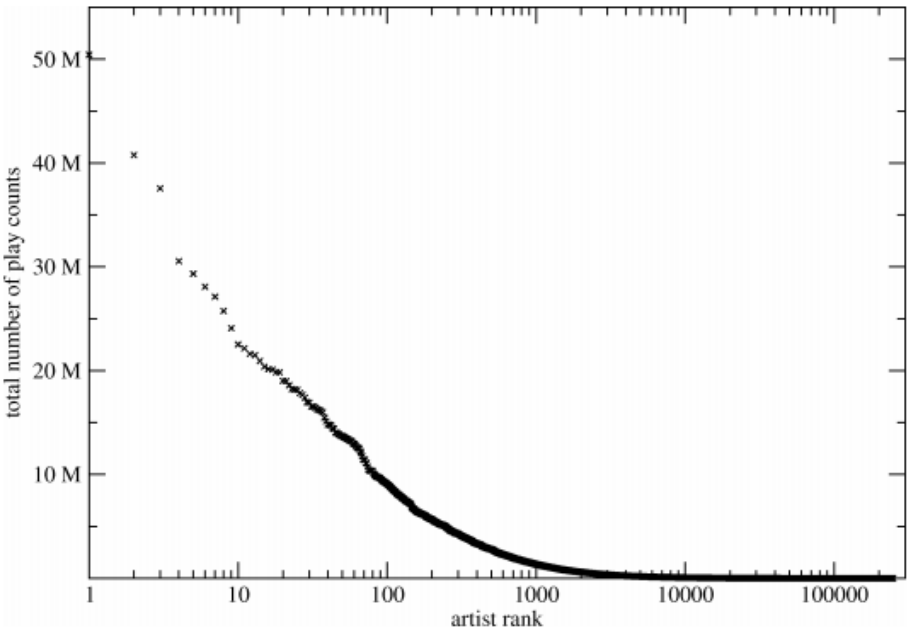

Figure 2 – Example of the long-tail effect: Artist popularity rank (Celma, 2008, p. 98)

The long tail effect is especially prevalent in recommendation algorithms that use a weighted result. The most popular items tend to be the most obvious recommendations, and often relevant recommendations for new users. But as Aggarwal highlights, there is value in recommending the less popular items (Aggarwal, 2016). For long term users, the less rated items become more relevant. For example, in a bookstore where the user can filter by categories, the most popular books in the Fantasy category might be Lord of the Rings and Harry Potter. These are good recommendations for people who do not regularly read fantasy books. However, people familiar with the genre might already have read those books, thus finding lesser known books, is more relevant for them.

2.1.7 Over-Specialization

When a recommendation system only recommends items with low novelty, the recommendation system is regarded as over-specializing. In other words, if the recommended items are too similar to what the user has already seen, then the algorithm has become too specialized (Aggarwal, 2016).

A way of describing the issue of over-specialization is to categorize the items and users into groups. Each group of items have items that are somehow related to each other. For example, one group could consist of items that people who bought a car often also buy. When a user rates an item, the system creates a connection from the user to that item. The recommendation system can then find items within each group that are similar to the rated item. In the car example, when the user rates a car, the system will want to recommend items from the mentioned group (for example car washing products) to the user. When a recommendation algorithm becomes overspecialized, it only recommends content from item-groups that the user has seen before. This might seem positive at first, since the user gets recommendations related to items they have rated. However, the user will also want to see new items. Since there are no connections to the items within the other groups, it will not recommend items from the other groups. A user with no experience in a group of the available items will not get recommendations items from said group. For example, if the user who bought and rated a car only gets recommendations for car-related products, it might be obstructive when the user is looking for a

new dishwasher. Then, the car product recommendations are irrelevant. This is the core of the over-specialization issue.

There are several possible solutions to the over-specialization problem. Randomization is a common way of dealing with the issue. By including random elements from groups that the user has not seen, new content can enter the user's list of recommendations. Both over-specialization and the long-tail effect can be encountered in the prototype created for this thesis. Any recommendation system that base its predictions on the preferences of others (a collaborative filtering system), will potentially encounter both of these issues. In the prototype over-specialization is countered by selecting between the three most recommended items. By selecting a random of the top three items, the prototype will make sure that the selected is relevant, but still lowering the probability of over-specialization. Additionally, the prototype features a knowledge-based filtering algorithm that selects tracks (items) based on their intensity. Different sections of the song will get different levels of intense tracks. The selection of tracks by intensity is ignorant to the popularity of the tracks, thus making sure that there is a more even spread of tracks. Both the knowledge-based filtering algorithm and the random selection of the top three recommended tracks are detailed further in Section 4.

2.2 Music Terms

Throughout this thesis, we use a handful of terms from music theory. These terms describe how fast the music is going, what the primary set of notes are, and the structure of a song.

Beats per minute (bpm) is a measurement in music that defines how fast or slow the music is. The number is based on a steady pulse, divided into a minute. The number of pulses, or beats, that fits within a minute is the bpm value. For example, a clock is running at 60 bpm because there are 60 seconds, or beats, in a minute. Naturally, 120 bpm is twice as fast as 60 bpm.

When describing or trying to play a song, it is important for musicians to know what collection of notes the song uses. This collection of notes is called the *key*. Many songs have one key, but there are exceptions where the key changes during the song. Longer compositions, such as symphonies or progressive rock songs often change keys during the song. For example, Bohemian Rhapsody by Queen[10] changes keys through the song.

There are two major categories of music keys: *major* and *minor*. These two categories are tied to what feelings the keys in them produce. Major keys tend to create happy or majestic sounding music, while minor keys create music that is sad or dark. During this thesis, when referring to key it is generally meant the core note found in the key. A common phrase to describe the key of a song is "*this song is in E minor*". This tells a musician that all the notes that are found in the E minor scale (series of notes) will fit with the song.

When notating music, it is common to split parts of the music into small sections. These sections are called *bars*. It is easier to read the music when divided up into segments. In sheet music, the separation between bars is noted as a vertical line. The current *time signature* of the song determines the length of a bar. Normally, the length of the bar corresponds to how many quarter-notes fit within the time

[10] The Official video of Bohemian Rhapsody by Queen: https://www.youtube.com/watch?v=fJ9rUzIMcZQ. The Wikipedia page for the song contains detailed information about the composition of the song: https://en.wikipedia.org/wiki/Bohemian_Rhapsody#Composition_and_analysis

signature. Most songs are in the time signature of $^4/_4$, meaning a bar will stop once four quarter notes have been counted. If a song is in the time signature $^3/_4$, there will be three quarter notes before a new bar is drawn.

There are a handful of common terms that are used to describe different sections of a song. They are*: verse, chorus* and *bridge*. There are certainly more terms, however it is these three that are most used.

The *verse* is often the section where most of the lyrics are sung. This is where most of the progression of the song is, and therefore it is the section that is used to build up to the chorus and move the song forward.

The *chorus* is the main part of the song. This is the section of the song where the main melody is repeated. The chorus is often independent from the verse and serves as the highlight of the song. The lyrics in the chorus are often repeated and sometimes contains the title of the song (Appen & Frei-Hauenschild, 2015).

A *bridge* is a section of a song that introduce either a new concept of the song, or that introduces a new variation of existing melodies. The bridge is used as a way of breaking up the rest of the song, in order to keep the listener interested. The terms *interlude* and bridge are used interchangeably to describe the same part of the song.

2.3 Content Creation

Content creation is a process when an agent or artefact creates new content. For example, a musician is a content creator. There is a vast selection of content creators in the digital landscape, some automated, others done by humans. The Deep Dream Generator[11] is an example of a partially automated digital content creator. The Deep Dream Generator is a set of tools that use an artificial intelligence (AI) to combine images. The visual style of one of the images is imprinted on the other image.

Automated content creation is very common in video games. Content creation in games is often in unison with the game and therefore often labelled as *procedural content creation*. For example, a video game where the game generates the world as the player explores has a *procedural world*. Minecraft[12], Spelunky[13] and No Man's Sky[14] are examples of games with this kind of world creation. These three games have procedurally generated worlds or levels. For example, in Minecraft when the player starts to get near the edge of the generated landscape, the game will begin to generate more terrain.

To create meaningful worlds (content), the generators in computer games follow rules. The complexity of these rules will vary with each game. For example, the universe-generating algorithms behind No Man's Sky are a lot more complex than the algorithms that create the levels in Spelunky. The scope of the world is not the only differentiating factor between the algorithms of these games. While No Man's Sky (and Minecraft) are games where the player can move the character in three dimensions, the player can only move the character in two dimensions in Spelunky. This makes the needed

[11] https://deepdreamgenerator.com/
[12] https://minecraft.net/
[13] http://spelunkyworld.com/
[14] https://www.nomanssky.com/

computation and generating a lot smaller. It will therefore be easier to highlight and compare the content creation algorithms from Spelunky.

Derek Yu, the creator behind Spelunky describes the rules he based the level generation on in his book about the game (Yu, 2016). The first rule is that there should be an entrance at the top of the level, and an exit at the bottom. Since the theme of the game is exploring old ruins (inspired by Indiana Jones and similar tropes), this is fitting as the exploration would lead into deeper depths. The second rule is that there should be a way for the player to get from the entrance to the exit without using any of the tools in the game, since the player might not have these at the time. The third rule is that the algorithm should not create any areas where the player can get stuck. They achieve this by making sure that there is stairs or similar ways of escaping pits. The final rule is that the edges of the level should be hard to notice. This rule will not get prioritized, since it is primarily aesthetic (Yu, 2016).

Continuing from these rules, the level creation algorithm will put rooms together from different types of templates. Differentiating the different template types (of themes) are based on what piece in the level they need to fill. Yu highlights "*basic room*", "*ladders*" and "*upper platforms*" as some examples of template types/themes. The game selects randomly from the list of templates. In addition to the selected pattern from the template, including some additional smaller room-parts enhance the randomness. This makes it seem like there is a larger set of rooms, despite the small number of templates. The game then adds game-objects, like enemies and traps, to the room.

The game has a series of rules that determine what the needed content is to create a valid game board. These rules tell the level generator what category of content to use. Then, the level generator selects randomly from the given categories. The selected content is small pre-generated content, in the case of the game this is tiles. This randomized rule-based selection of pre-generated pieces, highlights one of the common ways of generating content in games.

The prototype created for this thesis uses a similar type of content creation algorithm. We have created several small pieces of content (audio tracks), that an algorithm selects. The selection is based on several criteria or rules. Similar to Spelunky, the prototype features content with different categories. Where Spelunky had items where the biome was different (ice, jungle, etc), the prototype features items with different instruments. The other criteria are based on the preferences of the users. Spelunky does not include this type of criteria for its algorithms.

We have found few research projects that combine content creation and recommendation systems. Liebman and others present one of the most relevant projects (2017). Their project is a playlist generator. In other words, a system that generates playlists based on what songs user has previously listened to. The method of playlist generating highlighted in the paper consists of two main components. The task of the first component is to map out the user's preferences. This is a typical system found in normal recommendation systems. The second component creates the playlist, basing its selection of songs on the collected preferences. The focus of the paper is on the second component, since there has been a lot of research done on preference mapping (Liebman, et al., 2017). The research done for the paper continued to build on a prototype from a previous study. The prototype, named DJ-MC, use techniques from Reinforced Learning to generate playlists. In this paper, the authors improve the performance by using Upper Confidence Bound in Trees (UCT). The project described in (Liebman, et al., 2017) is similar to the prototype created for this thesis, in that both contain two primary components. Both systems aim to create a sequence of music that fit together, which is based on the preferences of the users. However, both the generated content and the implemented decision-agent is different.

3 RELATED WORK

3.1 RECOMMENDATION SYSTEMS

Ever since their introduction, recommendation systems have been changing. Alternative algorithms with better accuracy, relevance and novelty have been suggested. In the paper *Toward the next generation of recommend systems*, the authors present an overview of several potential areas of expansion for recommendation systems (Adomavicius & Tuzhilin, 2005). This is an overview paper and contains several sections. The last section of the paper is called "*extending capabilities of recommender systems*", and highlights seven different areas in which recommendation systems can be improved. Four of the seven areas are particularly relevant for this thesis. These are: *multidimensionality of recommendations, multicriteria ratings*, *nonintrusiveness* and *flexibility*.

Adomavicius and Tuzhilin (2005) first highlight the need for *multidimensionality* in recommendation systems. They explain that by including for example the time of year into the recommendation, the system might yield better predictions. They further argue that the normal two-dimensional ($\mathrm{User} \times \mathrm{Item}$) search space might not be sufficient. Others have argued the same. O'Donovan and Smyth (2005) explore if building computational models of trust can help in improving the users perceived quality (and therefore trust) in the recommendation algorithm. By adding a dimension of trust to the recommendation system, they intend to increase the quality of the recommendations. The trust should be measured and changed as the user interacts with the system. McNee, Riedl and Konstan (2006) also mention the importance of building the users trust toward the recommendation algorithm, however they do not directly tie it to the dimensionality of the recommendation engine. The trust of the user is important, since the user must feel that the recommendation engine "knows" them. This is relevant for the thesis in that the user should feel like the prototype creates a song based on their preferences. The user should trust the prototype to recognize their preferences. We argue that this is particularly important since the song creation process is automatic and contains several predictions. As explained in section describing the empirical evaluation, the user will run through multiple iterations of creating a song. This will build trust in the recommendation engine, since the user will directly hear how their preferences take effect.

In the *multicriteria ratings* section of the paper Adomavicius and Tuzhilin (2005) explain that basing the recommended ratings only on one score, might not reflect the complex nature of most types of content. By adding several criteria into the recommendation process (where one is the primary criteria and the others are secondary) recommendations might prove more insight for the user. The prototype created for this thesis is a suitable area for this kind of ratings, but an implementation of multicriteria ratings is not in the prototype.

Nonintrusiveness is also concept we consider relevant for this thesis. The same paper (*Toward the next generation of recommend systems*) ties nonintrusiveness into recommendation systems by pitching the idea of limiting the number of requests for explicit feedback by the user (Adomavicius & Tuzhilin, 2005). By requiring the user to rate items, the user is interrupted from the reason they visit the page. As a result, some recommendation systems leverage implicit feedback, and therefore a more non-intrusive approach. A recommendation system that is removed from its "normal" setting might need a different type of feedback. For example, in the case of the prototype, a new user must rate at least one iteration of clips before the recommendations are based on the collaborative filtering engine.

Toward the next generation of recommend systems (Adomavicius & Tuzhilin, 2005) briefly explains Flexibility, in terms of recommendation systems. The authors mention the creation of an SQL-like language (RQL). The goals of RQL is to add flexibility to recommendation engines. This is particularly interesting for the thesis. Although the prototype does not use RQL, it proves that flexibility and reuse of recommendation systems is an area that others have considered.

3.2 Music Generating

Music generation has been an appealing project for musicians and mathematicians for centuries. In fact, ancient philosophers have pointed out the similarities between music and math (Purwins, 2005, pp. 22-24). Since sound is waves, it is possible to use math to describe and change sounds. However, music generation is not only limited to creating and manipulating individual sound waves, but also the entire art of assembling sounds into songs. As a result, we categorize music generation into two groups. The main difference between these two groups is a matter of scope. The first group is concerned with the composition (or the structure) of the music, instead of on the notes, melodies and chords. Composition-generators pay close attention to the progression and the structure of the song. The second group is more concerned with the notes themselves, and therefore pay little attention to the structure of the song. The aim of the second type of song generator is to maximise the quality of the generated chain of notes, *i.e*. the melody.

The prototype made in the context of this thesis belongs to the first group, since it does not concern itself with generating notes. In fact, it is more closely related to dice music machine. Dice music generators were an invention that arose in the eighteenth century (Hedges, 1978). The generators consist of a series of tables with sheet music in the cells. The tables have numbered rows and columns, creating a basic coordinates system. An artist writes the content of the table in advance. For each column, the user rolls the dice. When the user rolls some dice (introducing the randomness), they pick the corresponding row to the number they rolled. They repeated this for each column, until they reach the end of the table. Naturally there is some variation in the generating process between the different versions of the game. However, the core principles of dice and pre-created content stay the same.

Using the dice music generator system, there is potentially quite a large set of possible variations of generated music. In the first recorded game, by Kirnberger, Hedges claims that there were 11^8 possible combinations (Hedges, 1978). Meaning there where 11 rows, since two six-sided dice will result in a range of 2-12 (11 numbers), and there was a total of 8 steps (dice rolls).

These dice machines have a major problem. If the dice rolls are added together the results will be weighted towards a specific number. This is always the case when you roll more than one dice and the results are added together. For example, the most common result of two six-sided dice (2d6) is 7. There are more results on the two dice that produce 7 than any other combination. For 3d6 it is 10.5, which results in either 10 or 11. The 2d6 annotation is a common way of describing multiple dice, often used in table-top roleplaying games. The formula is *xdy*, where x and y are numbers. The *x* annotates how many dice are used, and the *y* annotates what type of die to use (how many sides the dice has). In many table-top roleplaying games polyhedral are common. They typical ones are dice with 4, 6, 8, 10, 12 or 20 sides. See Figure 3 for an example of the dice. Each of the different dice types has a different most likely number. Using different types of dice in combination with each other can create a large variation of scales.

Due to the addition of two dice results, some results on the tables will occur more often than others. In the example of Kirnberger, the most likely result in his generator is the seventh row of each column, since they used two six-sided dice. However, if the game only rolls one die, or dice results are not added together, the problem disappears since the results are not dependent on each other.

Figure 3 – Polyhedral dice. From left to right, number of sides in parenthesis: icosahedron (20), dodecahedron (12), pentagonal trapezohedron (10), octahedron (8), cube (6) and tetrahedron (4)

The dice machine approach to generating music is close to the approach used in the prototype created for this thesis. A musician creates a series of premade segments of music in advance. The pieces are designed to fit together no matter what order they are arranged in. In the dice music-machines, the segments are presented as sheet music, while in this prototype it is sound files. The difference lies in the selection of the sound clips. The music machines of old use dice, while this prototype test whether recommendation systems can be used for such a purpose. The main difference between these two approaches is in the noting of user preference; the randomness of the dice is replaced by user preferences. The recommendation system approach will consider the preference of similar users, while the dice do not. We pose that consistency is key. A song generated only by randomness will naturally have a weaker consistency than something made from the preferences of users.

There have been other approaches to non-conventional music creation. A collaboration between a series of artists and a collection Artificial Intelligence (AI) systems, has yielded an album aptly named "Hello World" (Casey, 2018). The artists used the Flow Machines[15] project (FMp), which is a library of tools. The tools in the FMp are based on research combining creativity and machine learning, specialized in literature and music generating. The focus of the project is on imitating the *style* of the creators. The authors define style as *"an individual's uniqueness; style makes an artist's work recognised and recognisable"* (Ghedini, et al., 2015). In the case of the Hello World album, the collaborating artists gave the AI a series of songs which they want it to draw inspiration from. Then the system mimics the style and content of the songs. This is achieved through a neural network, which is created to find patterns in songs. There are different tools in the systems library, created for different tasks of music creation. For example, one of the tools is called FlowComposer, and it is designed to be an assistant tool for composing songs.

[15] http://www.flow-machines.com/

The development team behind FMp has published a series of papers in conjunction with the project. We highlight one of the papers, which we find relevant for this thesis[16]. *Assisted Lead Sheet Composition using FlowComposer* (Papadopoulos, et al., 2016) highlights a tool (FlowComposer) within the FMp toolset. This tool creates lead sheets (musical notation) that imitate a given musical style. The sheets are created using constraint-based Markov sequences. The constraints will ensure that the generated sheets imitate the style of the given music. The lead sheets consist of a melody section and chords that the melody fit within. Two Markov sequences are used to generate the sheets, one for the melody and one for the chords.

As reflected in the *Assisted Lead Sheet Composition using FlowComposer* (Papadopoulos, et al., 2016) paper, the project covers similar topics as this thesis. Both cover music generating, and to a certain extend allowing the user to influence the result. The main differences in generating technique, is that the FMp leverage machine learning techniques, whereas this thesis uses recommendation systems. FMp is designed to facilitate the composer to alter the music during the generating of the music. The project aims to create tools for artists and composers to use while creating music. FMp, and especially FlowComposer, are created to be an interactive composition tool (Papadopoulos, et al., 2016). The prototype for this thesis is more static, in that it generates a song and asks the user to rate it once the song has been generated. Both FMp and this thesis feature automatic content creation. Given that FlowComposer mimics the given reference songs and that the artist/user can change the suggested music, it is evident that the generated songs from FMp are intended to be used as inspiration or as a foundation for a new song. We therefore argue that the FMp project does not focus on generating personalized content but facilitate the personalization with the alteration tools. This is a key difference between FMp and the prototype for this thesis.

3.3 Music and Recommendation systems

Music and recommendation systems is a natural combination. As music libraries get bigger and bigger, the need for better recommendation systems also increase. Some of the recent studies into the combination of music and recommendation systems have focused on leveraging personality traits of users to enhance the recommendations (Ferwerda, et al., 2015; Ferwerda, et al., 2017). In *Personality Traits Predict Music Taxonomy Preferences* Ferwerda and others (2015) find correlations between the personality traits of the user and what type of taxonomy the user is most likely to search for music in. In this case, taxonomy refers to categories of music, for example music categorized by their mood (sad music, happy music, *etc.*). Another example, which is a more traditional taxonomy, is to categorize music by its genre. They argue that by understanding the personality traits of the user, a more directed recommendation, within a specific taxonomy, can be made. In *Personality Traits and Music Genres: What Do People Prefer to Listen to*, Ferwerda and others (2017) compares different personality traits and listening habits for a series of Last.fm[17] users. Some of the personality traits they highlight were *Openness to Experience, Conscientiousness* and *Agreeableness.* They found several correlations to previous work, but a direct comparison was difficult to make due to different clustering of results and data. As a conclusion they argue that it would be beneficial to include the user's music genre preferences in a personalized system (Ferwerda, et al., 2017).

[16] For a full list of papers visit: http://www.csl.sony.fr/publications.php?username=&keyword=flow+machines&pub_type=&year=

[17] https://www.last.fm/

There have also been studies into different aspects of popularity and music. More specifically, the "long-tail of music". The long-tail of music refers to the uneven distribution of popularity of artists or how few songs get most of the attention from the users. Celma (2008) aimed to further enhance music recommendations by leveraging the rarely recommended items (found in the long-tail). Celma explains that by shifting the focus from the recommendations systems *predictive accuracy* to *the users perceived quality*, better recommendations can be made. The predictive accuracy here refers to the best recommended item, and the users perceived quality is how well the user will like the recommended item. Celma argued that the focus of music recommendations should be on the novelty and relevance, thus circumventing the most popular items to get better recommendations (Celma, 2008).

Another approach to studying the correlation between popularity of music and music recommendations, is presented by Schedl and Bauer (2017). Their study compares different levels of demographics of music (culture, country, *etc*), and if recommendations based on these categorizations will improve the recommendation accuracy. Additionally, they try to leverage the most popular items (they define it as *mainstreaminess*) to enhance the recommendations. Schedl and Bauer find that their best result is achieved by both using a demographic filtering, based on the user's country, in addition to a filtering based on mainstreaminess (Schedl & Bauer, 2017).

Although these studies combine music and recommendation systems, the way they use the recommendation systems are quite different from the way proposed in this study. The presented studies suggest ways of improving the accuracy of recommendation systems by including new criteria, while the prototype created for this thesis use recommendation systems as a means of creating new content.

4 MUSIC GENERATION WITH RECOMMENDATION SYSTEMS

This section of the thesis elaborates on how we aim to answer the research questions: Can a music-generator create personalized music, and can new content be created by a recommendation system? In order to test these two questions, a prototype was created. The main purpose of the prototype is to generate content using recommendation algorithms as the primary decision-agent. In this case, the decision-agent is the process that selects what items to include in the generated content. By basing the selection of items on the predictions and recommendations from recommender systems, we attempt to make the generated content more personalized.

The motivation for the thesis is to facilitate the automation of creation of personalized content. Our goal is to create a system that read the user's preferences and then use them to create new content that is meaningful to the user. As highlighted in Section 2.3, one of the common ways of creating new content is to have a library of pre-created content, and then merge a selection of the content together. We therefore sought a medium in which this was possible; music.

Choosing music as the preferred medium to create the content in, is based on two main factors. The first factor is that music is a flexible medium. Flexible in that it can be split up into small chunks and put together in different patterns. Therefore, it is possible to create a library of music segments that can be arranged into songs by a smart selection. The second factor is that music is a medium most people know and can relate deeply to. Choosing it for content creation is therefore a safe choice. People will be able to recognize music patterns and will have opinions on what they like and do not like. In domains that the users are not familiar with, some training or explanation as to what to look for is needed. This is not the case with music. Therefore, the target audience of the prototype was anybody that is interested in creating music. This does not mean they have to be musicians.

The following sections further elaborate the song creating process used in the prototype. In section 4.2 the process of generating songs is explained (in addition to the feedback process), while section 4.3 explains the implemented feedback process. Section 4.4 consist of a more detailed description of the recommendation engine. This includes how and why the engine switches between the two recommendation algorithms, in addition to a detailed description of the implemented algorithms themselves.

4.1 PROTOTYPE DESCRIPTION

The primary functionality of the prototype, named RecOrder, is to generate songs. RecOrder is therefore a song creation application, that combine smaller clips of music into a song. There are three major components in RecOrder: *the song generator, the recommendation engine* and *the feedback collector*. Figure 4 illustrates these three components. The *song generator* initiates the song creation process, and it is this component that will output the finished song. The song generating is done through sequential iterations. Each iteration creates an audio track that is a subsection of the finished song. This subsection consists of three of the mentioned audio clips: clips with guitar, bass and drums respectively. Once the last iteration has been completed, these subsections are combined into the finished song. This is followed by a feedback process, where the user rates different parts of the generated song.

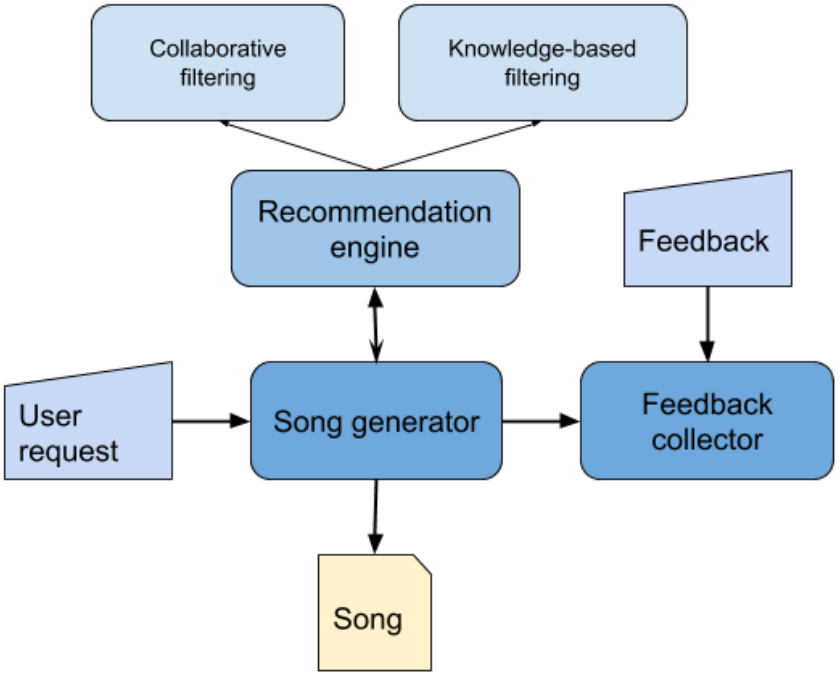

Figure 4 – Simple overview of RecOrder's structure

The recommendation engine selects which audio clips are to be used in each of the subsections of the song. As illustrated in Figure 4, there are two recommendation algorithms implemented in RecOrder: an *item-based collaborative filtering algorithm* and a *knowledge-based filtering algorithm*. RecOrder switches between these two algorithms once one of them fails to provide a recommendation. This makes RecOrder a *hybrid switching recommendation system*. The item-based collaborative filtering algorithm is the primary algorithm, and knowledge-based is the secondary. RecOrder defaults to the collaborative filtering algorithm, but if a recommendation cannot be made using that algorithm, the engine will switch to the knowledge-based filtering algorithm.

Figure 5 displays a detailed overview of the class-structure of RecOrder. In addition to the noted classes, RecOrder includes some minor classes. The tasks of these minor classes are mostly of the utility nature, such as cleaning output from queries, *etc*. The structure diagram below, does not include these classes, for the sake of visibility.

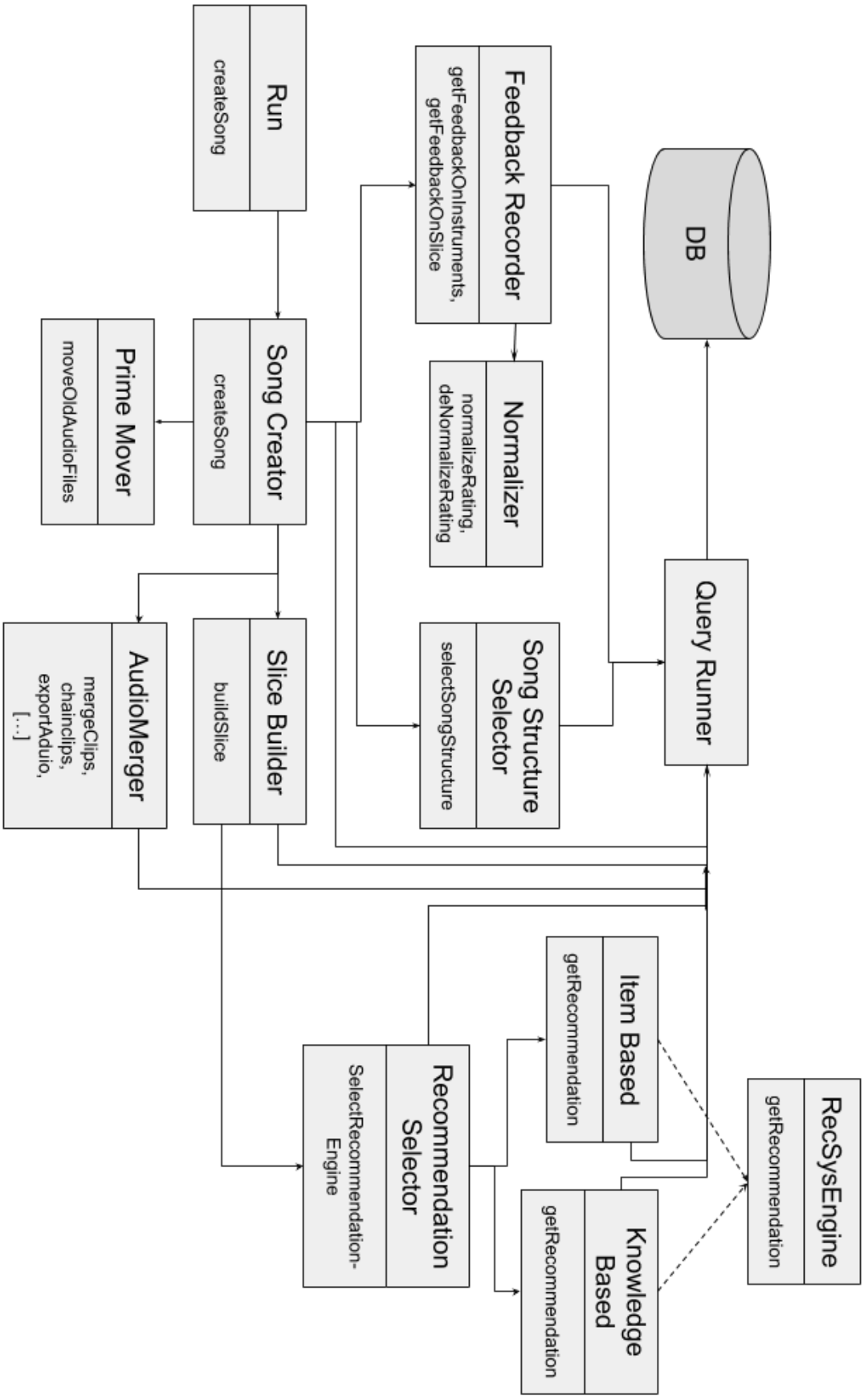

Figure 5 – More advanced overview of the structure of RecOrder.

RecOrder has two primary inputs and one output. Since the purpose of RecOrder is to generate songs, it then follows that the output will be the created song. The songs will vary in length. The length is based on how many sections it includes, and how long each of the included audio tracks are. The songs always contain three instruments: guitar, bass and drums.

RecOrder's primary input is a series of audio clips. These clips are short in length, typically the length of 4 or 8 *bars*[18] of music. In terms of seconds this usually translates into a couple of seconds. However, the exact amount of time is dependant of the *bpm* (the tempo). Each audio clip contains the recordings of a single instrument. The clips are recorded in advance and are used as components in the generated songs. There is a total of 37 audio tracks that the prototype can select from. 19 of the tracks are guitar tracks, 9 is bass tracks and the rest (9) are drum tracks. Since the guitar is a lead instrument, there is a larger amount of guitar tracks than the other two instruments. Most of the listeners attention will go to the guitar, thus a greater amount of variation is needed.

The second input of RecOrder is the user input. This input is used to guide the prototype in creating a song. There are three main properties the user needs to include: the bpm, the key and the intended length of the song. The first two describe actual musical properties, while the third is an abstraction. Bpm and key could be picked at random by the system, but they are included in order to facilitate a higher degree of user choice. However, in the prototype we only include audio tracks with the bpm of 105 and in the key of D. This was done to reduce the needed time to create audio tracks during the development phase. The input of the song length is limited to three choices: *short, medium and lon*g. Although these are abstractions they are intuitive enough to understand. In addition to the bpm, key and the length of the song, the system is also dependant on a profile ID. This dependency is in place mainly to differentiate among different user's ratings. Person A's ratings should not mix with Person B's. As a result, RecOrder has a basic user profile system. RecOrder does not store any information that can identify the user; it only stores the user's preferences and the profile ID. Implementing a larger form of profile structure is outside of the scope of this thesis' timeframe.

The main way for a user to interact with RecOrder is through a terminal window. A user interface was not prioritized during development. The priority was solely on the song creation process, thus keeping the interaction to a terminal window was the simplest option. However, a small webpage was created to start the song generation process, but none of the responses from RecOrder are displayed on that webpage. This was done to minimize the chance of accidentally starting the song creation process, and to make it easier for test subjects to start the process. The alternative to not having this webpage was to type a command in the terminal window, which unnecessarily complicates the process of creating a new song.

4.2 The Song Generating Process

The first step in generating a song is selecting an overall structure for the song. There are several available song structures, ranging from simple arrangements to more complex ones. The different song structures are split into three categories, which match with the three options the user had for song length ("*short*", "*medium*" and "*long*"). Naturally, the categories describe the intended length of the song, and then in turn how many sections the structure of the song should contain. The shorter song structures have few sections, while the complex ones have many sections. The song structures

[18] See Section 2.2 for a definition of *bar*

are stored in a database table as strings, and are represented as a series of letters, where each letter designates a different section of a song. The series of letters is a way of describing song structures. Examples of its use in literature are (Covach, 2005) and (Appen & Frei-Hauenschild, 2015). For example, many pop songs follow this structure: ABABCB. In that song structure A, B and C are different sections of the song. Most commonly A in that case is a *verse*, B a *chorus* and C a *bridge*.

When generating songs, it is preferable to generate smaller sections of the song at the time, and then combine these sections later. By dividing up the generated content, it is possible to reuse parts of the generated content (for example is there is multiple verses in a song). Additionally, generating a smaller piece of music is less demanding in terms of processing power, and it is easier to ensure a greater level of quality of content for the smaller piece. There are two options for dividing up the song. The first option is to divide the song into sections, where each section is a complete part of the song. Each section should then be listenable by itself. The other option is to generate the song each instrument track at the time, and then join the tracks together. For example, the guitar track can be generated first, then the bass track and finally the drums. The three tracks are then merged together. However, this approach to song generating can result in songs where the instrument tracks drift out of synch with each other. Because each instrument-track will consist of several smaller audio tracks, and since the smaller audio tracks might differ in length, the time when transitioning between the smaller tracks might not line up with the other instrument tracks. This difference in length between the smaller tracks causes the shortest track to continue into the next short track of the instrument track, before the short tracks of the other instrument tracks do. Figure 6 illustrates this. As seen, the first verse-guitar track is twice as long as the verse tracks for the other instruments. Therefore, the guitar-verse track continues into the chorus, which is the next section of the song. To avoid the described problem, the prototype generates songs in steps, where each step creates a section of the finished song, as described in the first option.

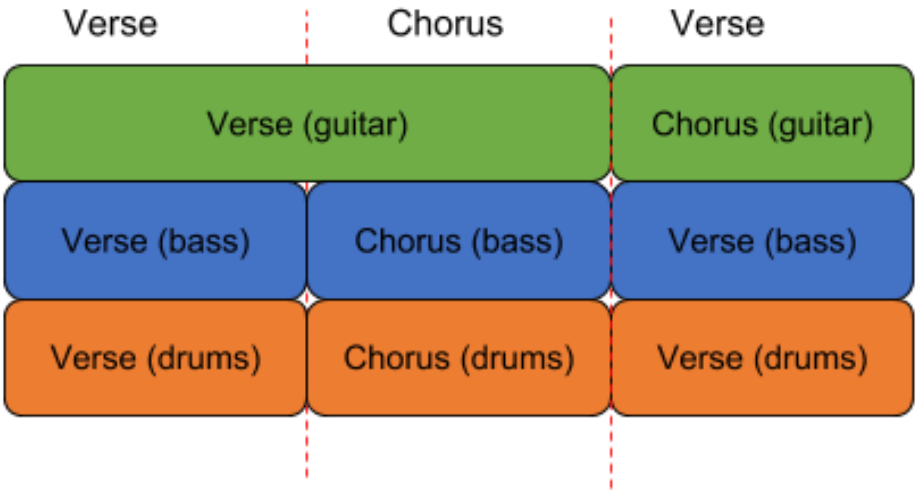

Figure 6 – Illustration of clips drifting out of sync

The song generating process therefore create smaller sections of the finished song and then combine these sections into the full piece. These sections contain the combined result of several audio tracks, each track containing the recordings of different instruments. The term *slice* is used to describe these sections. Slice is chosen because each generated section is a slice of the finished song. As a result, a slice is created for each unique letter from the selected structure of the song. A slice therefore

describe which instruments are playing in the given part of the song, and what these instruments are playing. To create further variation, each slice in a song has its own structure picked at random. Figure 7 illustrates the structure of a series of slices. The slice includes a row for each instrument, and each cell contains a binary range (i.e.: 1 or 0). The binary range defines whether an instrument is playing in that slice or not: 1 for playing, 0 for not playing. As shown in Figure 7, the drums and the bass are playing in every section of the song except section C. The selection of these slice-structures is weighted towards having all instruments playing at the same time.

Song structure	A	B	A	B	C	B
Has drums	1	1	1	1	0	1
Has guitar	0	1	0	1	1	1
Has bass	1	1	1	1	0	1

Figure 7 – Active instruments in multiple slices

Once RecOrder has chosen the structure for the slice at hand, it starts the process of creating the slice. For each instrument that is playing in the slice (based on the slice-structure), the active recommendation system attempts to give a recommendation. The recommendation engine first tries to get a recommendation from the collaborative-filtering engine. Failing that it, the active recommendation engine is switched to a knowledge-based recommendation engine. There are several reasons that it might not be able to provide a recommendation using the collaborative-filtering algorithm. The two most common ones are that the current user has seen every item that the other users have also rated, or that the user has not rated enough items (the *cold-start problem*). The recommendation engine is further defined in Section 4.4.

To create a song RecOrder combines audio clips recommended by the recommendation algorithm into a song. This process contains two axes. The first axis is the progression throughout the song and is shown on a horizontal line, thus referred to as the horizontal axis. This is a common way of displaying song progression and is used by for example Spotify. A bar fills up as the song progresses, following the horizontal line from left to right. This type of display is also used for general progression bars, for example when installing new software. Figure 8 shows how Spotify displays this horizontal line, and thus the song progression. The second axis is an expansion of the concept of the song progression. At any point in the song, typically there are multiple instruments playing. The instruments are layered on top of each other to create a blend of audio. For example, a very common blend of instruments in rock are two guitars, a bass and drums. The vertical axis is the view of these instruments and what they are playing, at any given time during the song. Slices describe this part of the axis. For example, looking only at the verse of a song, what instruments are playing as well as what they are playing, is the vertical axis.

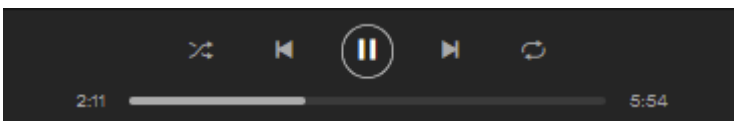

Figure 8 – Screenshot of how Spotify displays song progression

In addition to the two axes, the following sections use two terms to describe the process of combining the audio clips within the axes. These terms are *merging* and *appending*. Figure 9 illustrates the two axes and the difference between merging and appending. In short merging describes the vertical

addition of sound clips, while appending describes the horizontal action. *Chain* is also used to describe the horizontal sequences of appended audio tracks.

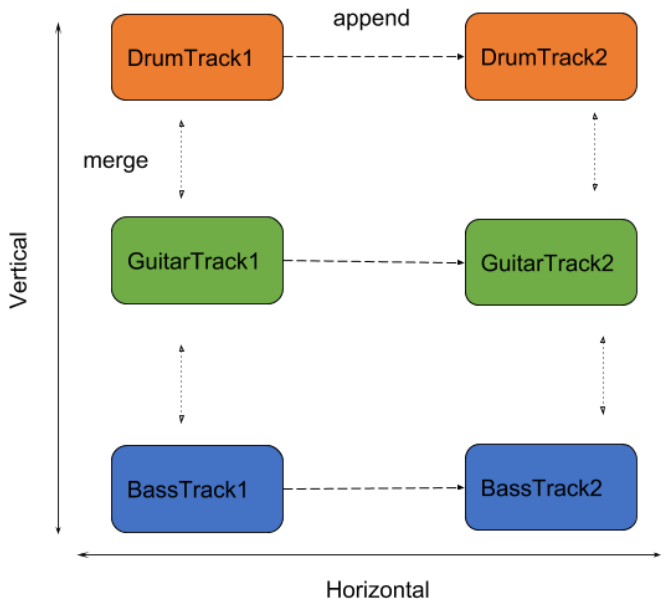

Figure 9 – Vertical merging and horizontal appending explained

The combining of audio tracks presents several challenges, most of which are related to making sure the tracks start and stop at the same time. In other words, it is important to ensure consistency in the horizontal axis. However, this could be challenging, since the clips might vary in length. When designing the merging process this challenge was taken into careful consideration. One of the precautions that will ensure that the tracks are merged in a sensible manner, is that it is easier to merge two tracks at a time. Once the two tracks have been merged the other tracks are added to the mix. In other words, when combining clips in the vertical layer, it is less complicated to merge pairs in contrast to merging an entire slice at a time. By gradually merging more tracks into the first track in the pair, the merging of a slice can be done iteratively. Using Figure 9 as an example to illustrate this, DrumTrack1 and GuitarTrack1 would merge first. Then, BassTrack1 would merge with the result of the merging of the drum and guitar track. If the prototype were to merge the entire slice at once, it is difficult to handle the difference in track lengths.

RecOrder takes some precautions to ensure that the lengths of the tracks within a slice align. For example, the shortest track can be looped in order to fit within the time of the longer track. This is useful for cases where the length of one of the tracks is twice the length of the other track. If the difference in length of tracks is not checked for, the longer track would be cut off while the shorter track is still playing. This is not a desired effect, and to prevent this from happening the lengths of the two tracks are measured. If the length differs by a factor of two, RecOrder loops the shortest of the two tracks until it fits within the length of the longer track. In order words, if the longest track is 8 *bars* long and the shorter one is *4 bars*, the shorter track plays twice and the longer plays once. This will ensure that the tracks are either of equal length or last for the same amount of time.

The song creation process is as follows: first select an appropriate song structure based on the user input (length of the song). Then, for each unique section of the song structure, find a structure for that

section. Each of the section structures determine what instruments are playing. These sections are called slices. For all the instruments that are playing in the slice, get a recommended track from the recommendation system. Once every active instrument track in the slice has a recommended track, the recommended tracks are merged to a single audio file. When the merging is done, the system has finished the slice. The created slice is then appended to the previous slice, unless is the first one generated. The order that the slices are appended to each other, are determined by the song structure that the system chose in the beginning of the generating process. For example, given the song structure ABABCAB, the system generates three slices (A, B and C). The appending order then follows the song structure, starting with A followed by B. As a result, a slice can be inserted in multiple spots in the song. Figure 10 illustrates how slices are created and how they relate to each other. The slices are appended after each other, until each slice has been inserted at their appropriate spot(s). The end of this step finishes the song.

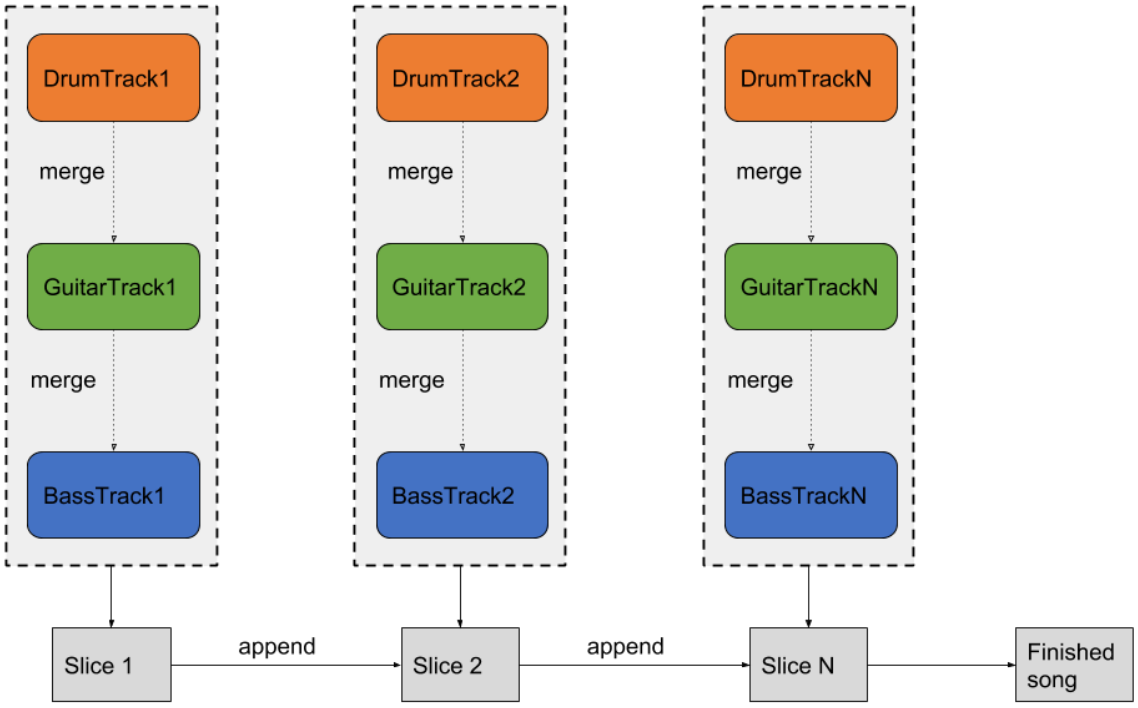

Figure 10 – The creation and appending of slices

4.3 The Feedback Process

Once the song has been generated, the feedback process begins. As mentioned in Preliminaries**,** recommendation systems are heavily dependent on feedback from the users. It is through feedback that the recommendation system finds out what the user likes, and as a result if the recommendation were accurate. Without sufficient feedback, there is no way for the recommendation algorithm to adjust its recommendations. Then the recommendation algorithms will recommend the same items multiple times. Therefore, the next steps of the song generating process is to get feedback from the user, on the generated song. In RecOrder, the feedback is collected through explicit feedback at the end of the song creation process. The user rates sections of the generated song, on a scale of 1 to 5, where 1 is the lowest score. By limiting the scale to five numbers the user has a smaller selection of score to

choose from, thus making the process easier. In addition, by having an odd number of choices, the user can choose a neutral option (3) or have a more nuanced opinion (2 or 4). Continuing with the two axes mentioned previously, the user will be asked to give ratings to clips both in the vertical axis and in the horizontal axis. The rating in the vertical axis will focus on the relation between the audio tracks within a slice, and the horizontal axis will focus on the same instrument track across the song. A random selection determines which parts of the song that is going to be evaluated. There are two stages in the rating process. First, the user rates the relation of items on the horizontal axis. The horizontal axis focusses on the relation between clips of the same instrument, from all the slices. The relation is highlighted as *a* in Figure 11. The user rates the quality of each individual instrument track in the song. In the second stage, the user is asked to rate how well the items within up to three different slices fit together. This are items on the vertical axis, highlighted by the arrows annotated by *b* in the same figure. By giving ratings on both axis, a larger portion of the song receives feedback. This also makes sure that the given feedback is more accurate, since the user rates a smaller selection, compared to rating the entire song with one score. By having both vertical and horizontal feedback the feedback process allows for a greater coverage of the user's preferences. A separate audio track is created for each of the relations the user rates. This makes the rating process easier, since the user can listen to the specific part that they are rating.

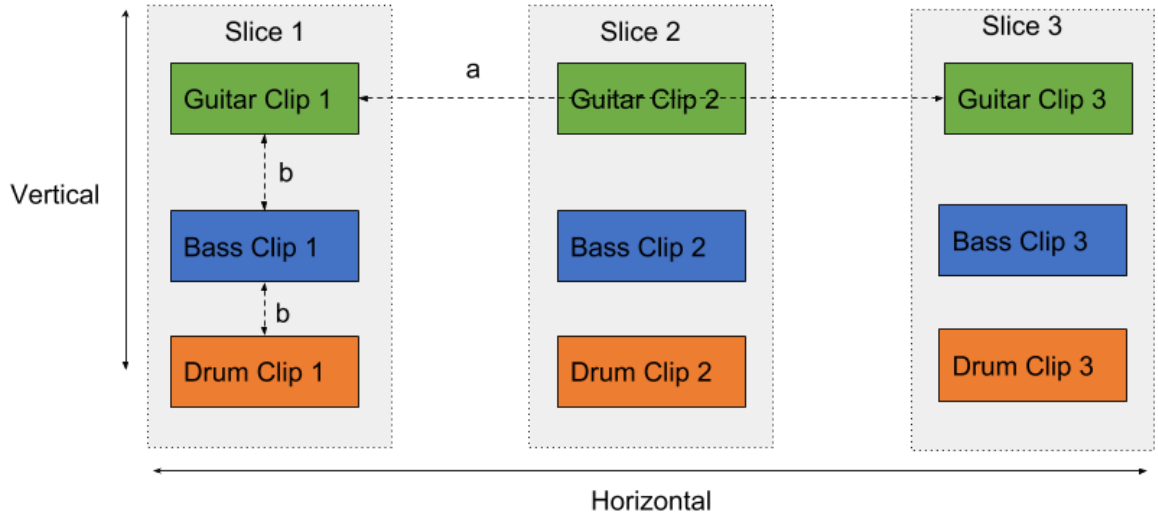

Figure 11 – Feedback evaluation axis

In terms of the feedback process, the RecOrder differentiates itself from regular use-cases of recommendation systems in two main ways. Firstly, RecOrder selects what content the user shall give feedback to. This is opposed to a regular recommendation system where the user either actively rates an item (explicit feedback) or gives ratings through their interactions with the system (implicit feedback). In RecOrder, the user gives feedback only at the end of the song creation process, as opposed to during the process. This makes the prototype solely reliant on the explicit feedback given by the user. Traditional recommendation systems often use a mix of explicit and implicit feedback to create a broader aspect of what a user likes and do not like. For example, if two items are recommended to a user and the user only clicks on the first item, one can assume that the user prefers the first item over the second. RecOrder loses this nuance, since the implemented song creation system selects the recommendations.

During development of the prototype, a different approach to the feedback system were implemented. Earlier prototype iterations had a more traditional version of a feedback system. They featured an active user choice between three recommended items. Figure 12 shows the input of this choice. The numbers within the brackets corresponds to the ID of a track. The tracks in the brackets are the top three recommended tracks, given by the recommendation algorithm. This results in the user taking an active part in the selecting of what tracks to include in the song. Additionally, the system could then interpret that the user thought the two other songs where be less fitting. In this version of RecOrder, this was the primary input of feedback. The selection between the three tracks had to be made for each of the clips that was going to be in the song. Naturally, this approach to feedback became tiresome when repeat multiple times and therefore this approach was abandoned.

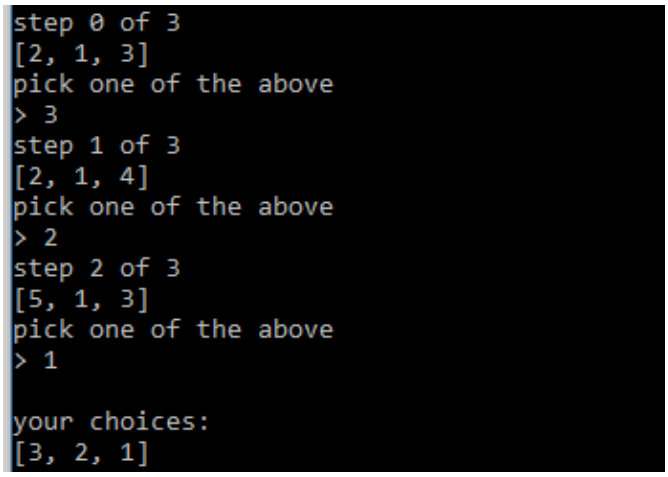

Figure 12 – User selection process in the early iterations of the prototype

There is a challenge between getting the most accurate feedback and exhausting the user with choices, as highlighted by the example in the previous paragraph. These two aspects (accuracy versus exhausting the user) can be posed as two opposites, both being potentially problematic. Their difference is in how many recommended items the user should rate. The first aspect is that the user rates every recommendation. This would become tiresome and boring, but it would be the most accurate way of gathering feedback. The other aspect is a fully automatic feedback system, where the user has no input in the feedback process. The technical details of such a system are not defined in this thesis, but its primary advantages and disadvantages are as follows: The main advantage of a fully automatic feedback system is that it would help reveal whether recommendation systems are truly fitting in content creation. If the system can learn from itself, it would (at least partially) prove that recommendation systems are in fact versatile enough to be applied for other uses. However, an automatic feedback system could easily fall into a self-reinforcing loop. Unwanted patterns could easily appear and detecting them in advance would be difficult. This is the primary disadvantage of a system that iterates over and learns from its own results. Due to the complex nature of a fully automatic feedback system, and the highlighted disadvantages, the prototype does not include an automated evaluation process. Therefore, the prototype has an automatic selection of tracks with a manual feedback system: The system randomly selects one of the top three recommended items. Then, after the song is created, the user rates multiple sections of the song. The randomness of the selection lowers the probability of selecting the same items multiple times.

The second way that the prototype differentiates itself from a regular recommendation system implementation, is that the prototype asks the user to rate the connection between two items (audio clips).

In a regular recommendation system, the rating defines how much the user liked the recommended item itself. Here the rating describes how well the user liked the relation between two items. This relation is referred to as the *clip-clip relation*, since the items at hand are audio clips. There are two main reasons for placing the ratings on the clip-clip relation rather than the items themselves. The first reason is to make the feedback process more focused on what part of the song is important: how recommended parts work together. When the user rates the relation between items, the focus of the rating is not on the quality of the items, but rather how the items work together. The sum of the parts is prioritized over the parts themselves. The purpose of the feedback section was to measure how well the recommendations fit together in a larger picture. The items themselves are not that relevant, but how they work together is the focus. The second reason for abstracting where the ratings are placed, is to further remove the recommendation algorithm from the items. By abstracting the layer between the items and the algorithm, the prototype is a larger step towards a more generalized system. Given that RecOrder generates songs that the users like, a generalized system will in turn prove the viability of recommendation systems being used in personalized content creation. A crucial step in the feedback process is therefore to find all the relevant connections between items in the generated song. In other words, the system must find all the clip-clip relations. This is done by registering every tracks relation to its neighbouring tracks, both in the vertical and horizontal direction. For each track that is in the song, each of the tracks that the track is merged with and appended to, is registered. For example, in Figure 10 the connection between DrumTrack1 and GuitarTrack1 is stored as a relation. In addition, the connection between DrumTrack1 and DrumTrack2 is also stored. When the user then rates, the rating is placed on that relation, instead of the items. Then, the rating is stored in the database, along with am identifier for the clip-clip relation the rating is placed on. The ratings are later used by the recommendation system to recommend similar tracks.

Since RecOrder is explicitly asking the user to rate specific sections of the song, it can be considered as an active learning system. The selection of which items to rate is slightly different from a typical active learning system. In RecOrder the selection of items is based on a *clip-clip relation* that the user has not rated before, while in a typical active learning system the selection is based on areas of the item catalogue where there is a greater amount of uncertainty of the user's preferences.

4.4 The Recommendation System

The song generating process is designed to be independent of the recommendation system part of the system. In other words, the song creation process is designed to work with any system that selects tracks, as long as the decision system returns song IDs. To achieve this independence, the recommendation engine returns the unique identifying signature of the recommended audio track. This signature is a database index, which the prototype (eventually) uses to build the songs actual audiofile.

To guide the recommendation engine in giving a relevant recommendation, certain parameters are included in the query for a recommendation. The parameters describe the wanted track's *bpm, key* and *instrument*. The recommendation engine will then limit the audio tracks it considers to only those who match all three of the parameters. This is a way of *pre-filtering* the available items. The reduction of the items from which the recommendation engine bases its selection, ensures that the recommendation is relevant. This hearkens back to the goals set by Aggarwal, McNee and others to maximise the quality of recommendations (2016, pp. 3-4; 2006). They highlighted that relevant recommendations were important, as mentioned in Section 2.1. The pre-filtering ensures this. If the pre-filtering was not in place, the system could recommend a track with a different tempo or with the wrong

instrument playing, which can result in unwanted outcomes and confusion. For example, if the generated song has two drum tracks where one is replacing the guitar, it quickly becomes tiresome and noisy for the listener. In terms of the other three goals that Aggarwal, McNee and others (2016, pp. 3-4; 2006) highlighted, both *novelty* and *increasing diversity* is less important in a content creation setting. Reuse and categorization of items can certainly help in creating better content. This is especially true in music creation. For example, a song often has repeating melodies. However, the reuse of items should not be overdone, meaning that the *serendipity* must still be high. Depending on the categorization, recommending items within a category is beneficial, for example if the recommended tracks are sorted by instrument. On the other hand, recommending tracks from different genres can create pleasant surprises and interesting combinations of music. Thus, the *increasing diversity* of recommendations is still relevant as a goal, although less important.

Several factors had to be considered in terms of what recommendation system RecOrder should implement. From a study of recommendation systems done prior to the development, it was apparent that a hybrid system was the most preferred approach. Due to the limited development time, there was not time to develop an advanced user-profile system, complete with log-in functionality. Therefore, RecOrder is aimed at a session to session basis, where a new user profile is created with each session. By session it is meant each time a new user interacts with RecOrder[19]. Since a new user has to be created frequently, RecOrder will often have little information of the user's preferences. This in turn means that the system has no *ground truth*[20] to base its predictions on. Thus, the recommendation system will have no data on the preference of the user when a new user is created, and a good way of dealing with the cold-start problem was therefore necessary. Additionally, the implemented recommendation system needs to be able to give recommendations when there are few items. This is necessary, since the prototype has a small content library. Creating an exhaustive library is time consuming and distracts from the goal of the thesis.

There are three primary requirements that determine what recommendation algorithm is fitting for the prototype. The requirements are derived from the limited timeframe for development of RecOrder. The first requirement is that the algorithm cannot depend on a large number of users. Creating a large user-base is well out of the scope for a master thesis. The same argument can be held towards items, which is the second requirement: the algorithm should not depend on a large item library. However, it is easier to generate items than users, so the emphasis on the number of items was slightly lower than the user centred one. The third requirement is that the algorithm should not depend on item-information, but on the ratings given on the items. We include this requirement to prioritize the personalization aspect of the research questions[21].

Three categories of recommendation systems were considered for the prototype; *item-based filtering, user-based filtering* and *content-based filtering*. Item- and user-based are both types of collaborative filtering. Item-based filtering was chosen as the most relevant algorithm type. The reasons and comparisons with the other algorithm-types are given below.

Since the prototype recommends audio clips based on what is needed in the song, content-based filtering seems fitting at first. However, not when compared to the goal of the thesis. As described in

[19] This also includes any recurring users, as long as there is a different user interacting with RecOrder in between.
[20] In this case, *ground truth* is a user-item combination that we know the rating for. This is used as a basis for comparison with other recommendations (Aggarwal, 2016).
[21] Although content-based filtering is certainly capable of providing personalized recommendations, we wanted to focus on the user opinion aspect of the algorithms.

Section 2.1.1, content-based algorithms are reliant on well described items. It was not within the scope of the thesis to create this type of item library. As mentioned, there is a pre-filtering of items that the recommendation algorithm can select from, ensuring that the recommended tracks are relevant. As a result, the recommended clip can be considered relevant without scanning the content of the individual clip, as one would have to in content-based filtering. It follows then that content-based filtering does not satisfy the requirements.

The two remaining algorithms (user-based and item-based) are both collaborative filtering algorithms. By examining the entire collaborative filtering category, we attempt to find the requirements and challenges implementations that collaborative filtering algorithms face. We use these challenges as a basis for discussing if collaborative filtering systems is suitable for the prototype.

There are certain domains in which collaborative filtering provides the best result. These domains have a set of properties that deem them suitable for implementing collaborative filtering (Schafer, et al., 2007). It is certainly possible to introduce collaborative filtering systems into domains that do not have these features. Schafer, *et al*. (2007) explain that the implementation of collaborative filtering algorithms into such domains will be challenging and might need additional adjustments to the algorithm and possibly the data. Using collaborative filtering algorithm in new domains is central to the thesis at hand, since the prototype aim to illuminate the challenges with using collaborative filtering to create new content. Content creation is certainly outside the normal domain of recommendation algorithms and a domain that does not fit with suggested characteristics. Therefore, what follows is an overview of the necessary properties of the data for an easy implementation of a collaborative algorithm. Schafer, *et al*. (2007) categorize the properties into three categories. These categories contain a different amount of properties and describe different elements of the data found in most suitable domains. Figure 13 displays these categories and the properties within.

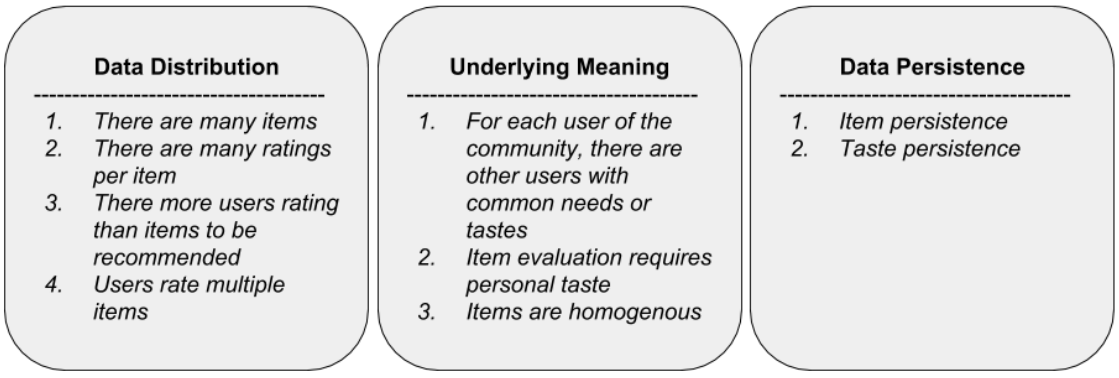

Figure 13 – Schafer, et al. (2007) collaborative-filtering domain properties

The first category is properties that describe the distribution of data, and therefore describe the size and shape of the data. There are four properties in this category, the first property being *"There are many items".* If a domain has few items, there is no need for a collaborative filtering system, since the user probably will have a good overview of the data set. *"There are many ratings per item"* is the second property in the category and means that each item should receive multiple ratings. If each item only has one rating, the system will not have enough information to differentiate the items. Without sufficient information, the recommendation algorithm has nothing base its predictions on. The third property of the data is: *"There are more users rating than items to be recommended".* In

other words, there should be more users giving ratings than there are items to be rated. Schafer *et al*. (2007) explains that a rating distribution is often not balanced, a small portion of items will get a larger amount of ratings. Thus, a lot of ratings are needed. Following this, the last property in this category is "*Users rate multiple items*". If the users only rate one or two items, there will not be enough user data for the recommendation system to create a prediction. Two of these properties are relevant for the domain the thesis: *there will be multiple ratings per item* and *each user will rate multiple items*. Both properties are true for each iteration of the song generating process. For each generated song, the user will rate more than one item. Additionally, the user will create more songs (when testing the prototype), meaning that the user will create even more ratings. It is only the time of the thesis that limits the first and third data properties. There is no limitation to the software to how many items the prototype can keep in its content library. When testing, the prototype had 37 tracks to select from. This number includes all the track with different instrument types. Adding more audio tracks does not require any changes in the code. This means that the first property also holds for the system, since the system does not need to change in order to accept more items. Additionally, 15 people partook in the testing. Neither the data, nor the system has a limit as to how many people can partake in the prototype, but there is a limit to how many items each user can rate. In other words, it is possible for a user to rate every item in the content library. If a user has given a rating to every item, it only stops the collaborative filtering from recommending new tracks. The knowledge-based algorithm will still be able to recommend items. We therefore argue that the third property also hold for the prototype.

The second category Schafer, *et al*. (2007) use to describe the suggested properties, they have named "*Underlying Meaning*". The first property of this category is "*For each user of the community, there are other users with common needs or tastes*". This property highlights one of the strengths of collaborative filtering. It will leverage the common taste that people share. However, the algorithm cannot make a recommend content to someone who does not have anything in common with others. The next property is "*Item evaluation requires personal taste*". In short, the users should rate items based on their preferences, not a uniform standard. This property argues that the data in the domain can neither be objectively good, nor objectively bad. Data in which there are some subjectivity and preferences are ideal for content-based algorithms. The last property in the category further clarifies the notion from the previous property: "*Items are homogenous*". This highlights that every item should be similar in nature and worth from an objective point of view. The differences in quality should only be derived from the user's subjective meanings (Schafer, et al., 2007). In terms of the prototype, the items in the content library are objectively equal. They are all short audio clips, and the prototype places no special emphasis on any subset of the clips. When creating the audio clip, special attention was made to making the audio tracks as equal as possible. However, some of the clips are objectively different from the others. Although different, we made sure that they were of equal quality as the rest of the tracks. For example, some of the tracks were given a slight shift in rhythm, making them less compatible with the other tracks. Even though these tracks are different, they are created with the same attention to quality as the other tracks. We therefore argue that the final property in the second category holds for our prototype. Additionally, when rating music the user is required to have personal taste. No standard is given to which the users should follow. Users are free to dislike or like the tracks, which is partially derived from the subjectivity of music itself. We therefore also argue that the second property (of the second category) holds, and then in turn the first property.

Third of Schafer, *et al.*'s (2007) categories are *Data Persistence,* and as the name suggests, it describes the longevity of the relevance of the data. There are two properties in this category: *Item persistence* and *taste persistence.* For how long the recommended items are relevant is critical to the accuracy of the collaborative filtering algorithm. Given a domain where the relevance half-life is short, it is harder to recommend new content, contrary to a domain where the items are always relevant. A similar

property can be raised regarding the user's taste. In domains where a person's taste might change rapidly it is harder to implement a collaborative filtering agent (Schafer, et al., 2007). Music clips will be relevant for as long as the prototype exists. Therefore, relevancy of the items in the content library does not really expire. On the other hand, the users' preference towards the sound clips might change. However, we argue that due to the session-based nature of the prototype, this issue is not that critical.

The arguments presented lead to the conclusion that collaborative filtering algorithms are suitable for RecOrder. The next step is therefore to find out whether user-based or item-based collaborative filtering is the most suitable for the project.

User-based filtering compares the ratings of the user (u1) with ratings of other users (u2). It then tries to find users with similar interests to u1. By comparing the u2 ratings on items u1 has not rated, to items both u1 and u2 have rated, the system can attempt to give a recommendation. This inherit focus on the similarity of users weakens user-based filtering in terms of both the first and the third of the requirement we posed. As a reminder, the first is that is does not depend on a large user-base, and the third is that the algorithm should not depend on item-information. A bigger emphasis is placed on the number of users. Without a sufficiently big user-base, user-based recommendations will be weak. In the case of the prototype, it is likely that there are more items than users. Therefore, the user-based filtering algorithms is less fitting for RecOrder.

Item-based filtering provides a solution to two of the requirements: Firstly, item-based filtering does not need a large group of users. The filtering algorithm serves valid recommendations for the user given that the engine has some access to previous recommendation (Lemire & Maclachlan, 2005). Second, the content itself is not considered in item-based filtering. Since the item-based filtering works on the predicted rating of the user based on previous ratings, the recommended item is still relevant. However, the algorithm does not assess the content of the items themselves.

The issue of a smaller number of items available is still present with an item-based algorithm, but it is less important, since the algorithm can still make valid recommendations. In contrast to user-based filtering, item-based filtering puts a greater weight on the user's own ratings, instead of how similar the users rating is to other user's ratings. Therefore, it is more critical to the recommendations that the user rates multiple items rather than that other users rate multiple items. As a result, the prototype uses an instance of an item-based algorithm.

4.4.1 Hybrid switching

The implemented recommendation system is a hybrid switching recommendation system. As described in Section 2.1.2, a hybrid switching system consists of more than one recommendation algorithm. There are several types of hybrid recommendation system architectures. The four that was considered for the prototype corresponds to the four that were described in Section 2.1.2: *Weighted, Mixed, Switching* and *Cascade*.

Each song needs multiple recommendations, requiring that the implemented configuration of recommendation systems must be a light-weight approach and be able to give recommendations quickly. Cascade, Mixed and Weighted all run multiple recommendation engines simultaneously (or after each other) for each needed recommendation. This could cause unnecessary long runtimes. For this reason, the Switching approach was chosen, both for its (relative) simplicity to implement and for its efficiency.

There are two main reasons for the recommendation selector to switch from one algorithm to the other. The first reason is triggered when a new user is created. This reason relates to the cold-start problem. If a newly created user tries to create a song, the recommendation algorithm will have no data on the users' preferences. As a result, the recommendation system cannot give a recommendation using the item-based collaborative filtering algorithm. The second reason that the recommendation selector will switch algorithms is if the user has rated too few items. This is the case if the system tries to recommend content using the item-based collaborative filtering algorithm. If there is not enough user data to compare the user to other users, the system cannot recommend an item.

When creating a new song, some parameters are included. One of these parameters is a user profile identifier. This identifier is tied to the ratings the user gives during the feedback process. In order to determine whether a recommendation can be made, the recommender selector queries a database table containing the preferences of all the users. From the table, every unique user-ID is selected. This selection will therefore result in a list of every user that has given feedback to a recommendation. The next step is to look for the ID of the user (one of the parameters needed to create a song) that requested the new song within the returned list. If the ID is found in the list, meaning the user has rated items before, the recommendation selector returns an instance of the collaborative filtering engine. However, if the user-ID is not found in the list, an instance of the knowledge-based engine is returned. The selected recommendation engine is returned, and the recommendations are retrieved from that engine. To better highlight the process, an example follows.

When a new user is created, the recommendation engine has no data on which to base a recommendation and recommendation cannot be made using the collaborative engine. The prototype tries to find the newly created user-ID within the list of users, but since it is looking for a new user it will not find her. Then, Recommender Selector switches the recommendation engine with the knowledge-based engine. Once a song has been generated, either the user profiles preferences are updated, or a new profile is created. The feedback the user gave to the song is then added to the user profile. A saturated user-profile is necessary for the item-based algorithm to make a recommendation. Only generating one song with the knowledge-based engine might not generate enough user data to saturate the user-preferences. Several iterations of knowledge-based recommendation generating might be needed. The number of iterations depends on how many other users have rated the same items. If few users have rated the items used in the song, the collaborative filtering makes weaker recommendations. As soon as there is a broad enough dataset for the item-based engine to make recommendations, the *Recommender Selector* returns to the collaborative item-based engine. The prototype continues to use the item-based engine until the user has rated every item. At that point the item-based engine cannot return any further recommendations, until new items are added. As a response to this, the system then switches back to the knowledge-based engine. The item-based engine is therefore most used and functions as the main recommendation system, while the knowledge-based engine serves as the backup engine.

4.4.2 The Primary Algorithm

The implemented item-based algorithm is *Weighted Slope One.* Slope One is an algorithm that leverages the ratings of other users (thus making it a collaborative filtering algorithm) to determine the current users predicted rating of the current item (Lemire & Maclachlan, 2005). By factoring in the number of ratings for each rated item, the algorithm becomes a weighted algorithm. The primary reason for selecting Slope One is that the algorithm can yield valid recommendation despite a small

data set. This is one of the strengths that Lemire and Maclachlan (2005) highlight as their goal with Slope One. Other strengths they highlight is effectivity at runtime, reasonable accuracy, ease of implementation and robustness with the addition of new ratings (Lemire & Maclachlan, 2005). Since this thesis has a small scope, these strengths are essential to the success of the prototype.

In the prototype, the slope one recommendation procedure returns several recommendations. The results are sorted by their predicted score, the highest recommended first. This score is an estimation of how well the user will like the item. The prototype then selects one of the top three results at random, to decrease predictability. These recommended items will only be items that the user has not rated and that other users have rated. Thus, one of the weaknesses with the weighted slope one algorithm is that if no users has rated an item, it will not appear in other recommendations. In other words, the cold-start problem for items is highly relevant for slope one. To counter this, we implemented a knowledge-based approach.

4.4.3 The Secondary Algorithm

The main reason for implementing a second algorithm where to counter the weaknesses of the primary algorithm. The primary algorithm, an item-based collaborative filtering algorithm, struggles with giving recommendations when no user data exists; the cold-start problem. The secondary algorithm therefore needs to be able to provide recommendations despite the lack of user data, and when new items is added. Of the highlighted algorithm types (*content-based, collaborative* and *knowledge-based*), only knowledge-based filtering is an option. The reasons given for why content-based filtering is not valid as a primary choice are still relevant for the secondary algorithm. Knowledge-based filtering (KBF) algorithms are good at mapping the needs of the user into suggestions of items (Burke, 2002). The user, in this case, would be the song creation algorithm. This is ideal for the needed algorithm.

When creating a new song, the prototype has clear instructions of what type of recommended track it needs. In KBF algorithms, it is typical for the user to express their needs or what type of items they desire. The previously mentioned pre-filtering of items (based on bpm, key and type of instrument) serves as the user's need and will help guide the KBF algorithm to a recommendation. However, the instrument, bpm and key alone are not sufficient to differentiate the tracks from each other. Many tracks can share the same values in all three properties, and still be very different. An additional item descriptor was therefore introduced: *intensity*. The intensity of an item describes how energetic the instruments in the track are playing. For example, if a drum track has a high intensity score, the drums might be playing a lot of rapid beats. The intensity does not affect the tempo of the track.

One of the weaknesses of knowledge-based filtering algorithms is that they do not learn from previous recommendations. This is partially due to not using a user profile. As a result, it is likely that they recommend the same content multiple times to the same user. To counter this, the prototype introduces some randomization to the knowledge-based algorithm. When the knowledge-based algorithm suggests items, it will return a selection of fitting items, and the prototype choses a random item from that selection. This randomization will help minimize the static recommendations often found in KBF algorithms. Using the intensity scores, some differentiation can be made between the tracks, and sense of progression can be emulated during the song. The secondary algorithm was therefore created to change the intensity of the recommended tracks based on what section of the song was being generated.

5 RecOrder – The Implementation

This section contains the details of the implementation of RecOrder. This includes description of the programming language used, the details of the recommendation algorithms as well as pseudocode for the different steps in the song creation process of RecOrder.

5.1 Programming language and Platform

We wanted to dedicate as much time as possible to implementing the song generator and the recommendation system, and as little as possible to creating the surrounding system. Thus, a programming language that was easy to write, flexible and modular was preferable. Additionally, we looked for a programming language where there were existing libraries of recommendation systems. By either directly using the existing libraries, or by using them as a basis for further development, we hope to cut down development time further. For these reasons, Python (The Python Software Foundation, 2018) was chosen as the programming language used to develop RecOrder. Python is an open source language (Open Source Initiative, 2007) with a focus on flexibility and modularity. It has several libraries dedicated to recommendation systems.

Alongside Python, HTML (HyperText Markup Language) and CSS (Cascading Style Sheets) were also used. These two languages were primarily used to facilitate the implemented web-framework and provide a basic user interface for starting the song creation process. HTML is used for creating the foundations and structure of websites, and CSS adds styling (colours, borders, etc) to the web pages (W3C, 2016).

MySQL was used as a database (Oracle Corporation, 2018). It is an open source database solution. MySQL was chosen for RecOrder because of the ease of creating tables and queries. The database system created for RecOrder contains several tables, primarily storing the user preferences and data about the audio tracks. In addition, the queries were saved as Stored Procedures[22], allowing for further reuse.

Three Python libraries were used: *Flask, Surprise* and *Pydub*. Flask (Ronacher, 2018) is a framework that facilitates the creation of webpages. The user can create modular webpages that are combined into the resulting webpage using the integrated templating language. Flask also provides the user with extensive debugging tools and unit testing support. The Flask framework integrates seamlessly into any Python script, making it an ideal way of creating a web user-interface. The second Python library used is Surprise. Surprise (Hug, 2017) is a library designed to build and analyse recommendation algorithms. There are several algorithms included in the library, and good documentation. The final iteration of the prototype did not end up using the library, but it was used extensively during the development as a basis for development of the implemented recommendation system. Pydub (Robert, 2017), the third library used, is a Python library created to manipulate audio. The library can for example split, merge, fade between and create audio tracks.

[22] https://dev.mysql.com/doc/connector-net/en/connector-net-tutorials-stored-procedures.html

5.2 The Implementation of the Song Creation Process

The user starts the song creating process. The system is designed for the user to enter their preferred song length, bpm and key. The preferred length of the song is limited to three options: "*short*", "*medium*" and "*long*". Once the system receives this input, the system initiates generating of the song. For this prototype we have limited the possible selection to short songs, in the key of D and at 105 bpm.

Pseudocode for Song creation process

Input: *bpm, key, songLength and profileKey*
Output: Generated song

```
1:  Get song structure based on songLength
2:  FOR EACH unique section of the song structure:
3:  |   Create a Slice:
4:  |   |   Get a random slice structure
5:  |   |   FOR EACH active instrument track in slice
    |   |   structure:
6:  |   |   |   Get a recommended track from the
    |   |   |   recommendation engine
7:  |   Save the slice
8:  Move away old audio files
9:  Put the song together:
10: |   Merge audio tracks in each slice
11: |   Append each slice based on song structure
12: |   Save finished song as audio file
13: Register all clip-clip relations in the generated song
14: Get feedback on the instrument tracks
15: Get feedback on k number if slices
```

Figure 14 – Pseudocode for the song creation process

Figure 14 displays the steps of the song creation process as pseudocode. The following paragraphs refer to the noted step numbers in Figure 14. The database is queried for song structures, based on the input from the user as shown as step 1. The query retrieves all structures that have the correct length property (for example "short"), and the system picks one of the results at random. This results in the prototype having a partially randomized song structure selection. The song structures consist of a series of letters, as highlighted in Section 4.2 (for example: ABABC). The prototype then parses the selected song structure. The parsing consists of splitting the structure up by each letter and fining every unique letter. A section of music is generated by the prototype for each unique letter found. These sections are also referred to as *slices* in Section 4. The steps for creating each slice is shown through step 2-7 in Figure 14. Each slice also has a structure selected partially randomly (step 4 in Figure 14). These structured determine which instruments are playing, and which are silent. The selection is partially random, because it is weighted towards having all the instruments playing at the same time.

The prototype implements the slices as a list of integers. The lists are filled by the recommended track identifiers from the recommendation engine. Each type of instrument has a defined position in the slice list. The guitar track identifier is placed in the first position of the list, then the drum track identifier follows, and finally the bass track identifier. These identifiers correspond to the name of an audio track. Therefore, each slice contains the "keys" needed to create that section of the song. For each instrument track that is active in the slice-list, a request for a track recommendation is sent to the recommendation system. When every slice has been completed, the song generating process continues. Steps 5-7 in Figure 14 highlight the slice creation stage of the song creation process.

Step 8 (in Figure 14) highlight a small process that moves any audio files from previous song generation processes into a separate folder. This is to ensure that there is no confusion when the user is going to listen to the generated song.

As noted in step 9 to 12, the next step of the song creation process is to put the song together. The system starts merging the tracks within each slice. This produces a complete audio file for each of the sections of the finished song. Once the song creation process has generated the audio track for each slice, the slice audio tracks are appended to each other. The order of which the slices are appended are based on the song structure, determined at the start of the process.

Step 13 to 15 in Figure 14 pertain to the feedback procedure in RecOrder. First all the clip-clip relation within the generated song is stored. This is stored both in a table in the database, and as list of tuples. The tuples contain the track identifiers of the two clips in the clip-clip relation. Following this, the feedback for each instrument track is collected from the user. Finally (as noted in step 15), feedback for k number of different slices is collected. The k defaults to 3, but it is possible to override this number. K number of slices is selected randomly from the generated song. An audio file is generated for each slice that is going to be rated. This is to make it easier for the user to find the relevant part of the song. The feedback (from both the instrument track and the slices) are stored in the database in a table containing user preferences.

5.3 The Implementation of the Recommendation algorithms

The prototype uses a *hybrid switching* recommendation system[23]. The class called *Recommender Selector* ensures that a recommendation always can be made. This is achieved by switching the recommendation engine when one fails to provide a recommendation. The Recommender Selector class does not return the actual recommendations but returns an instance of one of the recommendation engines. The song creation process system then queries the selected engine-instance for a recommendation. The process of switching between recommendation engines makes the prototype a *hybrid switching* system.

The two recommendation engines (*item-based* and *knowledge-based*) are implemented as subclasses of a superclass: *RecSysEngine*. The superclass is empty, except for a method called GetRecommendation. This method is also empty, except that it raises an exception when called from the superclass. Both the subclasses inherit this method since they extend the superclass. The inheritance from the superclass is displayed in Figure 15. Thus, the GetRecommendation method within both recommendation-classes have the same name and parameter structure. This ensures that the system can switch

[23] A definition of *hybrid switching recommendation systems* is found in Section 2.1.2

between the classes using the same method signature. In addition, if the need to change one of the recommendation algorithms arises, it is easier to implement a new engine.

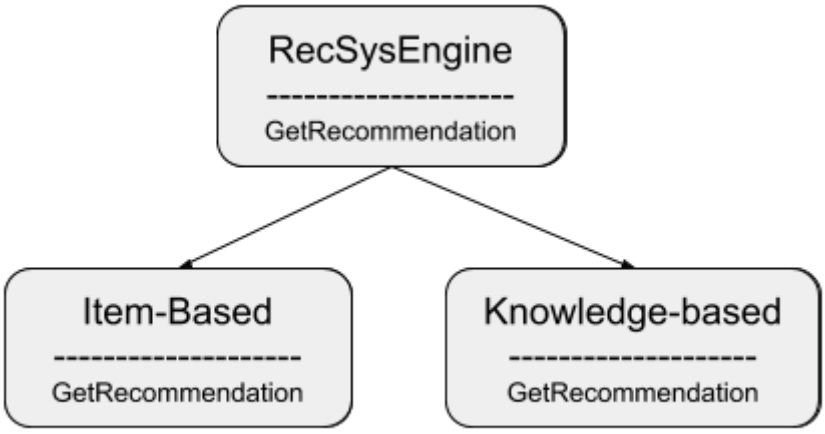

Figure 15 – Recommendation engine inheritance structure

As explained in Section 4, *weighted slope one* was chosen as the primary recommendation algorithm. The implementation of Slope One is based on instructions found in a tutorial (Zacharski, 2015). The tutorial is in the form of an online book, and it is an introductory guide to datamining techniques. It covers topics regarding recommendation systems, classification, Naïve Bayes and clustering.

Some modifications were made to the code. The tutorial used a different dataset from ours, and it is often necessary to do some adjustments when switching datasets. Most of these changes are preparing and formatting the data to fit the algorithms needed format. The Slope One implementation needs to receive a dictionary[24] for each user, containing what items the user has rated and the ratings themselves. The data is not stored in this way in the database, and therefore needs to be collected and put into a dictionary.

The prototype needs to have a pre-filtering of the available tracks in order to ensure that the recommended tracks will be compatible. This prefiltering is done in three stages. Figure 16 shows the pseudocode for these three stages. First, the prototype collects all tracks with the requested bpm, key and instrument into a list. These values are the values inserted at the start of the song creation process. The IDs of these tracks are stored in a list. This stage is shown in step 1 in Figure 16. Once the tracks are collected, all user's preferences are also gathered. In other words, every user's item-rating pair. This is the second stage in the pre-filtering, and in Figure 16 it is shown in step 2. The next stage is a for-loop that iterates through each of the user-preferences. The structure of the user's preference is noted in brackets behind the for-loop in line 4 in Figure 16. As seen, there are two items stored in the user' preference. These items are the two items are stored as a clip-clip relation, and the rating is placed on the relation between two items. Within the loop, the system checks if both the rated items exist in the list of tracks with the right bpm, key and instrument. If that is the case, the tracks and the

[24] Python dictionaries are very similar to hashmaps from other programming languages. The dictionary consists of several items, each with a unique key that retrieves the item.

rating for the track, are added to a dictionary. The key in the dictionary is the item-ID, and the value is the rating. Steps 4 to 11 in Figure 16 display these steps.

Pseudocode for Pre-filtering of Available Tracks

Input: *bpm, key and instrument*
Output: Dictionary of the user's preferences

```
1:    Get clips with relevant bpm, key and instrument
2:    Get the preferences of all the users
3:    Create an empty Preference Dictionary
4:    FOR EACH user's preferences [userID, item1, item2, rating]
5:    |    Create a Result Dictionary, and leave it empty
6:    |    Get the two items (item1 and item2) from the
      |    current user's preference.
7:    |    IF item1 is in the list of relevant clips
8:    |    |    Add item1's rating to Result Dictionary
9:    |    IF item2 is in the list of relevant clips
10:   |    |    Add item2's rating to Result Dictionary
11:   |    Save the store user preferences in Preference Dictionary
12:   RETURN Preference Dictionary
```

Figure 16 – Pseudocode for prefiltering of available tracks

Once the loop has iterated over every user-preference, the dictionary with the relevant items and ratings, are added to another dictionary, with the user-ID as each key. The structure of the returned dictionary is displayed in Figure 17. This is the least generalizable part of the item-based recommendation system, since it needs to access data properties.

```
{
        "User-ID 1" :
        {
                "Item-ID 1" : rating,
                "Item-ID 2" : rating
        },
        "User-ID 2" :
        {
                […]
        }
}
```

Figure 17 – Prefiltering dictionary format

Since the selected version of Slope One is weighted, the number of times each item has been rated needs to be counted. In addition, the deviation between each pair of items needs to be calculated. The next steps contain a series of for-loops that iterate over the pre-filtered list of ratings and then the items within. For each user in the list, find every item the user has rated, plus the rating. This is temporary stored as a pair. Then, for each item and rating that is not equal to the first pair, increase

the frequency by one and save the difference between the two ratings. Finally, divide each deviation-pair by its frequency to get the weighted result. Figure 18 shows the pseudocode for these steps.

Pseudocode for Deviation Calculation Method

Input: A dictionary all the users and their ratings
Output: A dictionary of the deviation between each item

```
1:  FOR EACH users' ratings:
2:  |   FOR EACH item1 and rating1 pair:
3:  |   |   Create empty frequency and deviation dictionaries for
    |   |       the item and rating pair
4:  |   |   FOR EACH item2 and rating2 pair that is not the first pair:
5:  |   |   |   Calculate the difference between the two ratings
6:  |   |   |   Store the difference in the deviation dictionary
7:  |   |   |   Increase the frequency of the item-item (item1 and item2) pair.
8:  |   |   |   Store the new frequency in the frequency dictionary
9:  FOR EACH item (item1) and ratings in the deviations
10: |   FOR EACH item (item2) in ratings
11: |   |   Divide the ratings of item2 with the frequency of item1 and item2
12: |   |   Update the ratings for item2 with the result of line 11
13: RETURN the deviation dictionary
```

Figure 18 – Pseudocode for the deviation calculation method

The calculation of deviation between each item would normally be done in an offline phase, in order to make runtime-recommendation quicker. Calculating the deviation between every combination of items each time the system needs a recommendation, will slow down the system quite a lot. This is especially true when more items and ratings are added. However, these calculations were not moved into an offline phase in the prototype, mainly because of the limited development time. Given more development time, a separate offline phase would be added to the prototype. At present, the number of items in the prototype is small enough not to significantly impact the speed of the needed calculations.

Once the deviation between each track has been computed, the recommendation process continues to the *Slope One* algorithm. Lemire and Maclachlan (2005) write the following formula to how Weighted Slope One returns a prediction for a user:

$$p^{wS1}(u)_j = \frac{\sum_{i \in S(u) - \{j\}} (dev_{i\,j} + u_i) c_{j,\,i}}{\sum_{i \in S(u) - \{j\}} c_{j,\,i}} \qquad (1)$$

Thus, *p* is short for prediction, wS1 is weighted slope one and $(u)j$ is the predicted item (j) for the user (u). $S(u)$ in the formula is every item that the user(u) has rated. $S(u) - \{j\}$ then becomes every item the user has rated, except the item we want a recommendation for (the item = $\{j\}$). $dev_{i,\mathrm{i}}$ is the deviation score between item i and j. u_i are the users rating for item i. Finally, the frequency of the item j $-$ i pair (*i.e.* cardinality) is noted as $C_{j,\,i}$. The numerator of (1) then becomes: for every item (i)

that user (u) has rated, except for item j, add the deviation between item i and item j with the users rating for item i. Then multiply the sum with the cardinality of items *j* and i. In other words, the number of times both item *j* and item i have been rated by the same user. The denominator of (1) can be summed up as: for every item (i) that user *u* has rated, sum the cardinality of *j* and i. The result of numerator over denominator then becomes the prediction. Figure 19 displays the pseudocode for *weighted slope one*, which is based on Zacharski's (2015) implementation.

Pseudocode for Weighted Slope One

Input: *userRatings*, *deviation* (dictionary), *frequencies* (dictionary)
Output: recommendation (dictionary)

```
create two empty dictionaries: recommendation and frequency
FOR EACH item and rating in userRatings:
|    FOR EACH diff-Item and diff-Rating in deviation:
|    |    IF diff-item is not in userRatings and item is in deviation:
|    |    |    get the frequency of diff-item and item from frequencies
|    |    |    add diff-Rating and rating and multiply the result by the
|    |    |    frequency of diff-item and item
|    |    |    save the result from line 6 in recommendation with
|    |    |    diff-Item as key
|    |    |    save the frequency of diff-item and item in frequency
|    |    |    with diff-Item as key
FOR EACH recItem in recommendation:
|    divide recItem by its corresponding item from frequency
sort recommendation in descending order
RETURN recommendation
```

Figure 19 – Pseudocode for Weighted Slope One

The prototype uses a basic implementation of a knowledge-based filtering (KBF) system as the secondary algorithm. The KBF system implemented is a constraint-based system, that narrow down the available items until there is only a handful of items left. Then, the KBF system selects a random item of the remaining items. As mentioned, there is a pre-filtering of items prior to the recommendation part. This pre-filtering is also done prior to the KBF recommendation. In addition to the pre-filtering, a new constraint is added: the intensity of the tracks. This constraint is in place to ensure that more relevant items are being recommended. The KBF-algorithm categorises three stages of intensities: *low, medium* and *high*. For each track there is a database table with the track-ID and an intensity score. The score describes how intense the track feels, and it ranges from one to five, where five is the most intense tracks. The *low* category describes tracks with intensity score 1 and 2, the *medium* intensity describes tracks with score 3 and 4, and *high* intensity cover tracks with score 4 and 5. It is important to note that the intensity score does not affect the tempo of the track: some tracks are "busier" than others, thus making them more intense.

The KBF-algorithm recommends different tracks based on what part of the song are being generated. Recall that each song is split into slices, each one given a letter. The recommendation request includes one of these section-letters, based on what section the generator is at. The KBF-algorithm then finds tracks with different intensity scores based on the section of the song. By including this score into the KBF-algorithm, it is possible to simulate a song progression, since the song will vary in intensity. The low intensity tracks are fetched at the start of the song (section-letter A). This creates a build-up and

later a break if the A-section is reused in the song. The medium intensity tracks are used when section B is created. This contrasts the low intensity of the first section. Finally, when section C is created a high intensity track is recommended. For all other sections, the prototype selects a random track, from all intensity levels. Once the available tracks are filtered based on intensity, the tracks are put into a list. The algorithm then picks a random track-ID from the filtered list. The selected track identifier is then returned to the song creation process. The steps for the KBF-algorithm is illustrated in Figure 20. In Figure 20, when talking about "all tracks" it is meant all the available tracks from the pre-filtered list of tracks.

Pseudocode for KBF algorithm

Input: *pre-filtered track list*, *input-section*
Output: *track-ID*

```
1:   create empty list: recList
2:   IF input-section IS "A":
3:   |     get all tracks with intensity 1 or 2, and add them to recList
4:   ELSE IF input-section IS "B":
5:   |     get all tracks with intensity 3 or 4, and add them to recList
6:   ELSE IF input-section IS "C":
7:   |     get all tracks with intensity 4 or 5, and add them to recList
8:   ELSE:
9:   |     get all tracks add them to recList
10:  select a random track from recList
11:  RETURN the ID of the selected track
```

Figure 20 – Pseudocode for the implemented knowledge-based filtering algorithm

6 EVALUATION METHOD

There are three primary categories of evaluating the effectiveness of Recommendation Systems: *user studies, online methods* and *offline methods* (Aggarwal, 2016, p. 227). In *user studies* the participants are actively recruited to test the implementation of the recommendation system. Feedback from the users (either during or after the testing) are used to determine the success of the recommendation system, and the accuracy of the recommendations. The *online methods* measure the direct impact that the recommendation system had on the users and the choices they took and is often done with real users in real use-cases. A comparison is made between the users that get recommendation from the recommendation system, and the users that do not (*A/B* testing). Therefore, the online methods of evaluation require a more complete system with the recommendation system implemented. The *offline methods* leverage data generated from user-interactions with the recommendation system. The data is analysed for patterns using different types of evaluation metrics/formulas (Aggarwal, 2016, pp. 227-229).

User studies was selected as the most suitable category of evaluation for RecOrder, due to the limited timeframe of the thesis. The online methods require more development time than user studies. Likewise, the offline methods require more in-depth study and implementation of evaluation metrics. Therefore, a user study was created. The user study consists of series of questions that aim to establish whether the prototype is successful in creating personalized content. In other words, if the prototype is able to utilize the users preferences when creating a song. The anticipated trend, prior to running the experiments, is that the songs should get closer to the preferences of the user with each iteration. However, this might not be the case. The following section describes the user study.

6.1 SET-UP DESCRIPTION

The testing done for this thesis is a combination of a survey and interaction with the created prototype. Interacting with the prototype is crucial in order to give feedback on the created songs. The participant needs to establish their preferences, and have multiple songs generated based on them. This will give them a good basis for comparison between different songs. Preferably, the songs should be generated in the same test-session. As explained in the prototype description section, the prototype will first use the knowledge-based approach, in order to map the user's preferences. Then, it will switch to the item-based approach. As a result, the user must generate at least one song before their preferences take effect. A total of four songs will be generated by the user during the testing. By limiting the number of generated tracks to four, the risk of the system getting over-specialized is lowered. We found that during the development of RecOrder, when generating more than four songs for each user, the item-based algorithm had a tendency to not be able to recommend new content. This was because every discovered item had gotten a rating, and thus there was nothing left for the item-based algorithm to recommend. This is most likely due to the low number of items. Therefore, the users will not create more than four songs when testing the prototype.

In addition to interacting with the prototype, it was necessary for the user to express thoughts about the process. A systematic comparison between each generated song was needed. Thus, we introduced a survey to use in union with the interaction with the prototype. The survey contains questions that ask the participant to compare the resulting songs from multiple song-iterations.

The survey is created in *Google Forms*, Google's survey tools [25]. This tool allows for quick and flexible survey creation, and the results can be exported to spreadsheets. There is also pre-calculated visual aid for the answered questions. For example, if the participants answer a multiple-choice question, a pie chart will be created, displaying the percentage of answers for each answer. An example is shown in Figure 21. Although the graphs used in the paper itself are created in Excel, the graphs created in Google Forms served as an inspiration for what types of graphs to use.

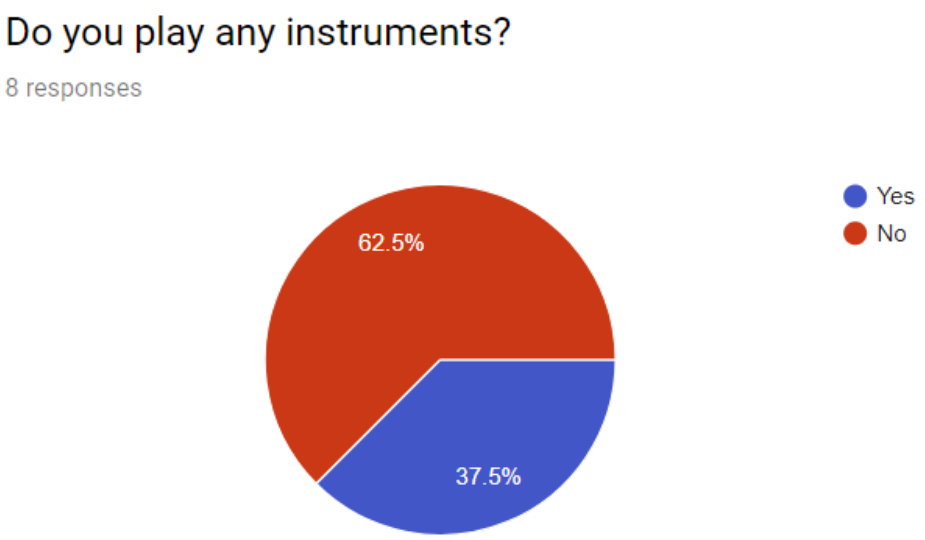

Figure 21 – Example of the generated graphs

Since the tester must listen carefully to tracks, often many times, the testing was held in silent rooms, with as few distractions as possible. The testing was done in study-rooms at the university campus (UiB). The participants are seated at a desk, with a laptop and a headset. The prototype is running on the laptop. The only persons in the room are the participant and the test supervisor. The test supervisor describes the procedure of the test and aids in any technical questions the participant might have. The supervisor does not see what the participant writes in the survey, what they rate the different sections of music, nor can he or she hear the generated music. This was intentional, as the supervisor should not be able to affect the results of the participant.

No particular skills are required of the participants in order to participate in the testing. They do not require any musical background, nor any programming skills. This disregard to musical experience was to get a broader group of people to test the general use of the prototype. Although the participants were asked briefly about their musical background, what they answer is not used in further analysis. The general quality of the generated songs was prioritized over more specialized analysis from, for example, people with musical education. The scope of the thesis did not lend enough time towards running detailed testing, thus, anyone could join in the testing.

The participants are not told specifically how the prototype generates music. Some of them are introduced to the concept of using recommendation systems to generate music, from prior conversations, however, none of the participants was told the steps in which the songs are created. This was deliberate in order to minimize bias towards the generated songs. The intention is that the participants

[25] https://www.google.com/forms/about/

must rate every song equally, even though some of the songs are not created by the primary recommendation engine.

Some circumstantial variables are expected to affect the generated result. For example, the recommended tracks will be different from person to person. Each person will also have different preferences to what kinds of tracks they like. If a track they really do not like is recommended, it might affect their perception of the entire song. They will rate the song with a lower score as a result. The question then becomes if the prototype can properly respond to that change. Will the user be able to affect the generated content enough to remove the tracks that they do not like? The user is asked to generate several songs and compare them in retrospect. This will reveal if this question is correct.

Since feedback is a vital aspect of recommendation systems it was also necessary to find out if the implemented way of giving feedback truly allowed the user to reflect their opinions. Therefore, the survey includes a series of questions regarding the feedback process. These questions will illuminate some of the challenges with implementing recommendation systems in other settings. As mentioned in Section 4.3, feedback was one of the challenges when creating the prototype. By including these questions about the feedback process, some indication to the success of the feedback-implementation might be gained, in addition to how giving feedback in this manner is perceived.

6.2 SURVEY RUN-THROUGH

The first part of the survey contains a slide with information on what the survey will contain. Additionally, the slide states that the survey does not collect any identifying information, and that the information collected will only be used in conjunction with the thesis. The user is then asked if they play any instruments. This question is not directly connected to the rest of the survey. However, the question was included since it might show interesting trends amongst users with and without a musical background. This marks the end of the introduction part of the survey. Figure 22 displays the different stages of the survey.

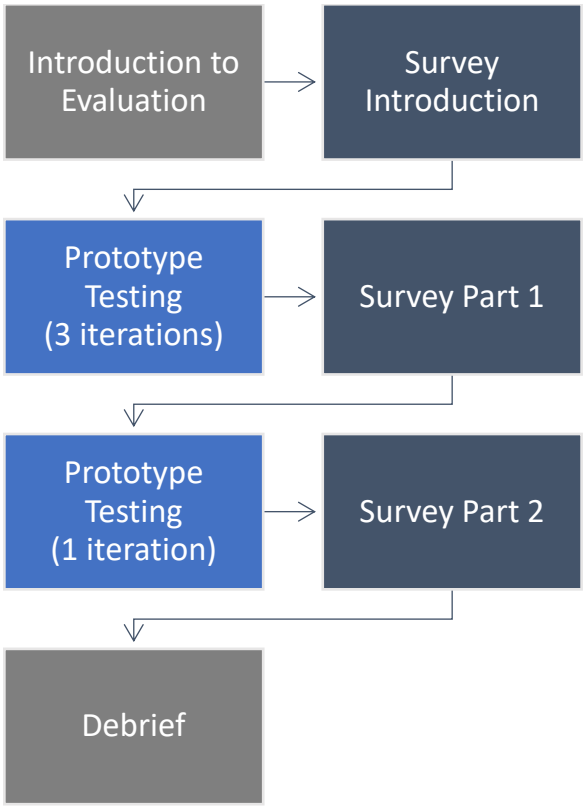

Figure 22 – Procedure overview of the experimentation

Following the introduction part, the prototype requires the user to generate three songs. This stage is named *Prototype Testing (3 iterations)* in Figure 22. The users receive a brief explanation of what the next steps will entail. The instructions are related to how the prototype will display its progress, where the feedback should be entered, and where the songs and clips they should listen to appears. Figure 23 shows the layout of the screen the user. The left part of the screen is a web browser. Within the browser, the user starts the song creation process (by clicking on the blue button). Additionally, the user can answer the survey in another tab in the browser window. The top right of the screen displays the folder where the generated songs and clips appear. The prototype terminal window fills the bottom right of the screen. This window displays the instructions from the prototype and this is also where the user inputs their ratings.

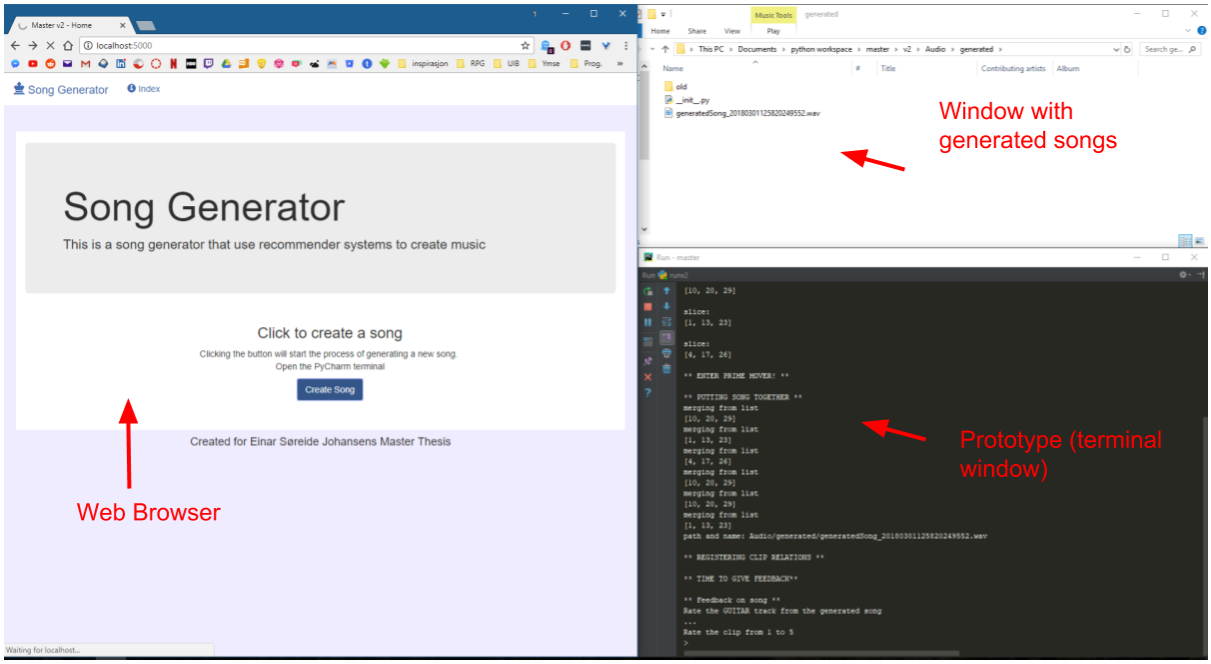

Figure 23 – Screenshot of the testing setup

For each generated song, the user listens to the song, rates the consistency of the guitar track, the bass track and the drum track. The feedback is limited to a score from 1 to 5. Once each of the tracks in the song have been rated, the prototype presents the user with up to three slices, which they should rate in a similar manner. When the three songs have been created and rated, the prototype have enough information on the user's preferences. The next generated song is expected to mostly contain well recommended tracks. In addition, the user has been exposed to the song creating process and are ready to answer questions about the experience. Thus, the user returns to the survey.

The following stage of the evaluation is named *Survey Part 1* in Figure 22, and consists of multiple questions regarding the song creation process. The user is asked to compare the results of the different iterations of the song creation process. There is no restriction to whether the user can listen to the previously generated tracks. In fact, they are encouraged to listen to the songs again, if they wish. The first question pertains to how well the different instrument tracks fit together. This is supposed to test whether the generated songs managed to have a consistency across slices[26]. The specific question is:

> *"There are three instrument tracks in these songs: the guitar track, the bass track and the drum track. From the first to the last song, was there a change in how well the different instrument tracks fit together?"*

This is a multiple-choice question, with five answers: "*They became worse towards the last song", "They became slightly worse towards the last song", "There was no notable change"," They became slightly better towards the last song"* and *"They became much better towards the last song*". The available answers are written in this manner to reflect the scoring system in the prototype. In addition, this allows for nuance in the answers. Several other questions follow this structure in the available answers.

[26] For a definition of Slice, see Section 4.2

The next question is similar in nature to the previous one, but the participant compares the parts of the song instead of each instrument track. This question aims to detail the progression of the slices themselves, from song to song.

> *"All songs can be split up into different parts. We split the song into parts where there is a different melody, or the song somehow changes (in many songs this is the verse, the chorus, etc). From the first to the last song, was there a notable change in how well the parts of the songs fit together?"*

This question is also a multiple-choice question, and the available answers are the same as the previous question.

The third question is a much simpler one: *"Was there a song that was better than the others?"* The question has two available answers: "*Yes*" and "*They were all equal / heard no difference*". If the participant answers yes, they are asked follow-up questions. There are two follow-up questions. The first one is "*What song did you like the most?"*. This is a multiple-choice question with three answers, one for each of the songs they generated prior to part 1 of the survey. The second question is *"What about that song was better than the others*". Here the participant is prompted to write as detailed as they would like, and there is no limit to the amount of words they can write.

If the participant answers "*They were all equal / heard no difference*", they continue to the next series of questions, which queries the participant about their feelings regarding giving feedback in the song generating process. Prior to the questions a short text explains what the questions in this section pertain to. The text reminds the participant that they rated songs on a scale from 1 to 5 (when giving feedback to the prototype), and that the following questions are about this process. There are four questions regarding this topic, three of which are mandatory. The first question in the section is *"Was it difficult to give feedback on pieces of music in this way?"* There are three answers for this question. The first two are "*No, it was easy*" and "*Neither difficult, nor easy*". The last answer is "Yes, it was difficult". If the participant selects this answer, they are asked to briefly write about what they thought was difficult. This question is not mandatory. The next question is a multiple-choice question, asking the participants of their impression of giving ratings like they did in the song creation process. The answers are ranged similarly as some of the multiple-choice questions previously in the survey. There are five alternatives, ranging from "*it was boring*" to "*it was very fun!*". Then, the participant is expected to write a short paragraph about their impression of giving feedback in this manner.

Similar to part 1, prior to part 2 the participant interacts with the prototype. This time however, the participant should generate only one song. Giving the prompted feedback to the song is not necessary, as only the resulting song is needed. At this point the prototype should know what the user likes and dislikes well enough to produce a song that is in accordance with their preferences. Part 2 is split into two sections. In the first section the participant will compare the newly generated song to the last song of the three previously generated. Two questions are then asked regarding this comparison. In the second section, four questions regarding only the newest generated song is asked. The participant is encouraged to listen to the two songs again, if necessary.

The first question, in the first section of part 2, asks the participant which of the two songs was the better one. There are four possible answers: *"the song generated now", "the last song from part 1", "they are of equal quality"* and *"I cannot really tell any difference"*. In the second question, the participant is asked to elaborate of their perception of the difference between the two tracks. They will compare both the songs at a general level, with a focus on the flow of the song and how well each part fit together.

The second section of part 2 starts with a multiple-choice question. The question is: "*How well did the different instrument parts fit together?"*, and there are five available answers. The answers follow the same structure as the other questions with five answers. The options range from "*Not well at all*" to "*very well*". If the participant answers one of the negative options (i.e. *"not very well" or "For the most part not very well"*), they are asked to elaborate on their feelings. The next question also has a follow-up question. The original question is: "*Did the song have a general consistency from start to end?"*, and the follow-up question is *"if you answered no, please elaborate".* Once the participant has answered this question, the survey is completed. The participant is then debriefed and thanked for their help by the test-supervisor. The debriefing consists of a casual conversation of their feelings about the testing process. This part is not recorded, in order to make the test subjects feel more at ease. This concludes the description of the survey.

7 Results and Discussion

The following section of the thesis consists of two parts, beginning with an analysis of the results from the survey, and ending with a discussion of said results. The discussion section is split into two section, one for each research question.

7.1 Results

The following sections detail an analysis of the results gathered from the user testing. There are two primary parts of the survey. The first part contains the analysis of the song generating and whether the prototype manages to produce songs in line with the user's preferences. The focus of the second part is at the feedback system of the prototype and the survey questions regarding that system.

The very first questions of the survey are whether the test subjects play any instruments. As seen in Figure 24. 66% of the test subjects did not play any instruments. Although the answers to this question is not factored into the analysis of the results, it is an interesting observation to keep in mind. Although some of the participants had previous experience with playing or creating music, most of them did not.

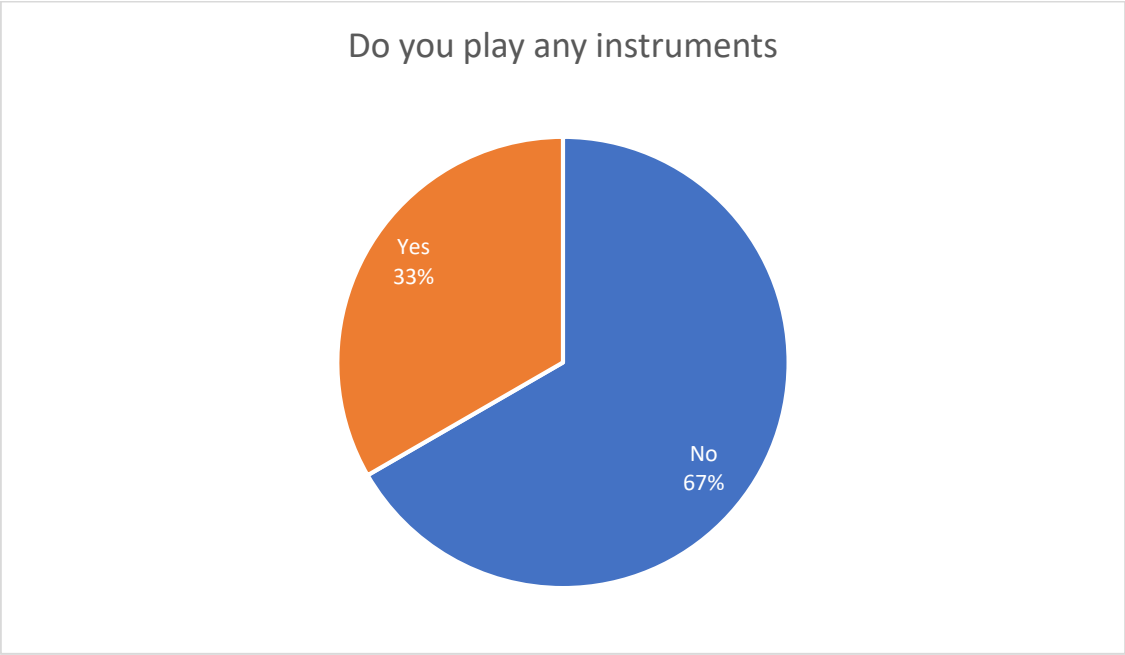

Figure 24 – Distribution of how many test subjects that play any instruments

The first stage of the survey will help to answer whether the songs will get more accurate with each iteration. By comparing what the test subjects answered on the first set of questions, we expect to determine if there is indeed an increase in accuracy across iterations. The first questions are asked right after the user has generated three songs using the prototype. When referring to "the last song" in these first questions, it is meant the last of the three songs.

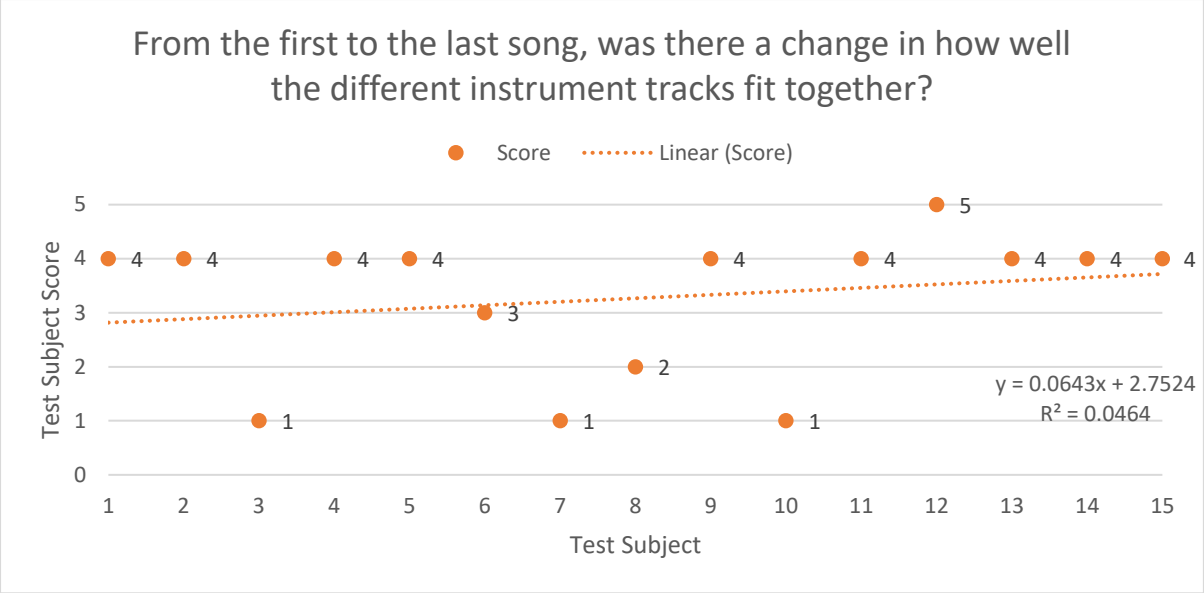

Figure 25 – The test subjects opinions of how well the instrument tracks of the songs fit together.

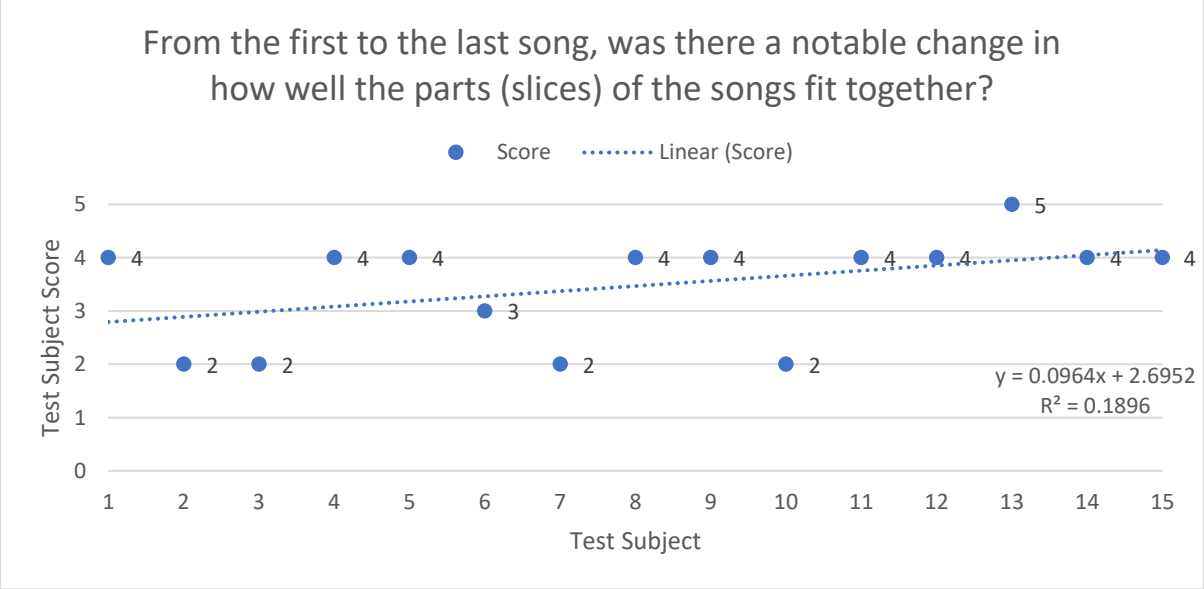

Figure 26 – The test subjects opinions of how well the parts of the songs fit together.

Figure 25 presents an overview of what each test subject thought of the development of how well the instrument tracks fit together in the three tracks they generated. In other words, based on the three tracks, did the instrument tracks become better or worse from the first to the last song.

The horizontal axis of Figure 25 consists of each of the test subject's (TS) answers. The vertical axis in the table is based on the available answers in the query and the answers are represented with numbers instead of text. 1 translates to "*They became worse towards the last song*", 2 to "*the became slightly worse towards the last song*", 3 to "*there was no notable change*", 4 to "*They became slightly better towards the last song*" and 5 to "*they became much better towards the last song*".

The answers from the second question are collected in Figure 26. The question asks the user to compare the different slices from the three generated songs to each other. Similar to the previous

question, this is a multiple-choice question. The vertical numbers have the same meaning as in Figure 25, as described in the previous paragraph.

We expected to see the quality of the songs to get better with each iteration. Therefore, the two graphs include trendlines. They illustrate the progression of the average TS opinion.

As seen in Figure 25, 9 of 15 results answer that the instrument tracks in the songs became slightly better towards the last song. This shows that a little over half (60%) of the participants found that the instruments in the generated songs were getting better. Include the one *"they became much better towards the last song "*(5 in the vertical axis) in that, and the amount goes up to 67% percent of the TS that are happy with the instruments tracks development. Only 4 answers were in the *slightly worse* (2) or *worse* (1) category, which is 26.7% of the answers.

Similar to Figure 25, Figure 26 shows that most people answered the *"They became slightly better towards the last song"* alternative (9 of 15: 60%). No participant answered with the lowest score, but the number of answers below 3 is the same as in Figure 26. In that sense, the answers in this graph (Figure 26) are more clustered than in the other graph (Figure 25). To better visualize the similarities between the answers of the two first questions we have collected the data of both Figure 25 and Figure 26 into one graph: Figure 27.

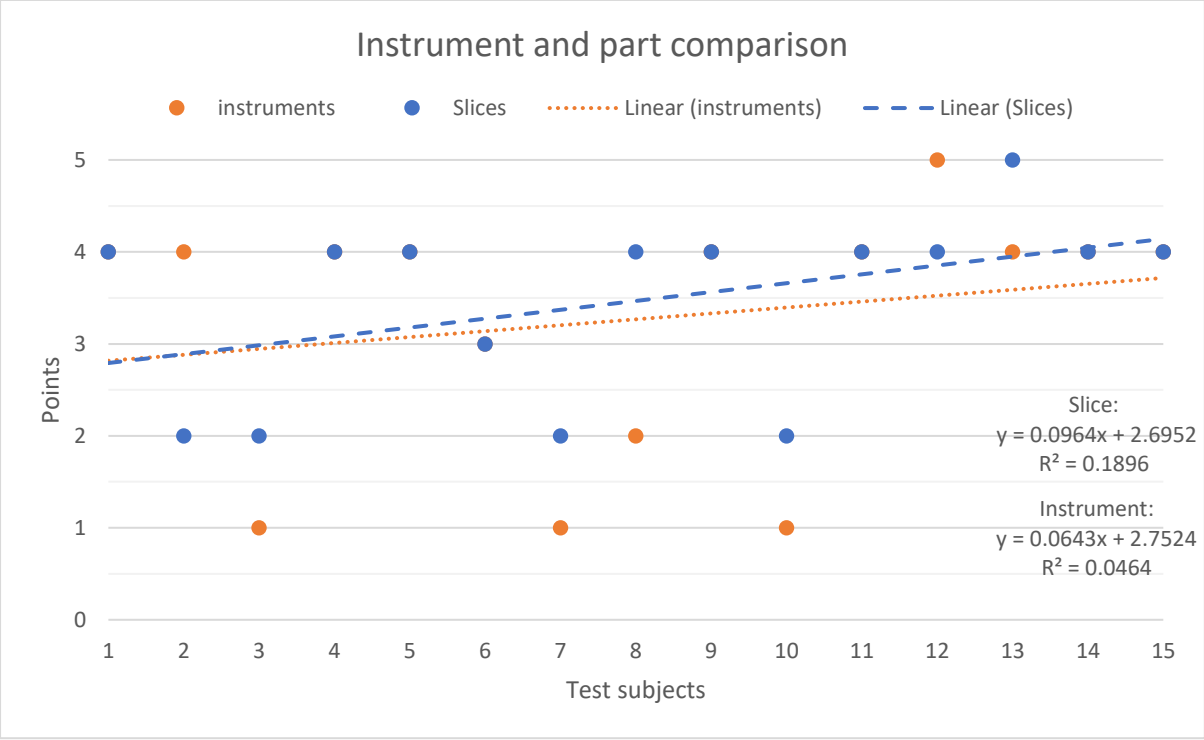

Figure 27 – Comparison between answers in Figure 25 and Figure 26

In Figure 27 the orange series is the data from Figure 25, while the blue series is from Figure 26. Where the two graphs overlap, only one of the series is displayed (for example in the answer of test subject 1). 8 of 15 test subjects gave the same answer on the two questions. This represents 53.3% of the total answers. Only two answers (Ts2 and Ts8) have a difference in category by more than one. Both questions have the same number of positive answers, as highlighted in previous paragraphs.

The test subjects are finding the *slices* more acceptable than the *instrument-tracks*, since the *instruments* have three 1s. Additionally, there are more instances of people liking the *slices* more than the *instruments*. In five instances, the *slices* get a higher score than the *instruments* (Ts3, Ts7, Ts8, Ts9, Ts10 and Ts13). On the other hand, there are only two cases where the *instruments* get a higher score than the *slices* (Ts2 and Ts12). The test subjects might feel that it is easier to evaluate a full range of instruments, rather than one instrument at the time. It is probable that time is a large factor in this, since a *slice* will be shorter than an *instrument*-track. Evaluating a longer piece of music is harder than a shorter piece, since the longer track might contain several elements the test subject both like and dislike.

As mentioned, the graphs include trendlines. We expected that the trendlines were going to reveal whether the slices and instrument tracks was getting better or worse over the course of multiple iterations. The expectation was that the songs would get better across test subjects, and that the trendlines could hint at this. Although both trendlines are gradually moving upwards, the scattered data makes it hard to conclude any real pattern. More test subjects are needed before a clearer and more visible pattern emerges. It is, however, evident that most of the test subjects thought that the generated songs were getting better with each iteration, but it is not enough to show that the overall quality of the generated songs moves in a positive direction.

The test subjects were asked which of the three generated songs they liked the best. Only one answered that they felt there was little difference between the songs. Of the 14 other responses, half said that the third song they generated was their preferred. Song 2 got in turn 5 votes (35.7%). Figure 28 displays the distribution of answers.

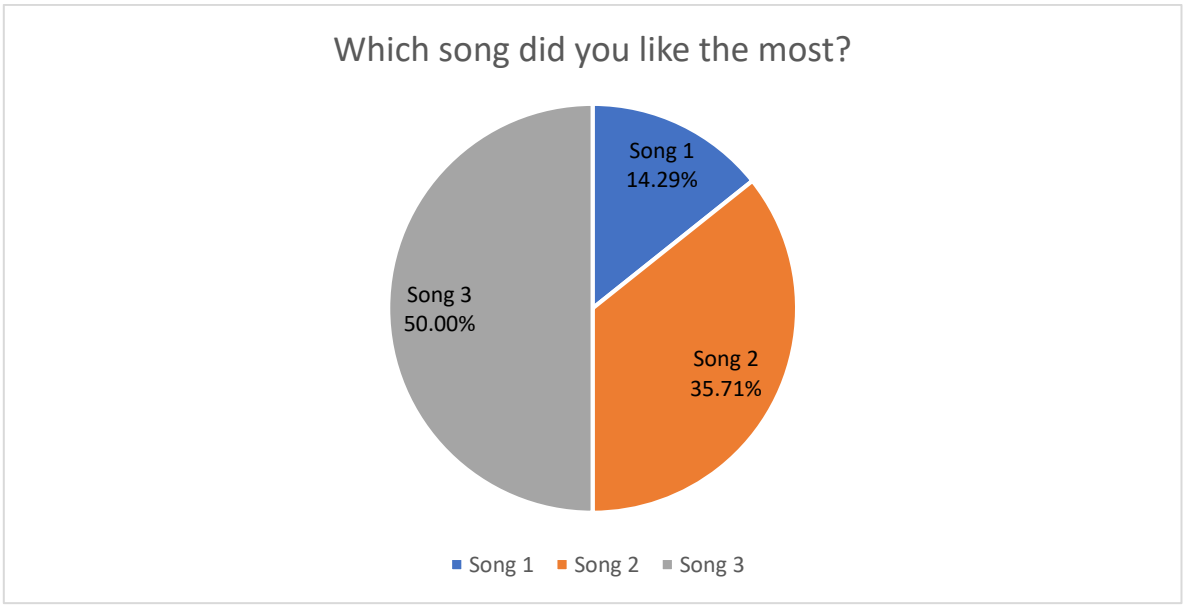

Figure 28 – Overview of what song the user liked the best

The combination of the results presented in Figure 27 and Figure 28 point towards a clear trend that the test subjects find the songs generated from their preferences to be better than the ones who are not. Most of the TS think that the songs are improving with each iteration, and song 2 and onward is considered the better songs. However, it is important to clarify that the songs are not improving by mere chance. As mentioned in Section 4 and Section 5, the algorithm has some built in randomization that it will select randomly between the top three recommended tracks. This random part of the song

generation could lead to some false positive results, especially with the relative small dataset we use here. We therefore asked the users to describe how the best song of the three was better than the others.

Two aspects were present in the descriptions, both related to the previous questions: transition between the parts and unison of instruments within the parts. We have tried to analyse the answers as objectively as possible. However, analysing free-form text is prone to confirmation bias. Therefore, we have tried to be as open with our interpretations as possible.

Of the 14 answers, 5 mentioned directly that the song they liked the best had better transitions between the slices. 3 people partially mentioned better transitions, meaning they described the progression of the song as being better than the other songs. For example: "*I liked the guitar much better in the last song. I* [Sic] *was a bit heavier in general, and then a little more soft* [Sic] *in the chorus."* This mention of the changing guitar is describing the transition between different guitar-parts. The total number of people mentioning or alluding to the transitions in the songs adds up to a little over half (57%) of the answers.

A larger portion of the test subjects mentioned how well the instruments played together. 11 of the answers directly describe how the unison between the different instruments was better than the other songs. Only two persons eluded to the unison of instrument. One of the eluding answers were: *"It appealed more to my taste, the softer parts also felt more "natural" in a weird way.* [...]*"* The last part of the quote is describing how the instrument parts in the softer parts work better in unison.

It is a lot easier for the TS to compare how instruments work together, contrary to how the song transitions to different parts. Additionally, it seems as if people base their preferences on the quality of how well the instruments play together, over the transitions between the parts. This is an interesting observation and further developments, or similar projects, could leverage this. Based on this observation, we argue that creating good parts/slices should be prioritized over creating good transitions between the parts. The test subjects notice bad combinations of instrument tracks much more than bad transitions. However, they certainly notice transitions that are too jarring.

Later in the survey, the test subjects are asked to create another song, and compare that to the last of the three prior ones. By comparing these two songs, we expected to determine whether the songs will get better with each iteration. However, during development we found that the fourth song often suffered from over-specialization or that the recommendation engine could not make a new recommendation. The last part is due to the user having explored every track that other users have rated. When that happens, the item-based algorithm cannot "find" new tracks, and the system returns to the knowledge-based approach. Therefore, we wanted to query the test subjects about their opinion of the difference between the third and fourth tracks. We realized after running the testing that a better alternative to the mentioned question would have been to have the user compare the song they liked best (from the first three) to the newly generated one. This would have been a more interesting observation, since the test subject already had selected their favourite.

We have created two graphs from the question about which of the mentioned two songs the user prefers. The first graph (Figure 29) displays the distribution of answers. As seen in the figure, 60% of the test subjects favour song 4 *("the song generated now*"). We also measured the progression of votes to this question. This is shown in Figure 30.

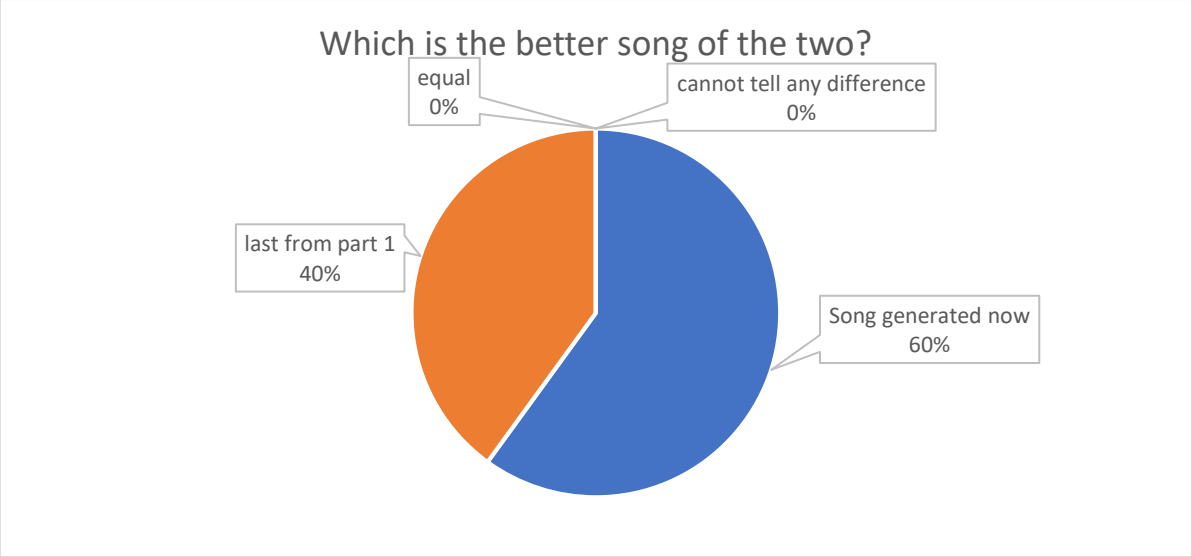

Figure 29 – User comparison of the two latest generated songs

The horizontal values in Figure 30 are an abbreviation for the answers. It was easier to map the values into the graph using numbers instead of text. The numbers translate to the following: 1 is "*Song generated now*", 2 is "*last from part 1*", 3 is "*equal*" and 4 is "*cannot tell any difference*". We also included a trendline in this graph. The trendline is more useful in Figure 30 than in the other graphs, since the points are less scattered. The trendline shows a shift from preferring the last generated song in part 1 towards the latest song. To create a more nuanced dataset of answers, it would be preferable with more data for this question. However, using the data at hand, this strengthens that the generated songs will get more accurate, both as a user used it more and as more users interacted with the prototype.

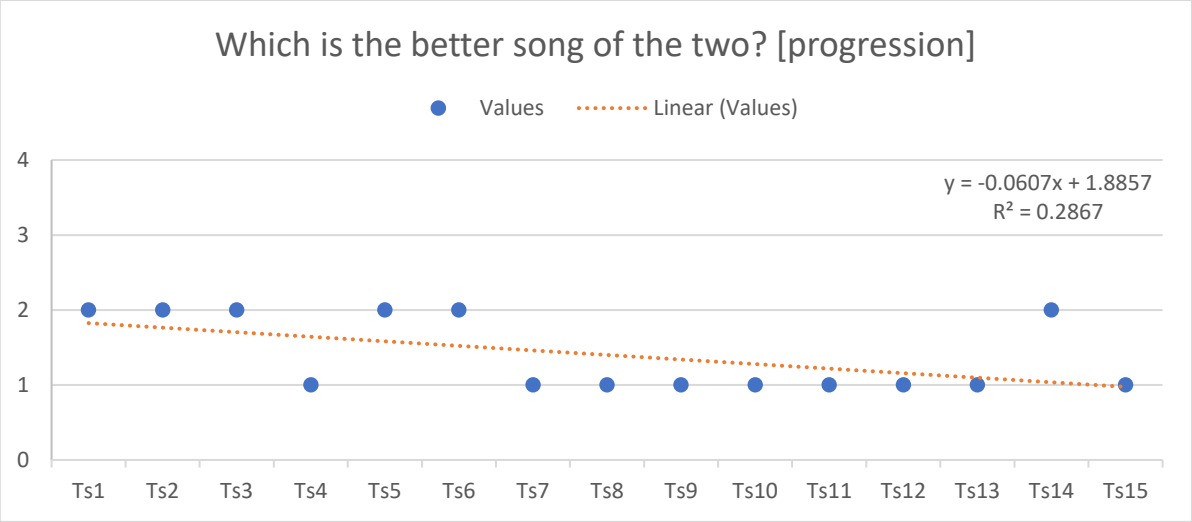

Figure 30 – Trendline of user comparison of the two latest generated songs

We also asked the test subjects if there was anything notably different between the two songs they compared. The answers are in free form text, meaning that the TS could write as much or as little as they would like. Most of the answers are regarding the song they did not like. Transition between the different slices/parts are a common theme for many of the answers. This is an interesting aspect to

compare to the findings earlier. In earlier questions the test subjects seemed to care more about the individual instruments fitting together rather than the transitions between slices. At the stage of the fourth generated song, the transitions seem to be more important. This aligns with the arguments we made earlier, about having good slices prior to having good transitions. The test subjects might feel that the fourth song (and perhaps even the third song) have good slices, so now the transitions are the focus. The answers to next two questions shed some light on this hypothesis.

The user is asked to focus solely on the last song they generated (the fourth song). Two questions, each with a follow-up questions are presented to the test subject. The first question is a multiple answer question, and the question is "*How well did the different instruments fit together?".* The five possible answers and what the test subjects answered is displayed in Figure 31.

As the pie chart in Figure 31 displays, 53% of the TS thought that the different instrument worked "*for the most part well*" together. On the other hand, the 27% that though the instruments for the most part did not fit well together is an interesting observation. Comparing these numbers to the previous results from Figure 27, we see that the number of negative responses has been stable. Circa 30% of the answers from both Figure 27 and Figure 31 have been negative. No test subjects answered "*not well at all*" about how the instruments fit together in the fourth song. From the first three songs, there were 3 answers that said that the instrument parts became "*worse towards the last song*". We therefore argue that the generating process has become slightly more accurate and has eliminated the worst combinations of audio tracks.

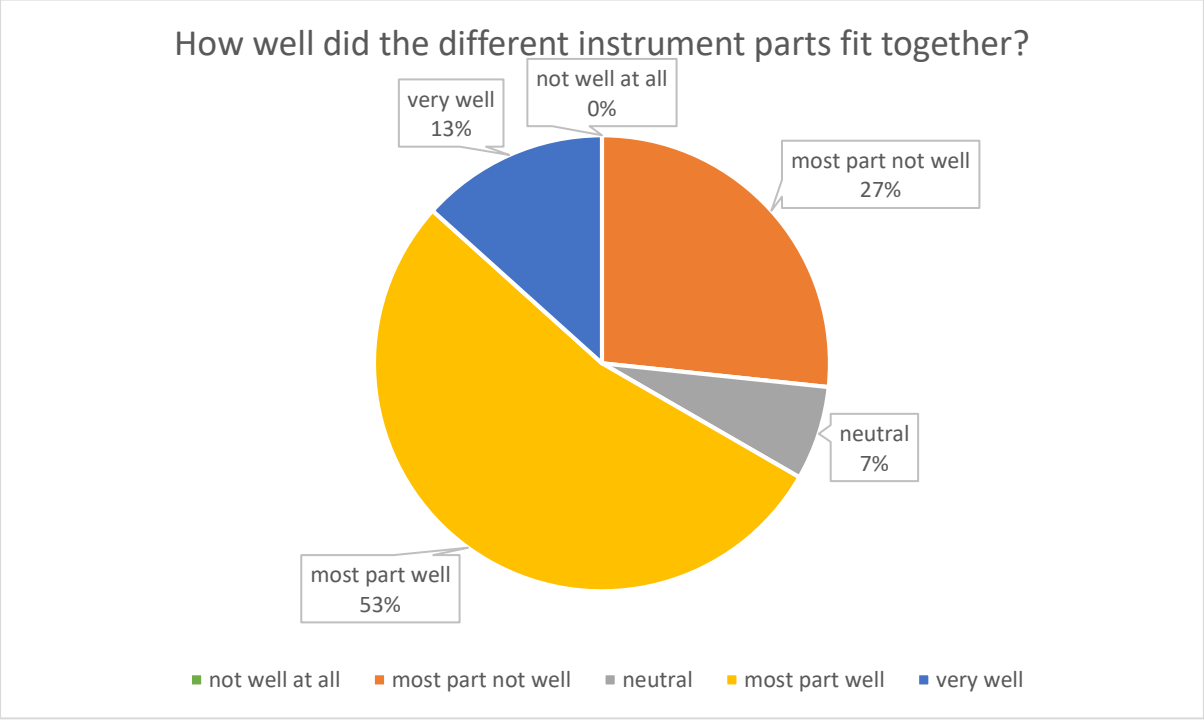

Figure 31 – User feedback to how well the instruments in song 4 fit together

The follow up question is aimed at the TS that answered (in the previous question) that some of the parts did not fit well together. Most of the answers detail specific sections of the songs they did not like, which is to be expected.

The answers often relate that the guitar track is not synchronized with the rest of the instruments in the mentioned section. This is a problem that other generated songs also have revealed. The issue with synchronization is not related to the recommendation engine, but to the audio clips and the song creation process (how the tracks are merged together). Thus, the synchronization problem is not directly related to the research questions, but they impact the results, which is unfortunate. On the other hand, the fact that the complaints are on the synchronization could mean that the recommended tracks are all valid recommendations, and that the errors only arise in the merging of the tracks. However, there is not enough data to sufficiently confirm this hypothesis.

Like in earlier questions, it is necessary to measure the success of the transitions between the different sections in the fourth song generated. This is done through a question about the general consistency of the newest song. Figure 32 displays the results. 4 out of 15 TS answered that the song did not have a general consistency, which makes 27%. The rest, 11 out of 15 TS gave a positive response. The interesting results here is the people that did not think there was a general consistency of the song. These four TS were asked to elaborate on why they though there was a lack of consistency. The four answers are regarding jarring transitions due to guitar effects found in the audio tracks. For example, one of the comments is regarding some reverb[27] present in one of the guitar tracks. The following guitar track does not have reverb, so the effect ends when the new track begins. From this, and the previous arguments regarding merging of tracks, it is evident that the individual audio tracks have a great amount of influence to the song in general. The user will notice faulty merging or audio tracks that are somehow disturbing the flow of the composition.

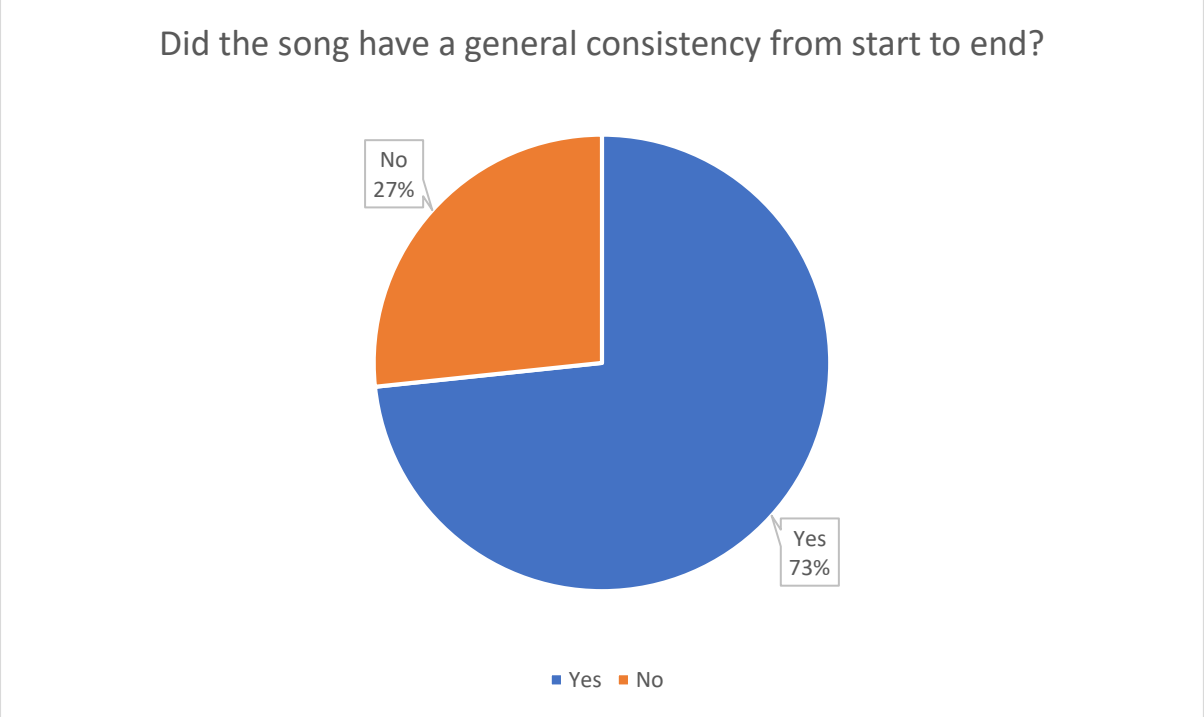

Figure 32 – User's opinion of the consistency of song 4

[27] Reverb is an effect that makes it sound as if the instrument is playing in a large room.

The generated songs are getting increasingly personalized with each iteration. The test subjects report that they think the songs are getting better, which leads to the conclusion that their preferences are taken into account when generating the songs.

The survey contained questions regarding the test subject's opinion on the implemented feedback system. These questions are asked right after the test subject has answered the questions regarding the first three generated songs. Including the questions about the feedback process at this point in the survey, served as a break from the song creation loop. These questions are regarding the 1-to-5 feedback that the prototype asks the user to give, during the song generating process. There are four questions, where one of the questions is a follow-up question.

The first of the feedback questions simply ask the test subject if they though that giving feedback in the prototype was difficult. There are three possible answers, which are displayed together with the distribution of answers in Figure 33. Forty percent of the test subjects though it was difficult to rate the different sections when interacting with the prototype. Similarly, forty percent also found it neither difficult nor easy to rate the items in this manner. Only 20% (3 out of 15) of the test subjects thought it was easy. This highlights the need for change in the feedback system, as the system might be perceived as too difficult or to abstract to use. A follow-up question was asked in order to find out what aspect of the feedback process was difficult. Seven people answered the follow-up question. These seven answers revealed three areas of difficulty.

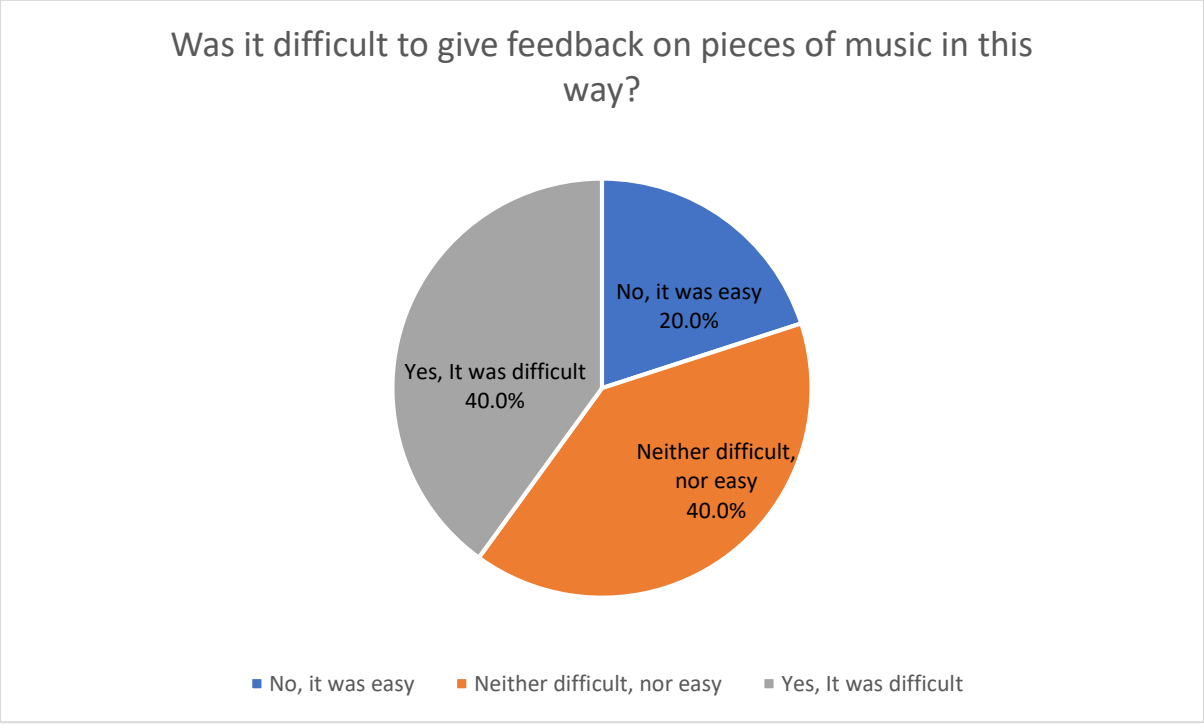

Figure 33 – User response to the difficulty of giving feedback

The first area was that it was difficult to tell the different sections in the songs apart. One of the answers highlighted that a slice would contain multiple small parts that they liked, but equally many parts they did not like. For example, if the drums end to intense, it might lessen the quality of the entire slice. A test subject felt that there was no real way of expressing this, thus making it difficult to give a rating. Two other subjects also expressed this difficulty, however one of them has bad hearing. Differentiating the different instruments and sections with minor changes was difficult for him. He

explains that the instruments blend into each other. Although the design of the prototype did not prioritize accessibility, it is an interesting point to keep in mind for later work.

The second area of difficulty highlighted by the test subjects is the lack of a graphical user interface (GUI). As detailed in the prototype description section (Section 4.1), no real development time was spent on creating a user interface. The test subjects interact with a terminal window. Since sound is the primary aspect of the prototype, little time was spent on creating a GUI. Two of the test subjects noted that a GUI with instructions or guidelines would have made the process easier and more intuitive.

The last area of difficulty was determining how the feedback was being used. One of the test subjects noted that without knowing how the feedback was being used, it was difficult to know what aspects to focus on. The user later told the test supervisor more about this. He said that he was unsure if elements in the recording should be considered when determining if the clip was fitting. For example, some of the clips have different left-right panning[28], which becomes extra noticeable when they are appended after each other. The user felt it was unclear what to prioritize during the rating process. Circumventing this issue with a GUI or with better instructions from the test supervisor is certainly possible.

The prototype is created as a product for entertainment. Therefore, the song process needs to be enjoyable. The only part of the process that is not automated is the feedback section. As a result, some evaluation of the feedback system was necessary. We asked the test subject (TS) to rate how they felt about the feedback process. There were five answer alternatives to this question, and the answers are detailed below, in Figure 34. Alongside this question, a freeform question was also included. In this freeform question the TS could write as detailed they like about the process of rating music in this manner. With these two questions we expected to uncover whether this feedback process was something the user liked to do and if it allows the user to express themselves.

[28] Panning is best illustrated when listen with a headset. For example, audio that is panned completely to the left is only audible in the left ear. Panning ranges from full left, through the centre (both ears) and then to the full right.

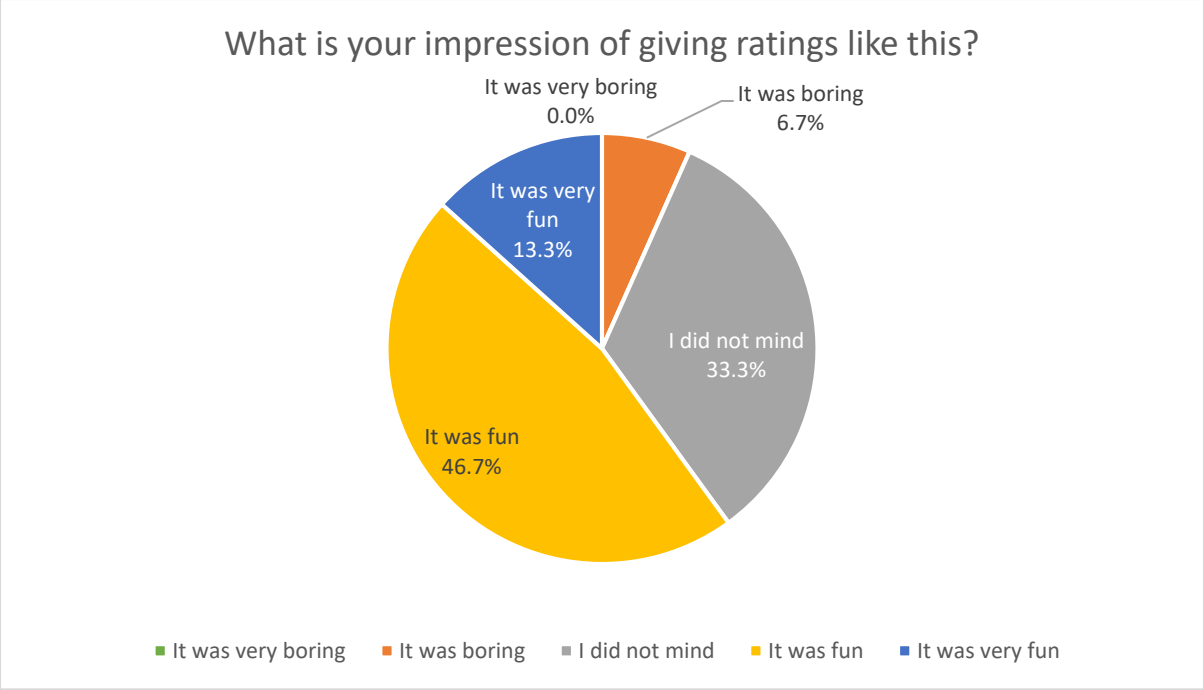

Figure 34 – User impression of giving 1-5 ratings on pieces of music

As seen in Figure 34, almost half (46.7%) of the answers reported that the process of giving ratings was fun. This constitutes 7 out of 15 answers. Additionally, two test subjects thought the process was very fun, making the total positive percent amount to 60%. Most people thought that the feedback process was fun to do, however we still do not know if the TS felt that it allowed them to express themselves. This is particularly relevant since one of the previous questions revealed some uncertainty about how the feedback was being used and what parts of the clips to focus on.

Most of the answers in the freeform question, following the question illustrated in Figure 34, state that the TS found the feedback process enjoyable and interesting. Four of the answers highlight that having to rate both slices and instruments tracks was a good thing, since they felt that it would have been hard to rate an entire song with a single score. One of the four explains that rating the slices was a lot easier, since it was more contained. However, when asked to rate the instrument tracks, he felt that it was harder. The main reason was that the instruments tracks demanded the test subject to listen to "*the big picture*" of the song. The potential for more things to be of varying quality is greater when analysing the horizontal axis[29] of the song, contrary to the vertical axis. This is due to the number of items/tracks in each axis. The vertical axis only contains three tracks that are merged together. The horizontal axis on the other hand can contain a lot more tracks. The number of tracks is dependent on how many sections the song has, since each section will contain one track.

Two of the TS recommend that the user is given more information about what aspects to focus on when rating. If the recoding of a track is of poor quality, but the music in the audio track is good, the entire track might get a bad score because the user does not know what to focus on. Another suggestion from the TS was an alteration to the rating scale. Instead of the one-to-five rating system, the TS suggested that the ratings should be based on feelings instead. "*Maybe a bit hard to give a score to a piece of music. Maybe each score could relate to a feeling instead. e.g 1 = boring, 2 = meh, 3 = interesting etc.*" This is a sign that the user did not really know what to listen for in the song. The

[29] The axis is defined in Section 4.2

implemented scoring system was too vague in terms of what the numbers symbolize. This highlights one of the arguments made earlier about a user interface and a clearer set of instructions.

7.2 DISCUSSION

This section contains the discussion of the success of the research questions. Thus, the section is divided into two sub-sections, one for each research question. In the first section, the success of the first research question is discussed: whether RecOrder is successful in creating personalized content. The following section discuss the second research question: if RecOrder is a successful approach to use recommendation systems to create new content.

7.2.1 Personalized Computer-Generated Music (RQ 1)

Over 70% of the users think that the songs have a general consistency, and that the songs were improving. We therefore assume that the users accept the recommendations. By accept we mean that they feel they are accurate. This is reflected in the answers shown in Figure 27. Although the acceptance of recommendations is not directly indicative of personalized content, the user's perceived accuracy of the recommendation is critical to them feeling that the system "listens" to their feedback. By getting higher satisfaction scores, the user show that they trust the system, allowing it to further map their preferences.

The songs generated after the second song are the ones that are the most liked. This shows that the songs based on the recommendations are more liked. Once the item-based engine starts recommending tracks, the user feels that the songs get better. Additionally, as more songs are generated, the users start to favour the last generated songs. This points towards an increase in song-quality over multiple iterations. This increase in reported quality of songs is also a sign that the recommendations are getting more accurate, and that the prototype creates better content.

Based on the arguments presented in the two paragraphs above, we argue that the prototype created for this thesis is successful in generating personalized music. The personalization aspect is true, since the recommendation system selects the song components, and that the users feel that the prototype selects audio clips based on their preferences. A clear trend towards positive development in the songs highlights the improving accuracy of the recommendation engine. As a result, we consider the prototype a possible solution to generating personalized music.

7.2.2 Using Recommendation Systems to Create New Content (RQ 2)

To answer the second research question, we need to determine if recommendation systems are fitting for content creation. To this end we must consider the limitations recommendation engines impose. These limitations stem from the core functionality of recommendation systems, and directly affect the viability of implementing recommendation systems into content creation systems.

Firstly, the cold-start problem is an important challenge to consider. Recommendation engines cannot make predictions (recommendations) without user preferences. The prototype created for this thesis, or any recommendation system, must learn what the user likes and dislikes before it can recommend content. Therefore, the first content a user creates, in a system using recommendation algorithms as decision-agents will never be based on their preferences. This is one of the major challenges to implementing recommendation systems as content creation engines. Overcoming the cold-start problem is

therefore critical to the success of recommendation algorithms as decision-agents in a content creation system.

The prototype features a hybrid switching engine, which combines the strengths of knowledge-based filtering and collaborative item-based filtering. The prototype manages to create content despite the lack of user preferences, thus successfully circumventing the cold-start problem. As shown by the response of the test subjects, the songs based on people's preferences are superior to the ones that are not. Implementing a more complex knowledge-based filtering (KBF) method can even out the quality between the songs generated from the KBF and the item-based algorithm. However, by creating a more detailed set of rules the system can risk becoming less robust. The stricter rules might impose a smaller selection of tracks, thus making the results less dynamic. The user might feel that the system always creates the same set of songs when using the knowledge-based approach. Having the user answer a small query about their music taste is also a solution to the cold-start problem. The user could then be asked to select their favourite genre or listen to a series of songs and report to the system if they like them or not. By having a small query prior to creating the first song, the system can map out the most foundational of the user's preferences.

The second limitation is that the recommendation system can only recommend content if there is content the user has not rated. In other words, if the user has rated every item in the item-library, RecOrder cannot make any new recommendations using the item-based algorithm. This is less of an issue in settings with a larger selection of available items, but still an aspect that needs careful consideration. The experiment setup for RecOrder was created with this limitation in mind. The number of generated tracks was limited to four, to minimize the probability of the user rating every item. During development of RecOrder, we found that after the fourth generated song the collaborative filtering algorithm was not able to give any valid recommendations. This is due to the low number of audio tracks (37 tracks) being split into three categories (one for each instrument). In each slice of the song there needs to be one track of each category, meaning the possible permutations of tracks is made smaller. It is a possibility that the low number of tracks put a limit to how close RecOrder was able to get to the users' preferences. However, we argue that the system still got close enough for the users to notice their preferences shaping the songs. The users' answers from the evaluation support this claim. We suggest a solution to the problem of rating every item in the library: During the offline phase (explained in Section 2.1.1), the system can delete ratings based on some parameters. For example, deleting tracks that are older than a certain number of days. This way, the system minimizes the probability of a user discovering every item. However, the system will lose the full history of the user-preference. This can have many disadvantages. Most prominently, this increase the probability of the system to regenerate content that the user did not like. The user might remember the tracks and get annoyed that the tracks they know they rated poorly still appear as recommendations. Therefore, it is necessary to have a more nuanced selection of the tracks will be deleted.

Due to the collaborative nature of the implemented algorithm, recommendations can only be made for tracks that other users have rated. This is the third limitation, and it often leads to an uneven distribution of items. The long-tail effect is a common result often seen in systems with recommendation systems (Schafer, et al., 2007)**.** The solution we used in the prototype was successful in stopping the long-tail effect. As explained in Section 4, the algorithm finds the three most recommended tracks, and selects randomly between them. This creates a slightly more distributed selection of tracks. Figure 35 displays how many times each track in the prototype has been rated. The different instrument types are colour-coded: green is the guitar tracks, orange is the bass tracks and blue the drum tracks. Every track in the content-library have received a rating, and the smallest number of ratings for a track is four. There are more guitar tracks than the other two instrument tracks.

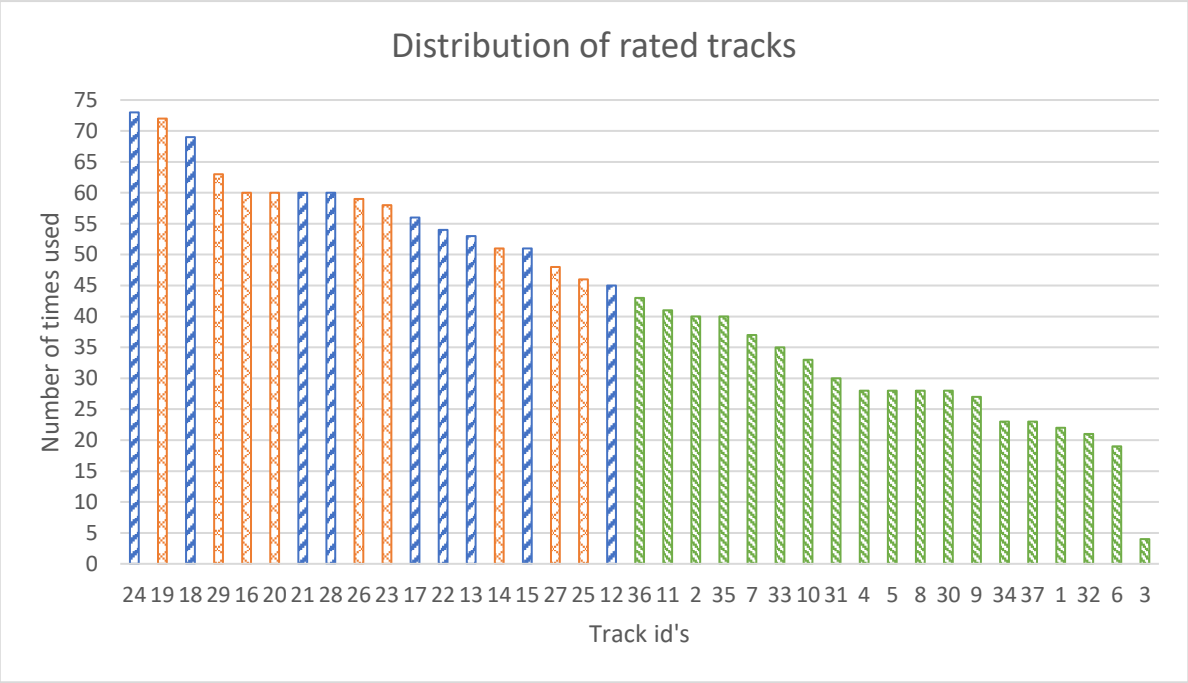

Figure 35 – Distribution of rated tracks. Instruments are colour coded: green is guitar, blue is drums and orange are bass

It appears, due to the uneven distribution of tracks per instrument, that the guitar tracks are the least used tracks. This is because there is a larger set of guitar tracks for the prototype to select from. Adding the total number of times used for each instrument reveals a more even distribution. The guitar tracks are used 550 times, the bass tracks are used 521 and the drums 517 times. This distribution almost equals an even split between the instruments. The guitar track has 35% of the ratings, while the bass tracks have 33% and the drums have 32%. However, we still see a linear regression of how many times each instrument has been used. The distribution of tracks is too linear to be indicative of a long tail effect. It is therefore evident that the prototype prioritizes tracks that are previously rated, but still manages to vary the selection.

A potential issue that needs to be considered is the case of over-specialization. Over-specialization can stem from the combination of limited feedback due to uneven distribution of recommended items. If the system only recommends items from a certain category, the user will only see items from that category. This is an inherit flaw with content-based systems. Additionally, over-specialization can appear when collaborative filtering algorithms only recommend content that are similar to the content the user has already seen. Our prototype has to few items for over-specialization to occur, thus we cannot determine if it appears in the system.

The forth limitation is that recommendation system is dependent on feedback on their recommendations. The feedback is used to make predictions. As more feedback is collected from the users, the accuracy of the recommendations will increase, however, when combining automation and recommendation systems an issue of how to collect feedback arise. Our solution was to slightly compromise the automation by having the user rate different sections of the song. The generating part is still fully automated, but the feedback part is not. A possible solution to turning the feedback process into a fully automatic process is to only use implicit feedback. Several of the test subjects suggested this as an improvement to the feedback process. They suggested that the preferences should be determined from the users listening history. By analysing the actions of the user, one could try to imply if they like

or dislike the generated content. When using implicit ratings, the user does not have to rate any tracks. However, implicit ratings will never be able to discover the nuances of a person's preferences, and this would result in lesser accuracy. Generated content like in the case of this prototype is often complex and consist of multiple parts. The distribution of the feedback across the different parts is a challenge. The user should be able to specify specific parts they did not like. This is hard to achieve using only implicit feedback. In addition, there is a temporal aspect to the process. Implicit feedback might not be able to catch the change in a user's preferences as quick as explicit feedback will. On the other hand, a combination of both implicit and explicit feedback could yield a more nuanced result. We suggest an interactive user interface where the user can give feedback to parts of the song. For example, in the case of generated songs, the user could highlight sections of the song and rate those sections alone. The system would then get specific feedback for smaller sections of the song.

Based on the highlighted limitations and how the prototype deals with those limitations, in addition to the results from RecOrder, we conclude that recommendation systems can be used as an automated content creation engine. Domains (or settings) where the user can give feedback, either explicitly or implicit is necessary for the accuracy of the prediction from the recommendation algorithm. In settings where the user is not able to return feedback to the recommendation system, the recommendation engine will provide weak predictions. This is especially true if the content creation aims to generate personalized content, tailored to the user. In that case, feedback is critical to the success of recommendation algorithms in the role of decision-agents. That is, if recommendation algorithms are to be the system that selects what content to include in the new generated content, there is a need for feedback from the user. Additionally, since recommendation systems take feedback and improve the recommendations based on them, the domain also needs to facilitate an iterative content creation cycle. Settings where there is a possibility of the user discovering every item in the content library, weakens the validity of choosing recommendation systems as the decision-agent. Recommendation systems typically do not recommend the same item more than one time to the same user (given that the user gave a rating to the item). This is a weakness that the implementation needs to consider.

In short; using recommendation systems to create new content is possible, if they are implemented as decision-agents in certain settings. The settings need to facilitate user feedback, have an iterative content creation process or allow the user to create more than one piece of content. For collaborative filtering algorithms, the setting also needs to facilitate multiple users. There should also be a low risk of the user discovering every item in the system.

8 CONCLUSION

8.1 SUMMARY

Two research questions were posed for this thesis: *how can computer generated music be personalized?* and *how can a recommendation system be used to create new content*? We created a prototype to explore these research questions. The prototype aimed to see if recommendation algorithms could be used in unison with content creation, and therefore generate new personalized content. Music was chosen as domain for the prototype. By merging several small audio clips together, the prototype creates new songs. The prototype consists of three major components: the song generator, the recommendation system (containing two recommendation algorithms) and the feedback system.

The song generator creates music consisting of three instrument tracks: drum, bass and guitar. First, the song generator selects a random song structure. The generator then fills in the different sections of the song structure with audio clips. The recommendation system selects the audio clips from the content library. Once the song generator has created every section, the song generator merges the sections together, creating the song.

The recommendation system selects what audio clips the song shall use. The selection is based on the preferences of the user. The recommendation system consists of two recommendation system algorithms and is structured as a hybrid switching system. This means that the system will select the most fitting recommendation algorithm based on what is needed in the song. The primary algorithm is *weighted slope one*, an item-based collaborative filtering algorithm. The secondary algorithm is a knowledge-based algorithm, created specifically for this prototype. The selection of *weighted slope one* was based on a paper (Lemire & Maclachlan, 2005), and the implementation was directly based on the code from a tutorial (Zacharski, 2015). The code for the slope one implementation was slightly modified to fit the project, however, most of the code remained unchanged.

After creation of the song is done, the prototype starts the feedback process. This process queries the user for feedback regarding the generated song. The prototype asks the user to rate different sections of the song, creating a view of the user's preferences. The ratings are limited to a 1 to 5 score. These ratings are leveraged in later iteration of the song creation, to make more accurate recommendations of tracks.

We tested the prototype by running a combination of a survey and prototype interaction. Fifteen test subjects partook in the testing of the prototype. We selected the test subjects at random, with little regard to their musical experience. The test subjects were asked about their musical background, but the selection of test subject was ignorant of their musical background. During the testing the test subjects created multiple songs using the prototype and the survey asked them to compare the different generated songs. Each test subject created a total of four songs with the prototype.

Our findings indicate that the test subjects were overall happy with the generated content. They report an increase in quality as more songs are generated. The generated songs that was best liked by the users were the songs generated last. This points toward an increase in the quality of the generated songs, or at least an increase in recommendation accuracy. Additionally, we saw tendencies towards an improvement in the recommendations across users, although we do not have enough data to fully conclude this. When examining which of the songs the test subjects liked best, across multiple users, we found that the later songs became increasingly more popular as more tested the prototype. We used these arguments to conclude that the prototype successfully includes the users' preferences into the song creation process.

Our implemented feedback system received mixed reviews. Although most of the test subjects thought it was a fun process, some of the users found it challenging. The output and input that the feedback system used was displayed through a terminal window and not every test subject was comfortable interacting with that type of user interface. Some of the test subjects requested the need for a more complete user interface, to make the feedback process easier. Additionally, some reported that they found it challenging (but often fun at the same time) to rate sections of music with numbers.

8.2 FUTURE WORK

We include some suggestions to future work on this project. There are four primary areas of improvement for the prototype.

The first area of improvement is a better user interface. As mentioned, some of the test subjects requested a more user-friendly graphical user interface (GUI). Additionally, by adding a GUI the feedback process could have been made easier, since the user interface might have visualized the rating process better. We therefore suggest developing a more functional GUI.

The second area of improvement is that more audio clips would have been beneficial to the prototype. By including audio clips with different beats per minute and in different keys, a more nuanced set of songs could have been generated. Additionally, adding more genres and clips with different instruments to the selection of clips could have been interesting. A more formalized method of creating the audio tracks would have been beneficial. When creating the audio tracks to that are used in RecOrder, there was no defined method of creation. This resulted in variance in left-right panning, volume and effects (for example reverb for the guitar). This was noted by some of the test subjects. Small differences in the audio quality might impact the resulting song. As mentioned, several of the test subjects reported that several songs suffered from bad transitions, where the previous audio track would be cut off too early or continue too long. Although a better method of merging the tracks together can solve some of these issues, some of the fault is also in the audio tracks themselves. In fact, this will always be the case when creating a library of small elements that are merged together: minor inconsistencies will impact the generated content.

The third area of improvement is regarding the recommendation system algorithms. Comparing the result from different recommendation algorithms, could yield interesting results. Especially in regard to the degree of perceived personalization of the generated music. We suggest not only testing different item-based algorithms, but also to use user-based algorithms and content-based filtering algorithms. Machine learning algorithms could also serve as an alternative to recommendation system. It would be interesting to compare the degree of personalization of the generated music between a system that use machine learning, versus a system that use recommendation system. Additionally, by creating similar applications within different domains, we can achieve a broader sense of the success of the personalization of the generated content. To create a comparable system to the prototype created for this thesis, the domain should have content that it is possible to split into smaller segments and to combine them into new content. In other words, the pieces of content need to be modular and flexible.

Finally, we also suggest creating a more complex song creation process. The process we implemented into the prototype is specifically created for three instruments. It is possible to create a more dynamic system. Our suggestion is to abstract the instrument layers, from guitar, bass and drums to a more general naming convention. For example: rhythm, backing section and lead section. That way, it is

possible to place more instruments into each instrument layer. Another aspect of the song creation process that could have been made better is the selection of clips in the same slice. The users notice bad combinations of audio clips (in a slice) more often than weak transitions between clips. This assertion is based on the number of responses highlighting bad combinations over good combinations. We therefore argued that getting good recommendations for each slice is more important than having good transitions between the slices. We also suggest that additional work into the algorithms that select the structures of the song section. A more complex system can create more dynamic songs. For example, in the start of the song it could be fitting to only have the guitar playing and then introduce the other instruments one by one. This process is also open for a smarter and more flexible decision-aiding system, that can learn what type of song structure the user likes. Either a recommendation system or a machine learning system is fitting for this role.

REFERENCES

Adomavicius, G. & Tuzhilin, A., 2005. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *IEEE Transactions on Knowledge and Data Engineering,* pp. 734-749.

Aggarwal, C. C., 2016. *Recommender Systems - The Textbook.* New York: Springer.

Appen, R. v. & Frei-Hauenschild, M., 2015. AABA, Refrain, Chorus, Bridge, Prechorus - Song Forms and Their Historical Development. *Samples: Online-Publikationen der Gesellschaft für Popularmusikforschung / German Society for Popular Music Studies e. V.*, 10 03.

Bobadilla, J., Ortega, F., Hernando, A. & Jesús, B., 2012. A collaborative filtering approach to mitigate the new user cold start problem. *Knowledge-Based Systems,* pp. 225-238.

Burke, R., 2002. Hybrid Recommender Systems: Survey and Experiments. *User Modeling and User-Adapted Interaction,* November, 12(4), pp. 331-370.

Burke, R., 2007. Hybrid Web Recommender Systems. In: *The Adaptive Web.* Berlin: Springer-Verlag, pp. 377-408.

Casey, K., 2018. *'Earworm melodies with strange aspects' – what happens when AI makes music.* [Online]
Available at: http://robohub.org/earworm-melodies-with-strange-aspects-what-happens-when-ai-makes-music/
[Accessed 05 April 2018].

Celma, Ò., 2008. *Music Recommendation And Discovery In The Long Tail.* s.l.:Springer.

Covach, J., 2005. Form in rock music. In: *Engaging music: Essays in music analysis.* s.l.:Oxford University Press, pp. 65-76.

Feil, S., Kretzer, M., Werder, K. & Maedche, A., 2016. *Using Gamification to Tackle the Cold-Start Problem in Recommender Systems.* San Francisco, ACM, New York, pp. 253-256.

Ferwerda, B., Tkalcic, M. & Schedl, M., 2017. *Personality Traits and Music Genres: What Do People Prefer to Listen to?.* Bratislava, ACM.

Ferwerda, B., Yang, E., Schedl, M. & Tkalcic, M., 2015. *Personality Traits Predict Music Taxonomy Preferences.* Seoul, ACM.

Ghedini, F., Pachet, F. & Roy, P., 2015. Creating Music and Texts with Flow Machines. In: G. a. A. S. Corazza, ed. *Multidisciplinary Contributions to the Science of Creative Thinking (Creativity in the Twenty First Century).* s.l.:Springer.

Hedges, S. A., 1978. Dice Music in the Eighteenth Century. *Music & Letters, Vol 59, No 2,* April.

Herlocker, J. L., 2000. *Understanding and Improving Automated Collaborative Filtering Systems.* s.l.:ResearchGate - University of Minnesota.

Herlocker, J. L., Konstan, J. A. & Riedl, J., 2000. *Explaining Collaborative Filtering Recommendations.* Philadelphia, ACM, pp. 241-250.

Hug, N., 2017. *Surprise, a Python library for recommender systems.* [Online]
Available at: http://surpriselib.com/

Lam, X. N., Vu, T., Le, T. D. & Duong, A. D., 2008. *Addressing cold-start problem in recommendation systems.* Suwon, ACM. New York, pp. 208-211.

Lemire, D. & Maclachlan, A., 2005. Slope One Predictors for Online Rating-Based Collaborative Filtering. *SIAM Data Mining,* pp. 21-23.

Liebman, E., Khandelwal, P., Saar-Tsechansky, M. & Stone, P., 2017. *Designing Better Playlists with Monte Carlo Tree Search.* Texas, s.n.

Lika, B., Kolomvatsos, K. & Hadjiefthymiades, S., 2014. Facing the cold start problem in recommender systems. *Expert Systems with Applications,* pp. 2065-2073.

McNee, S. M., Riedl, J. & Konstan, J. A., 2006. Being Accurate is Not Enough: How Accuracy Metrics have hurt Recommender Systems. *CHI 2006.*

O'Donovan, J. & Smyth, B., 2005. Trust in Recommender Systems. *Adaptive Information Cluster,* pp. 167-174.

Open Source Initiative, 2007. *The Open Source Definition.* [Online]
Available at: https://opensource.org/osd

Oracle Corporation, 2018. *MySQL.* [Online]
Available at: https://www.mysql.com/
[Accessed 29 03 2018].

Papadopoulos, A., Roy, P. & Pachet, F., 2016. *Assisted Lead Sheet Composition using FlowComposer.* Toulouse, Sony CSL Paris.

Purwins, H., 2005. *Profiles of Pitch Classes: Circularity of Relative Pitch and Key – Experiments, Models, Computational Music Analysis, and Perspectives.* Berlin: Technische Universitat Berlin.

Robert, J., 2017. *Pydub.* [Online]
Available at: http://pydub.com/

Ronacher, A., 2018. *Flask Overview.* [Online]
Available at: http://flask.pocoo.org/

Rubens, N., Kaplan, D. & Sugiyama, M., 2011. Chapter 23 - Active Learning in Recommender Systems. In: F. Ricci, L. Rokach, B. Shapira & P. B. Kantor, eds. *Recommender Systems Handbook.* s.l.:Springer Science, pp. 735-769.

Schafer, J. B., Frankowski, D., Herlocker, J. & Sen, S., 2007. Collaborative Filtering Recommender Systems. In: *The Adaptive Web.* Berlin: Springer-Verlag, pp. 291-324.

Schedl, M. & Bauer, C., 2017. *Introducing Global and Regional Mainstreaminess for Improving Personalized Music Recommendation.* Salzburg, s.n., p. 8.

Statistic Brain, 2016. *Statistic Brain.* [Online]
Available at: https://www.statisticbrain.com/youtube-statistics/
[Accessed 12 January 2018].

The Python Software Foundation, 2018. *About Python.* [Online]
Available at: https://www.python.org/about/

W3C, 2016. *HTML & CSS.* [Online]
Available at: https://www.w3.org/standards/webdesign/htmlcss

Yu, D., 2016. *Spelunky.* 1st ed. Los Angeles: Boss Fight Books.

Zacharski, R., 2015. *A Programmer's Guide to Data Mining.* [Online]
Available at: http://guidetodatamining.com/
[Accessed 11 2017].

APPENDIX A: TABLE OF FIGURES

APPENDIX B: GIT REPOSITORY

Link to Github repository for RecOrder: https://github.com/ejo034/RecOrder