Advanced Music Recommendation System

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

Master of Technology

by

Aritra Saha
Roll No.: Y8127125

under the guidance of

Dr. Arnab Bhattacharya

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
May, 2014

Abstract

Dedicated to my parents.

Acknowledgement

Aritra Saha

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recommendation systems is an extensive class of web applications that involve predicting user responses to options. It is a subclass of information filtering system. The main purpose of recommendation systems is to generate plausible options to users for items or products of their interest. These systems build up the user profile usually based on their past search history or ratings and compares it with some reference characteristics.

Recommendation systems are broadly categorized on the basis of how they analyze data sources to develop notions of affinity between the user and the items which can be used to identify well-matched pairs. They are:

- Content-based: The user's profile, created with their preferences and any available history, is compared with various candidate items and the best matching items are recommended.

- Collaborative-based: Large amounts of information on several users' preferences and activities are analyzed to obtain the similarity among users and thus predicting the future interests. Collaborative filtering is capable of recommending complex items without requiring its "understanding" accurately.

- Hybrid: This approach involves both content-based and collaborative-based filtering while recommending.

Recommendation systems are of great importance for the success of e-commerce and the IT industry and are gradually gaining popularity and becoming an active area for research. They enhance user experience by assisting them in finding information and reducing search and navigation time. In addition, recommendation systems increase productivity and creditability of a user. These systems have evolved to fulfill the natural dual need of buyers and sellers by automating the generation of recommendations based on data analysis.

Recommendation systems are gaining popularity in the field of music as well. One of the well known example is *Pandora Internet Radio*[1], which is an automated music recommendation service. Music recommendation systems ask for users ratings, like or dislike for particular artists, songs or albums, and based on these parameters it recommends choices closer to their taste.

1.1 Motivation

Most of the modern recommendation systems we come across use hybrid filtering, i.e., take into consideration a user's choices and cross-reference it against large amounts of user data finding similar interests. Even though this method works fine for some cases, there is a scope for improvement.

Recommending music to a user based on her pre-set choices may not be a success if she is in a totally different *mood*. We can determine the *mood* of a user by analyzing her recent history. The recent history considered to predict the mood is kept rolling forward, so as to accommodate the fact that a user's mood changes with time and music. This mood cross-referenced with content and collaborative filtering tend to give better recommendations.

[1] `http://www.pandora.com/`

1.2 The Problem

The mood of a music enthusiast can be determined by the songs she has heard recently. The mood usually changes gradually over time and can be modeled by the genre of the songs. Mood once modeled can be collaboratively compared and thus help conclude the preference of a song after a set of given songs (which determine the mood) as heard by other "similar" users.

1.3 Contributions of this Thesis

There are two main contributions of this thesis. The first is to determine the rolling current mood of a user based on her recently played tracks. The challenge here is to optimize the monitored rolling time duration and to come up with a method to compare two set of moods. The second contribution is to come up with a set of weights for each of the contributing factor, i.e., mood, user's preferences and collaborative filtering to obtain a confidence score for each recommendation at any given instant.

1.4 Organization of this Thesis

The rest of this thesis is organized as follows. Chapter 2 presents related work done. Chapter 3 discusses the various background work that one must be acquainted with in order to understand the work presented. Chapter 4 discusses the data sources and APIs used to set up a sample infrastructure of the recommendation system. Chapter 5 then discusses in detail the implemented methods and algorithms. Chapter 6 presents a summary of the results that was achieved and also talks about what can be done further.

Chapter 2

Related Work

There is a lot of interesting product and implementation related projects in the field of music recommendation. There are two primary ways to categorize and identify similar music, either by analyzing and mathematically formulating the audio signals or by mining from a plethora of available music related metadata. Some of the well known projects have been introduced below.

2.1 Pandora Radio

Pandora Radio[1], one of the most popular music recommendation and discovery services on the Internet today, bases its recommendations on data from the Music Genome Project (2.1.1) [7]. It uses *musicological analysis* [10] form of recommendation. Pandora has no concept of genre, user connections or ratings. When a user listens to a radio station on Pandora, it uses a pretty radical approach to delivering users personalized selections; having analyzed the musical structures present in the songs one likes, it plays other songs that possess similar musical traits.

An important aspect of Pandora is its feedback system. This allows users to like or dislike a presented song. Pandora makes efficient use of proximity measure algorithm [6] to recommend music from its database that matches users choice. Based on this, Pandora then recommends music and adapts its recommendations to

[1] `http://www.pandora.com/`

match the users taste.

2.1.1 Music Genome Project

The Music Genome Project[2] [3] assigns a vector of up to 400 'genes' (or attributes) to every song. These attributes capture the musical identity of a song and many other significant qualities that are relevant to understand the musical preferences of listeners. These 'genes' correspond to attributes of the track such as *gender of lead vocalist*, *type of background vocals*, *level of distortion*, etc. Each determined gene is given a score in the range of 0 to 5, with intervals of 0.5. Given the vector of one or more songs, a list of other similar songs is constructed using a distance function.

The project employs musical analysts who listen to music and rate songs based on those attributes. These analytics then gets imported into Pandora computer analytics system that is presented to the users for their feedback. Pandora takes that feedback and develops playlist metrics and recommends it to the users.

2.2 Modelling Internet Radio Streams

The radio can provide useful data regarding the popularity of a song and those that are trending. Radio usually plays music as per their listener's requests or based on prediction which will increase their user base. Either way, it is a fair source to determine a song's popularity. Internet Radio are no way behind in this regard, but they also happen to provide very structured information regarding the played songs and possibly the upcoming playlist.

Yahoo! attempts to mine this data obtained from several internet radio stations over a considerable period of time [1]. It can then be used to create all sorts of popularity and/or trends related charts for songs, artists, albums etc. Such kind of information often proves vital while recommending songs. A music enthusiast may like to hear a trending song, even if the songs does not match her preferred genre.

[2] `https://www.pandora.com/about/mgp`

Chapter 3

Background

The algorithms and data structures used in this thesis have been introduced and discussed below.

3.1 Levenshtein Distance

Levenshtein distance [9] or evolutionary distance is a concept from information retrieval. It is one of the most common variants of *edit distance* named after the Soviet Russian computer scientist Vladimir Levenshtein. Edit distance [13] describes the number of edits that has to be made in order to convert one string to another by performing minimum number of operations like insertions, deletions and substitutions. It is the most common measure to expose the dissimilarity between two strings; the greater the distance, the more different the strings are.

This measure allows to assess similarity between strings (or words) and has many applications that include spell-checking, examining correctness of pronunciation and affinities between dialects, analyzing the DNA structure or web mining. The Levenshtein distance can also be computed between two longer strings.

For example, the Levenshtein distance between "contest" and "context" is 1, since just one edit is required to convert one into the other, i.e conte**s**t $\rightarrow$ conte**x**t (substitution of s by x). Similarly, levenshtein distance between "incubate" and "incubus" is 3, since three edits are required to convert one into another and there

is no way to do it with fewer than three edits:

1. incubat**e** → incubat (deletion of e)
2. incuba**t** → incuba**s** (substitution of t by s)
3. incu**b**as → incu**b**us (substitution of a by u)

3.1.1 Definition

Mathematically, the Levenshtein distance between two strings a and b is given by $lev_{a,b}(|a|, |b|)$, where

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0, \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

where $1_{(a_i \neq b_j)}$ is the indicator function equal to 0 when $a_i = b_j$ and 1 otherwise.

3.1.2 How it works

Firstly, with the help of most common approach called dynamic programming, Levenshtein algorithm calculates the minimum number of operations that are required to convert one string to another. A matrix is initialized measuring in the (m,n)-cell the Levenshtein distance between the m-character prefix of one with the n-prefix of the other word. The matrix can be filled from the upper left to the lower right corner. Each horizontal jump corresponds to an insert and each vertical jump corresponds to a deletion. For each of the operation, the cost is normally set to 1. The diagonal jump can have either cost 0 or 1. If the cost is 0, it means that characters match and if it is 1, it means that characters do not match. The cost is always locally minimized by each cell. In this way the number in the lower right corner is the Levenshtein distance between both words. Figure 3.1 is an example that

features the comparison of “meilenstein” and “levenshtein” (where ‘=:’ Match; ‘o : Substitution; ‘+ : Insertion; ‘− :’ Deletion).

The two possible paths through the matrix that produces the least cost solution is described in Figure 3.2 [1].

		m	e	i	l	e	n	s	t	e	i	n
	0	1	2	3	4	5	6	7	8	9	10	11
l	1	**1**	2	3	3	4	5	6	7	8	9	10
e	2	2	**1**	**2**	3	3	4	5	6	7	8	9
v	3	3	2	**2**	**3**	4	4	5	6	7	8	9
e	4	4	3	3	3	**3**	4	5	6	6	7	8
n	5	5	4	4	4	4	**3**	4	5	6	7	7
s	6	6	5	5	5	5	4	**3**	4	5	6	7
h	7	7	6	6	6	6	5	**4**	4	5	6	7
t	8	8	7	7	7	7	6	5	**4**	5	6	7
e	9	9	8	8	8	7	7	6	5	**4**	5	6
i	10	10	9	8	9	8	8	7	6	5	**4**	5
n	11	11	10	9	9	9	8	8	7	6	5	**4**

Figure 3.1: Levenshtein distance example for ‘levenshtein‘ and ‘meilenstein‘

Figure 3.2: Least cost solution to compute Levenshtein distance

3.2 Hungarian Algorithm

The Hungarian method is an algorithm which finds an optimal assignment for a given cost matrix. It is also known as the *Kuhn-Munkres Algorithm*. The original algorithm [8] had a time-complexity of $O(n^4)$; however, it was later modified to obtain a running time complexity of $O(n^3)$.

[1] `http://levenshtein.net/`

3.2.1 Definition

Assuming that numerical costs are available for each of n persons on each of n jobs, the *assignment problem* is the quest for an assignment of persons to jobs so that the sum of the n costs so obtained is as small as possible.

Let $c_{i,j}$ be the cost of assigning the i^{th} resource to the j^{th} task. We define the cost matrix to be the $n \times n$ matrix

$$C = \begin{bmatrix} c_{1,1} & c_{1,2} & \dots & c_{1,n} \\ c_{2,1} & c_{2,2} & \dots & c_{2,n} \\ \vdots & \vdots & & \vdots \\ c_{n,1} & c_{n,2} & \dots & c_{n,n} \end{bmatrix}$$

An assignment is a set of n entry positions in the cost matrix, no two of which lie in the same row or column. The sum of the n entries of an assignment is its cost. An assignment with the smallest possible cost is called an *optimal assignment.*

3.3 Trie

Trie [5] is an ordered multi-way tree data structure that is used to store strings over an alphabet. It is a tree data structure that allows string with similar character prefixes to use the same prefix data and store only the tails as separate data. One character of the string is stored at each level of the tree, with the first character of the string stored at the root. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. Each node contains an array of pointers, one pointer for each character in the alphabet and all the descendants of a node have a common prefix of the string associated with that node. The root is associated with the empty string and values are normally not associated with every node, only with leaves.

A trie can also be used to represent data types that are objects of any type, for example, strings of integers. Various operations such as searching, deletion, insertion

etc. can be performed on a trie. One of the advantages of the trie data structure is that its tree depth depends on the amount of data stored in it. Each element of data is stored at the highest level of the tree that still allows a unique retrieval.

Applications of trie may include sotring a predictive text or an autocomplete dictionary like the one found on a telephone. It is also useful in implementating approximate matcing algorithms including those used in hyphenation and spell checking software.

The insertion to and serching in a *trie* has a time complexity of *O(key_length)*, each. The space complexity however, is of *O(alphabet_size * key_length)*

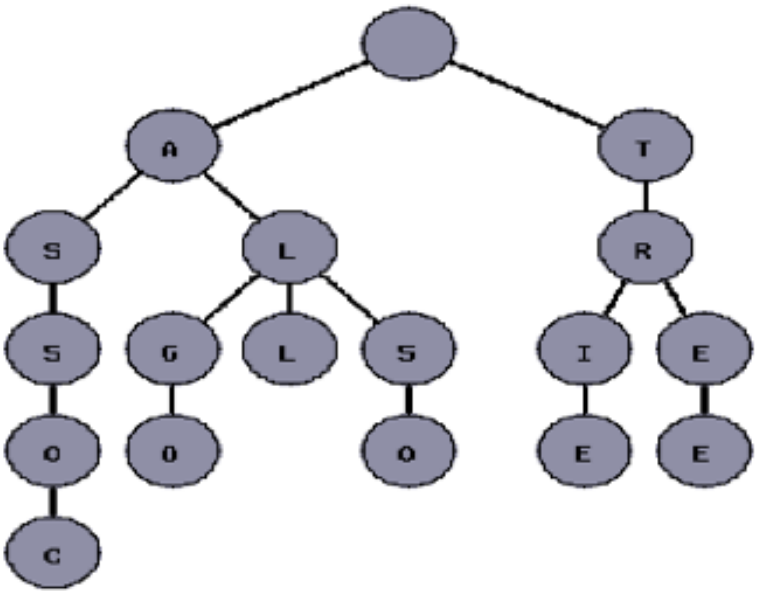

Figure 3.3: Trie with the words “tree”, “trie”, “algo”, “assoc”, “all”, and “also”

3.4 Cosine Similarity

Cosine Similarity [12] measures the similarity between two vectors of an inner product space by measuring the angle cosine between them. A cosine similarity of 1 indicates 0°, thus having the same orientation, however, a similarity of 0 indicates 90°. Even though the value of the cosine ranges from -1 to 1, the cosine similarity is particularly used in the positive space, i.e., $[0, 1]$.

Cosine similarity is commonly used in high-dimensional positive spaces such as

in text mining and information retrieval. The popularity of this similarity owes to the fact that it can evaluate very efficiently for sparse vectors as only the non-zero dimensions need to be considered.

3.4.1 Definition

The cosine of 2 vectors can be obtained by the use of the Euclidean Dot Product:

$$a.b = \|a\|\|b\|cos\theta$$

The cosine similarity for 2 vectors A and B, $cos\theta$ is representated by

$$similarity = cos\theta = \frac{A.B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

Chapter 4

Data Acquisition

Several websites contain information about songs, artists and other relevant media descriptions, some of which also includes their signal analysis. In this section, the Million Song Dataset (Section 4.1) is briefly presented and few of its features are described.

Obtaining a considerable amount of user's song listening history is crucial to be able to recommend songs to other similar users. The publicly available Last.fm API (Section 4.2) used for this purpose has also been introduced and its features briefly described.

4.1 Million Song Dataset

The Million Song Dataset[1] is a freely-available collection of audio features and metadata for a million contemporary popular music tracks. The dataset started as a collaborative project between *LabROSA* and *The Echo Nest*[2]. It includes data contributed by other similar communities doing similar work like *Last.fm*[3], *Musicbrainz*[4], *SecondHandSongs*[5], etc.

[1] `http://labrosa.ee.columbia.edu/millionsong/`
[2] `http://echonest.com/`
[3] `http://last.fm/`
[4] `http://musicbrainz.org/`
[5] `http://secondhandsongs.com`

4.1.1 Data

It contains [2]:

- 280 GB of data
- 1,000,000 songs/files
- 44,745 unique artists
- 7,643 unique terms (*Echo Nest* tags)
- 2,321 unique *Musicbrainz* tags
- 43,943 artists with at least one term
- 2,201,916 asymmetric similarity relationships
- 515,576 dated tracks starting from 1922

The songs/files are stored in HDF5[6] format to be able to efficiently handle variety of audio features. It contains basic meta-data like *title*, *artists*, *year* of composition, *duration*, *IDs* mapped to other popular song databases (*Last.fm*, *Musicbrainz*, *The Echo Nest*), etc. as well as MFCC (4.1.2) features like *beats*, *danceability*, *energy*, *tempo*, *loudness*, and several other features.

4.1.2 Mel-Frequency Cepstrum Coefficients (MFCC)

The extraction and selection of the best parametric representation of acoustic signals are important tasks in the design of any speech recognition system. It significantly affects the recognition performance. A compact representation would be provided by a set of *Mel-Frequency Cepstrum Coefficients* [14], which are the results of a cosine transform of the real logarithm of the short-term energy spectrum expressed on a mel-frequency scale [11]. The MFCCs are proved more efficient [4].

[6] `http://www.hdfgroup.org/HDF5/`

4.2 Last.fm API

Last.fm scrobbles user music listening activity via plugins installed on the user's devices or directly from the music players. It uses these mined data to evaluate popularity of the songs, compare music history to find similar users and finally recommend songs. Last.fm uses collaborative filtering for recommendation. Last.fm has built an extensive database of songs and its related meta data like artists, albums, genres etc. It also provides several statistical data regarding songs, artists and albums.

Last.fm API[7] has been used to fetch user's song history and the genres of the songs obtained from the Million Song Dataset.

[7] `http://last.fm/api`

Chapter 5

Implementation

5.1 Dictionary of Songs

The Trie (Section 3.3) data structure is used to load the Million Song Dataset (Section 4.1). Trie enables faster searching and computation of Levenshtein Distances (Section 3.1) for the titles of the songs. The titles of the songs are used as an index for the *trie*. Each node of the *trie* stores with itself some data which is used frequently, for example, its unique track ID using which all other song informations can be referenced from the MIllion Song Dataset and artist information. We store the artist, as a song is uniquely identified by its title and artist, which is used for other API calls in the subsequent steps. The $999,999$ songs from the dataset occupies $6,510,645$ nodes.

5.2 User History

A set of 2614 users a with considerable amount of playback history has been compiled. These users' histories will serve as the basis for obtaining recommendations from similar users using *collaborative filtering*. The limit on the number of recent tracks taken per user is 200. This gives us a considerable number of tracks to perform analysis upon.

Each track is cross-referenced with those in the MIllion Song Dataset by searching

within the above discussed *trie*. Only those tracks which find a match are kept and the others are discarded. However, the track names scrobbled by Last.fm are often custom written by 3rd party and/or the users themselves, in which case even a slight change in the name will deem it to be discarded. *Levenshtein Distance* has been used to match each of the fetched song. A threshold distance of 2 gives the optimal number of track filtering. Anything lesser would make the filter too strict thus losing too much of valuable user history; whereas keeping it higher will corrupt the data with too many incorrect matches.

A few other information available with the *Million Song Dataset*, for example, *loudness*, *tempo*, *hotness* are also loaded into memory at this step. This is a part of pre-processing of the data and would considerably lower the runtime of generating the recommendations. These features has been described below in Section 5.5.

5.3 Model User Interests

A set of 127 commonly recognized genres (often referred to as track tags) has been compiled. Each song is classified in one or more of these genres. These genres compiled for a given set of tracks will dictate the musical taste of a user. This goes forward to define the user's music preferences. The genres of each song uniquely identifed by its *title* and *artist name* from the user's history is fetched and added to the pool of genres.

The pooled genres for a user is represented using a vector of integers, each representing the total count of the respective genre in her given music history.

5.4 Determine Similar Users

The similarity between users are determined by their music preferences determined by the pool of genres obtained from each of the songs from their respective music histories. The similarity between user X and user Y is calculated on their respective vectors as follows.

$$similarity(X, Y) = \sum_i min(X_i, Y_i) - |X_i - Y_i|$$

Using kNN with $k = 200$, the top k similar users have been chosen. These users' histories will be searched and used to recommend songs for any given user.

5.5 Recommend Songs

5.5.1 Defining Mood

Assuming an average song has a runtime duration of $3minutes$, the mood of a user is determined by the latest 10 songs from her history, i.e., 30 minutes. It is assumed that the mood of a user changes gradually with time and the songs she listens and is considerably constant over the period the 10 songs. Thus, a rolling window of latest songs is used at any point of time.

5.5.2 Collaborative Filtering

The mood of the current user, i.e., the recent 10 songs, is compared to other rolling windows in the histories of the "similar" users. The rolling window in the "similar" user's history can be in any order, but necessarily consecutive. The track listened to by the "similar" user just after the rolling window in consideration, is taken as the recommendation for that window. So, every window has a recommendation with the similarity to the given user's mood as its confidence score. These recommendations for each window for each of the "similar" users are then sorted and the best 5 recommendations are presented to the given user.

The similarity between the windows of the given user and a similar user is determined by the use of the *Hungarian Algorithm* (Section 3.2). The cost matrix used by the algorithm is generated by calculating the similarities (subsection 5.5.3) between each pair of songs, 1 from each of the windows. The algorithm is applied on the thus formed 10 x 10 matrix.

5.5.3 Song Similarity

The following parameters define certain properties of a song.

- Loudness: Overall song loudness is a formula combining segments: local maximum loudness, dynamic range, overall top loudness, and segment rate. The greater the dynamic range, the more influential it is on lowering the overall loudness. The loudness reference is currently -60dB.

- Tempo: The average beats per minute of a song contributes to the tempo. This determines the pace of the song, for example, if the song is a slow one or a fast.

- Hotness: This is a measure of the popularity of the song among users. It plays a significant role in the recommendation as users might want to hear new and top of the chart songs even if it does not flare well according to her mood.

Over and above these parameters, the comparing the artist is crucial, as more often than not, users choose to remain loyal to a certain set of artists. The Million Song Dataset happens to provide us with a vector of genres for each artist. The cosine similarity (Section 3.4) is used to compare the genre vectors of the artists for a given pair of songs.

These 4 parameters contribute to the determination of the similarity between a pair of songs. A weight of 0.25 has been provided to each as they are equally significant towards recommendation.

5.6 Work Flow Summary

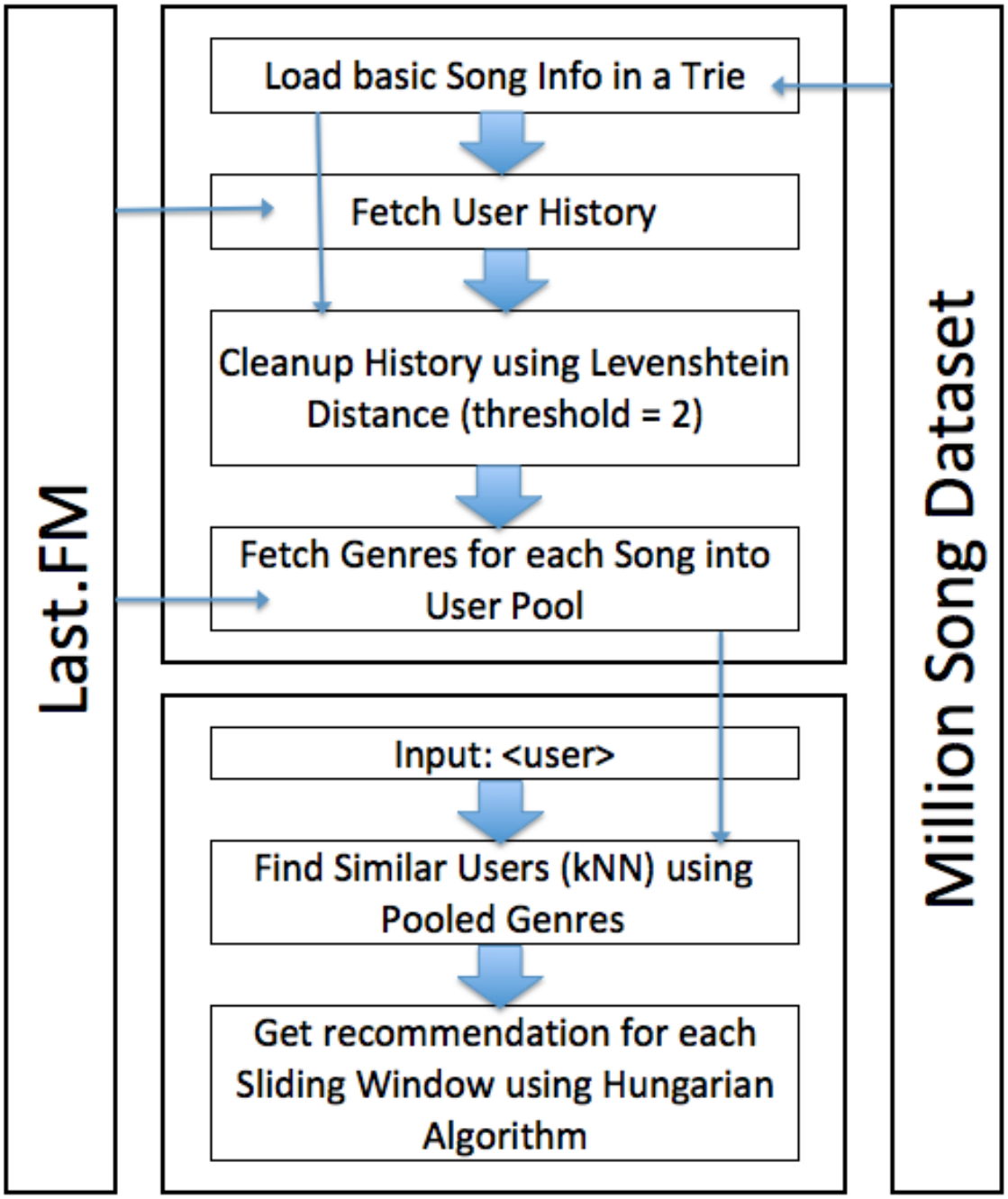

Figure 5.1: Work Flow

Chapter 6

Results and Conclusions

Appendix A

Development Details

A.1 Tools/Libraries

- Eclipse: An Integrated Development Environment helpd speed up the process of coding and its subsequent debugging.
- Java: Being a very common and widely used programming language, finding documentation and 3rd party libraries.
- Maven: A dependency resolver fetching all the required dependencies given the name and version of the required libraries.
- Jedis (Redis): A java implementation of the Redis DB server. Redis is an in-memory key-value pair DB.
- Apache Tomcat: An HTTP servelet implement in and for java execution environments.

A.2 System Requirements

The code and tests have been successfully run on the following configuration. Any system with a configuration equal or higher than this should be able to not only to the job but faster.

- RAM: 16GB; fails to run on 8GB, due to the high amount of in memory data.
- CPU: Intel Core i7, 4th Generation; have used all 8 virtual cores with hyper-threading, with lower CPU, processes would take longer to complete.
- HDD: 280GB dedicated; Million Song Dataset is the only component using considerable persistent memory. Storing user history and codes consume very minimal data storage (in MBs).
- Internet Connection: A suitably fast internet connection to getch user history and a few song information. Slower internet speeds might slow down the entire workflow.

A.3 Optimizations

- Parallelization: Due to most of the code being independent to each other in nature, they can very well be run in parallel as a multi-threaded task. 16 thread has been found to be the optimal for speed on a CPU with 8 virtual cores (2 hyperthreads per core).
- On-Demand: Most songs from the dataset would not be required for recommending, and might consume some valuable runtime. The songs are loaded in the memory at the first missed access. This not only ensures faster runtime but also a lower comsumption of physical memory.
- Load Minimal Data Efficiently: Only song data like *loundess*, *tempo* and *hotness* that is required for recommending is loaded into memory. This data is stored in a serialized JSON format. This lightens the overhead of high-level data structures.
- Prevent File Access: Each song's information in the million song dataset is stored away in a separate file for each. This heightens the overhead of several files being opend and closed at runtime. Also, there is a limit imposed by the

OS to the number of files that can remain open at any given point of time. Thus, only the required information has been extracted, compressed into a serialized format and loaded in memory on an on-demand basis.

A.4 Complications

- Last.FM API: The API is not very robust when multiple calls are made in a short span of time. Several of the calls tend to get failed resulting in the obtained history being corrupted. This restricts the use of multi-threading to fetch user data, which would have improved the runtime significantly.

A.5 Improvements

- Distributed Systems: Using over an NFS could not only help parallel processing significantly without the overhead of context switching in threads, but also could share the dataset on a single drive. It could also load balance the recommendation requests. Also, the issue of making parallel call to Last.FM would be solved thus acheiving a faster pre-processing.

Bibliography

[1] Natalie Aizenberg, Yehuda Koren, and Oren Somekh. Build your own music recommender by modeling internet radio streams. In *Proceedings of the 21st international conference on World Wide Web*, pages 1–10. ACM, 2012.

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

[3] Michael Castelluccio. The music genome project. *Strategic Finance*, 88(6):57–58, 2006.

[4] Steven Davis and Paul Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(4):357–366, 1980.

[5] Ulrich Germann, Eric Joanis, and Samuel Larkin. Tightly packed tries: How to fit large models into memory, and make them load fast, too. In *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, pages 31–39. Association for Computational Linguistics, 2009.

[6] Michael Howe. Pandoras music recommender.

[7] Joyce John. Pandora and the music genome project. *Scientific Computing*, 23(10):40–41, 2006.

[8] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[9] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[10] Terence Magno and Carl Sable. A comparison of signal based music recommendation to genre labels, collaborative filtering, musicological analysis, human recommendation and random baseline. In *ISMIR*, pages 161–166, 2008.

[11] LCW Pols. Spectral analysis and identification of dutch vowels in monosyllabic words [dissertation]. *Free University, Amsterdam, The Netherlands*, 1966.

[12] Amit Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[13] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[14] Fang Zheng, Guoliang Zhang, and Zhanjiang Song. Comparison of different implementations of mfcc. *Journal of Computer Science and Technology*, 16(6):582–589, 2001.