CITY UNIVERSITY LONDON

MSC IN DATA SCIENCE

PROJECT REPORT 2019 / 20

FINDING SIMILAR SONGS IN A LARGE MUSIC COLLECTION BASED ON THEIR CHORD CONTENT

ELENA LESTINI

SUPERVISED BY DR JOHAN PAUWELS

NOVEMBER 2020

Declaration: By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the coursework instructions and any other relevant programme and module documentation. In submitting this work I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the Programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct.

Signed: Elena Lestini

Abstract

Digitalisation, data storage and internet technologies led to a drastic change and modernisation of the music industry. The advent of online music platforms and the fast growth of digital music collections has given the opportunity to users to be exposed to a much wider range of music and artists than in the past. Music information retrieval technology plays a fundamental role in enabling easy searchability and browsability of large music collections.

This project implemented a procedure based on Locality Sensitive Hashing banding technique via MinHashing to find similar pairs and triplets of songs from a large music collection using their chords as main features. The MinHash technique was used to bridge the Jaccard similarity to the Locality Sensitive Hashing to validate the similarity findings and narrow down the similarity search to the pairs of songs that are more likely to be similar ("candidate pairs"), instead of comparing all the possible pairs combinations within a dataset.

Keywords: Big Data, Music Information Retrieval, Jaccard similarity, MinHash similarity, Locality Sensitive Hashing

Acknowledgments

I would like to thank Dr Johan Pauwels for giving me the opportunity to work on such an interesting project and for the continuous mentorship and discussion.

Table of Contents

Chapter 1. Introduction

1.1 Musical Information Retrieval (MIR) in the digital era.

Digitalisation and the availability of supercomputers, data storage and internet technologies have brought radical changes to the music industry. The traditional music industry was based on a (label-centred) business model that relied on talent-scouts to find and sign the next big artist, promoted the musical products to create hit songs through radio and TV channels and arranged the printing and distribution to the retailers of the final physical product (e.g., cassettes, CDs, LPs) [1].

This traditional model has been disrupted by the new digital model, based on online music distribution. As a result, there has been a sharp decrease in the production and distribution of physical media such as CDs and LPs and a corresponding rapid growth of online suppliers of music contents. Online music streaming (OMS) is nowadays one of the most popular form of digital information accessed daily by hundreds of millions of users worldwide. The shift of music products from physical media (such as CDs or LPs) to digital formats has resulted in a vast number of online services and platforms easily accessible to everybody. In 2019 the revenues from OMS has grown by 20% and it accounted for ca. 80% of the overall revenues of the music industry (and digital downloads accounted for ca. 8%), while physical products such as CDs accounted for 10% of those revenues [2].

Nowadays, music is generally available to the public online, due to the easy access to online musical digital platforms. With the advent of the digital era, the business model in the music industry has shifted from a label- to a user-centred paradigm. Accessibility and convenience has become a priority for the music industry, whilst the importance of sound quality has decreased. Customers can now avail of a wider range of music than ever and can stream music 24/7 on their smartphones, other portable devices, digital radios and smart-TVs [3]. Moreover, social network technology has enabled the expansion of the music community to a global level where music customers/users/fans can, for example, share and rate music. The exponential growth of data has helped personalise customer experience, and Big Data analytics is becoming essential for the success and development of the music industry, as it allows on the one hand to reach, and manage the customer relationship with, vast audiences of very diverse and active users, on the other hand to detect inactive subscribers and predict and reduce subscribers/customers churn rate [1, 4].

Digital music collections comprise of tens of millions of songs or other musical pieces, for example Spotify can account for over fifty million pieces, similarly to rival OMS platforms such as Apple, Amazon Music and Deezer [5]. Due to the massive music collections and the high expectations from

the users in terms of easy searchability and browsability, music information retrieval (MIR) has become of paramount importance in the music industry. MIR is a very broad and interdisciplinary field of research that covers many different areas, such as computer science, engineering, mathematics, physics, psychology, biology, psychoacoustics, music, musicology, music performance and history [6]. MIR is relevant to a diverse range of users, including platform subscribers, industry bodies and musicians, authors and other professionals (e.g., patent attorneys). The most common MIR techniques are based on metadata, such as author and title of a music record. More advanced MIR methods use instead content-based descriptions, which is what happens when for example an unlabelled piece of music is played to an online music recognition platform (e.g., Shazam) through a portable device (e.g., a smartphone). Behind the hood, this method relies on an algorithm which allows to link the audio signal to metadata such as author and title. Once the metadata is retrieved, it is sent back to the users within milliseconds [6, 7, 8, 9]. The most advanced MIR methods can also rely on symbolic information such as chords, rhythm or lyrics [6].

MIR techniques can be used to search for information that ranges from a specific piece of music to a trend in a collection or music similarities between different pieces. MIR tasks are classified based on the level of specificity of the query. For example, using metadata (text) to find music that belongs to a certain music genre (such as Jazz or Rock) or that has a certain "mood" (e.g., joyous, energetic, melancholic, meditative or calming) can be considered a low-specificity MIR task. The use of an input signal to address a query can be considered a mid- to high-specificity MIR task. Examples of high-specificity MIR tasks are those based on the use of an input signal to address queries around melody, performance alignment, audio fingerprinting or music similarity (e.g., in the context of plagiarism detection and copyright monitoring) [6]. Some MRI tasks are so important to the music/streaming industry that companies such as Pandora, rdio, Adobe, Shazam and YouTube have copyrighted them [10].

Advanced MIR tasks often follow a common workflow: in the first step, the input signal (e.g., audio) is processed. For example, an input audio signal is transformed from stereo to mono and segmented into frames of milliseconds. In the next step, several features are extracted from each audio frame and mapped to a feature space (usually a highly dimensional space). The type of features is usually selected on the basis of the type of query or problem to be addressed (e.g., pitch-based features for a transcription query; timbre-based features for a genre query). Finally, once the features are available, machine learning is applied to predict, for example, the mood of a musical piece. In this case, supervised machine learning (e.g., supervised support vector machine) can be useful.

1.2 Big Data in MIR

As far as the specific machine learning algorithm used in each MIR task is concerned, the focus is on the type of data that needs to be used for the machine learning process.

Some MIR tasks intrinsically need large music collections, whist other MIR tasks can be useful for single songs (such as generating a music sheet for a song).

For example, simple text-based MIR applications operate on large music collections by comparing and matching text-based queries by the user with songs metadata (e.g., retrieving a song by its title or/and author).

More advanced MIR applications such as content-based audio fingerprinting technologies also work on large-scale collections. Audio fingerprinting is an audio-based short digital signature that preserves as much as possible the relevant content of an original audio file [10].

A "brute force" approach would be to compare the features of a new incoming audio signal against the feature vector that identifies all songs in a database by a combination of clustering and nearest neighbour search. In the nearest neighbour task, given several training points (feature vectors in a database) and a new incoming point (test point) the aim is to find which training sample is the closest to the test sample (new point). To complete this task, the distance of each training point to the test point should be evaluated and the smallest distance(s) between training and test points found. This approach is slow and therefore not computationally feasible. For this reason, MIR uses an algorithm known as Locality Sensitive Hashing (LSH) that allows to approximate the nearest neighbour task by reducing the complexity of such task in the size of a training set from linear (Big-O notation: O(N); where N is the size of the training set) to sublinear (e.g. O(log N)) [11]. The aim is to find similar characteristics within songs or music records by approximate nearest neighbours [7-10, 12].

In LSH, given two points in a feature space, if these are close to one another (therefore similar in some way), then it is statistically likely that they have the same or similar hashes. On the contrary, if the points are apart, their hashes will be likely to be different. An LSH table, created using a defined number of buckets, allows to find those points that are similar, as their corresponding hashes are likely to collide in the same bucket, whilst those that are different should not. The LSH algorithm reduces the dimensionality of the dataset and maintains the local relations among data, contrary to cryptographic secure hashing where small perturbations in the data points lead to dramatic changes in their hashes. This way, LSH can be used successfully in MIR and in the multimedia industry, allowing for distortion or perturbation of the input audio signal. This has a wide range of applications, such as audio fingerprinting or detection of cover songs, genre, mood or plagiarism or recommendations of music pieces to the users based on their preferences. For example, LSH allows to detect similarity between

different versions of the same song or musical record (i.e., cover song detection), which would be impossible by cryptographic security hashing [10]. Among the possible applications of LSH, fingerprinting is designed to find exact song matches, and similar songs have different fingerprints, whereas some other LSH applications require finding similar songs. Our focus is on the latter types of LSH applications, i.e. on the use of LSH applications to find similarities between songs.

1.3 The aim and approach of this project

This work aimed to find similar music pieces in a large collection of songs using scalable similarity techniques for massive datasets [13, 14]. The use of such techniques should allow for a fast and a non-memory intensive assessment of songs similarity based on musically meaningful representations, such as their chords and other chord-related features [15], as opposed to arbitrary fingerprints.

The original music pieces are available on Jamendo.com, a large online music catalogue comprising half a million free songs from 40,000 artists around the world [16,17]. The music records analysed in this project are available from Dr Johan Pauwels' personal MongoDB database (ca. a hundred thousand records) and are represented by symbolic information based on music chord content and derived chord-related features (e.g., chord ratios defined within a certain confidence interval) [15]. The information on the songs is retrievable as JSON files where song features are organised in a list of nested dictionaries which include song chords, chord ratio, chord sequence, distinct chords number, duration of the song (in seconds) and a measure of confidence for the quality of chords sequence.

The project comprises of three main sections. Firstly, the Jaccard similarity and the weighted Jaccard similarity for song pairs and triplets has been evaluated by using chord-ratio derived features stored in a large characteristic matrix. In this case, comparison of every single pair of songs by a "brute force" approach is not computationally feasible for massive datasets, due to the long time required for execution and the large memory space needed, leading to poor performance as described by the big O notation (for N documents, N choose 2 complexity or combinations). The scalability of the process has been assessed by monitoring the computational time and memory for an increasing number of songs. As expected, due to a high number of inputs, the "brute force" approach adopted in section 1 was inefficient, and effective indexing and filtering techniques have been necessary in order to overcome the scalability problem.

For such reason, in Section 2 the large characteristic matrix has been hashed to a smaller signature matrix by using the MinHashing technique. Such technique allows to preserve the similarity characteristics of the main sets in the large characteristic matrix. Song similarity has been evaluated using the MinHash signatures of the songs and the results compared to those obtained from Jaccard similarity in order to assess whether song similarity can be successfully approximated for compressed

data. Time and memory have also been evaluated and compared. Finally, Locality Sensitive Hashing (LSH) has been implemented using different hashing algorithms in order to reduce the search to only potentially useful pairs/triplets (so-called "candidate pairs") [13]. The results (accuracy, precision and recall and computing time) obtained for a different number of documents using the MinHash similarity as metric have been evaluated and compared to the results obtained in section 1.

In conclusion, the main questions addressed in this project are the following: can data mining similarity techniques and distance measurements allow to find similar songs in a scalable manner? Do we need to compare all the documents available in a catalogue in a pair-wise fashion or can we just focus on those that are most likely to be similar, thus making the process computationally efficient, and avoiding the bottleneck of scaling?

1.4 Big data with Pyspark DataFrames

The exponential growth of structured, semi-structured and unstructured data (Big Data) has led to the development of dedicated software packages. Among all the available packages, Spark is the most popular as it overcomes the limitations of traditional Big Data data processing frameworks such as MapReduce. Indeed, "Apache Spark has become one of the key big data distributed processing frameworks in the world" [18]. Spark has several core abstractions: Resilient Distributed Datasets (RDDs), datasets, and DataFrames, all of which represent a distributed collection of data. An RDD represents an immutable fault tolerant collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are at the core level of Spark, they are low level in nature and strongly typed. RDDs can handle unstructured data, as there is no need to set up a schema in order to load/process the data.

Datasets were introduced with Spark 1.6 in 2016 and provide a more functional programming interface to work with structured data. Datasets APIs are available only in Scala and Java. DataFrames is a dataset organised as a table, allowing for a much easier interface to interact with the data. DataFrames can handle a wide range of data such as structured data files, tables in Hive, external databases or existing RDDs and the API is available also in Python and R. Although Spark architecture is built on core RDDs, operations with DataFrames are generally optimised and faster than RDDs [19]. RDDs, Datasets and DataFrames data objects once created are immutable, thus ensuring that data can be easily distributed and shared among processes and nodes in a cluster. RDDs can be created at any point in time taking advantage of caching, sharing and replication. Due to the immutable nature of data objects in Spark (i.e., RDDs, Datasets, DataFrames), every operation performed on them results in the creation of new data objects (RDDs, Dataset and DataFrames), giving the advantage of fault tolerance. Another important feature of Apache Spark is Spark lazy evaluation. Data operators are lazily evaluated instead of eagerly evaluated and that is one of the main reasons why Spark is often faster than other big data

processing engines such as Hadoop and MapReduce. Lazy evaluation in Spark means that the actual execution will not start until an action is triggered, Spark will perform all the transformations that it has collected up to that point. Spark lazy evaluation plays a key role in saving computational overheads.

In this project Pyspark DataFrames has been used to work with Big Data, as operations are faster and optimised when compared to Spark RDDs core data abstraction or other big data processing engines such as Hadoop and MapReduce.

Chapter 2. Similarity search: the theory

2.1 Similarity search on large scale datasets

Similarity techniques for massive collections of data have important applications in a wide range of fields, such as document clustering [20, 21, 22], classification, ranking [23], recommender systems [24, 25], plagiarism detection [26, 27, 28], improved web search engines [29].

Nowadays, due to the large volume of data (both structured and unstructured) with high dimensional features, the reduction in the computational time and memory required for similarity queries is of paramount importance, in both industry and academia. Some of the most popular techniques adopted to solve this problem are Principle Component Analysis (PCA) and neural networks [30], hashing and filtering methods (e.g. MinHashing [13, 31] and Locality Sensitive Hashing (LSH) [13, 32]), as well as parallel algorithms run on a cluster of computers with a MapReduce framework [33, 34].

The aim of this project was to find similar songs pairs and triplets from a large music catalogue by using the Jaccard coefficient as similarity metrics and to scale up the process by using the MinHashing and LSH banding techniques [13]. It is important to note that the Jaccard similarity works efficiently (time- and memory-wise) for small datasets.

The Jaccard similarity coefficient works for sets of data, but not for textual documents. Therefore, when it comes to comparing two textual documents a pre-processing step known as "k-shingling" is usually necessary. The k-shingling technique allows to create (for each document) a set of unique elements, each being a string of continuous characters with length k (i.e., k-shingle).

Each document in a dataset can be represented by a set of unique k-shingles. A k-shingle is a sequence of k- tokens, (e.g., strings of length k, k-words, or else depending on the applications). If there are several documents (in this case songs) and their relationship is to be investigated with regard to the sets of k-shingles, a '*characteristic matrix'* can be created. The characteristic matrix comprises of one row per document and one column per k-shingle. As a characteristic matrix represents all the possible combinations of sets of k-shingles across all documents in a dataset, its size (number of columns x number of rows) is large. The similarity search for a large collection of documents becomes computationally not efficient and for such reason a more compact representation of the characteristic matrix is necessary.

However, "k-shingling" was not implemented in this work, as distinct chords were used directly as 1-shingles. Indeed, chord ratio was chosen as the main feature to investigate songs pairs similarity, so the work focused on finding such general similarity in terms of chords used anywhere in a song.

It is important to mention that long shingles are more suitable for finding song duplicates and near duplicates. But this was not the aim of this work.

For large datasets, MinHashing allows to scale up the process by compressing the characteristic matrix to a smaller matrix (i.e., signature matrix) whilst preserving the nature and characteristics of the main sets in the characteristic matrix. The similarity search via MinHash technique provides an estimated value of the Jaccard similarity, whilst LSH narrows down the similarity search to pairs for a given similarity threshold.

Figure 1 shows a pipeline comprising of three main steps: *Shingling, MinHashing* and *LSH*. A document is *shingled* (step 1) to a set of strings of length k (k is an integer; k-shingles). For pairwise comparison, the documents are represented by a sparse (M x N) Boolean matrix (*characteristic matrix*) where M is the Universal set of all shingles and N is the set of all documents. The Jaccard similarity is evaluated from the *characteristic matrix* for all possible pairs of documents present in a collection. In order to scale up the process MinHashing and LSH techniques are employed (step 2 and 3, respectively).

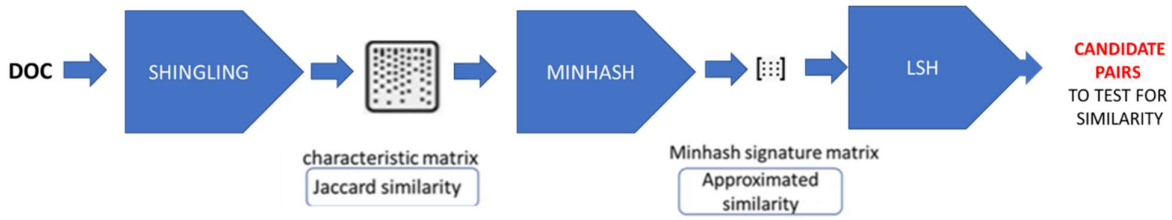


Figure 1. Pipeline for similarity search in Big Data.

The process can be summarised as follows:

- *The Shingling technique* converts a text document into a set of strings of length k (k-shingles, k-grams). Therefore, a document can be represented by a set of k-shingles.[13]

- The *Jaccard similarity* (or Jaccard index) is a metric used to measure the similarity of documents (application: near-duplicate pages, mirror pages), by assessment of the relative intersection. For textual documents the similarity is measured by finding the intersection of their sets of k-shingles [13].

- The *Characteristic matrix* [13] visualises the relationship between documents through their k-shingles. In general, the more similar two documents are, the more shingles they have in common. The matrix comprises of one row per document and one column per k-shingle. This

is a sparse Boolean matrix, as each document contains just a part of the k-shingles generated across all the documents in the characteristic matrix. The Boolean values 1 and 0 indicate whether or not a document contains a certain k-shingle.

- The *MinHashing* technique condenses the large sets of unique shingles present in the characteristic matrix into shorter integer vectors called "signatures" by maintaining the underlying similarity across the document sets. Therefore, the large characteristic matrix is compressed into a smaller matrix, the signature matrix. An approximated measure of similarity (e.g., for documents pairs) can be estimated using the MinHash signatures. The larger the signature, the more accurate the similarity measure [13].

- The *Locality Sensitive Hashing* (LSH) banding technique maps each band of a MinHash signature to an integer number called bucket. This process is carried out across all the bands present in a signature, leading to an array of integers or number of buckets.

- Similar documents have a higher probability of hashing (colliding) in the same bucket than those that are not similar. This technique allows to narrow the search of similar documents to those whose signatures agree in all the rows of at least one band (e.g., pairs). These pairs are called "candidate pairs" and their similarity should be assessed. This solves the problem of a "brute force" pairwise comparison of all documents [13].

2.2 Jaccard Similarity

The Jaccard coefficient J(S1, S2) [13, 35] is successfully used in a range of fields to measure the degree of similarity of sets of data which are binary (e.g., presence-absence (0,1) of certain features) and high dimensional. For example, the Jaccard coefficient J(S1, S2) is used for genome comparison in biology [36], biodiversity evaluation in ecology [37], to find mirror web pages [38], to detect plagiarism [39], among other uses.

J(S1, S2) for two sets (e.g., feature vectors) S1and S2 is defined as the ratio between the size of the intersection of two sets S1 and S2 over their union (1):

$$J(S1, S2) = |S1 \cap S2|/|S1 \cup S2| \qquad (1)$$

J(S1, S2) can vary between zero (S1 and S2 have no common elements) and one (S1 and S2 are identical sets):

$0 \leq J(S1, S2) \leq 1$

The Jaccard Similarity computes similarity between all pairs of items. It is a symmetrical algorithm, as the similarity of S1 to S2 is the same as the similarity of item S2 to S1. It is computationally expensive for big datasets as $O(n^2)$ comparisons are necessary for sets of n objects.

From J(S1, S2) it is possible to derive the Jaccard distance d(S1, S2) (2), which provides the measure of dissimilarity for two sets (S1 and S2) [13, 40, 41]:

$$d(S1, S2) = 1 - J(S1, S2) \quad (2)$$
$$0 \leq d(S1, S2) \leq 1$$

It is clear from Equation (2) that d(S1, S2) can be readily converted into d(S1, S2) and vice versa. When S1, S2 are identical, d(S1, S2) is zero; when S1, S2 are completely different, d(S1, S2) is one. When S1, S2 differ in some measure, d(S1, S2) is positive but smaller than 1.
The Jaccard distance d(S1, S2) is a distance metric, therefore it satisfies the rules of symmetry and the triangle inequality axiom. This means that the distance between two sets S1 and S2 must be the same as the distance between S2 and S1. For sets of triplets S1, S2, S3, the d(S1, S2) cannot be greater than the sum of d(S2, S3) and d(S1, S3) [41].

Both J(S1,S2) and d(S1,S2) can be applied to either binary or quantitative sets of data [41].

A chord ratio value represents the fraction of time a chord (out of the sixty distinct chords available) is played in a song. The chord ratio is zero when a given chord is not present in a song, otherwise it is equal to a positive value (0< chord ratio <1). For example, the song from the Jamendo dataset with id 214 has 19 distinct chords and 19 corresponding non-zero chord ratio values in its feature vector. The chord ratio values in the feature vector can range between 0 and 1 and their aggregate value adds up to 1.

The songs from the Jamendo dataset were compared by analysing their chord ratios feature vectors. Each feature vector comprises of sixty chord ratio values, one for each of the sixty distinct chords present in the chord vocabulary determined by an algorithm used to analyse the audio collection [15], which can be potentially found in any song of the dataset. Figure 2 shows the chord vocabulary that comprises of 60 distinct chords, resulting from the combination of 12 root notes (i.e., A, Bb, b, C, Dd, D, Eb, E, F, Gb, G and Ab) and 5 chord types (i.e., maj, min, 7, maj7 and min7) [42].

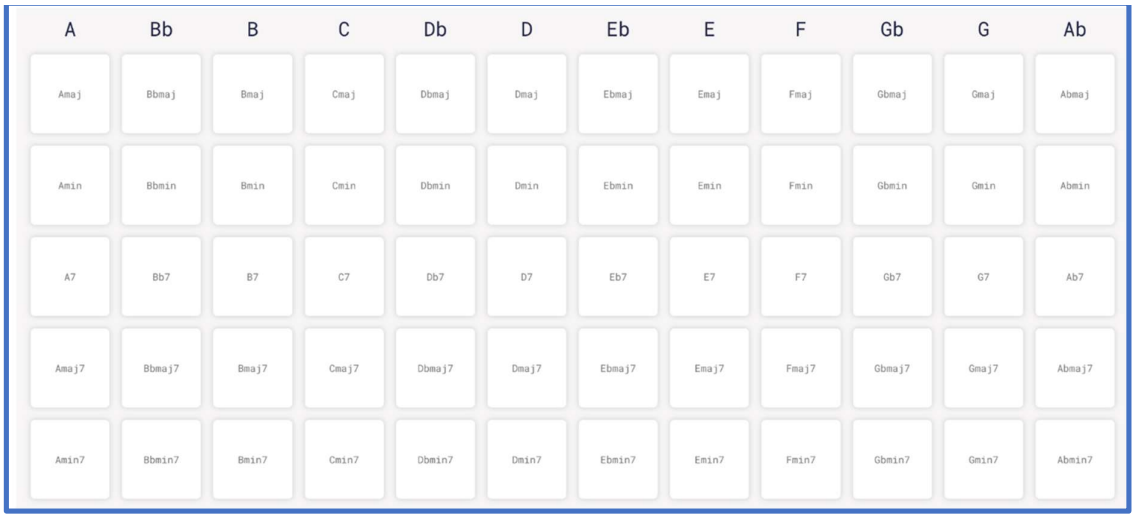


Figure 2. Chord vocabulary of the 60 distinct chords available in the dataset. [42]

It should be noted that generally some chords are played more often than others. Figure 3 shows the occurrences of the 60 chords in the songs of the Jamendo dataset. Cmaj and Gmaj are most frequently played (52% of the songs in the dataset) while Gbmaj7 is rarely played (6.1% of the songs in the dataset) [42].

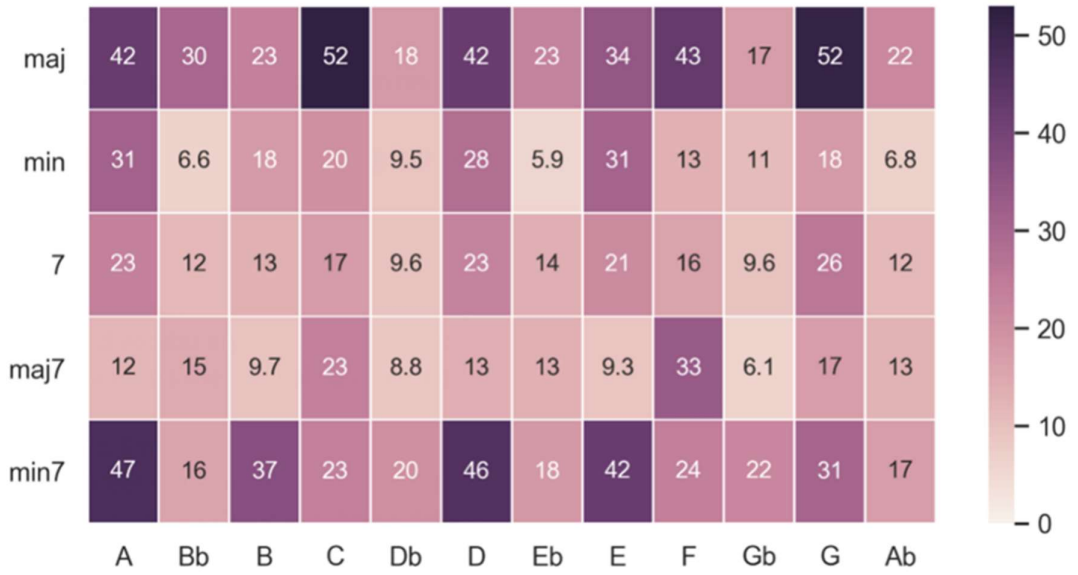


Figure 3. Chord occurrence in the Jamendo dataset. [42]

As each song has a chord ratio value limited to 60 distinct chords and the sets for two songs are comparable in size, the Jaccard similarity is a suitable choice for the task of similarity search. However, as the Jaccard similarity algorithm has $O(n^2)$ time complexity, it cannot be efficiently employed for Big Data (i.e., for scale-up).

In order to determine the Jaccard similarity, each song was represented by a sparse feature vector (Figure 4). In such vector, non-zero chord ratio values and the corresponding indices were conveniently stored in two arrays (value and index arrays, respectively). The size of the maximal array is 60, which is equal to the number of distinct chords present in the Jamendo dataset.

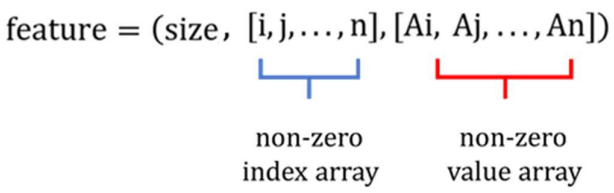


Figure 4. Sparse vector representation used to calculate the Jaccard similarity for song pairs.

For each pair of songs compared, the Jaccard similarity was calculated as the intersection of two sets of indices (i.e. [i, j, ..., n]) over their union. Identity (i.e., songs with the same id) and duplicates of a pair of songs (i.e., where the similarity for songs (id1, id2) is measured, whilst the similarity for songs (id2, id1) is not) were excluded from the computation, as the Jaccard similarity is a symmetric algorithm. In this case, the similarity is calculated considering only those indices of the feature vectors that the two songs share (i.e., the two songs have chord ratio values that are not null at the same indices), whilst disregarding those indices that they do not share (i.e., the chord ratio values are not null, but at different indices). This approach does not consider the individual feature values of each chord ratio belonging to each song. However, similarity searches can be improved by using the weighted Jaccard similarity for a pair of non-negative real vectors (3) [43]:

$$J(S1,S2) = \frac{\sum_{i=1}^{n} min(S1i,S2i)}{\sum_{i=1}^{n} max(S1i,S2i)} \quad (3)$$

$$if \sum_{i=1}^{n} max(S1i,S2i) > 0$$

This work focused on the Jaccard similarity. The weighted Jaccard similarity is mentioned as a potential avenue for future work.

2.3 MinHash Technique

MinHash is an algorithm that computes the approximate similarity for two binary sets [31], instead of calculating the Jaccard similarity, which requires the computationally intensive intersection (AND operation) and union (OR operation) of two sets (each set representing a document / song). Similarity

search based on Jaccard similarity is not feasible for Big Data, as for n documents it requires n*(n-1)/2 pairwise comparisons with $O(n^2)$ time complexity.
Moreover, as a characteristic matrix M represents all the possible combinations of sets of k-shingles across all documents in a dataset, space complexity is a problem for Big Data and a smaller signature matrix that preserves the characteristic of the main sets in the characteristic matrix is needed. Space complexity can be considerably reduced via the MinHash technique [13, 14, 31], which converts a sparse characteristic matrix M (i.e., rows with zeros in most of the column) into a smaller matrix of signatures, allowing for fewer data to be stored and compared, although on the other hand it gives just an approximated value of the similarities between the data investigated. The technique allows to sample elements (or k-shingles) from a set of documents in the characteristic matrix M. The MinHash algorithm computes a hash value for each shingle in a document and samples the shingle with the smallest hash value, reducing memory overheads linked to data sparseness.

There are two ways to MinHash documents (or sets) represented by a column C in a sparse characteristic matrix M, described at i) and ii) below:

i) via explicit random permutations (usually several hundreds) and sorting of the rows in the characteristic matrix M. Each random permutation defines a MinHash function h that applied to a column C in the characteristic matrix M gives a hash value, which is the first row in the permuted order in which column C has a 1. However, this method is computationally not viable, as it is time consuming for the number of permutations and the large number of rows in the characteristic matrix M [13];

ii) by using a random hash function that simulates the effect of a random permutation. This method is computationally viable, and it was used in this work [13].

2.3.1 MinHash by random permutations

Figure 5 shows an example of MinHash technique via explicit permutations of the rows of a characteristic matrix M for a toy dataset. The four documents (or sets) and the seven possible elements (e1,e2,e3, etc.) across all documents are represented by columns S1, S2, S3 and S4 and by rows with indices 1 to 7, respectively. Each random permutation defines an independent MinHash function h_i that applied to each column of M generates a smaller signature matrix. The rows of the MinHash signature matrix correspond to the independent hash functions generated by the permutations, while the columns correspond to the documents (or sets) S1, S2, S3, S4. Each hash function is one component of the vector, which is the signature for a document S [13].

The random permutation P1 (Figure 5) generates a hash function, which applied to each column (or set) S1, S2, S3, S4 generates a corresponding MinHash value for each column. The MinHash algorithm starts from row 1 (in the permuted order) and checks whether there are columns that have a value 1. In the example at Figure 5, row 1 in the permuted order (see permutation P1) has a value 1 in both columns S2 and S3. Therefore, both hash values for S2 and S3 are assigned to row 1 (in the permuted order).

As the hash values for columns S1 and S2 are still unknown, the algorithm moves to row 2 (in the permuted order) and checks which columns have a value 1. Both columns S3 and S4 have a value 1, but the hash value for S3 has already been assigned in the previous iteration. Therefore, only the hash value of column S4 is assigned to row 2 (in the permuted order). The hash value for column S1 is still outstanding, so the algorithm moves to row 3 (in the permuted order) and checks if this row has a value 1 in column S1. The hash value for column S1 is 3 as row 3 (in the permuted order) has a value 1 in column S1. The algorithm stops when all the columns (or sets) in the characteristic matrix M have been assigned a hash value. As this is the case for the hash function generated by the first permutation, the search stops, and the signature vector is stored in the MinHash signature matrix (row 1 in the MinHash signature matrix, Figure 5).

A second independent MinHash function is generated by a second (P2) random permutation of the rows in M. The algorithm proceeds to find the MinHash values for the columns S1, S2, S3 and S4 starting from row 1 (in the permuted order) for permutation P2. Row 1 (in the permuted order) has a value 1 in column S3, so the hash value for S3 is assigned to row 1 (in the permuted order). As the hash values for columns S1, S2, S4 are still unknown, the algorithm proceeds to row 2 (in the permuted order). Row 2 (in the permuted order) has value 1 in both columns S1 and S2. So, their outstanding hash values are assigned to row 2 in the permuted order. The algorithm moves to row 3 (in the permuted order), as the hash value for column S4 is still unknown from the previous two iterations. Row 3 has value 1 in column S4, therefore the hash value for S4 is assigned to row 3. The algorithm stops as all the columns have hash values. The vector of hash values for the second hash function is stored in the second row of the MinHash signature matrix.

In the example at Figure 5, a third permutation is performed and a third MinHash function is generated and applied to the columns of M. The MinHash values are stored in the third row of the MinHash signature matrix. In the example at Figure 5, for the sake of simplicity just three random permutations were computed.

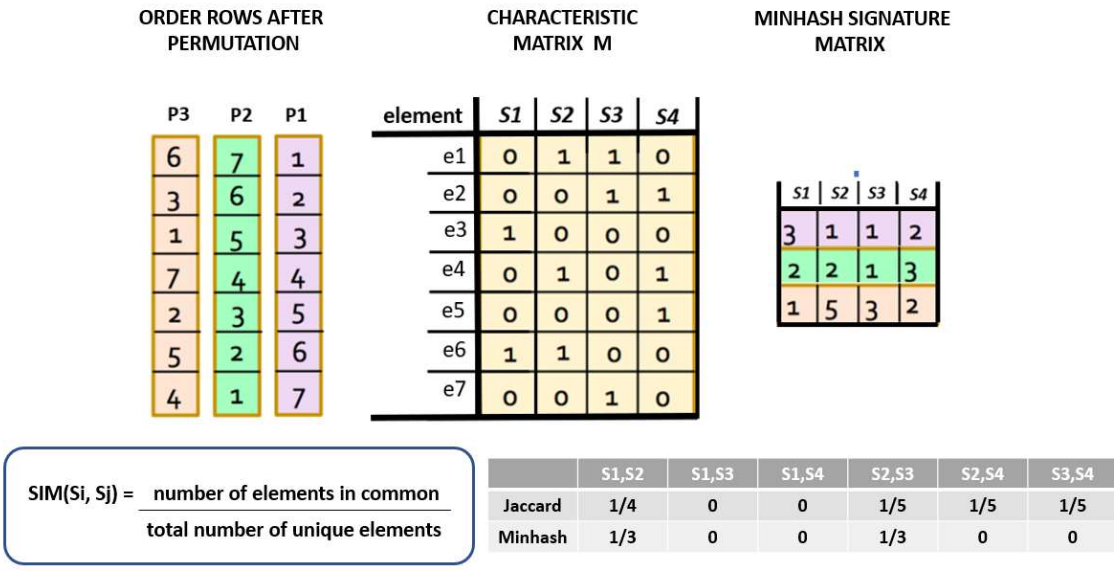


Figure 5. Example of Minhash technique via explicit permutations of the rows in a characteristic matrix M.

Once the MinHash signature matrix is available, the approximated similarity between two documents can be calculated as the ratio between the number of MinHash values that two documents (or sets) have in common and the total number of hash values in the signatures (for signatures of the same length). The more permutations performed, the more MinHash functions and hash values are generated for each document. The more hash values, the more accurate the measure of similarity for two documents, compared with the ground truth Jaccard similarity. Figure 5 shows the results of similarity for pair of documents (e.g. S1, S2; S1, S3; etc). It is clear that the MinHash similarity provides an approximate estimation of the Jaccard similarity.

The main findings regarding MinHash by random permutations are as follows:

- *n* random permutations of rows in a characteristic matrix M generate *n* independent hash functions h;

- the MinHash function for a set is the index of the *first* row (in the permuted order) in which a column S has value 1;

- the probability that the MinHash function h for a random permutation of rows produces the same value for two columns (e.g. S1, S2) equals the Jaccard similarity J(S1,S2) of those sets:

$p\big(h(S1) = h(S2)\big) = J(S1, S2)$ (4);

- the similarity of the signatures for two documents (or sets) is the fraction of the MinHash functions (rows) in which they agree [13]:

$(S1, S2) = common\ hashes / total\ uniquehashes$ (5);

- The higher the number of permutations, the longer the signatures for each document (or set) and the lower the error of the MinHash similarity when compared with the Jaccard similarity.

2.3.2 Implementation of the MinHash technique without physical permutations

The MinHash technique, which computes signatures of the sets (columns) of a characteristic matrix via permutations, is not computationally feasible for large datasets. For a dataset of n rows, the number of possible permutations is n! (factorial). Picking a number of random permutations is not efficient, due to high space and time complexity. Indeed, the process is sequential, and the matrix needs to fit in memory, which leads to high space and time complexity.

Instead of randomly permuting the rows of the characteristic matrix M, the rows permutation is simulated by a number of randomly chosen hash functions (h1, h2, …, hn), which hash the row numbers to n numbers in the range (0, number of rows). Such values represent the order in which the row appear in the permuted order. In general, a hash function maps integers to bucket numbers. The idea is *"to use a hash function h to permute a row r to position h(r) in the permuted orde*r"[13].

As hash functions are not real permutations, some rows will be hashed to the same bucket (i.e., bucket collision), whilst some other rows will be left out of the permutation (i.e., unfilled buckets). However, the more hash functions (i.e., the more permutations and buckets), the fewer the collisions.

As an initial step, a signature matrix is created with a number of columns equal to the number of documents (sets) in the characteristic matrix M and a number of rows equal to the number of hash functions (h1, h2, …, hn) used in the simulation. Initially, each element of the signature matrix is set to ∞.

A set of hash functions is chosen for the task. For example, Equation 6 shows the set of hash functions used in this project:

$h(x) = ((a * x + b)\%p)\%N$ (6) a: random integers

a ≠b

p: fixed large prime number (p>N)

For row r in the characteristic matrix M, a set of hash functions h1(r), h2(r), . . . , hn(r) is computed for those columns c that have 1 in row r. If the computed hash value $h_i(r)$ for column c in row r is smaller than the value stored in the signature matrix for column c and hash function h_i, then the signature matrix is updated with the new smaller value $h_i(r)$.

Figure 6 shows an implementation of the MinHash algorithm on a toy dataset, using a toy hash function $(a * x + b)\%p$ specifically for the sake of simplicity of the calculations in the example. The toy characteristic matrix M comprises of two columns S1 and S2 (e.g., songs) and 11 elements (i.e., chord ratios). Figure 6 shows how the MinHash algorithm works step by step.

A signature matrix is created, with the value of all the elements set to ∞. Then the algorithm starts from row 1 (top of characteristic matrix M) and computes the hash functions (e.g., x%11, 2x+2%11) of the row for those columns (in row 1) that have a 1. In step 1, both S1 and S2 in row 1 (i.e., Amaj) have 1, so the hash values are computed and then compared with the values in the signature matrix. In this case, the hash values for h(x) and g(x) are both smaller than ∞ for both S1 and S2, so the signature matrix is updated in step 1. The process continues row by row until the last row of the characteristic matrix M is reached and the final signature matrix is updated one last time. The algorithm stops when the computation reaches the last row of the characteristic matrix M.

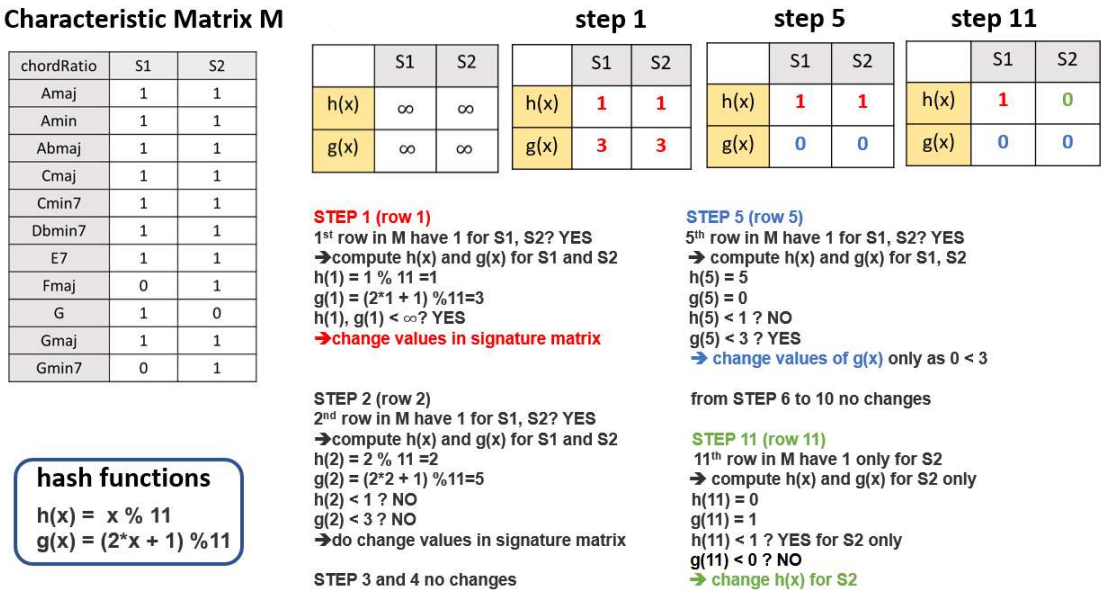


Figure 6. Example of MinHashing technique without physical permutations on a toy dataset. The final signature matrix is obtained at step 11.

2.4 Locality Sensitive Hashing

The Locality Sensitive Hashing (LSH) technique allows to narrow the search to the most similar pairs of documents (or sets) with underlying similarity above a set similarity threshold. Such documents are called *candidate pairs* [13]. The main purpose of LSH is to shortlist candidate pairs and find their similarity, instead of investigating all the possible pairs of documents present in a dataset.
Setting a similarity threshold t means that if a pair of signatures for two sets are at least t fraction similar in their rows, then the two sets they originate from are likely to be t similar and they can be considered candidate pairs.

There are two main approaches to LSH, being:

1. hashing the columns of a signature matrix several times with several hash functions under the assumption that similar pairs will hash to the same bucket; or
2. dividing each signature (for a column c in a signature matrix M(i, c)) into b bands of rows r. Each band is hashed with the same hash function. Dividing a signature into bands is important as it helps to minimise the number of candidate pairs that are false positives. Signatures with the same rows in different bands will hash to different buckets.

The approach at 2. above was chosen for the last part of this project. The LSH banding technique divides a MinHash signature matrix M(i, c) into b bands of rows r. Each band for a column c is then hashed to a hash table with a large number of buckets.

Candidate column pairs in a signature matrix M are those that hash to the same bucket for all the rows in at least one band.

The number of bands b and rows r (b*r = n with n = length of the signature) can be tuned in order to minimise the number of false positives and false negatives for a similarity threshold *t* that is function of b and r (7):

$$t \sim (1/b)^{1/r} \qquad (7)$$

For example, to minimise the number of false negatives or false positives b and r can be tuned to produce a lower or higher threshold limit [13].

The probability that the signatures agree in all the rows of at least one band, and therefore become a candidate pair, also depends on b and r (8):

$$P = 1 - (1 - s^r)^b \qquad (8)$$

Therefore, the choice of such parameters is an important aspect of this study.

Chapter 3. Results and discussion

3.1 Jaccard similarity

In this work, we aimed to find the Jaccard similarity coefficient J(S1, S2) between pairs of songs (e.g., S1 and S2) by measuring the degree to which their chord ratio composition is alike. On the other hand, the dissimilarity coefficient *d* provides a measure of the degree to which songs differ in composition and it can be expressed as 1-J(S1,S2).

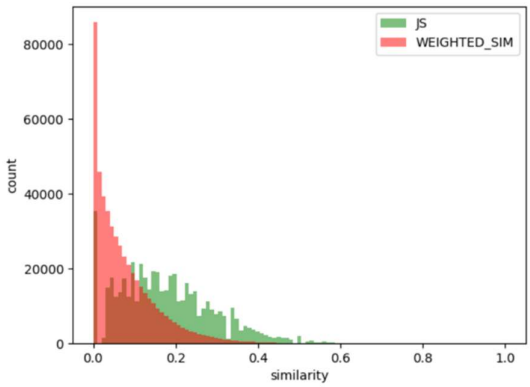
As previously discussed in Chapter 2, a pre-processing step known as "k-shingling" for textual data is usually necessary to implement the Jaccard similarity coefficient.

However, in this project such pre-processing step was not necessary as we were interested in the general similarity of songs pairs. The Jamendo songs dataset comprises of ca. a hundred thousand songs, each of which is represented by sets of chords and by their corresponding chord ratios. These are the elements that represent each song (the equivalent to "1-shingles") and for such reason the k-shingling technique was not used.

The chord ratios are organised in a sparse vector, one per song. Each sparse vector comprises of an index array for the chords (among all the possible chords available in the dataset) which are present in the corresponding song, whilst their actual values (each, a positive float number smaller than one) are stored in a second array. All feature vectors have the same length, as there are sixty distinct chords available across all the Jamendo songs (Chapter 2.2).

The chord ratio provides meaningful information, as it represents the length of time that a specific chord is played for in a song. For example, if the chord ratio for the Amaj chord in a song is 0.3, this means that the Amaj chord is played for 30% of the time in that song, whilst if it is zero then that chord is not played for any fraction of time in that song.

Both the simple Jaccard similarity coefficient (JS) and the weighted Jaccard similarity (WJS) were employed in the song pairs similarity measures and the results were analysed and compared. When comparing pairs and triplets of songs, the simple JS was calculated by taking into account the presence or absence of a chord and by ignoring the length of time for which that chord is played (referred to as "abundance"). On the other hand, WJS was evaluated by using the chord ratios values. Figure 1 shows the superimposed histograms for JS and WJS for half a million song pairs.



summary	JS	WJS
count	499500	499500
mean	0.17655719071386514	0.07771538584691884
stddev	0.11636436587175325	0.07869709108648987
min	0.0	0.0
max	1.0	0.995800524934383

Figure 7. Superimposed histograms for JS (green) and WJS (red) computed for ca. half a million song pairs from the Jamendo dataset.

JS is a set-based similarity measure, whilst WJS is a vector-based similarity measure. The difference between JS and WJS is evident. Similarities are more spread out in JS than in WJS (Figure 7, standard deviation).

Memory and time of computation were also evaluated for both JS and WJS. In a series of experiments, JS and WJS were computed for an increasing number of pairs: 5k, 500k, fifty million. The results are reported for both JS and WJS as mean values (with the standard error) calculated over five experiments under the same set of conditions (i.e., number of song pairs). The standard error (SE) on the mean was calculated as the ratio between the standard deviation and the square root of the sample size n [44] (i.e., five trials for each number of song pairs).

Figure 8 (D) shows that JS and WJS have a comparable memory requirement and that such requirement increases (at a similar rate) with an increasing number of pairs analysed (Figure 8 plot D).

Timewise, JS scaled up better than WJS. Figure 8 shows the time of computation in logarithmic scale for JS and WJS (plot C). The time of computation has an exponential growth for WJS, and this is not feasible for Big Data. On the contrary, JS works more efficiently as it requires minimal data and a shorter time of computation**.** Histograms A and B in Figure 8 highlight the remarkable difference between JS (orange histograms) and WJS (blue histograms).

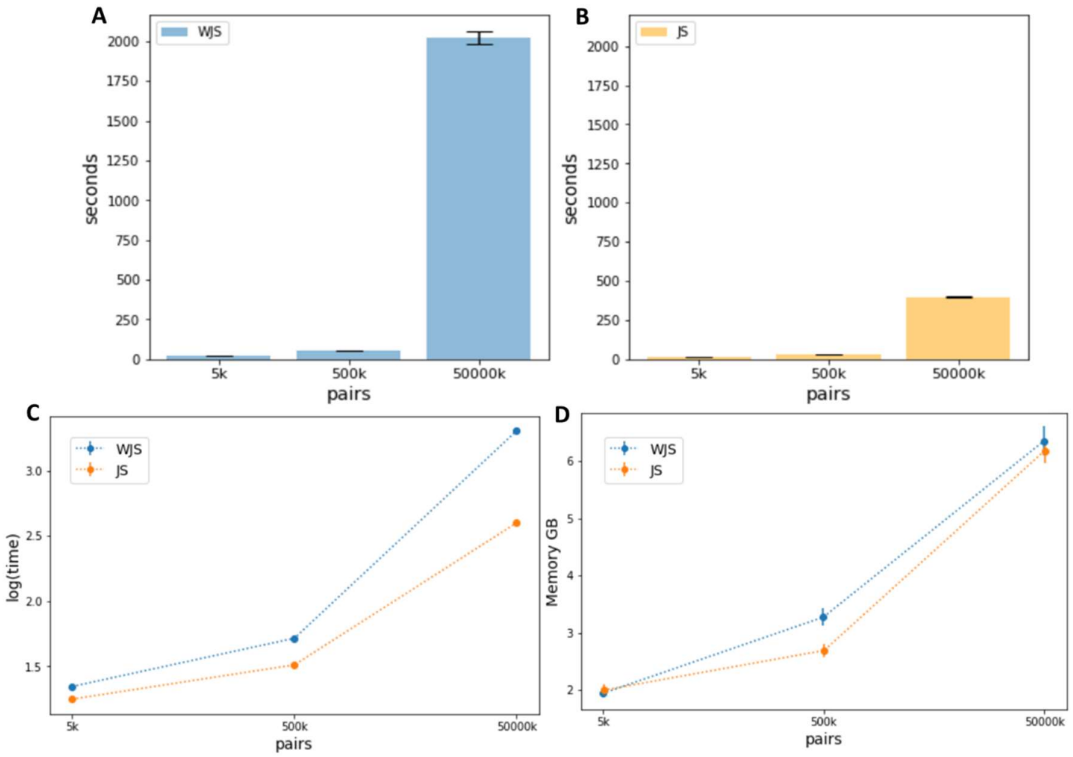


Figure 8. Time and memory for Jaccard and weighted Jaccard similarities computed on an increasing number of song pairs (top, histograms A and B). Superimposed line plots (bottom, histograms C and D) show time (logarithmic scale) and memory (Gigabyte) of computation for JS and WJS, respectively. Results are expressed as mean $\pm$ 1SE, calculated over five trials.

When it comes to scalability, time efficiency is a key parameter to consider. For such reason, JS was chosen as metric to assess the similarity across the Jamendo songs and used as ground truth in the next step of this work.

3.2 MinHash similarity

MinHash dimensionality-reduction is a popular technique [31, 45, 14] widely used to estimate the Jaccard similarity coefficient for large datasets [46, 47, 48, 49]. Indeed, the precise calculation of such coefficient is usually computationally intensive (with high complexity in terms of time and memory), due to lengthy operations (i.e., intersection and union), which require to keep in memory the pairs of sets to compare (i.e., high space complexity).

The MinHash technique reduces space complexity, but it is not able to address the problem of $O(n^2)$ time complexity, as it needs to compare every document to every other document, as is the case for the Jaccard similarity algorithm. Time complexity can be resolved by Locality Sensitive Hashing (LSH,

discussed in the next paragraph), which allows to circumscribe the similarity search to only those documents that are most likely to be similar (for a certain similarity threshold) [13, 32].

The MinHash technique is a fundamental step towards the implementation of LSH. The latter technique is key to solving the problem of time complexity posed by the Jaccard similarity algorithm in Big Data.

The results from a series of MinHash experiments on the pairwise comparison of Jamendo songs are presented and discussed in this section.

The first important point that needs to be addressed is how well a similarity measure based on the MinHash algorithm approximates the ground truth Jaccard similarity.

It is generally known that the higher the number of MinHash functions, the larger the signature and the more accurate the measure of similarity for a pair of documents [13].

We were interested in determining the impact of varying the number of MinHash functions on the accuracy and precision (and other related metrics (Recall and F1 score)) of the approximate similarity for a pair of signatures. In these experiments, the Jaccard similarity coefficient was used as a benchmark to compare the results obtained with the MinHash technique for a minimum similarity cut off (threshold t). The relevant metrics were calculated in terms of whether the similarity assessment were correct (with TP: true positives, TN: true negatives) or wrong (with FP: false positives, type I error; FN: false negatives, type II error).

Figure 9 shows the following: at the top, the equations for accuracy, precision, recall and F1 score; at the bottom, the definitions of TP, TN, FP and FN. The metrics for similarity were evaluated for half a million songs pairs and reported as mean $\pm$ 1SE. The standard error (SE) was calculated over five experiments carried out under the same conditions (i.e., number of songs pairs, number of MinHash functions, similarity threshold t).

TP, TN, FP and FN are defined in terms of approximated similarity SIM(S1,S2) and benchmark similarity J(S1, S2) for a cut off similarity value t. In particular, for a set similarity threshold t, when both SIM(S1, S2) and J(S1, S2) fall at or above such cut off value, then a data point SIM(S1,S2) is classified as TP (Figure 3, A); when they both fall below t, then SIM(S1,S2) is classified as TN (Figure 9, B). On the contrary, when SIM(S1,S2) falls above t and J(S1,S2) does not, then SIM(S1,S2) is classified as a FP data point, and vice versa as a FN data point when SIM(S1,S2) falls below t, but J(S1,S2) does not.

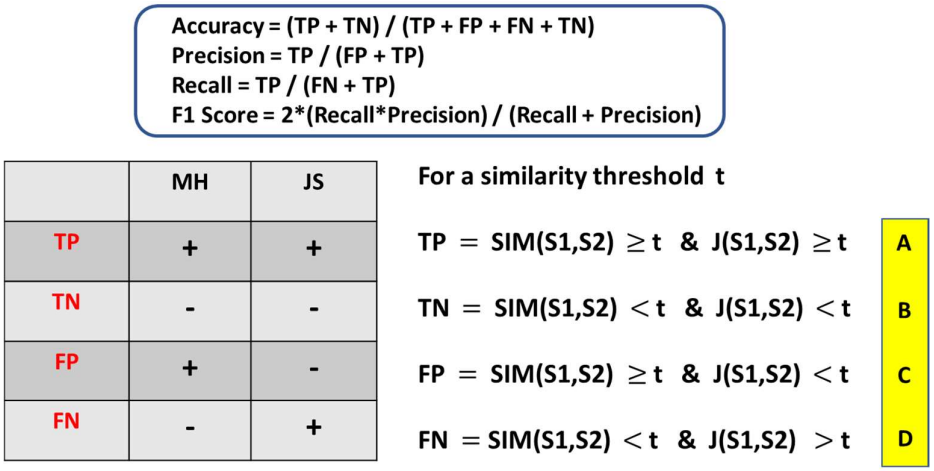


Figure 9. Equations for metrics (top) and how the following data points are assessed: true positive (TP), true negative (TN), false positive (FP) and false negative (FN) (bottom). The table depicts TP, TN, FP and FN on the basis of whether a data point SIM(S1, S2) or J(S1, S2) falls above (+) or below (-) the threshold t.

In our experiments, three similarity cut off values t were chosen (i.e., 0.1, 0.3 and 0.6) and the approximate similarity SIM(S1, S2) for songs pairs was computed for a different number of MinHash functions (i.e., 200, 300, 400, 500 and 600). All the experiments were carried out on the same number of songs pairs (i.e. half a million pairs).

Figure 10 shows four line-plots, as follows: A for the accuracy; B for the precision; C for recall; and D for F1 score. The metrics are plotted against each cut off value t and the different colours represent each metric computed for a different number of hash functions. In particular, blue, orange, green, red and purple were chosen to depict computation for 200, 300, 400, 500 and 600 MinHash functions, respectively. Each point in the plots represents the mean value $\pm$ 1SE, calculated over five experiments under the same set of conditions (i.e., number of songs pairs, number of hash functions and threshold t). The standard error on the mean was calculated as the ratio between the standard deviation and the square root of the sample size n, which in this case is five, as the number of trial runs for each unique set of conditions.

Plot A (Figure 10) shows that the accuracy for SIM(S1,S2) increases almost linearly with increasing t. For the lowest similarity threshold (i.e., t=0.1), the most accurate results are obtained for larger signatures (i.e., 500 and 600 MinHash functions; red and purple points, respectively, for t=0.1) rather than shorter signatures (i.e., 200 Minhash functions). However, the difference is not striking (at most, ca. 1%).

Plot B (Figure 10) shows that the precision of the approximated similarity SIM(S1,S2), compared with the Jaccard similarity J(S1, S2), depends on the length of the signatures. The larger the signature (more

MinHash functions), the fewer the FPs. Indeed, for the largest signature (i.e., 600 MinHash functions) the precision is the highest (ca. 0.87), which means that ca. nine out of ten SIM(S1,S2) values are relevant results (TPs), whilst for the shortest signature (i.e., 200 MinHash functions) only seven out of ten values are TPs (i.e., the precision is ca. 0.75). Another interesting finding is that the precision seems to depend only on the number of MinHash functions used and not on the cut off value considered.

Plot C and D (Figure 10) show the recall and F1-score for the similarity measure(s) based on the MinHash technique. Plot C (Figure 10) shows that the higher the similarity threshold t, the lower the recall (i.e., more FNs); the higher the number of MinHash functions, the higher the recall (i.e., less FNs).

The F1-score is the weighted average of precision and recall. The F1-score takes into account both FPs and FNs. Plot D (Figure 10) shows the F1-score values for a different number of MinHash functions.

The plot confirms that the number of MinHash functions plays an important role in the correctness of the similarity measure(s). The lowest F1-scores are found for shortest signatures (i.e., 200 and 300 MinHash functions), whilst larger signatures (i.e., 500 and 600 Minhash functions) are more likely to give approximated similarities SIM(S1, S2) with a lower error (FPs and FNs), especially for a high degree of similarity.

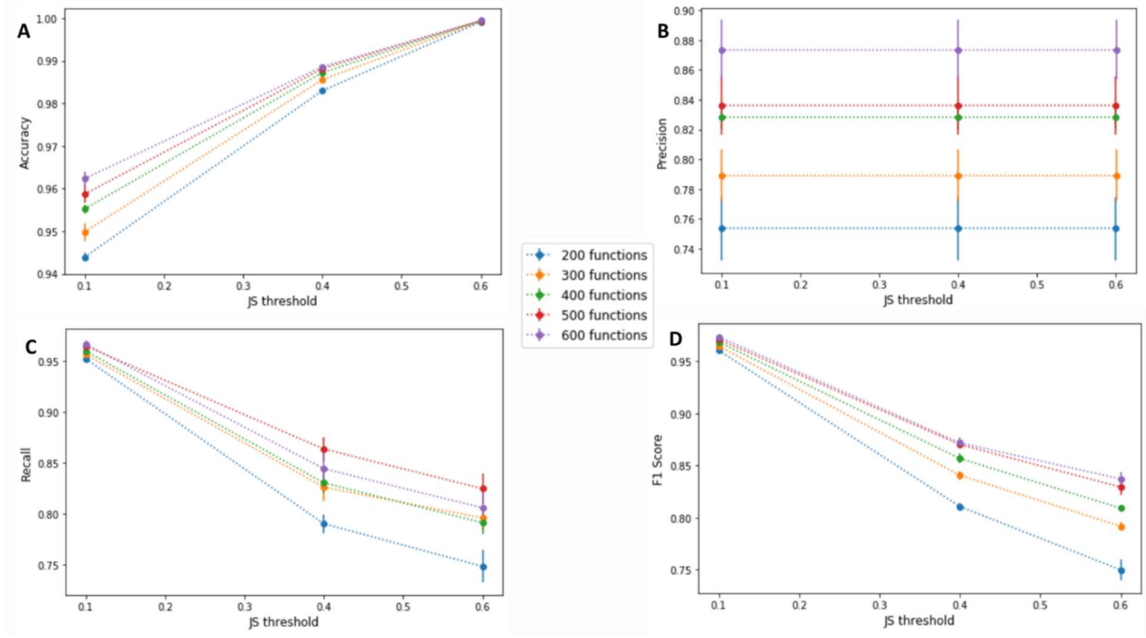


Figure 10. Line plots for accuracy (A), precision (B), recall (C) and F1-score (D). Metrics are expressed as mean $\pm$ 1SE, calculated over five trials. Similarity query carried out for ca. half a million song pairs.

Computational memory and time for similarity measure(s) (i.e., time and space performance) were also analysed in relation to the number of MinHash functions used to *write* each song signature. Although the MinHash technique is not meant to solve the problem of time complexity, it is important to understand the costs (in terms of both time and memory) associated with, for example, high precision for the similarity measure(s) (but this applies also to any other metric investigated earlier). Indeed, precision, accuracy, recall and F1-score all depend on the number of MinHash functions used in the MinHash process.

When time complexity is a problem, a choice needs to be made. For example, a trade-off between precision and performance. This means that a better performance (i.e., fast computation) implies a lower precision of the results (i.e., the lower the precision, the more FPs).

Figure 11 shows the superimposed line plots of time (in seconds, represented by blue dots and a dotted line) and memory (in Gigabyte, represented by red dots and a dotted line) versus the number of MinHash functions employed for the computation of SIM(S1, S2) of ca. half a million of songs pairs using the MinHash technique. Each point is expressed as the mean$\pm$ 1SE of time or memory calculated over five trials carried out under the same conditions (i.e., number of songs pairs and MinHash functions). Figure 11 shows that both computational time and memory of similarity measure(s) increase with the number of MinHash functions employed. Indeed, similarity measures between shorter signatures is computed on average in ca. 55 seconds (i.e., 200 MinHash functions), whilst for longer signatures such computation takes on average ca. over 63 seconds. Moreover, the longer the signatures (e.g., 600 MinHash functions), the more precise for example the similarity measures (Figure 10, plot B). The more precise the similarity measures, the longer the computational time and memory (Figure 11) associated with the task.

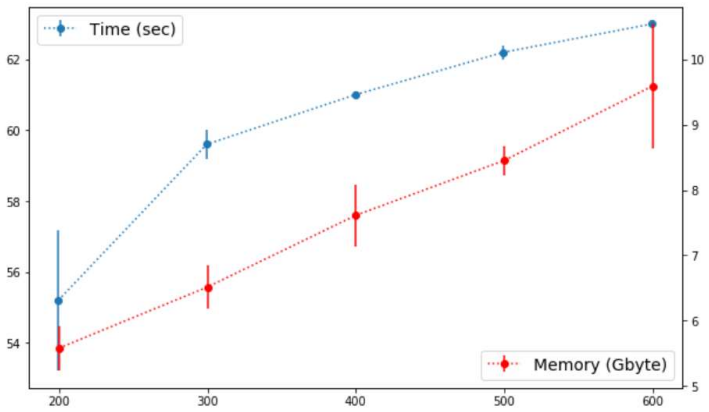


Figure 11. Superimposed line plots for Time (seconds, blue) and Memory (Gigabyte, red) vs number of MinHash functions for the computation of approximated similarity via MinHash technique of ca. half a million songs pairs. Time and memory values are expressed as mean $\pm$ 1SE, calculated over five trials.

It can be concluded that when both precision and recall are important, a higher number of MinHash functions is necessary in order to minimise the error. On the contrary, in order to improve time and memory performance a trade-off between performance and precision is usually needed. This can be achieved by employing less MinHash functions, as the performance improves for shorter signatures.

In this section we have applied successfully the MinHash technique to compute the approximate similarity of Jamendo songs pairs.

We have carried out a number of experiments in order to establish the number of MinHash functions necessary to minimise the error of the process (e.g., maximise the F1-score). Performance was also investigated in relation to the number of MinHash functions.

The analysis carried out in this section is a necessary step towards the implementation of the of Locality Sensitive Hashing technique presented in the last section of this work (Chapter 3.3). The LSH technique reduces time complexity and enables the scaling up of the process.

3.3 Locality Sensitive Hashing

The Locality Sensitive Hashing (LSH) technique [50, 51] hashes items into buckets using a family of functions. Similar items are hashed to the same bucket with high probability. LSH probabilistic approach to finding similar items in a large dataset solves the problem of time complexity [50, 51].

In this work we implemented the LSH banding technique via MinHash. In the previous section (Chapter 3.2) we analysed MinHash signatures of a number of songs. Such signatures are now analysed with the LSH technique to find those songs pairs which are most likely to be similar (i.e., candidate pairs).

LSH divides the minhashes *m* into a series of bands *b*. Each band comprises of a number of rows *r*. The relation between m, b and r is the following (1):

$$b * r = m \qquad (1)$$

For example, when 600 minhashes are divided into 100 bands, each band comprises of six rows (or MinHash signatures). At this stage each band is hashed to a bucket by a hashing function; in our work we used a modular hashing function (2):

$$\sum_{i=0}^{r}(Ai * Xi + Bi)\%p \qquad (2)$$

The LSH hash function is calculated as the product between randomly generated coefficients *Ai* and the *r* MinHashes (i.e., MinHash signatures) *Xi* in a band. In our work, *Ai* were randomly generated odd numbers smaller than p, with p a fixed prime large number (p = 338563 in our work).

If two songs have the exact same MinHash values on every row of at least one band (i.e., identical rows), they will be hashed to the same bucket and they will be considered *candidate pairs*. The probability of becoming a candidate pair is given by (3) with s, b and r being similarity, bands and rows, respectively:

$$P(s) = 1 - (1 - s^r)^b \quad (3)$$

When P(s), for certain b and r values, is plotted against s (i.e., s can be the Jaccard similarity J(S1, S2) or the approximated similarity SIM(S1,S2)), a S-shaped curve is obtained (i.e., the S-curve) [13]. P(s) describes the probability of performing the comparison of two MinHash signatures.

The optimal similarity threshold (*t*) for a candidate pair can be estimated from (4) *b* and *r*, as follows:

$$t \cong (1/b)^{1/r} \quad (4)$$

An optimal balance between the number of False Positives (FPs) and the number of False Negatives (FNs) is achieved at *t*, with FPs defined as the song pairs that share a bucket but have similarity below t and FNs defined as the song pairs that do not share a bucket but have similarity above *t*.

The songs pairs with SIM(S1, S2) above *t* are more likely to become *candidate pairs* than those with similarity below such threshold. Therefore, the choice of b given a MinHash signature of length m MinHashes is crucial in the search for similar pairs with a similarity threshold t.

In conclusion, a similarity query performs better (timewise) if the comparison is carried out for those songs pairs which are candidate pairs . Therefore, the LSH probabilistic approach to finding similar items in a large dataset solves the problem of time complexity [50, 51]. However, LSH does not guarantee to find all similar pairs, as some similar pairs are expected to be missed (i.e., FNs). The tuning of b is particularly important as the higher *b*, the lower t (4) and the more likely the FPs; on the contrary, the lower *b*, the higher *t* and the more likely the FNs.

LSH with banding technique was used to find candidate pairs across the Jamendo songs. Table 1 shows the similarity thresholds t for MinHashes (MH) with different length (i.e., from 200 to 600), number of bands *b* (keeping in mind that the number of MinHashes must be divisible for the number of bands) and the number of rows *r* per band calculated by using equation (1). Threshold values t were determined by using equation (4). The LSH technique enables to find the candidate pairs with a similarity over t for a given number of MinHashes and bands. For example, a MinHash signature MH of length 200 can be divided into 10, 20, 50 and 100 bands, each comprising of 20, 10, 4 and 2 rows, respectively. For songs with MinHash signatures of length 200 which are divided into 10 bands, we can expect that the retrieved

candidate pairs will be at least 89% similar. It is clear from the table that the number of options available in terms of t values depends on the length of MH and b. Indeed, for MH with length 600 the number of possible threshold values is nine, whilst for MH with length 200 just four threshold values are possible, which indicates that the similarity search is less specific for 200 MH than for 600.

MH	b	r	t
200	10	20	0.89
200	20	10	0.74
200	50	4	0.38
200	100	2	0.10
300	10	30	0.93
300	30	10	0.71
300	50	6	0.52
300	100	3	0.22
300	150	2	0.08
400	10	40	0.94
400	20	20	0.86
400	40	10	0.69
400	50	8	0.61
400	100	4	0.32
400	200	2	0.07
500	10	50	0.95
500	50	10	0.68
500	100	5	0.40
500	250	2	0.06
600	10	60	0.96
600	20	30	0.90
600	30	20	0.84
600	50	12	0.72
600	60	10	0.66
600	100	6	0.46
600	150	4	0.29
600	200	3	0.17
600	300	2	0.06

Table 1. LSH technique. Similarity threshold t for candidate pairs for MinHash signatures of different length and number of bands.

An S-curve visualises the relationship between similarity of document pairs and the probability that all their rows are identical in at least one band (3), when FPs and FNs occur. Ideally, with no FPs and FNs such curve would show a sharp cut-off point as it would be an infinitely steep straight line at *t*, without curved endings (like a step function).

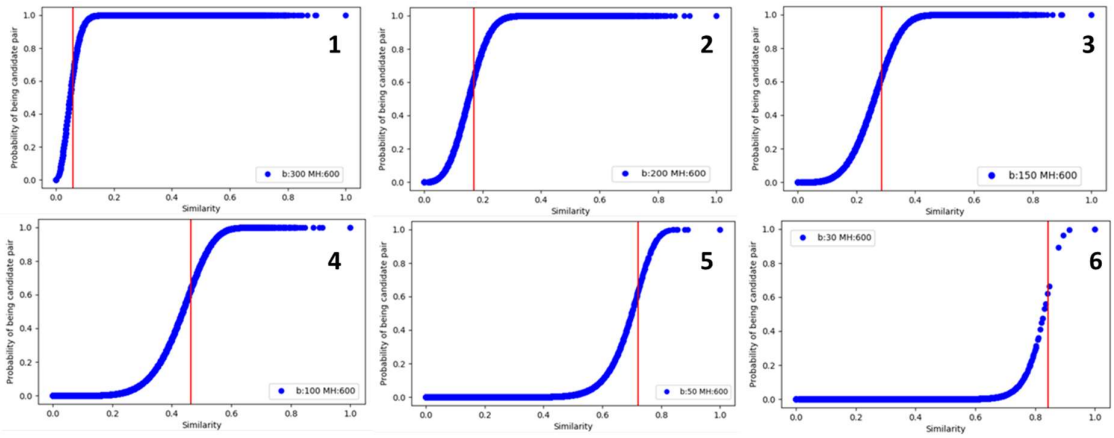
The trade-off between FPs and FNs at an optimal similarity threshold *t* can be visualised as the area below and above the S-curve for similarity values smaller than *t* and larger than *t*, respectively.

Figure 12 shows a series of S-curves computed for ca. half a million Jamendo song pairs computed for different numbers of bands. Songs similarity (x-axis plots, Figure 12) was evaluated via MinHash technique using six hundred MinHashes per signature per song. The probability of being a candidate pair was calculated using equation (3) and evaluated for different numbers of bands and rows (Table in Figure 12). The similarity threshold *t* for different numbers of bands and rows was also evaluated using approximation (4).

The aim of the LSH technique is to find a division into rows such that if a pair of songs is compared, that means that the probability of such pair of songs actually being similar is high. The S-curve has low values until it reaches a point where it grows rapidly (step). The step should occur as close as possible to the similarity value we are searching for. Approximation (4) shows that the threshold t (step) depends on the number of bands and rows, which means that their choice is important to focus and narrow the similarity search.

For example, if we consider a pair of songs as being similar when their MinHash similarity is greater than 0.7, and their signatures comprise of 600 MinHashes per song, the step of the S-curve needs to be as close se possible to 0.7 (plot 5, Figure 12). Therefore, we need to divide the 600 MinHash values into twenty bands, each band comprising of twelve rows. Table B (Figure 12) shows the probability of being a candidate pair at the S-curve step, also known as threshold *t* (0.7, highlighted in yellow). Table B shows that if the similarity of a song pair is lower than 0.7 (70% similar), the probability of the two songs being similar is low, therefore it is unlikely that they will be compared. On the contrary, when the similarity is higher than 0.7 then the probability of the two songs being similar is high and it is quite likely that they will be compared.

The choice of number of bands and rows is an important task in finding candidate pairs with a certain degree of similarity. The length of a MinHash signature also plays a role in this, as the shorter the signature, the fewer the bands and rows to choose from. Vice versa, *t* allows to find the number of bands and rows which ultimately determine the length of a signature needed to find candidate pairs with similarity above an optimal *t*, where there is an optimal balance between FPs and FNs.



A

MH	BANDS	ROW	threshold
600	20	30	0.9
600	50	12	0.7
600	100	6	0.5
600	150	4	0.3
600	200	3	0.2
600	300	2	0.1

B

MH	BANDS	ROWS	SIM	Probability
600	50	12	0.1	4.99989E-11
600	50	12	0.2	2.048E-07
600	50	12	0.3	2.65717E-05
600	50	12	0.4	0.000838516
600	50	12	0.5	0.0121343
600	50	12	0.6	0.103231694
600	50	12	0.7	0.501872171
600	50	12	0.8	0.971553685
600	50	12	0.9	0.999999938
600	50	12	1	1

Figure 12. S-curves for different numbers of rows and bands. The red line represents the similarity threshold t for a number of bands and rows. Table A shows threshold values for different numbers of rows and bands. Table B shows the probability of being candidate pair at the step (or threshold t).

The LSH banding technique can be extended to triplets of songs. Preliminary results are shown in Figure 13. Histogram (A) shows the superimposed distribution of Jaccard and MinHash similarity, whilst plot B shows the S-curve calculated for the triplets using the similarity values obtained from the comparison of triplets MinHash signatures, each signature comprising of three hundred MinHash values. The S-curve was obtained for *ca.* nine million triplets. The signatures were divided into fifty bands comprising each of four rows for a similarity threshold value at *ca.* 0.38.

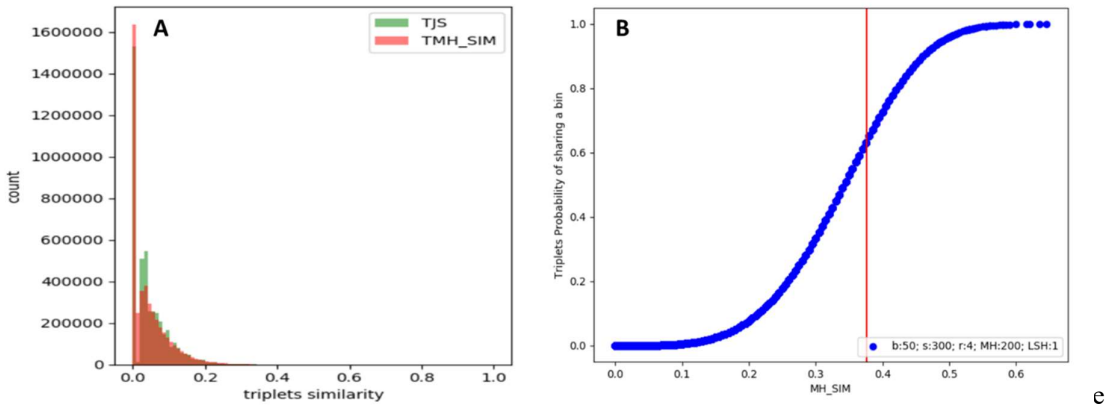


Figure 13. Superimposed histograms of Jaccard and MinHash similarities for songs triplets (A); S-curve for ca. nine million triplets with MinHash signatures comprising of 300 MinHash values each, divided into 50 bands, each band comprising of four rows (B).

Once the number of bands and rows for a threshold similarity *t* is chosen, the optimal number of LSH functions (i.e., hash tables) to be computed for each band needs to be evaluated in order to retrieve as many candidate pairs as possible with the minimum error.

Table 2 shows the metrics (i.e., precision, accuracy, recall and F1-score) for short-signature (i.e., 200 MinHash values) candidate pairs using 1, 3 and 5 LSH functions. The metrics are reported as mean values, calculated across at least five consecutive experiments under the same set of parameters, with the corresponding standard errors. Each short signature was divided into 20, 50 and 100 bands for the evaluation of candidate pairs at a similarity threshold value of *ca.* 0.7, 0.4 and 0.1, respectively.

MH	LSH	band	TP_mean	TP_1SE	TN_mean	TN_1SE	FP_mean	FP_1SE	FN_mean	FN_1SE	Acc_mean	Acc_1SE	Prec_mean	Prec_1SE	Recall_mean	Recall_1SE	F1_mean	F1_1SE	SIM_t
200	1	20	2	0	499420.4	4.377	0	0	77.6	4.377	1	0	1	0	0.03	0.002	0.05	0.003	0.741
200	1	50	36	9.236	467786.6	402.142	0	0	31677.4	399.884	0.94	0.001	1	0	0	0	0	0.001	0.376
200	1	100	66253.5	1716.09	146863.3	1855.119	80	10.501	286303.3	1862.122	0.43	0.004	1	0	0.19	0.005	0.32	0.006	0.1
200	3	10	2	0	499497.5	0.5	0	0	0.5	0.5	1	0	1	0	0.83	0.167	0.9	0.1	0.891
200	3	50	2	0	499425.8	3.72	0	0	72.2	3.72	1	0	1	0	0.03	0.001	0.05	0.003	0.741
200	3	50	2539.4	171.713	466499.8	1209.553	228.6	30.276	30232.2	1080.198	0.94	0.002	0.92	0.011	0.08	0.003	0.14	0.005	0.376
200	3	100	339339	2268.604	103680.6	2307.697	38956.2	1341.834	17524.2	1093.083	0.89	0.001	0.9	0.003	0.95	0.003	0.92	0.001	0.1
200	5	10	2	0	499496.5	0.645	0	0	1.5	0.645	1	0	1	0	0.64	0.131	0.76	0.093	0.891
200	5	20	10.6	1.288	499424.2	2.8	1.6	0.245	63.6	2.064	1	0	0.87	0.015	0.14	0.014	0.24	0.02	0.741
200	5	50	18366.2	662.229	446175.4	2694.977	23838.4	1931.082	11120	208.277	0.93	0.004	0.44	0.013	0.62	0.005	0.51	0.009	0.376
200	5	100	334818.8	3515.8	109162.2	2946.325	35509	1829.659	20010	1529.938	0.89	0.002	0.9	0.004	0.94	0.004	0.92	0.002	0.1

Table 2 . Results of experiments with LSH banding technique for song pairs with short signatures (i.e., 200 MinHash values), carried out with 1, 3 and 5 LSH functions. The standard error is calculated on at least five consecutive experiments.

Figure 14 shows the bar charts (i.e., 1a, 2a, 3a) for the mean number of TPs (blue), TNs (orange), FPs (green) and FNs (read) with the corresponding standard error (i.e., 1 SE) at the relevant threshold values (i.e., *ca.* 0.1, 0.4, 0.7) for 1, 3 and 5 LSH functions, respectively. Figure 14 also shows the corresponding superimposed line plots for precision and recall (i.e., 1b, 2b, 3b), each calculated as the mean value over at least five experiments. Precision and recall are reported with the respective standard errors (1SE). The results show that a trade-off between precision and recall is necessary. Indeed, for one LSH function (and one hash table) the recall is quite low (1b: ca. 0.2; 2b: close to zero; and 3b: ca 0.03) independently of the value of the similarity threshold t, which means that the system returns very few results, with very good precision (1a, 1b). So, in this case, the system is not detecting the majority of candidate pairs (there are many FNs). The problem can be fixed by increasing the number of LSH functions (3 and 5). However, this can affect the precision of the results (i.e., too many FPs). The results indicate that 3 LSH functions are necessary to detect most of the candidate pairs with a good precision.

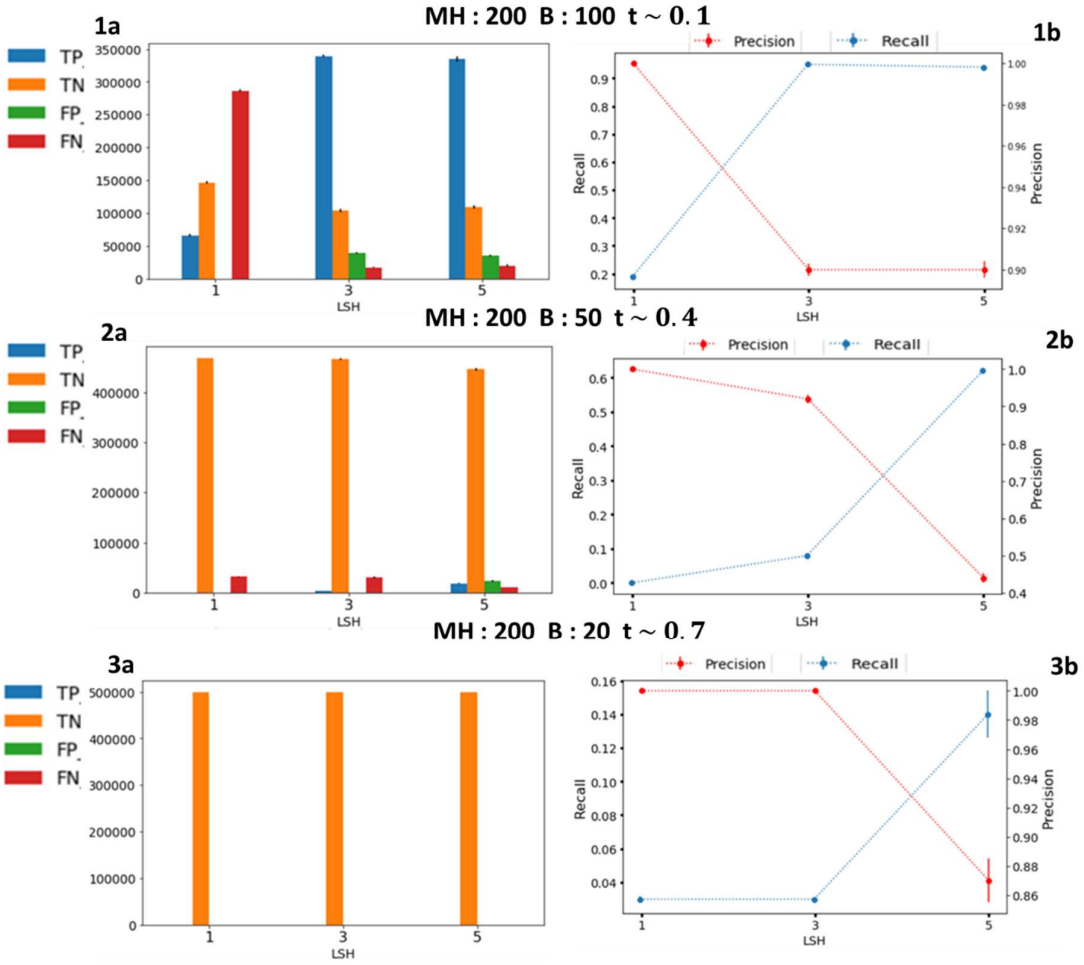


Figure 14. Bar charts and plots for LSH experiments carried out on pairs of songs with short MinHash signatures (i.e., 200 MinHash values) for an increasing number of LSH functions (i.e., 1, 3, 5). Bar charts (1a, 2a and 3a) show TPs, TNs, FPs and FNs for 1, 3 and 5 LSH functions, whilst line plots (1b, 2b and 3b) show the corresponding trade-off between precision and recall.

A series of similar experiments were also carried out for longer MinHash signatures (i.e., 600 MinHash values). The reason for such further experiments is that we know that the longer the MinHash signature, the more accurate the approximated similarity with respect to the ground truth Jaccard similarity. Therefore, it is important to understand whether the previous results can be extended to longer signatures.

Figure 15 shows the results as bar charts (i.e., 1a, 2a, 3a) and line plots (i.e., 1b, 2b, 3b). In this case, the line plots indicate that 3 and 5 LSH functions provide the best chance of the system detecting most of the candidate pairs, given a certain similarity cut off value. The numerical results are also available in Table 3.

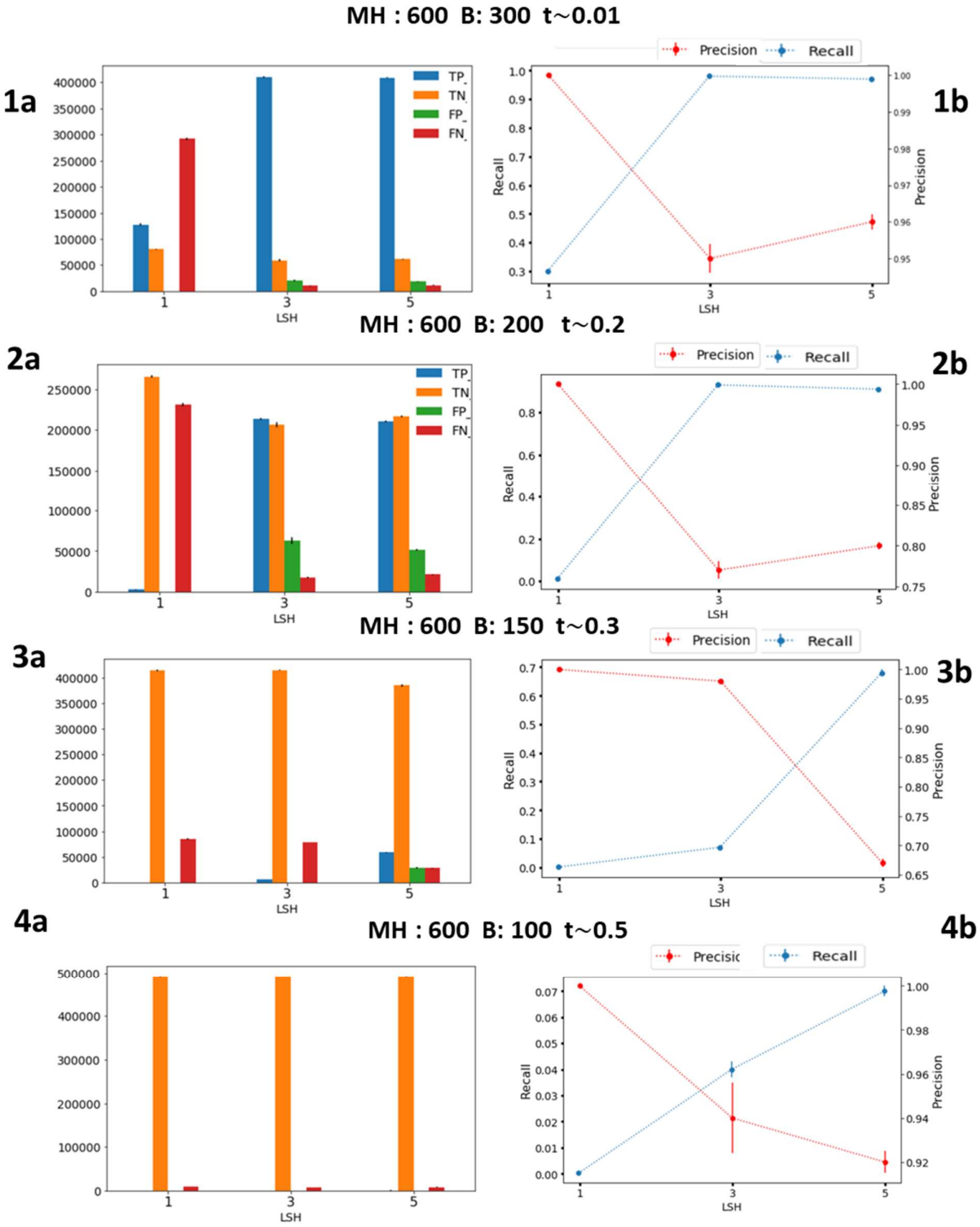


Figure 15. Bar charts and plots for LSH experiments carried out on pairs of songs with long MinHash signatures (i.e., 600 MinHash values) for an increasing number of LSH functions (i.e., 1, 3, 5). Bar charts (1a, 2a, 3a and 4a) show TPs, TNs, FPs and FNs for 1, 3 and 5 LSH functions, whilst line plots (1b, 2b, 3b and 4b) show the corresponding trade-off between precision and recall.

MH	LSH	band	TP_mean	TP_1SE	TN_mean	TN_1SE	FP_mean	FP_1SE	FN_mean	FN_1SE	Acc_mean	Acc_1SE	Precision_mean	Prec_1SE	Recall_mean	Recall_1SE	F1_mean	F1_1SE	SIM_t
600	1	300	127469.6	2758.482	80139.4	327.885	18.4	8.687	291872.6	2866.687	0.42	0.006	1	0	0.3	0.007	0.47	0.008	0.058
600	1	200	2451	160.851	265795	1666.826	0.2	0.2	231253.8	1587.064	0.54	0.003	1	0	0.010479	0.000654	0.02074	0.00128	0.171
600	1	150	77.8	6.719	414433.6	1347.672	0	0	84988.6	1350.253	0.83	0.003	1	0	0.000917	8.50E-05	0.00183	0.00017	0.286
600	1	100	2.2	0.2	490812.2	277.088	0	0	8685.6	277.08	0.98	0.001	1	0	0.000254	2.30E-05	0.00051	4.70E-05	0.464
600	1	50	2	0	499407.6	1.72	0	0	90.4	1.72	1	0	1	0	0.021675	0.000405	0.04243	0.00078	0.722
600	1	30	2	0	499493.5	0.645	0	0	4.5	0.645	1	0	1	0	0.32	0.032	0.48	0.037	0.844
600	3	300	410194	1614.392	59155.8	1877.955	20037.6	1740.122	10112.6	1451.379	0.94	0.001	0.95	0.004	0.98	0.003	0.96	0.001	0.058
600	3	200	213350.8	992.707	206171.8	3163.943	62670.4	3906.789	17307	886.796	0.84	0.006	0.77	0.011	0.93	0.004	0.84	0.005	0.171
600	3	150	6250.2	360.166	414661.4	634.486	131.4	22.787	78457	539.885	0.84	0.001	0.98	0.002	0.07	0.004	0.14	0.007	0.286
600	3	100	352.75	33.51	491383.5	308.381	23.25	7.341	7740.5	273.324	0.98	0.001	0.94	0.016	0.04	0.003	0.08	0.005	0.464
600	3	50	2	0	499407.5	3.175	0	0	90.5	3.175	1	0	1	0	0.02	0.001	0.04	0.002	0.722
600	3	30	2	0	499493.8	0.401	0	0	4.17	0.401	1	0	1	0	0.33	0.023	0.5	0.025	0.844
600	5	300	409038	1143.087	61052	828.409	18521	781.6	10889	824.93	0.94	0.001	0.96	0.002	0.97	0.002	0.97	0.001	0.058
600	5	200	210291	1122.959	216704.6	594.848	51404	1152.937	21100.43	733.962	0.85	0.001	0.8	0.004	0.91	0.003	0.85	0.002	0.171
600	5	150	58426.5	1299.055	384587.3	1911.39	28608.75	1506.528	27877.5	885.774	0.89	0.001	0.67	0.007	0.68	0.012	0.67	0.003	0.286
600	5	100	563.6	31.101	490894.8	345.241	46.6	4.792	7995	319.984	0.98	0.001	0.92	0.005	0.07	0.002	0.12	0.004	0.464
600	5	50	2.25	0.25	499406.5	2.901	0	0	91.25	3.065	1	0	1	0	0.02	0.003	0.05	0.006	0.722
600	5	30	2.17	0.167	499494.2	0.601	0	0	3.67	0.667	1	0	1	0	0.39	0.055	0.56	0.055	0.844

Table 3 . Results of experiments carried out with LSH banding technique for song pairs with long signatures (i.e., 600 MinHash values) with 1, 3, 5 LSH functions. The standard error is calculated on at least five consecutive experiments.

Time and memory of computation were also evaluated for 1, 3, and 5 LSH functions for both short signatures (i.e., 200 MinHashes) and long signatures (i.e., 600 MinHashes). The results in Figure 16 show that both time and memory increase linearly with the number of LSH functions employed.

Moreover, memory and time depend also on the size of the MinHash signature. Indeed, the larger the signature, the longer the time and the larger the memory needed for computation. For example, for short signatures and one LSH function, computation takes ca. 80 seconds and ca five gigabyte, versus ca. 200 seconds and ca. 8 gigabytes for longer signatures. Similar results were found for 3 and 5 LSH functions.

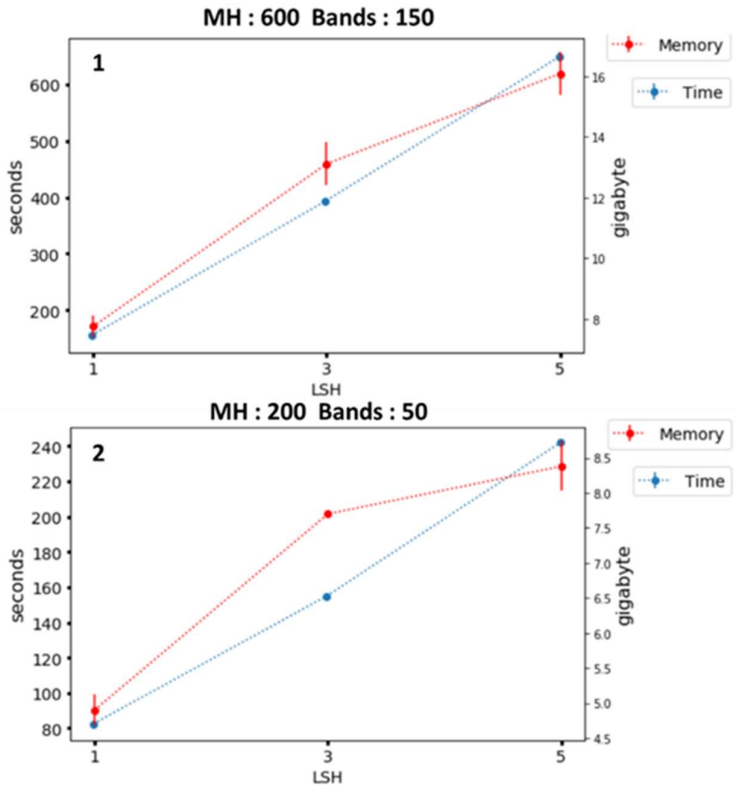


Figure 16. Line plots of memory and time against number of LSH functions. Plot 1 shows the results for a long signature, whilst plot 2 shows the results for a short signature.

3.4 Summary

In conclusion, this chapter shows the implementation of similarity searches for Jamendo songs pairs via MinHash and LSH techniques. To start with, the Jaccard and weighted Jaccard similarities for pairs and triplets of songs were computed and compared. The simple Jaccard similarity was chosen as a benchmark for the successive steps of this work as it is computationally faster than the weighted Jaccard similarity.

In the successive step, the signature of each song was implemented via the MinHash technique and the approximated MinHash similarity computed in order to verify whether the signatures would preserve the Jaccard similarity of any pair of songs. The accuracy and precision of the results were evaluated by comparing the approximated similarity against the ground truth Jaccard similarity. It was found that the larger the signature, the better the approximation, however, the larger the signature, the less efficient the computation.

When the MinHash technique is used solely for the purpose of measuring the degree of similarity between two documents, its efficiency highly depends on the level of sparseness of the data and the number of shingles it is applied to. We know that the MinHash technique works efficiently by drastically reducing the dimension of multivariate and highly sparse datasets.[13] This does not seem to be the case for our dataset, due to the limited number of shingles used to represent the songs (i.e., sixty shingles, each of length one), leading to a low degree of sparseness of data in the characteristic matrix.

However, our aim was to use the MinHash technique as a prior step to the use of the Locality Sensitive Hashing banding technique. The main point is to reduce the number of pairs to be compared, therefore improving the overall efficiency of the process.

In the last section, the search of candidate pairs via the LSH banding technique was implemented on the songs' signatures, thus decreasing the number of possible pairs to be compared. The process is faster for short signatures and a trade-off between precision and recall is possible by choosing the number of LSH functions to compute.

Chapter 4. Conclusion and future work

4. 1 Conclusion

The aim of this work was to find similar songs in a large dataset. The similarity search, carried out for pairs and triplets of songs was implemented in the following steps:

- Sixty chord ratios were chosen as feature vectors to represent each Jamendo song. Each chord ratio was used as a shingle of length one and used to build a characteristic matrix.

- The Jaccard similarity was evaluated for songs pairs and triplets as the intersection over the union of binary sets of chord ratios.

- The weighted Jaccard similarity was also computed for songs pairs and triplets taking into consideration both chord ratios indices and values.

- The computational efficiency of the Jaccard similarity and weighted Jaccard similarity was studied. It was found that the simple Jaccard similarity was computationally more efficient than the weighted Jaccard similarity. For that reason, the simple Jaccard similarity was chosen as a benchmark value to be compared with the MinHash approximated similarity.

- The MinHash technique was used to create songs signatures from the characteristic matrix, to be used as inputs in the implementation of the Locality Sensitive Hashing technique. To make sure that the similarity among any pairs of songs was preserved after MinHashing, we computed the MinHash approximated similarity for pairs and triplets of signatures and compared them with the ground truth Jaccard similarity. An estimate of the Jaccard similarity was found for two hundred MinHash functions per signature. A close estimate was found for six hundred MinHash functions.

- The Locality Sensitive Hashing technique was implemented as the last step of this work. Such technique was useful to narrow down the number of songs pairs to be compared to those with similarity above a certain similarity threshold, thus making the similarity search faster.

In this work, we were able to implement Locality Sensitive Hashing via MinHash technique. The results were reported in terms of accuracy, precision, recall and F1-score and were averaged over at least five consecutive experiments under the same conditions and parameters.

Time and memory were also evaluated at each step of this work. The main finding is that Locality Sensitivity Hashing is indeed effective in finding candidate pairs. Time complexity of similarity search decreases using Locality Sensitivity Hashing, as there are fewer pairs to be compared.

LSH reduces remarkably the time complexity of a similarity search via MinHash. Indeed, the time of MinHash similarity computation decreases by over 60% after LSH.

However, the time complexity for the Jaccard and the MinHash similarity after LSH is similar. This is due to the choice of representing each song with shingles of length one, meaning that each chord ratio is a shingle. This choice leads to a low sparsity of the characteristic matrix. This statement is based on the fact that we started with sixty shingles of length one, and we ended up after MinHash with at least 200 MinHash values per signature. This was necessary in order to best approximate the MinHash similarity to the Jaccard similarity. Therefore, we increased the number of elements in each song array from 60 to (at least) 200. Hence, in conclusion the data were not compressed, actually quite the opposite. The MinHash technique is designed to compress highly sparse data. For example, in textual documents where every k consecutive character is a shingle, there are hundreds or thousands of shingles, leading to a highly sparse characteristic matrix.

In conclusion, we implemented a similarity search procedure based on LSH via MinHash technique for the Jamendo songs. The MinHash technique was used to bridge the Jaccard similarity to the Locality Sensitivity Hashing technique.

4.2 Future work

In this work we evaluated the similarity for songs pairs and triplets. In terms of future work, scale up of shingles for songs pairs should be considered, as this would remarkably reduce the space and time complexity of the similarity search. The same approach .should be extended also to songs triplets

In order to solve the issue of the low sparsity of the characteristic matrix, it would be necessary to scale up the number of shingles. In this regard, it would be interesting to consider sequences of chords instead of single chord ratios as shingles.

For example, if we consider each shingle of length 2 as the combination of two distinct chord ratios, then the number of possible combinations would lead to 3,540 shingles instead of 60. If we consider a sequence of three chords (each shingle with length three), we would obtain ca. 200 thousand shingles. This high number of shingles would result in a sparse characteristic matrix and its compression with 200 or 600 MinHash functions would become effective leading to quite remarkable results.

Increasing the length of the shingles would allow to evaluate whether two or more songs have the same chords in the same sequence. This would be useful for duplicates detection.

For a larger number of shingles, the tuning of the number of MinHash functions could become key in near duplicates or duplicates detection. For larger MinHash signatures, the LSH technique would allow to find duplicate candidate pairs, whilst for shorter MinHash signatures it would allow to find both duplicates and near duplicates pairs.

It would also be interesting to study in more detail the dissimilarity of songs triplets. As each triplet comprises of three songs, it would be interesting to find three songs that are all dissimilar to each other.

References

[1] Hujran, O., Alikaj, A., Durrani, U.K. and Al-Dmour, N., 2020, January. Big Data and its Effect on the Music Industry. In *Proceedings of the 3rd International Conference on Software Engineering and Information Management* (pp. 5-9).

[2] Techcrunch (2020). Streaming services accounted for nearly 80% of all music revenue in 2019. Available at: https://techcrunch.com/2020/02/26/streaming-services-accounted-for-nearly-80-of-all-music-revenue-in-2019/?guccounter=1&guce_referrer=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvbS8&guce_referrer_sig=AQAAAIy2-J2VWHHb6xphnm-3QapOdi2zfevE2wlQLJg8E5rcTpAMfGYSWbixSmOpB6p9X_29ACVEUebqCMOt9ZD5awUhKO_XR7LDKhDQ-UTCqD1p4F6zrckHMZYbj_m5p6E-Iro77OlKDgMU4mayr190V_MvWyaYBTBa8k-DmUMeWQqB (Accessed: 10 October 2020).

[3] Digital Initiative (2017) *Digitalisation radically changes the music industry*. Available at: https://digital.hbs.edu/platform-rctom/submission/digitalization-radically-changes-the-music-industry/ (Accessed: 09 October 2020).

[4] Ascarza, E., Neslin, S.A., Netzer, O., Anderson, Z., Fader, P.S., Gupta, S., Hardie, B.G., Lemmens, A., Libai, B., Neal, D. and Provost, F., 2018. In pursuit of enhanced customer retention management: Review, key issues, and future directions. *Customer Needs and Solutions*, *5*(1-2), pp.65-81.

[5]Business of apps (2020). Spotify Usage and Revenue Statistics. Available at: https://www.businessofapps.com/data/spotify-statistics/#4 (Accessed: 10 October 2020).

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

[7] Wang, A., 2003, October. An industrial strength audio search algorithm. In *Ismir* (Vol. 2003, pp. 7-13).

[8] Haitsma, J. and Kalker, T., 2002, October. A highly robust audio fingerprinting system. In Ismir (Vol. 2002, pp. 107-115).

[9] Baluja, S. and Covell, M., 2006. Content fingerprinting using wavelets.

[10] Rackspace Developers, Steve Tjoa (2014). *Music Information Retrieval Using Locality Sensitive Hashing*. Available at: https://www.youtube.com/watch?v=SghMq1xBJPI (Accessed: 11 October 2020).

[11] Cormen, T.H., 2013. *Algorithms unlocked*. Mit Press.

[12] Casey, M.A. and Slaney, M., 2006, October. Song Intersection by Approximate Nearest Neighbor Search. In ISMIR (Vol. 6, pp. 144-149).

[13] Leskovec, J., Rajaraman, A. and Ullman, J.D., (2020) "Finding similar items". Mining of massive data sets. Cambridge: Cambridge university press, pp.73-125.

[14] Broder, A.Z., 1997, June. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)* (pp. 21-29). IEEE.

[15] Pauwels, J., O'Hanlon, K., Fazekas, G. and Sandler, M.B., 2017. Confidence Measures and Their Applications in Music Labelling Systems Based on Hidden Markov Models. In ISMIR (pp. 279-285).

[16] Pauwels, J. and Sandler, M.B., 2019. Finding new practice material through chord-based exploration of a large music catalogue.

[17] Jamendo Music (no date) About us. Available at https://www.jamendo.com/about/ (Accessed: 14 October 2020).

[18] InfoWorld, Ian Pointer (2020) What is Apache Spark? The big data platform that crushed Hadoop. Available at https://www.infoworld.com/article/3236869/what-is-apache-spark-the-big-data-platform-that-crushed-hadoop.html#:~:text=Berkeley%20in%202009%2C%20Apache%20Spark,machine%20learning%2C%20and%20graph%20processing (Accessed: 14 October 2020).

[19] Apache Spark 2.4.3 (no date) Available at https://spark.apache.org/docs/2.4.3/sql-programming-guide.html (Accessed: 14 October 2020).

[20] Carpineto, C., Osiński, S., Romano, G. and Weiss, D., 2009. A survey of web clustering engines. *ACM Computing Surveys (CSUR)*, *41*(3), pp.1-38.

[21] Gupta, M.K. and Chandra, P., 2020. A comprehensive survey of data mining. *International Journal of Information Technology*, pp.1-15.

[22] Mehta, V., Bawa, S. and Singh, J., 2020. Analytical review of clustering techniques and proximity measures. *ARTIFICIAL INTELLIGENCE REVIEW*.

[23] Wang, C., Song, Y., Li, H., Zhang, M. and Han, J., 2015, November. Knowsim: A document similarity measure on structured heterogeneous information networks. In *2015 IEEE International Conference on Data Mining* (pp. 1015-1020). IEEE.

[24] Li, L., Zheng, L., Yang, F. and Li, T., 2014. Modeling and broadening temporal user interest in personalized news recommendation. *Expert Systems with Applications*, *41*(7), pp.3168-3177.

[25] Feng, C., Khan, M., Rahman, A.U. and Ahmad, A., 2020. News Recommendation Systems-Accomplishments, Challenges & Future Directions. *IEEE Access*, *8*, pp.16702-16725.

[26] Hoad, T.C. and Zobel, J., 2003. Methods for identifying versioned and plagiarized documents. *Journal of the American society for information science and technology*, *54*(3), pp.203-215.

[27] Gurjar, K. and Moon, Y.S., 2018. Comparative Analysis of Music Similarity Measures in Music Information Retrieval Systems. *Journal of Information Processing Systems*, *14*(1).

[28] Gyawali, B., Anastasiou, L. and Knoth, P., 2020. Deduplication of Scholarly Documents using Locality Sensitive Hashing and Word Embeddings.

[29] Kakulapati, V., Kolikipogu, R., Revathy, P. and Karunanithi, D., 2011, July. Improved Web Search Engine by New Similarity Measures. In *International Conference on Advances in Computing and Communications* (pp. 284-292). Springer, Berlin, Heidelberg.

[30] Ture, M., Kurt, I. and Akturk, Z., 2007. Comparison of dimension reduction methods using patient satisfaction data. *Expert Systems with Applications*, *32*(2), pp.422-426.

[31] Broder, A.Z., Charikar, M., Frieze, A.M. and Mitzenmacher, M., 2000. Min-wise independent permutations. *Journal of Computer and System Sciences*, *60*(3), pp.630-659.

[32] Gionis, A., Indyk, P. and Motwani, R., 1999, September. Similarity search in high dimensions via hashing. In *Vldb* (Vol. 99, No. 6, pp. 518-529).

[33] Xia, D., Wang, B., Li, Y., Rong, Z. and Zhang, Z., 2015. An efficient MapReduce-based parallel clustering algorithm for distributed traffic subarea division. *Discrete Dynamics in Nature and Society*, *2015*.

[34] Wei, P., He, F., Li, L., Shang, C. and Li, J., 2020. Research on large data set clustering method based on MapReduce. *Neural Computing and Applications*, *32*(1), pp.93-99.

[35] Jaccard, P., 1901. Distribution de la flore alpine dans le bassin des Dranses et dans quelques régions voisines. *Bull Soc Vaudoise Sci Nat*, *37*, pp.241-272.

[36] Besta, M., Kanakagiri, R., Mustafa, H., Karasikov, M., Rätsch, G., Hoefler, T. and Solomonik, E., 2020, May. Communication-efficient jaccard similarity for high-performance distributed genome comparisons. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (pp. 1122-1132). IEEE.

[37] Chung, N.C., Miasojedow, B., Startek, M. and Gambin, A., 2019. Jaccard/Tanimoto similarity test and estimation methods for biological presence-absence data. *BMC bioinformatics*, *20*(15), p.644.

[38] Verma, V. and Aggarwal, R.K., 2020. A comparative analysis of similarity measures akin to the Jaccard index in collaborative recommendations: empirical and theoretical perspective. *Social Network Analysis and Mining*, *10*, pp.1-16.

[39] Niwattanakul, S., Singthongchai, J., Naenudorn, E. and Wanapu, S., 2013, March. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists* (Vol. 1, No. 6, pp. 380-384).

[40] Levandowsky, M. and Winter, D., 1971. Distance between sets. *Nature*, *234*(5323), pp.34-35.

[41] McCune, B., Grace, J.B. and Urban, D.L., 2002. *Analysis of ecological communities* (Vol. 28). Gleneden Beach, OR: MjM software design.

[42] Xambó, A., Pauwels, J., Roma, G., Barthet, M. and Fazekas, G., 2018. Jam with Jamendo: Querying a Large Music Collection by Chords from a Learner's Perspective. In *Proceedings of the Audio Mostly 2018 on Sound in Immersion and Emotion* (pp. 1-7).

[43] Chierichetti, F., Kumar, R., Pandey, S. and Vassilvitskii, S., 2010, January. Finding the jaccard median. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms* (pp. 293-311). Society for Industrial and Applied Mathematics.

[44] Altman, D.G. and Bland, J.M., 2005. Standard deviations and standard errors. *Bmj*, *331*(7521), p.903.

[45] Li, P. and König, A.C., 2011. Theory and applications of b-bit minwise hashing. *Communications of the ACM*, *54*(8), pp.101-109.

[46] Manaa, M.E. and Abdulameer, G., 2018. Web Documents Similarity using K-Shingle tokens and MinHash technique. J. Eng. Appl. Sci, 13, pp.1499-1505.

[47] Aksakalli, C.G. and Welke, P., 2016. Minhashing for Graph Similarity Computation. *Proc edings of the 3rd CSCUBS.*

[48] Henzinger, M., 2006, August. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval* (pp. 284-291).

[49] Bayardo, R.J., Ma, Y. and Srikant, R., 2007, May. Scaling up all pairs similarity search. In *Proceedings of the 16th international conference on World Wide Web* (pp. 131-140).

[50] Indyk, P. and Motwani, R., 1998, May. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing* (pp. 604-613).

[51] Indyk, P., Motwani, R., Raghavan, P. and Vempala, S., 1997, May. Locality-preserving hashing in multidimensional spaces. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (pp. 618-625).

Glossary

Jaccard similarity: It is a set-based index used to measure the similarity between finite sets of documents. It calculated as the size of the intersection over the size of union of the two sets of documents.

Weighted Jaccard similarity: It is a vector-based index used to measure the similarity between two vectors. It is calculated as the intersection of two vectors over their union.

MinHash: It is a hash function defined on two/more sets such that the collision probability of such sets is approximately the Jaccard similarity.

MinHash similarity: It approximates the Jaccard similarity.

Locality Sensitive Hashing (LSH): It is a technique that hashes similar items to the same bucket with high probability. It is used to speed up the search of similar pair of documents.

k-Shingles: It is a sequence of k- tokens, (e.g., strings of length k, k-words, or else depending on the applications).

Bucket: It is an integer number obtained by hashing data points.

Music information retrieval (MIR): It is the science of retrieving information from music.

Pyspark: It is a software package dedicated to the computation of Big Data.

Appendix 1: Project Proposal E. Lestini

Finding similar songs in a large music collection based on their chord content.

1. Introduction

This work aims to find similar music pieces in a large collection of music based on their chord content, chords sequence and other chord-related features (e.g. chords ratios, confidence interval) [1] using similarity techniques for massive datasets [2]. String representation of chords names will be necessary in order to apply information retrieval techniques. Textual similarity will be evaluated by the Jaccard similarity metric of the characteristic matrix of the k-shingled musical pieces [2]. To compare every single pair of documents would be a naïve approach to such algorithm. Indeed, for a big number of documents this approach is not feasible due to the long time of execution and too much memory space needed, leading to poor performance (Runtime), as described by the big O notation (for N documents, N choose 2 complexity or combinations). Due to a high number of inputs the function does not scale well, and effective indexing and filtering techniques are needed to overcome the scalability problem.

1.1 Characteristic and signature matrices

A characteristic matrix of k-shingled musical pieces comprises of all possible unique sets of consecutive substrings of length k found in a document. We will now consider a simple string to see how this works in practical terms. The string *'On Monday people are back to work'* gets shingled into a list of unique two-word-long (e.g. k =2) substrings (known as shingles; in this case as k=2 we obtain 2-shingles) as follows: *'on monday', 'monday people', 'people are', 'are back', 'back to', 'to work'*. It should be noted that identical shingles will appear only once in the set of 2-shingles (k=2). In this example there are no duplicates, so all these shingles will be used to build a '*characteristic matrix'*. If we have several documents and we want to investigate their relationship with regard to the above set of 2-shingles, we can create a *'characteristic matrix'*. The matrix rows comprise of all the possible 2-shingles sets across all documents and the columns represent the documents (one column per document). To represent this collection of sets we can use a metric representation of binary vectors (1s and 0s). The characteristic matrix (example in Figure 1) is therefore a Boolean matrix, since it is populated only by 0s and 1s. For example, 1 in row *'i'* and column *'j'* means that the shingle *'i'* is present in the document *'j'*, on the contrary 0 means that shingle *'i'* is not present in document *'j'*.

Document / Shingle	Document _1	Document_2	Document_3	Document_4
on monday	0	1	1.	1
monday people	0	1	0	1
people are	0	1	0	1
are back	1	0	0	1
back to	1	1	0	1
to work	0	0	1	1
...	...	...	...	...
jennifer and	1	1	1	0
and i	1	0	1	0
i plan	1	1	1	0

Figure 1. Example of characteristic matrix. Such matrix is usually sparse as most shingles do not appear in all documents.

To find the similarity of sets we compare the documents present in a characteristic matrix pair-wise, checking which sets of shingles are present in both documents. The rows where 1s appear show which shingles are present in a document. The Jaccard similarity coefficient (known also as Jaccard Index) provides a measure of similarity between two documents. For example, for document_1 and document_2 (Figure 1, column1 and 2, respectively) in the characteristic matrix (Figure 1) we look at the intersection (AND operation) and union (OR operation) of their shingle sets. The Jaccard similarity coefficient is obtained as the ratio between intersection and union for two documents. Figure 1 shows that the first shingle *'on monday'* is present in document_2 but not in document_1. Therefore, the intersection is zero (the documents do not have *'on monday'* in common), while the union is one as such shingle is present in one of the two documents. This is computed for all the sets of shingles in a characteristic matrix and a measure of similarity for document_1 and document_2 is found.
As a characteristic matrix represents all the possible combinations of sets of k-shingles across all documents in a dataset, the columns are usually very long and the size (number of columns x number of rows) of the characteristic matrix is large. The similarity search for a large collection of documents becomes impractical. Therefore, a shorter representation is needed.
Data compression of the large characteristic matrix to a smaller signature matrix, preserving the characteristic of the main sets in the characteristic matrix, can be achieved via Min Hashing technique [2, 3] while investigating the similarity of subsets that are most likely to be similar pairs. In practice, by using Locality-Sensitive Hashing (LSH) it is possible to reduce the search to only potentially useful pairs (called candidate pairs) [2].
The results (accuracy, precision and recall and computing time) obtained for a different number of documents using the Jaccard similarity as metric and high dimensional data similarity techniques (Min Hashing, LSH) will be evaluated and compared to the results obtained for a large characteristic matrix of hashed k-shingles. It would also be interesting to investigate how varying the trade-off between precision and recall can speed up the similarity search.
This work aims to provide a methodology based on scalable similarity techniques [2, 3] for a non-memory intensive songs similarity search based on their chords and other chord-related features [1].

1.2 The questions to address in the thesis are the following,

Can data mining similarity techniques and distance measurements allow to find similar songs in a scalable manner? Do we need to compare all the documents available in a catalogue in a pair-wise fashion or can we just focus on those that are most likely to be similar making the process computationally efficient, avoiding the bottleneck of scaling?

2. Critical Context

Identification of duplicate and near-duplicate documents is one of the biggest issues for large-scale digital collections [4, 5, 6]. A simple search on the Google (search) engine on the topic *'near duplicate documents collections'* provides nearly forty-three million results in 0.54 seconds. When the search is narrowed down to music collections the number of results is still staggering (over fourteen million). This is a hot topic in the field of Musical Information Retrieval (MIR) and Musical Industry (MI), for several reasons.
The presence of near duplicates in a large collection leads to an undesired increase in processing overheads and storage [7]. Near-duplicates in a collection can originate from non-integrated heterogeneous document sources (e.g. the same document has a different data

structure depending on the source), from mirror web pages, that differ from each other only in a small part of their content (e.g. different banners) [8], from plagiarism [9], etc.
Detection of near duplicates helps improve the quality and diversity of the output from a search query, thus making users' experience more pleasant and the search more effective.[10, 11] Therefore, detection of near duplicates can improve accuracy and efficiency of a search query, detect fraudulent activity (e.g. plagiarism) but it can also be used to improve the efficiency of recommender systems [12]. In order to avoid duplicates, online music platforms such as Spotify have implemented a system that allows an artist to upload his/her music directly, blocking third parties from doing so [13].

3. Approaches: Methods & Tools for Design, Analysis & Evaluation

3.1 Literature Survey

The literature survey was carried out using the City University Library, Google Scholar search engines. CiteThemRight was used to check the citation and reference Harvard style. The main textbook used in the Big Data module was used as reference guide to understand the algorithms used in Big Data on the topic of documents similarity. The literature survey will be further expanded at a chapter level during the course of the thesis.

3.2 Data Source The original music pieces are available on Jamendo.com, a large music catalogue ("half a million free songs from 40000 artists around the world") [14, 15]. The chord-based data used in this work originate from a chord-based transcription of the original songs available on Jamendo.com, using an automatic chord transcription algorithm that provides as output chords, chord sequence and a related measure of confidence for such chords' labels [1, 14]. The songs are available from a dataset with a JSON structure where songs are organised in a list of dictionaries, one dictionary per song. Each dictionary comprises of nested dictionaries which include song chords, chord ratio, chord sequence, distinct chords number, duration of the song (seconds) and a measure of confidence for the quality of chords sequence. The dataset is available from Dr Johan Pauwels' personal MongoDB database.

3.3 Software and MongoDB-Spark Pipeline

A Spark Connector will be used to read the database into a Spark DataFrame [16]. Installation of Anaconda navigator, PySpark and PixieDust for Windows is completed. The MongoDB and Spark connector implementation developed by Dr Johan Pauwels is provided and ready to use.
I will use GPU and Big Data servers (possibly at City University or renting Google Cloud or AWS). The work will be stored in a GitHub repository or/and Google Drive folder shared with the supervisor. The notebooks will be run from GitHub, Colab from Google drive and locally.

3.4 Similarity System Development

Implementation of the similarity search will be broken down as follows:
3.4.1 Data pre-processing, features selection and manipulation.
3.4.2 *k*-Shingle (or *k*-gram) the documents (songs) for different shingle sizes (e.g. k=5, 10) and for a given number of songs (e.g. 300). As the dataset comprises of both numerical data (chords ratios, chords sequence, and duration) and text data (chords names), obtaining a *k*-gram profile of the chord sequence of each song could be beneficial to this work. For example, a *k*-gram profile for the chord sequence could be obtained using a similar approach to that already reported in literature where each *k*-gram in the profile could be time or beat weighted [17].
3.4.3 Create the *'characteristic matrix'* (Boolean matrix) where the rows represent all the possible combinations of *k*-shingles sets across all the songs and each column represents one song.

3.4.4 Calculate the Jaccard similarity using the *'characteristic matrix'* and determine the similarity between songs. This is feasible for a small number of songs while it would become computationally inefficient for an increasing number of songs. Therefore, it should not scale well (big O notation: for N documents, N choose 2 combinations).
3.4.5 Use the Min Hashing technique using several hash functions (e.g. 100) to generate a signature matrix from the *'characteristic matrix'*. The signature matrix is smaller than the *'characteristic matrix'* as it maps only the 1s in the *'characteristic matrix'*, that is usually sparse, preserving the similarity between documents.
3.4.6 Calculate the 'approximate' Jaccard similarity from the signature matrix obtained by Min Hashing the *'characteristic matrix'*. The probability of the Minhash signature values for two columns is similar to the Jaccard similarity of the columns of the sets they represent due memory requirement.
3.4.7 Compare the Jaccard similarity for the *'characteristic matrix'* with the approximated Jaccard similarity of the signatures. Pair-wise comparison of the songs' signatures should improve performance but still should not be memory efficient; Next step is narrowing the similarity search to the best candidate pairs.
3.4.8 Locality Sensitive Hashing (LSH) technique to find 'candidate pairs'. Partition the signature matrix obtained from the Min Hashing technique into b bands. For each band, hash the portion of the columns (r) in such band to a hash table with k buckets. When hashed subsets for two columns end up in the same hash, these columns are good candidates for comparison as the columns have good probability to be similar in the entire signature as well. Indeed, good candidate pairs hash at least once to the same bucket. Set a similarity threshold (t) for two sets to be hashed in the same bucket. Ideally, the threshold value should result in a sharp cut-off point for good candidate pairs, meaning that songs that are falling above the threshold are highly likely to be similar. In reality, there is no sharp cut-off point, but a sigmoid curve (S-curve) whose slope is function of the chosen number of bands (b) and of the rows in each band (r). Therefore, by tuning the number of signatures coming in from the Min Hash signature matrix M, the number of bands (b) and the number of rows per band (r), a balance between rates of False Positive and False Negative can be achieved.
3.4.9 Scale up: Repeat the procedure (from step 3.4.2 to 3.4.8) for a higher number of songs (e.g. 1000) and evaluate the results.

3.5 Results Evaluation

The Jaccard similarities for pair of songs will be evaluated for the *'characteristic matrix'* and compared to the 'approximated' Jaccard similarities for the Min Hashed signature matrix. The accuracy of the Minhash signature will be compared to that of the *'characteristic matrix'* (step 3.4.3) and the trade-off length of signature vs accuracy should be investigated. The accuracy will be expressed as confusion matrix, from which precision and recall can also be derived.
Analysis of the LSH results will be carried out for the false positive candidate pairs checking whether they are actually similar in the full representation of the Minhash signature matrix (3.4.5). For candidate pairs that are similar, their similarity can be evaluated in the *'characteristic matrix'* of hashed shingles (step 3.4.2).
This part of the work will be carried out during two separate weeks. The results (i.e. small number of songs) will be evaluated first at the start of the 5th week, whilst the results from the scale up will be evaluated at a later stage (9th week).

3.6 Complete Report

The introduction and backbone of the report will be written in the first two weeks of the thesis. The results and evaluation of the results will be added as soon as they are obtained. In this respect an agile method will be used: early results on small batches of songs (e.g. 300) will be

added and discussed first, giving shape to the thesis. The procedure developed for a small batch will be then scaled up (e.g. applied to 1000 songs) and the results will be added at a later stage (six weeks after the start of the thesis), allowing time for discussion with the supervisor and review of the work.

3.7 Additional Time Allowance

One week at the beginning of the thesis will be used for ensuring that the MongoDB and PySpark connectors are working well and the database can be easily accessed, and the songs retrieved. This time will be used for data pre-processing and feature selection and manipulation. The project kick-off meeting with the supervisor should take place at the beginning of the first week the project commences. The kick-off meeting is likely to take place on-line (not face-to-face) due to lockdown and Covid-19 pandemic. After the kick-off meeting some time will be also used to exchange emails with the supervisor (e.g. review part of the research tasks, technical problems).

3.8 Project Meetings

One kick-off meeting and four 'progress meetings' and are planned to take place during the 12-week project period commencing from the 1st of June 2020. Progress meetings scheduled roughly every fortnight, as an opportunity to discuss with the supervisor the work done and results obtained, possible problems and blockers encountered and how they were resolved/how they could be resolved. These meetings should help keep the project on track and solve problems/blockers at an early stage.

3.9 Risks Table 1 (risks register) shows the possible risks that could arise during the thesis. At each risk there is associated a range of values for likelihood (1-3), consequences (1-5) and impact (likelihood x consequences). The last column shows the possible solution/plan B in respect of these risks.

Description	Likelihood (1-3)	Consequence (1-5)	Impact (LxC)	Mitigation
Raw songs data cannot be accessed	2	5	10	Ask the supervisor to share a batch of 1000 songs in a JSON file
Code difficult to write/does not work as expected	2	5	10	Ask Johan Pauwels for a code review or for resources as soon as the problem arise
Data pre-processing and features manipulation takes long time	2	4	8	Start this task early. Discuss with Johan Pauwels possible solutions/shortcuts.
Scale up takes long	3	4	12	Use City University Server (GPU)/ use a smaller number of data than planned.
City University servers are busy/down/no access	2	4	8	Use GPU on Google Cloud/Ask Johan Pauwels if other options are available/pay as you go server.
Unexpected results	2	3	6	Find a reasonable explanation/ discuss with Johan Pauwels in project meetings.
Files loss/damaged	2	5	10	Back up the work every time there is a progress on a GitHub repository/Google server.
Sickness	2	5	10	Discuss with Johan Pauwels/apply for an extension if that applies.

Table 1. Risk register

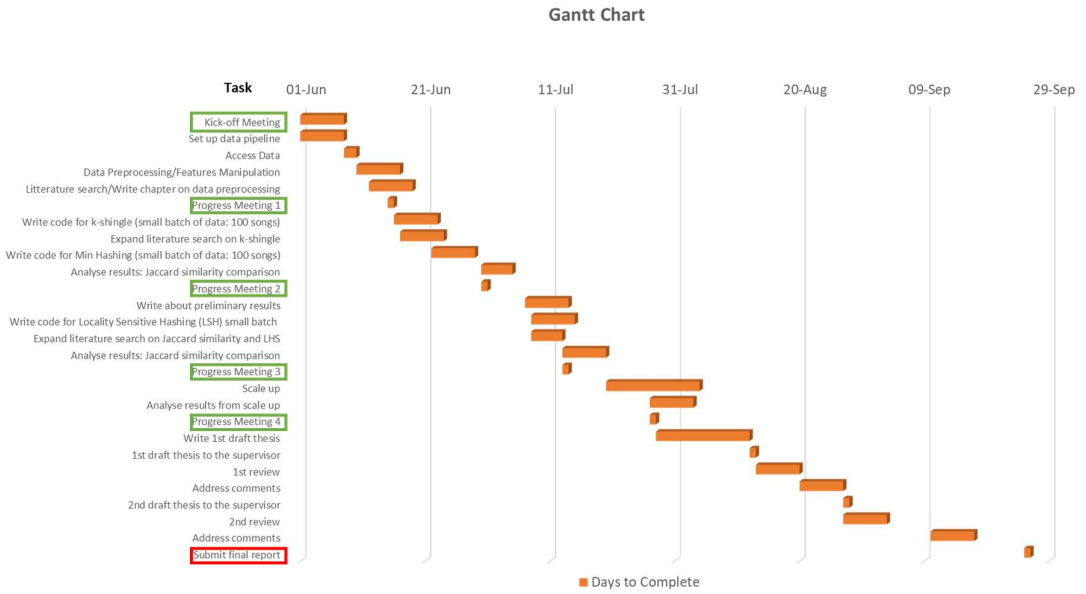


Figure 2. Project work plan.

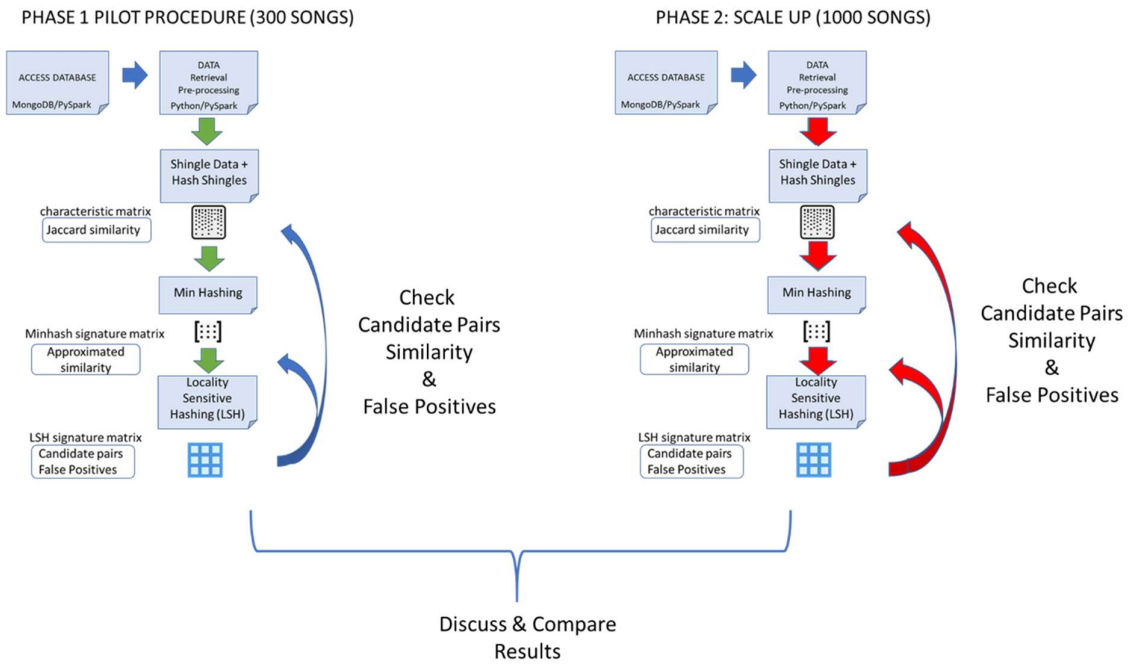


Figure 3. Work scheme.

References

[1] Pauwels, J., O'Hanlon, K., Fazekas, G. and Sandler, M.B., 2017. Confidence Measures and Their Applications in Music Labelling Systems Based on Hidden Markov Models. In *ISMIR* (pp. 279-285).
[2] Leskovec, J., Rajaraman, A. and Ullman, J.D., (2020) 'Finding similar items'. *Mining of massive data sets*. Cambridge: Cambridge university press, pp.73-125.
[3] Broder, A.Z., 1997, June. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)* (pp. 21-29). IEEE.
[4] Varol, C. and Hari, S., 2015. Detecting near-duplicate text documents with a hybrid approach. *Journal of Information Science*, *41*(4), pp.405-414.
[5] Xiao, C., Wang, W., Lin, X., Yu, J.X. and Wang, G., 2011. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, *36*(3), pp.1-41.
[6] Rezaeian, N. and Novikova, G.M., 2017. Detecting near-duplicates in russian documents through using fingerprint algorithm Simhash. *Procedia Computer Science*, *103*(C), pp.421-425.
[7] Manning, C.D. Raghavan, P. and Schütze, H. 2008. *Introduction to Information Retrieval*. Cambridge: Cambridge University Press. doi:10.1017/CBO9780511809071, pp. 437-440. Cambridge: Cambridge University Press.
[8] Theobald, M., Siddharth, J. and Paepcke, A., 2008, July. Spotsigs: robust and efficient near duplicate detection in large web collections. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval* (pp. 563-570).
[9] Hoad, T.C. and Zobel, J., 2003. Methods for identifying versioned and plagiarized documents. *Journal of the American society for information science and technology*, *54*(3), pp.203-215.
[10] Conrad, J.G., Guo, X.S. and Schriber, C.P., 2003, November. Online duplicate document detection: signature reliability in a dynamic retrieval environment. In *Proceedings of the twelfth international conference on Information and knowledge management* (pp. 443-452);
[11] Mathew, M., Das, S.N. and Vijayaraghavan, P.K., 2011. A novel approach for near-duplicate detection of Web pages using TDW matrix. *International Journal of Computer Applications*, *19*(7), pp.16-21.
[12] Beel, J. and Dinesh, S., 2017, April. Real-World Recommender Systems for Academia: The Pain and Gain in Building, Operating, and Researching them. In *BIR@ ECIR* (pp. 6-17).
[13] The Verge (2018) *Spotify will now let artists directly upload their music to the platform*. Available at: https://www.theverge.com/2018/9/20/17879840/spotify-artist-direct-upload-independent-music (Accessed: 04 May 2020).
[14] Pauwels, J. and Sandler, M.B., 2019. Finding new practice material through chord-based exploration of a large music catalogue.
[15] Jamendo Music (no date) *About us*. Available at https://www.jamendo.com/about/ (Accessed: 08 May 2020).
[16] MongoDB (no date) *Spark Connector Python Guide*. Available at https://docs.mongodb.com/spark-connector/current/python-api/ (Accessed: 04 May 2020).
[17] Ogihara, M. and Li, T., 2008, September. N-Gram Chord Profiles for Composer Style Representation. In *ISMIR* (pp. 671-676).

Research Ethics Review Form: BSc, MSc and MA Projects

Computer Science Research Ethics Committee (CSREC)

http://www.city.ac.uk/department-computer-science/research-ethics

Undergraduate and postgraduate students undertaking their final project in the Department of Computer Science are required to consider the ethics of their project work and to ensure that it complies with research ethics guidelines. In some cases, a project will need approval from an ethics committee before it can proceed. Usually, but not always, this will be because the student is involving other people ("participants") in the project.

In order to ensure that appropriate consideration is given to ethical issues, all students must complete this form and attach it to their project proposal document. There are two parts:

PART A: Ethics Checklist. All students must complete this part.
The checklist identifies whether the project requires ethical approval and, if so, where to apply for approval.

PART B: Ethics Proportionate Review Form. Students who have answered "no" to all questions in A1, A2 and A3 and "yes" to question 4 in A4 in the ethics checklist must complete this part. The project supervisor has delegated authority to provide approval in such cases that are considered to involve MINIMAL risk. The approval may be ***provisional*** – *identifying the planned research as* likely to involve MINIMAL RISK. In such cases you must additionally seek ***full approval*** from the supervisor as the project progresses and details are established. ***Full approval*** must be acquired in writing, before beginning the planned research.

A.1 If you answer YES to any of the questions in this block, you must apply to an appropriate external ethics committee for approval and log this approval as an External Application through Research Ethics Online - https://ethics.city.ac.uk/		*Delete as appropriate*
1.1	Does your research require approval from the National Research Ethics Service (NRES)? *e.g. because you are recruiting current NHS patients or staff?* *If you are unsure try - https://www.hra.nhs.uk/approvals-amendments/what-approvals-do-i-need/*	**NO**
1.2	Will you recruit participants who fall under the auspices of the Mental Capacity Act? *Such research needs to be approved by an external ethics committee such as NRES or the Social Care Research Ethics Committee - http://www.scie.org.uk/research/ethics-committee/*	**NO**
1.3	Will you recruit any participants who are currently under the auspices of the Criminal Justice System, for example, but not limited to, people on remand, prisoners and those on probation? *Such research needs to be authorised by the ethics approval system of the National Offender Management Service.*	**NO**
A.2 If you answer YES to any of the questions in this block, then unless you are applying to an external ethics committee, you must apply for approval		*Delete as appropriate*

from the Senate Research Ethics Committee (SREC) through Research Ethics Online - https://ethics.city.ac.uk/		
2.1	Does your research involve participants who are unable to give informed consent? *For example, but not limited to, people who may have a degree of learning disability or mental health problem, that means they are unable to make an informed decision on their own behalf.*	**NO**
2.2	Is there a risk that your research might lead to disclosures from participants concerning their involvement in illegal activities?	**NO**
2.3	Is there a risk that obscene and or illegal material may need to be accessed for your research study (including online content and other material)?	**NO**
2.4	Does your project involve participants disclosing information about special category or sensitive subjects? *For example, but not limited to: racial or ethnic origin; political opinions; religious beliefs; trade union membership; physical or mental health; sexual life; criminal offences and proceedings*	**NO**
2.5	Does your research involve you travelling to another country outside of the UK, where the Foreign & Commonwealth Office has issued a travel warning that affects the area in which you will study? *Please check the latest guidance from the FCO - http://www.fco.gov.uk/en/*	**NO**
2.6	Does your research involve invasive or intrusive procedures? *These may include, but are not limited to, electrical stimulation, heat, cold or bruising.*	**NO**
2.7	Does your research involve animals?	**NO**
2.8	Does your research involve the administration of drugs, placebos or other substances to study participants?	**NO**
A.3 If you answer YES to any of the questions in this block, then unless you are applying to an external ethics committee or the SREC, you must apply for approval from the Computer Science Research Ethics Committee (CSREC) through Research Ethics Online - https://ethics.city.ac.uk/ Depending on the level of risk associated with your application, it may be referred to the Senate Research Ethics Committee.		*Delete as appropriate*
3.1	Does your research involve participants who are under the age of 18?	**NO**
3.2	Does your research involve adults who are vulnerable because of their social, psychological or medical circumstances (vulnerable adults)?	**NO**

	This includes adults with cognitive and / or learning disabilities, adults with physical disabilities and older people.	
3.3	Are participants recruited because they are staff or students of City, University of London? *For example, students studying on a particular course or module.* *If yes, then approval is also required from the Head of Department or Programme Director.*	**NO**
3.4	Does your research involve intentional deception of participants?	**NO**
3.5	Does your research involve participants taking part without their informed consent?	**NO**
3.5	Is the risk posed to participants greater than that in normal working life?	**NO**
3.7	Is the risk posed to you, the researcher(s), greater than that in normal working life?	**NO**
A.4 If you answer YES to the following question and your answers to all other questions in sections A1, A2 and A3 are NO, then your project is deemed to be of MINIMAL RISK. **If this is the case, then you can apply for approval through your supervisor under PROPORTIONATE REVIEW. You do so by completing PART B of this form.** **If you have answered NO to all questions on this form, then your project does not require ethical approval. You should submit and retain this form as evidence of this.**		*Delete as appropriate*
4	Does your project involve human participants or their identifiable personal data? *For example, as interviewees, respondents to a survey or participants in testing.*	**NO**

Appendix 2: Source Code

---------- PAIRS ----------

DLSH.py

This is the main python file for similarity songs pairs it computes Jaccard similarity, MinHash similarity, LSH banding technique. It computes also metrics for MinHash similarity vs Jaccard similarity and TP, TN, FP, FN for LSH. Plots S-curve for probability of being a candidate pairs vs similarity

LSH_SIM_MH.py
AIM: measure time and memory of similarity search by LSH via Minhashing
It is a shorter version of DLSH.py as it does not compute plots, Jaccard similarity, metrics and csv files. It computes Minhash signatures, LSH candidate pairs and MinHash similarity for those pairs which are candidate pairs.

WJS_JS.py
source code for the computation of Jaccard similarity and weighted Jaccard similarity for songs PAIRS.
It computes also the superimposed histograms of Jaccard and weighted Jaccard similarity.
It computes the statistics for Jaccard and weighted Jaccard similarity.

SONG TRIPLETS ----------
TLSH.py
This is the main source code for TRIPLETS, computes Jaccard similarity, MinHash similarity, LSH banding technique, metrics for MinHash similarity vs Jaccard similarity. It computes TP, TN, FP, FN for LSH, S-curve for probability of being a candidate triplets vs MinHash similarity. It computes simple Jaccard similarity and weighted Jaccard similarity for triplets of songs

SERVER ----------

runSparkMongo.sh
file was used to run the above python/Pyspark files above in the City's server
runSparkMongo.sh source code was provided by Dr J. Pauwels

SOFTWARES ----------
PuTTy was used as user interface between my PC and the City server to submit the experiments.

FileZilla was used as user interface between my PC and the City server to transfer files, code generated png, CSVs and server output text documents.

OpenVPNConnect was used to connect to the server remotely.

Python Packages ----------
Pyplot and python pandas.
Pyspark for Big Data.
Python Notebooks.

GITHUB REPOSITORY----------

All the source code files (DLSH.py, LSH_SIM_MH.py, WJS_JS.py, TLSH.py), results from the server and Python Notebooks are available on GitHub

https://github.com/Ellie2020/Data-Science-MSc-Individual-project

runSparkMongo.sh

NB. On GitHub login credentials were removed from the connector to MongoDB. As the folder is public.

==========

DLSH.py

```
#importing main libraries
import sys
import socket
import os
import time
import psutil
import random
from random import randrange
import builtins
import functools
import operator
import itertools

import findspark
findspark.init()
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
from pyspark.sql.types import IntegerType
from pyspark.sql.functions import desc
from pyspark.sql.functions import asc
#from pyspark.sql.functions import sum as Fsum
from pyspark.sql import Row
from sparkaid import flatten

import pyspark.sql.functions as F
from pyspark.sql.functions import array, udf, col, lit, mean, min, max, concat, desc, row_number,
monotonically_increasing_id
from pyspark.sql.functions import *
from pyspark.ml.linalg import Vectors, VectorUDT, _convert_to_vector
from pyspark.sql.functions import desc, row_number, monotonically_increasing_id

from pyspark.sql.window import Window
import scipy.sparse

from pyspark.sql.types import ArrayType, StringType, DoubleType, IntegerType, FloatType
import pyspark.sql.functions as f
import pyspark.sql.types as T
import pyspark

import numpy as np
import pandas as pd
from matplotlib import pyplot

import builtins

##### input variables ####
songs=1000
num_minhashfuncs=600   ## enter number of MinHash functions here
num_bands=100          ## enter number of LSH bands here
num_LSHfuncs=3         ## enter number of LSH functions here
num_rows=(num_minhashfuncs/num_bands) ## given the inputs above get the number of rows per LSH band
# input parameters for MinHash and LSH function here

p=338563   ### large prime number for MinHash function
m=20117    ### large prime number for MinHash function
m1=20117   ### large prime number for LSH function
########################################
## Connector to the MongoDB database owned by Dr. Johan Pauwels and load data.
```

```
## Code written and provided by Dr. Johan Pauwels
sc = pyspark.SparkContext.getOrCreate()
spark = pyspark.sql.SparkSession.builder \
    .config("spark.mongodb.input.uri", "mongodb://city-projects:dbsZPUMXm7QB8WgP@c4dm-xenserv-
virt5.eecs.qmul.ac.uk/admin")\
    .getOrCreate()

######################## similarity threshold ##############################################

def SIM_threshold(number_bands, number_minhashfunctions):
    '''Given number of bands and length of MinHash signature
                it returns the optimal similarity threshold t
            '''
    num_rows=(number_minhashfunctions/number_bands)   ## rows per band
    return np.round((1/number_bands)**(1/num_rows), 3)

num_rows=int(num_minhashfuncs/num_bands)

#set your threshold
#threshold=0.478    #comment out if you want to set a sim threshold that is different from the optimal one

threshold=SIM_threshold(num_bands, num_minhashfuncs)  #comment out to use optimum theshold
print(f"LSH SIMILARITY THRESHOLD {threshold} for {num_bands} bands and {num_rows} rows per band and
{songs} songs")

#########################################################################################################

df = spark.read.format("mongo").option('database', 'jamendo').option('collection', 'chords').load()
sample_300_df=df.select(["_id","chordRatio"]).limit(songs)

myFunc = f.udf(lambda array_to_list: [int(0) if e is None else int(1) for e in array_to_list],
T.ArrayType(T.IntegerType()))
sample_300_df2=sample_300_df.withColumn('chordRatioMinHash', myFunc('chordRatio'))
df0=sample_300_df2.select(["_id", "chordRatio", "chordRatioMinHash"])

from pyspark.sql import Row
from pyspark.sql.functions import col
from sparkaid import flatten
df0_flat=flatten(df0)
columns_list1=df0_flat.columns[1:-1]
array_df=df0_flat.select('_id', 'chordRatioMinHash',array(columns_list1).alias('chordRatioJS'))

#fill NaNs with zeros in the array column
df2_flat=df0_flat.na.fill(float(0))
columns_list2=df2_flat.columns[1:-1]
array_df2=df2_flat.select('_id', 'chordRatioMinHash',array(columns_list2).alias('chordRatioJS_no_Nulls'))

###
to_vector = udf(lambda a: Vectors.dense(a), VectorUDT())
data = array_df2.select('_id', 'chordRatioMinHash', "chordRatioJS_no_Nulls",
to_vector("chordRatioJS_no_Nulls").alias("chordRatioWJS"))
data.show(1, truncate=False)

import scipy.sparse
from pyspark.ml.linalg import Vectors, _convert_to_vector, VectorUDT
from pyspark.sql.functions import udf, col
## from dense to sparse array
def dense_to_sparse(vector):
    return _convert_to_vector(scipy.sparse.csc_matrix(vector.toArray()).T)

to_sparse = udf(dense_to_sparse, VectorUDT())
data_sparse=data.withColumn("sparseChordRatioJS", to_sparse(col("chordRatioWJS")))
#data_sparse2=data_sparse.select('_id', 'chordRatio_for_minHash', 'sparseChordRatioJS')

indices_udf = udf(lambda vector: vector.indices.tolist(), ArrayType(IntegerType()))
values_udf = udf(lambda vector: vector.toArray().tolist(), ArrayType(DoubleType()))
data_sparse3=data_sparse.withColumn('indicesJS', indices_udf(F.col('sparseChordRatioJS')))\
```

```
.withColumn('values', values_udf(F.col('sparseChordRatioJS')))

## renaming columns for the following steps
simil_DF=data_sparse3.select('_id', 'chordRatioMinHash', 'chordRatioJS_no_Nulls','indicesJS')
simil_DF = simil_DF.select(col("_id").alias("_id"),
                           col("chordRatioMinHash").alias("chordRatioMinHash"),
                           col("chordRatioJS_no_Nulls").alias("chordRatioWeightJS"),
                           col("indicesJS").alias("indicesJS"))

minHashDF1=simil_DF.select('_id', 'chordRatioMinHash', "chordRatioWeightJS", "indicesJS"). \
withColumnRenamed('_id','id1'). \
withColumnRenamed('chordRatioMinHash', 'chordRatioMinHash1'). \
withColumnRenamed("chordRatioWeightJS","chordRatioWeightJS1"). \
withColumnRenamed("indicesJS", "indicesJS1")

### From Pyspark DataFrame with chordRatioMinHash to Matrix 2Darray
data1=np.array(minHashDF1.select('chordRatioMinHash1').collect())
#print('rows: '+str(len(data1)), 'columns: '+str(len(data1[0])))
flattened = np.array([val.tolist() for sublist in data1 for val in sublist])

### MinHash function generator
def make_random_hash_fn():
    a =randrange(1, p-1, 2) # odd random numbers
    b =randrange(0, p-1)    #all numbers even and odd
    if a!=b:
        return lambda x: ((a * x + b) % p) % m
### N MinHash functions generator
def generate_N_hashFunctions(N):
    hashfuncs=[make_random_hash_fn() for i in range(N)]
    return hashfuncs

### minhash the matrix
### Adapted from ref
#https://www.bogotobogo.com/Algorithms/minHash_Jaccard_Similarity_Locality_sensitive_hashing_LSH.php

def minhash1(data, hashfuncs):
    '''
    Returns the minhash signature matrix for the songs using a number of
    hash functions randomly generated
    '''
    '''songs x hashfuncs matrix with values=infinite to start with'''

    sigmat1= [[sys.maxsize for x in range(len(hashfuncs))] for x in range(len(data))]

    for c in range(len(data[0])):
        hashvalue = list(map(lambda x: x(c), hashfuncs))

        for r in range(len(data)):
            if data[r][c] == 0:
                continue
            for i in range(len(hashfuncs)):
                if sigmat1[r][i] > hashvalue[i]:
                    sigmat1[r][i] = hashvalue[i]

    return sigmat1

### To measure time and memory comment off
#import time
#import os
#import psutil
#tt0=time.time()

hash_fnc_list1=generate_N_hashFunctions(num_minhashfuncs)
sign_matrix_rf=minhash1(flattened , hash_fnc_list1)
#print('rows: '+str(len(sign_matrix_rf)), 'columns: '+str(len(sign_matrix_rf[0])))

signatures_df=spark.createDataFrame(sign_matrix_rf)
```

```
signatures_df.columns

#from minhash signature matrix to pyspark dataframe
def columns_to_array_column(N):
    '"N MH functions"'
    return array([col("_"+str(N)) for N in range(1, N+1)])

a=columns_to_array_column(num_minhashfuncs)
signatures_df2 = signatures_df.withColumn("signature1", a)
signatures_df3=signatures_df2.select('signature1')

# joining the minhash dataframe with the mean one
df1 = minHashDF1. \
withColumn('id', row_number().over(Window.orderBy(monotonically_increasing_id())) - 1)

df2 = signatures_df3. \
withColumn('id', row_number().over(Window.orderBy(monotonically_increasing_id())) - 1)

df3=df1.join(df2,on='id',how='left').orderBy('id',ascending=True)  #this is correct way to do it
df3=df3.drop('id')

### To measure time and memory comment off
#df3.collect()
#process = psutil.Process(os.getpid())
# memory=process.memory_info().rss # in bytes
#time_MH=time.time()-tt0
#print("Total time to create Min Hash signatures: "+str(time.time()-tt0) + " secs;", "Memory:
"+str(memory*0.000001) +" Mb")
#print("Total time to create Min Hash signatures: "+str(time.time()-tt0) + " secs;")

### Locality Sensitive Hashing
df4=df3.select('id1', 'signature1')

### Get the buckets for LSH
def get_LHS_buckets(list1, number_bands, number_hashfuncs, P, M):
MYRDN=random.Random(13)
    items_per_band=int(len(list1)/number_bands)
    coeff=[[MYRDN.randrange(1, P-1, 2) for _ in range(items_per_band)] for _ in range(number_hashfuncs)]
###OK
    list_of_tuples=list(zip(*([iter(list1)] * items_per_band)))
    polynomial_results=[[builtins.sum(x*c for x, c in zip(t, cf))%m] for t in list_of_tuples for cf in coeff]
    return polynomial_results

### To measure time and memory comment off
#import time
#import os
#import psutil
#tt0=time.time()

### LSH UDF
get_LHS_buckets_udf = udf(lambda x, y, w, z1, z2: get_LHS_buckets(x, y, w, z1, z2),
ArrayType(ArrayType(IntegerType())))
# these are the parameters for get_LHS_buckets(list1, number_bands, number_hashfuncs, P, M)
df4=df3. \
withColumn('LSH1', get_LHS_buckets_udf("signature1", lit(num_bands), lit(num_LSHfuncs), lit(p), lit(m1)))

### To measure time and memory comment off

#df4.collect()
# process = psutil.Process(os.getpid())
# memory=process.memory_info().rss # in bytes
#time_LSH_sign=time.time()-tt0
#print("Total time to create LSH signatures: "+str(time.time()-tt0) + " secs;", "Memory: "+str(memory*0.000001) +"
Mb")
#print("Total time to create LSH signatures: "+str(time.time()-tt0) + " secs;")

LSH_DF1=df4
```

```
LSH_DF2=df4.select('*'). \
withColumnRenamed('id1','id2'). \
withColumnRenamed('chordRatioMinHash1','chordRatioMinHash2'). \
withColumnRenamed('chordRatioWeightJS1','chordRatioWeightJS2'). \
withColumnRenamed('indicesJS1', 'indicesJS2'). \
withColumnRenamed('signature1', 'signature2'). \
withColumnRenamed("LSH1","LSH2")

# crossjoin without identities and doubles with swapped ids
spark.conf.set("spark.sql.crossJoin.enabled", True)

### Comment off to cache and reuse the DataFrame
#crossjoin_minHashDF=minHashDF1.join(minHashDF2,minHashDF1.id1>minHashDF2.id2).cache()
crossjoin_LSH_DF=LSH_DF1.join(LSH_DF2,LSH_DF1.id1>LSH_DF2.id2). \
select('id1', 'id2', 'chordRatioMinHash1','chordRatioMinHash2',
       'chordRatioWeightJS1','chordRatioWeightJS2',
       'indicesJS1', 'indicesJS2', 'LSH1', 'LSH2',
       'signature1', 'signature2')

# comment off to print number of columns and rows of the Pyspark DataFrame
#print((crossjoin_LSH_DF.count(), len(crossjoin_LSH_DF.columns)))

### LSH candidate pairs
#minhash similarity function
import builtins
import time
from pyspark.sql.functions import size

##If the signature values of two documents agree in all rows of at least one band
##then these two documents are likely to be similar

import builtins
def LHSSimilarpairs(l1,l2,n):
    ''' The function compare the bands of two MinHash signature and return the
        number of buckets for a pair of songs
            '''
    L3=[1 if x==y else 0 for s1, s2 in zip(l1,l2) for x, y in zip(s1,s2)]
    L4=[builtins.sum(L3[i:i+n]) for i in range(0, len(L3),n)]
    LSH=len([x for x in L4 if x>=n])
    return LSH

#minhash similarity udf
num_rows=int(num_minhashfuncs/num_bands)
LHSSimilarity_udf = udf(lambda x, y, z: LHSSimilarpairs(x, y, z), IntegerType())

df5=crossjoin_LSH_DF. \
withColumn('SHARED_BUCKETS', LHSSimilarity_udf('LSH1', 'LSH2', lit(num_rows)))

## MinHash similarity
import builtins
def minHashSimilarity(m1, m2):
    '''The function takes two MinHash signatures m1 and m2 and returns
               the approximate MinHash similarity
            '''
    size=len(m1)
    if size != len(m2): raise Exception("Signature lengths do not match")
    if size == 0: raise Exception("Signature length is zero")
    if size == len(m2):
        try:
            count=builtins.sum(1 for x, y in zip(m1, m2) if x==y)   #ok in one liner
            SIM=count/size

        except ZeroDivisionError:
            pass

        #print('elapsed'.format(elapsed))
```

```
        return SIM

#minhash similarity udf
minHashSimilarity_udf = udf(lambda x, y: minHashSimilarity(x, y), FloatType())

def jaccard_similarity_handling_zero_error(list1, list2):
    ''' This function takes two arrays list1 and list2
                and returns the simple Jaccard similarity '''
    intersection = len(set.intersection(set(list1), set(list2)))
    union = len(set.union(set(list1), set(list2)))
    try:
        sim = intersection/union
    except ZeroDivisionError:
        #return 0.0
        pass
    return sim

#Jaccard similarity UDF function
jaccard_HZE_udf=udf(lambda x, y: jaccard_similarity_handling_zero_error(x, y), FloatType())

#updating table df6 to df7 with the results
df11=df5.withColumn('MH_SIM', minHashSimilarity_udf('signature1', 'signature2')). \
     withColumn('JS', jaccard_HZE_udf('indicesJS1', 'indicesJS2')). \
             select('id1', 'id2', 'SHARED_BUCKETS', 'JS', 'MH_SIM').cache()

##### Metrics for LSH with respect to SIM(S1,S2) for a given similarity threshold t

FN_DF=df11.filter((df11.SHARED_BUCKETS==0) & (df11.MH_SIM>=threshold))
FP_DF=df11.filter((df11.SHARED_BUCKETS!=0) & (df11.MH_SIM<threshold))
TP_DF=df11.filter((df11.SHARED_BUCKETS!=0) & (df11.MH_SIM>=threshold))
TN_DF=df11.filter((df11.SHARED_BUCKETS==0) & (df11.MH_SIM<threshold))


FN=df11.filter((df11.SHARED_BUCKETS==0) & (df11.MH_SIM>=threshold)).count()
FP=df11.filter((df11.SHARED_BUCKETS!=0) & (df11.MH_SIM<threshold)).count()
TP=df11.filter((df11.SHARED_BUCKETS!=0) & (df11.MH_SIM>=threshold)).count()
TN=df11.filter((df11.SHARED_BUCKETS==0) & (df11.MH_SIM<threshold)).count()

## Metrics for SIM(S1,S2) with respect to ground truth J(S1,S2) for a given similarity threshold t

MH_TP=df11.filter((df11.JS>=threshold) & (df11.MH_SIM>=threshold)).count()
MH_TN=df11.filter((df11.JS<threshold) & (df11.MH_SIM<threshold)).count()
MH_FP=df11.filter((df11.JS<threshold) & (df11.MH_SIM>=threshold)).count()
MH_FN=df11.filter((df11.JS>=threshold) & (df11.MH_SIM<threshold)).count()


print(f"retrieved {songs} songs; {num_minhashfuncs} minhash functions; {num_bands} LSH bands;
{num_LSHfuncs} LSH function")


def ACCURACY(t_pos, t_neg, f_pos, f_neg):
   ''' function that takes FP, FN, TP and TN
       and returns accuracy.
               It handles ZeroDivisionError
   '''
    try:
        return (t_pos+t_neg)/(t_pos+f_pos+t_neg+f_neg)
    except ZeroDivisionError:
        pass

lsh_accuracy=ACCURACY(TP,TN,FP,FN)
mh_accuracy=ACCURACY(MH_TP, MH_TN, MH_FP, MH_FN)

def PRECISION(t_pos, f_pos):
    ''' function that takes FP, TP
       and returns precision.
               It handles ZeroDivisionError
```

```
    '''
    try:
        return t_pos/(t_pos+f_pos)
    except ZeroDivisionError:
        pass

lsh_precision=PRECISION(TP, FP)
mh_precision=PRECISION(MH_TP, MH_FP)

def RECALL(t_pos, f_neg):
    ''' function that takes FP, FN
        and returns recall.
                It handles ZeroDivisionError
    '''
    try:
        return t_pos/(t_pos+f_neg)
    except ZeroDivisionError:
        pass

lsh_recall=RECALL(TP, FN)
mh_recall=RECALL(MH_TP, MH_FN)

def F1SCORE(recl, precs):
    ''' function that takes recall and precision
        and returns F1-SCORE.
                It handles ZeroDivisionError
                & type None
    '''
    if (recl!=None and precs!=None):
        try:
            return 2*recl*precs/(recl + precs)
        except ZeroDivisionError:
            pass

lsh_F1=F1SCORE(lsh_recall, lsh_precision)
mh_F1=F1SCORE(mh_recall, mh_precision)

###saving metrics in a csv file
### 1. create a dictionary to store results and
metrics={'#songs': songs,'#MHfuncs': num_minhashfuncs,'#LSHfuncs':num_LSHfuncs,
'#bands':num_bands,'mMH':m, 'mLSH':m1,'pMH':p,
'TP':TP, 'TN': TN,'FP':FP, 'FN': FN,'LSH_Accuracy':lsh_accuracy,'LSH_Precision':lsh_precision,
'LSH_Recall':lsh_recall,'LSH_F1':lsh_F1,
'MH_Accuracy': mh_accuracy,'MH_Precision':mh_precision,
'MH_Recall':mh_recall, 'MH_F1':mh_F1,
'threshold':threshold}

### 2.from dictionary to Pandas DataFrame and then metrics exported as csv file
metrics_df = pd.DataFrame([metrics], columns=metrics.keys())
metrics_df.to_csv('./slurm-logs/metrics_LSHfun'+ str(num_LSHfuncs)+
                '_TP'+str(TP)+
                '_MHfuncs'+str(num_minhashfuncs)+'_bands'+str(num_bands)+
                '_threshold'+str(threshold)+'_songs'+str(songs)+'_m'+str(m)+'_p'+str(p)+
                                        '_F1'+str(lsh_F1)+
                '.csv', encoding='utf-8', index=False)

##exporting TP, FP, TN and FN in csv files
TP_DF_csv=TP_DF.toPandas().to_csv('./slurm-logs/TP_LSH'+ str(num_LSHfuncs)+
                '_TP'+str(TP)+
                '_MHfuncs'+str(num_minhashfuncs)+'_bands'+str(num_bands)+
                '_threshold'+str(threshold)+'_songs'+str(songs)+str(m)+
                '.csv', encoding='utf-8',  index=False)

TN_DF_csv=TN_DF.toPandas().to_csv('./slurm-logs/TN_LSH'+ str(num_LSHfuncs)+
                '_TP'+str(TN)+
                '_MHfuncs'+str(num_minhashfuncs)+'_bands'+str(num_bands)+
                '_threshold'+str(threshold)+'_songs'+str(songs)+str(m)+
```

```
                '.csv', encoding='utf-8',  index=False)

FN_DF_csv=FN_DF.toPandas().to_csv('./slurm-logs/FN_LSH'+ str(num_LSHfuncs)+
                '_FN'+str(FN)+
                '_MHfuncs'+str(num_minhashfuncs)+'_bands'+str(num_bands)+
                '_threshold'+str(threshold)+'_songs'+str(songs)+str(m)+
                '.csv', encoding='utf-8', index=False)

FP_DF_csv=FP_DF.toPandas().to_csv('./slurm-logs/FP_LSH'+ str(num_LSHfuncs)+
                '_TP'+str(FP)+
                '_MHfuncs'+str(num_minhashfuncs)+'_bands'+str(num_bands)+
                '_threshold'+str(threshold)+'_songs'+str(songs)+str(m)+
                '.csv', encoding='utf-8',  index=False)

######## S-curve ######
#probability of sharing a bucket
def sharing_buckets_probability(s, r, b):
    '''The function takes similarity values, number of rows and number of bands.
               It return the probability of  a pair of songs to be candidate pairs
            '''
    probability=1-np.power((1-np.power(s, r)), b)
    return float(probability)

### sharing_buckets_probability UDF
sharing_buckets_probability_udf=udf(lambda x,y,z: sharing_buckets_probability(x, y, z), FloatType())

### adding a column with the probability values to the Pyspark DataFrame (i.e. df11)
df_prob=df11. \
withColumn('Probability', sharing_buckets_probability_udf('MH_SIM', lit(num_rows),lit(num_bands)))

### from Pyspark DataFrame to pandas DataFrame to csv file
df_prob_pandas=df_prob.toPandas().to_csv('./slurm-logs/Probability'+ str(num_LSHfuncs)+
                '_MHfuncs'+str(num_minhashfuncs)+'_bands'+str(num_bands)+
                '_threshold'+str(threshold)+'_songs'+str(songs)+str(m)+
                '.csv', encoding='utf-8',  index=False)

####### Results Visualization  ######
########### S-curve plot ###########
df_prob_pandas=df_prob.toPandas()
print(f"{len(df_prob_pandas)} rows")
import matplotlib
from matplotlib.pyplot import figure
figure(figsize=(7,4))
x=df_prob_pandas.MH_SIM
y=df_prob_pandas.Probability

#label on the plot
pyplot.plot(x, y, 'o', color='blue',  label='bands:'+str(num_bands)+'; songs:'+str(songs)+
          '; rows:'+str(int(num_rows))+ '; MHfuncs:'+str(num_minhashfuncs)+
          '; LSHfuncs:'+str(num_LSHfuncs))

pyplot.plot(x, y, 'o', color='blue',  label='b:'+str(num_bands)+' MH:'+str(num_minhashfuncs))

pyplot.axvline(threshold, color='r', linestyle='-')
pyplot.legend(loc='upper left')
pyplot.legend(loc=(0.7, 1.05), ncol=2)
pyplot.xlabel("Similarity")
pyplot.ylabel("Probability of being candidate pair")
matplotlib.pyplot.savefig('./slurm-
logs/probability_bin_sharing_num_rows'+str(int(num_rows))+'_songs'+str(songs)+
                      '_bands'+str(num_bands)+'_MHfuncs'+str(num_minhashfuncs)+
                      '_LSHfuncs'+str(num_LSHfuncs)+'m'+str(m)+'.png')

pyplot.show()

############# Scatter plots ############
import numpy as np
```

```
from matplotlib import pyplot

df11_plot=df11.select('id1', 'id2', 'MH_SIM', 'SHARED_BUCKETS', 'JS')
df11_pandas=df11_plot.toPandas()
z1=df11_pandas.MH_SIM
z2=df11_pandas.SHARED_BUCKETS
z3=df11_pandas.JS

f = pyplot.figure(figsize=(13,5))

pyplot.subplot(1,3,1)

pyplot.plot(z2, z1, 'o', color='r', label='Shared bins vs MHS')
pyplot.legend(loc='upper right')
pyplot.xlabel("LSH: Number of shared bins")
pyplot.ylabel("MH similarity")
pyplot.axhline(y=threshold, color='b', linestyle='-')

pyplot.subplot(1,3,2)

pyplot.plot(z2, z3, 'o', color='g',  label='Shared bins vs JS')
pyplot.legend(loc='upper right')
pyplot.xlabel("LSH: Number of shared bins")
pyplot.ylabel("JS similarity")

pyplot.axhline(y=threshold, color='r', linestyle='-')

############# histograms JS and MHS ############
pyplot.subplot(1,3,3)
bins=np.linspace(0,1,100)
pyplot.hist(z3,bins,alpha=0.5,color='g', label='JS')
pyplot.hist(z1,bins,alpha=0.5,color='r', label='MH_SIM')
pyplot.legend(loc='upper right')
pyplot.xlabel('similarity')
pyplot.ylabel('count')

pyplot.tight_layout()

### saving the plot as png image
matplotlib.pyplot.savefig('./slurm-logs/scatterplot_hists_with_rows'+str(int(num_rows))+'_songs'+str(songs)+
                          '_bands'+str(num_bands)+'_MHfuncs'+str(num_minhashfuncs)+
                          '_LSHfuncs'+str(num_LSHfuncs)+'m'+str(m)+'.png')
pyplot.show()

############# histograms JS and MHS
print('TASKS DONE AND DUSTED')
spark.stop()
```

TLSH.py

```
#importing main libraries
import sys
import socket
import os
import time
import psutil
import random
from random import randrange
import builtins
import functools
import operator
import itertools

import findspark
findspark.init()
from pyspark.sql import SparkSession
```

```
from pyspark.sql import SQLContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
from pyspark.sql.types import IntegerType
from pyspark.sql.functions import desc
from pyspark.sql.functions import asc
#from pyspark.sql.functions import sum as Fsum
from pyspark.sql import Row
from sparkaid import flatten

import pyspark.sql.functions as F
from pyspark.sql.functions import array, udf, col, lit, mean, min, max, concat, desc, row_number,
monotonically_increasing_id
from pyspark.sql.functions import *
from pyspark.ml.linalg import Vectors, VectorUDT, _convert_to_vector
from pyspark.sql.functions import desc, row_number, monotonically_increasing_id

from pyspark.sql.window import Window
import scipy.sparse

from pyspark.sql.types import ArrayType, StringType, DoubleType, IntegerType, FloatType
import pyspark.sql.functions as f
import pyspark.sql.types as T
import pyspark

import numpy as np
import pandas as pd
from matplotlib import pyplot

import builtins

############ input variables #############
songs=300            ## enter number of songs to compare
num_minhashfuncs=600   ## enter number of MinHash functions here
num_bands=100          ## enter number of LSH bands here
num_LSHfuncs=3         ## enter number of LSH functions here
num_rows=(num_minhashfuncs/num_bands) ## given the inputs above get the number of rows per LSH band
# input parameters for MinHash and LSH function here

p=338563   ### large prime number for MinHash function
m=20117    ### large prime number for MinHash function
m1=20117   ### large prime number for LSH function
#########################################################################
## Connector to the MongoDB database owned by Dr Johan Pauwels.
## Connector provided by Dr Johan Pauwels

sc = pyspark.SparkContext.getOrCreate()
spark = pyspark.sql.SparkSession.builder \
    .config("spark.mongodb.input.uri", "mongodb://city-projects:dbsZPUMXm7QB8WgP@c4dm-xenserv-
virt5.eecs.qmul.ac.uk/admin")\
    .getOrCreate()

#### similarity threshold

def SIM_threshold(number_bands, number_minhashfunctions):

    '''Given number of bands and length of MinHash signature
            it returns the optimal similarity threshold t
    '''

   num_rows=(number_minhashfunctions/number_bands)   ## rows per band
   return np.round((1/number_bands)**(1/num_rows), 3)

num_rows=int(num_minhashfuncs/num_bands)
threshold=SIM_threshold(num_bands, num_minhashfuncs)
```

```
print(f"LSH SIMILARITY THRESHOLD {threshold} for {num_bands} bands and {num_rows} rows per band and
{songs} songs")


df = spark.read.format("mongo").option('database', 'jamendo').option('collection', 'chords').load()
sample_300_df=df.select(["_id","chordRatio"]).limit(songs)

myFunc = f.udf(lambda array_to_list: [int(0) if e is None else int(1) for e in array_to_list],
T.ArrayType(T.IntegerType()))
sample_300_df2=sample_300_df.withColumn('chordRatioMinHash', myFunc('chordRatio'))
df0=sample_300_df2.select(["_id", "chordRatio", "chordRatioMinHash"])

from pyspark.sql import Row
from pyspark.sql.functions import col
from sparkaid import flatten
df0_flat=flatten(df0)
columns_list1=df0_flat.columns[1:-1]
array_df=df0_flat.select('_id', 'chordRatioMinHash',array(columns_list1).alias('chordRatioJS'))

### fill NaNs with zeros in the array column
df2_flat=df0_flat.na.fill(float(0))
columns_list2=df2_flat.columns[1:-1]
array_df2=df2_flat.select('_id', 'chordRatioMinHash',array(columns_list2).alias('chordRatioJS_no_Nulls'))

to_vector = udf(lambda a: Vectors.dense(a), VectorUDT())
data = array_df2.select('_id', 'chordRatioMinHash', "chordRatioJS_no_Nulls",
to_vector("chordRatioJS_no_Nulls").alias("chordRatioWJS"))
data.show(1, truncate=False)

import scipy.sparse
from pyspark.ml.linalg import Vectors, _convert_to_vector, VectorUDT
from pyspark.sql.functions import udf, col

## from dense to sparse array
def dense_to_sparse(vector):
   return _convert_to_vector(scipy.sparse.csc_matrix(vector.toArray()).T)

to_sparse = udf(dense_to_sparse, VectorUDT())
data_sparse=data.withColumn("sparseChordRatioJS", to_sparse(col("chordRatioWJS")))
#data_sparse2=data_sparse.select('_id', 'chordRatio_for_minHash', 'sparseChordRatioJS')

indices_udf = udf(lambda vector: vector.indices.tolist(), ArrayType(IntegerType()))
values_udf = udf(lambda vector: vector.toArray().tolist(), ArrayType(DoubleType()))
data_sparse3=data_sparse.withColumn('indicesJS', indices_udf(F.col('sparseChordRatioJS')))\
.withColumn('values', values_udf(F.col('sparseChordRatioJS')))

## renaming columns for the following steps
simil_DF=data_sparse3.select('_id', 'chordRatioMinHash', 'chordRatioJS_no_Nulls','indicesJS')
simil_DF = simil_DF.select(col("_id").alias("_id"),
                           col("chordRatioMinHash").alias("chordRatioMinHash"),
                           col("chordRatioJS_no_Nulls").alias("chordRatioWeightJS"),
                           col("indicesJS").alias("indicesJS"))

minHashDF1=simil_DF.select('_id', 'chordRatioMinHash', "chordRatioWeightJS", "indicesJS"). \
withColumnRenamed('_id','id1'). \
withColumnRenamed('chordRatioMinHash', 'chordRatioMinHash1'). \
withColumnRenamed("chordRatioWeightJS","chordRatioWeightJS1"). \
withColumnRenamed("indicesJS", "indicesJS1")

### From Pyspark DataFrame with chordRatioMinHash to Matrix 2Darray
data1=np.array(minHashDF1.select('chordRatioMinHash1').collect())
#print('rows: '+str(len(data1)), 'columns: '+str(len(data1[0])))
flattened = np.array([val.tolist() for sublist in data1 for val in sublist])

### minhashing function generator
def make_random_hash_fn():
  '''It generates a minhash function'''
```

```
    a =randrange(1, p-1, 2) #odd numbers
    b =randrange(0, p-1)  #all numbers even and odd
    if a!=b:
        return lambda x: ((a * x + b) % p)% m

### N MinHash functions generator
def generate_N_hashFunctions(N):
    '''It generates N MinHash functions '''
    hashfuncs=[make_random_hash_fn() for i in range(N)]
    return hashfuncs

#minhash the matrix
### minhash the matrix
### Adapted from ref
https://www.bogotobogo.com/Algorithms/minHash_Jaccard_Similarity_Locality_sensitive_hashing_LSH.php
def minhash1(data, hashfuncs):
    '''
    Returns the minhash signature matrix for the songs using a number of
    hash functions randomly generated
    '''
    '''songs x hashfuncs matrix with values=infinite to start with'''
    sigmat1= [[sys.maxsize for x in range(len(hashfuncs))] for x in range(len(data))]

    for c in range(len(data[0])):
        hashvalue = list(map(lambda x: x(c), hashfuncs))

        for r in range(len(data)):
            if data[r][c] == 0:
                continue
            for i in range(len(hashfuncs)):
                if sigmat1[r][i] > hashvalue[i]:
                    sigmat1[r][i] = hashvalue[i]

    return sigmat1

### comment off to measure time and memory
#import time
#import os
#import psutil
#tt0=time.time()

hash_fnc_list1=generate_N_hashFunctions(num_minhashfuncs)
sign_matrix_rf=minhash1(flattened , hash_fnc_list1)
#print('rows: '+str(len(sign_matrix_rf)), 'columns: '+str(len(sign_matrix_rf[0])))

signatures_df=spark.createDataFrame(sign_matrix_rf)
signatures_df.columns

#from minhash signature matrix to pyspark dataframe
def columns_to_array_column(N):
    '''N is equal to number of MH functions'''
    return array([col("_"+str(N)) for N in range(1, N+1)])

a=columns_to_array_column(num_minhashfuncs)
signatures_df2 = signatures_df.withColumn("signature1", a)
signatures_df3=signatures_df2.select('signature1')

# joining the minhash dataframe with the mean one
df1 = minHashDF1. \
withColumn('id', row_number().over(Window.orderBy(monotonically_increasing_id())) - 1)

df2 = signatures_df3. \
withColumn('id', row_number().over(Window.orderBy(monotonically_increasing_id())) - 1)

df3=df1.join(df2,on='id',how='left').orderBy('id',ascending=True)  #this is correct way to do it
df3=df3.drop('id')
```

```
### comment off to measure time and memory
#df3.collect()
#process = psutil.Process(os.getpid())
# memory=process.memory_info().rss # in bytes
#time_MH=time.time()-tt0
#print("Total time to create Min Hash signatures: "+str(time.time()-tt0) + " secs;", "Memory:
"+str(memory*0.000001) +" Mb")
#print("Total time to create Min Hash signatures: "+str(time.time()-tt0) + " secs;")

##LSH
df4=df3.select('id1', 'signature1')

#  NB items_per_band=int(N_hashfuncs/N_bands) needs to be defined before the get_LHS_buckets
# m is a large prime number 1.3 x hashtable

def get_LHS_buckets(list1, number_bands, number_hashfuncs, P, M):
    '''takes a MinHash signature list1 and divide it in number of bands
               and compute the LSH function for all the bands
            '''
    MYRDN=random.Random(13)
    items_per_band=int(len(list1)/number_bands)
    coeff=[[MYRDN.randrange(1, P-1, 2) for _ in range(items_per_band)] for _ in range(number_hashfuncs)]
###OK
    list_of_tuples=list(zip(*([iter(list1)] * items_per_band)))
    polynomial_results=[[builtins.sum(x*c for x, c in zip(t, cf))%m] for t in list_of_tuples for cf in coeff]
    return polynomial_results

### comment off to measure time and memory
#import time
#import os
#import psutil
#tt0=time.time()

##LSH UDF
get_LHS_buckets_udf = udf(lambda x, y, w, z1, z2: get_LHS_buckets(x, y, w, z1, z2),
ArrayType(ArrayType(IntegerType())))
# these are the parameters for get_LHS_buckets(list1, number_bands, number_hashfuncs, P, M)
df4=df3. \
withColumn('LSH1', get_LHS_buckets_udf("signature1", lit(num_bands), lit(num_LSHfuncs), lit(p), lit(m1)))
### comment off to measure time and memory
#df4.collect()
# process = psutil.Process(os.getpid())
# memory=process.memory_info().rss # in bytes
#time_LSH_sign=time.time()-tt0
#print("Total time to create LSH signatures: "+str(time.time()-tt0) + " secs;", "Memory: "+str(memory*0.000001) +"
Mb")
#print("Total time to create LSH signatures: "+str(time.time()-tt0) + " secs;")

LSH_DF1=df4

LSH_DF2=df4.select('*'). \
withColumnRenamed('id1','id2'). \
withColumnRenamed('chordRatioMinHash1','chordRatioMinHash2'). \
withColumnRenamed('chordRatioWeightJS1','chordRatioWeightJS2'). \
withColumnRenamed('indicesJS1', 'indicesJS2'). \
withColumnRenamed('signature1', 'signature2'). \
withColumnRenamed("LSH1","LSH2")

LSH_DF3=df4.select('*'). \
withColumnRenamed('id1','id3'). \
withColumnRenamed('chordRatioMinHash1','chordRatioMinHash3'). \
withColumnRenamed('chordRatioWeightJS1','chordRatioWeightJS3'). \
withColumnRenamed('indicesJS1', 'indicesJS3'). \
withColumnRenamed('signature1', 'signature3'). \
withColumnRenamed("LSH1","LSH3")
```

```
# crossjoin without doubles and swapped ids
spark.conf.set("spark.sql.crossJoin.enabled", True)
#crossjoin_minHashDF=minHashDF1.join(minHashDF2,minHashDF1.id1>minHashDF2.id2).cache()
crossjoin_LSH_DF=LSH_DF1.join(LSH_DF2,LSH_DF1.id1>LSH_DF2.id2). \
select('id1', 'id2', 'chordRatioMinHash1','chordRatioMinHash2',
       'chordRatioWeightJS1','chordRatioWeightJS2',
       'indicesJS1', 'indicesJS2', 'LSH1', 'LSH2',
       'signature1', 'signature2')


crossjoin_LSH_DF2=crossjoin_LSH_DF.join(LSH_DF3, (crossjoin_LSH_DF.id1>LSH_DF3.id3) &
(crossjoin_LSH_DF.id2>LSH_DF3.id3))


crossjoin_LSH_DF2.columns
### LSH candidate pairs

import builtins
def LHSSimilarpairs(l1,l2,l3,n):
            '''takes three arrays of LSH polynomial results for three LSH arrays and
               compare whether 3 songs have at least one identical band (all rows in the band
               identical). If so return the number of LSH buckets
            '''
    #count=[1 for s1, s2 in zip(l1,l2) if x==y for x, y in zip(s1,s2) else 'A']
    l4=[1 if x==y==z else 0 for s1, s2, s3 in zip(l1,l2,l3) for x, y, z in zip(s1,s2,s3)]
    l5=[builtins.sum(l4[i:i+n]) for i in range(0, len(l4),n)]
    LSH=len([e for e in l5 if e>=n])
    return LSH


#minhash similarity udf
num_rows=int(num_minhashfuncs/num_bands)
LHSSimilarity_udf = udf(lambda x, y, w, z: LHSSimilarpairs(x, y, w, z), IntegerType())

df5=crossjoin_LSH_DF2. \
withColumn('T_SHARED_BUCKETS', LHSSimilarity_udf('LSH1', 'LSH2', 'LSH3', lit(num_rows)))

def minHashSimilarity(m1, m2, m3):
  '''It computes the MinHash similarity for a triplet of MinHash signatures
     and it returns the MinHash similarity for a triplet of songs
  '''
    size=len(m1)
    if size != len(m2): raise Exception("Signature lengths do not match")
    if size != len(m3): raise Exception("Signature lengths do not match")
    if size == 0: raise Exception("Signature length is zero")
    if size == len(m2) and size == len(m3):
        try:
            count=builtins.sum(1 for x, y, z in zip(m1, m2, m3) if x==y and x==z)   #ok in one liner
            SIM=count/size

        except ZeroDivisionError:
            pass

        #print('elapsed'.format(elapsed))
        return SIM

#minhash similarity udf
minHashSimilarity_udf = udf(lambda x, y, z: minHashSimilarity(x, y, z), FloatType())

def jaccard_similarity_handling_zero_error(list1, list2, list3):
    '''It computes the Jaccard similarity for a triplets of songs'''
    intersection = len(set.intersection(set(list1), set(list2), set(list3)))
    union = len(set.union(set(list1), set(list2), set(list3)))
    try:
        sim = intersection/union
    except ZeroDivisionError:
        #return 0.0
```

```
        pass
    return sim

#Jaccard similarity UDF function
jaccard_HZE_udf=udf(lambda x, y, z: jaccard_similarity_handling_zero_error(x, y, z), FloatType())


#results in table
df6=df5.withColumn('TMH_SIM', minHashSimilarity_udf('signature1', 'signature2', 'signature3')). \
            withColumn('TJS', jaccard_HZE_udf('indicesJS1', 'indicesJS2', 'indicesJS3')). \
            select('id1', 'id2', 'id3', 'T_SHARED_BUCKETS', 'TMH_SIM', 'TJS').cache()

df6.show()

##### Metrics for LSH Banding technique for song triplets ####

FN_T_DF=df6.filter((df6.T_SHARED_BUCKETS==0) & (df6.TMH_SIM>=threshold))
FP_T_DF=df6.filter((df6.T_SHARED_BUCKETS!=0) & (df6.TMH_SIM<threshold))
TP_T_DF=df6.filter((df6.T_SHARED_BUCKETS!=0) & (df6.TMH_SIM>=threshold))
TN_T_DF=df6.filter((df6.T_SHARED_BUCKETS==0) & (df6.TMH_SIM<threshold))

FN=FN_T_DF.count()
FP=FP_T_DF.count()
TP=TP_T_DF.count()
TN=TN_T_DF.count()

print(f"retrieved {songs} songs; {num_minhashfuncs} minhash functions; {num_bands} LSH bands;
{num_LSHfuncs} LSH function")


### metrics functions

def accuracy_t(t_pos, t_neg, f_pos, f_neg):
    '''from TP, TN, FP and FN computes the accuracy for triplets
                it handles ZeroDivisionError
            '''
    try:
        return (t_pos+t_neg)/(t_pos+f_pos+f_neg+t_neg)
    except ZeroDivisionError:
        pass
   # return a

accuracy=accuracy_t(TP,TN,FP,FN)


def precision_t(t_pos, f_pos):
    '''from TP,FP computes the precision for triplets
                it handles ZeroDivisionError
            '''
    try:
        return t_pos/(t_pos+f_pos)
    except ZeroDivisionError:
        pass


precision=precision_t(TP,FP)

def recall_t(t_pos, f_neg):
    '''from TP, FN computes the recall for triplets
                it handles ZeroDivisionError
            '''
    try:
        return t_pos/(t_pos+f_neg)
    except ZeroDivisionError:
        pass
  #  return r
```

```
recall=recall_t(TP, FN)
#recall=TP/(FN+TP)

def F1Score_t(recl, precs):
    '"from recall and precision computes F1-score for triplets
                it handles ZeroDivisionError and None types
            '"
    if (recl!=None and  precs!=None):
        try:
            return 2*recl*precs/(recl+precs)
        except ZeroDivisionError:
            pass


F1Score=F1Score_t(recall, precision)



# ##saving metrics in a csv file
metrics={'#songs': songs,'#MHfuncs':
num_minhashfuncs,'#LSHfuncs':num_LSHfuncs,'#bands':num_bands,'mMH':m, 'mLSH':m1,'pMH':p,
 'TP':TP, 'TN': TN,'FP':FP, 'FN': FN, 'Accuracy': accuracy,'Precision':precision ,'Recall':recall,'F1': F1Score,
'threshold':threshold}
                        #
metrics_df = pd.DataFrame([metrics], columns=metrics.keys())
metrics_df.to_csv('./slurm-logs/metrics_TRIPLETS_LSHfun'+ str(num_LSHfuncs)+
                '_TP'+str(TP)+
                '_MHfuncs'+str(num_minhashfuncs)+'_bands'+str(num_bands)+
                '_threshold'+str(threshold)+'_songs'+str(songs)+'_m'+str(m)+'_p'+str(p)+
                                                '_F1'+str(F1Score)+
                '.csv', encoding='utf-8', index=False)

##saving FN_DF and TP_DF in csv files
TTP_DF_csv=TP_T_DF.toPandas().to_csv('./slurm-logs/TRIP_TP_LSH'+ str(num_LSHfuncs)+
                '_TP'+str(TP)+
                '_MHfuncs'+str(num_minhashfuncs)+'_bands'+str(num_bands)+
                '_threshold'+str(threshold)+'_songs'+str(songs)+str(m)+
                '.csv', encoding='utf-8',  index=False)

TTN_DF_csv=TN_T_DF.toPandas().to_csv('./slurm-logs/TRIP_TN_LSH'+ str(num_LSHfuncs)+
                '_TP'+str(TN)+
                '_MHfuncs'+str(num_minhashfuncs)+'_bands'+str(num_bands)+
                '_threshold'+str(threshold)+'_songs'+str(songs)+str(m)+
                '.csv', encoding='utf-8',  index=False)

TFN_DF_csv=FN_T_DF.toPandas().to_csv('./slurm-logs/TRIP_FN_LSH'+ str(num_LSHfuncs)+
                '_FN'+str(FN)+
                '_MHfuncs'+str(num_minhashfuncs)+'_bands'+str(num_bands)+
                '_threshold'+str(threshold)+'_songs'+str(songs)+str(m)+
                '.csv', encoding='utf-8', index=False)

TFP_DF_csv=FP_T_DF.toPandas().to_csv('./slurm-logs/TRIP_FP_LSH'+ str(num_LSHfuncs)+
                '_TP'+str(FP)+
                '_MHfuncs'+str(num_minhashfuncs)+'_bands'+str(num_bands)+
                '_threshold'+str(threshold)+'_songs'+str(songs)+str(m)+
                '.csv', encoding='utf-8',  index=False)

######## S-curve ######
#probability of sharing a bucket
def sharing_buckets_probability(s, r, b):
    '"It computes probability for a triplet of being candidate pairs from
                similarity, number of rows and number of bands
            '"
    probability=1-np.power((1-np.power(s, r)), b)
    return float(probability)

sharing_buckets_probability_udf=udf(lambda x,y,z: sharing_buckets_probability(x, y, z), FloatType())
```

```
df_prob=df6.withColumn('T_Probability', sharing_buckets_probability_udf('TMH_SIM',
lit(num_rows),lit(num_bands)))

df_prob_pandas=df_prob.toPandas().to_csv('./slurm-logs/TRIP_Probability'+ str(num_LSHfuncs)+
                '_MHfuncs'+str(num_minhashfuncs)+'_bands'+str(num_bands)+
                '_threshold'+str(threshold)+'_songs'+str(songs)+str(m)+
                '.csv', encoding='utf-8',  index=False)


##########  Visualization ##########
############ S-curve plot ############
df_prob_pandas=df_prob.toPandas()
print(f"{len(df_prob_pandas)} rows")
import matplotlib
from matplotlib.pyplot import figure
figure(figsize=(8,6))
x=df_prob_pandas.TMH_SIM
y=df_prob_pandas.T_Probability

pyplot.plot(x, y, 'o', color='blue',  label='bands:'+str(num_bands)+'; songs:'+str(songs)+
          '; rows:'+str(int(num_rows))+ '; MHfuncs:'+str(num_minhashfuncs)+
          '; LSHfuncs:'+str(num_LSHfuncs))
pyplot.axvline(threshold, color='r', linestyle='-')
pyplot.legend(loc='lower right')
pyplot.xlabel("MH_SIM")
pyplot.ylabel("Triplets Probability of sharing a bin")
matplotlib.pyplot.savefig('./slurm-
logs/TRIP_probability_bin_sharing_num_rows'+str(int(num_rows))+'_songs'+str(songs)+
                    '_bands'+str(num_bands)+'_MHfuncs'+str(num_minhashfuncs)+
                    '_LSHfuncs'+str(num_LSHfuncs)+'m'+str(m)+'.png')

pyplot.show()

############## Other plots ##############
############## Scatter plots ############
import numpy as np
from matplotlib import pyplot

df11_plot=df6.select('id1', 'id2', 'id3','TMH_SIM', 'T_SHARED_BUCKETS', 'TJS')
df11_pandas=df11_plot.toPandas()
z1=df11_pandas.TMH_SIM
z2=df11_pandas.T_SHARED_BUCKETS
z3=df11_pandas.TJS

f = pyplot.figure(figsize=(13,5))

pyplot.subplot(1,3,1)

pyplot.plot(z2, z1, 'o', color='r', label='Triplets shared bins vs TMHS')
pyplot.legend(loc='upper right')
pyplot.xlabel("LSH: Triplets number of shared bins")
pyplot.ylabel("TMH similarity")
pyplot.axhline(y=threshold, color='b', linestyle='-')

pyplot.subplot(1,3,2)

pyplot.plot(z2, z3, 'o', color='g',  label='Triplets shared bins vs TJS')
pyplot.legend(loc='upper right')
pyplot.xlabel("LSH: Triplets number of shared bins")
pyplot.ylabel("TJS similarity")

pyplot.axhline(y=threshold, color='r', linestyle='-')

###############  VISUALIZATION ###################
############## SIMILARITY for TRIPLETS #############
############## histograms TJS and TMHS ############
```

```
pyplot.subplot(1,3,3)
bins=np.linspace(0,1,100)
pyplot.hist(z3,bins,alpha=0.5,color='g',label='TJS')
pyplot.hist(z1,bins,alpha=0.5,color='r',label='TMH_SIM')
pyplot.legend(loc='upper right')
pyplot.xlabel('triplets similarity')
pyplot.ylabel('count')

pyplot.tight_layout()

matplotlib.pyplot.savefig('./slurm-
logs/TRIP_scatterplot_hists_with_rows'+str(int(num_rows))+'_songs'+str(songs)+
                    '_bands'+str(num_bands)+'_MHfuncs'+str(num_minhashfuncs)+
                    '_LSHfuncs'+str(num_LSHfuncs)+'m'+str(m)+'.png')
pyplot.show()

print('TASKS DONE AND DUSTED')
spark.stop()
```

LSH_SIM_MH.py

```
#importing main libraries
import sys
import socket
import os
import time
import psutil
import random
from random import randrange
import builtins
import functools
import operator
import itertools

import findspark
findspark.init()
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
from pyspark.sql.types import IntegerType
from pyspark.sql.functions import desc
from pyspark.sql.functions import asc
#from pyspark.sql.functions import sum as Fsum
from pyspark.sql import Row
from sparkaid import flatten

import pyspark.sql.functions as F
from pyspark.sql.functions import array, udf, col, lit, mean, min, max, concat, desc, row_number,
monotonically_increasing_id
from pyspark.sql.functions import *
from pyspark.ml.linalg import Vectors, VectorUDT, _convert_to_vector
from pyspark.sql.functions import desc, row_number, monotonically_increasing_id

from pyspark.sql.window import Window
import scipy.sparse

from pyspark.sql.types import ArrayType, StringType, DoubleType, IntegerType, FloatType
import pyspark.sql.functions as f
import pyspark.sql.types as T
import pyspark

import numpy as np
import pandas as pd
from matplotlib import pyplot
```

```
import builtins

##### input variables ####
songs=1000            ## enter number of songs to compare
num_minhashfuncs=600   ## enter number of MinHash functions here
num_bands=100          ## enter number of LSH bands here
num_LSHfuncs=3         ## enter number of LSH functions here
num_rows=(num_minhashfuncs/num_bands) ## given the inputs above get the number of rows per LSH band
# input parameters for MinHash and LSH function here

p=338563   ### large prime number for MinHash function
m=20117    ### large prime number for MinHash function
m1=20117   ### large prime number for LSH function

########################################
## Connector to the MongoDB database owned by Dr. Johan Pauwels and load data.
## Code written and provided by Dr. Johan Pauwels

sc = pyspark.SparkContext.getOrCreate()
spark = pyspark.sql.SparkSession.builder \
    .config("spark.mongodb.input.uri", "mongodb://city-projects:dbsZPUMXm7QB8WgP@c4dm-xenserv-
virt5.eecs.qmul.ac.uk/admin")\
    .getOrCreate()

######################## similarity threshold ################################################

def SIM_threshold(number_bands, number_minhashfunctions):
    '''Given number of bands and length of MinHash signature
                it returns the optimal similarity threshold t
            '''
    num_rows=(number_minhashfunctions/number_bands)   ## rows per band
    return np.round((1/number_bands)**(1/num_rows), 3)

num_rows=int(num_minhashfuncs/num_bands)

#set your threshold
#threshold=0.478    #comment out if you want to set a similar threshold that is different from the optimal one

threshold=SIM_threshold(num_bands, num_minhashfuncs)  #comment out to use optimum threshold
print(f"LSH SIMILARITY THRESHOLD {threshold} for {num_bands} bands and {num_rows} rows per band and
{songs} songs")

#############################################################################################
#

df = spark.read.format("mongo").option('database', 'jamendo').option('collection', 'chords').load()
sample_300_df=df.select(["_id","chordRatio"]).limit(songs)

myFunc = f.udf(lambda array_to_list: [int(0) if e is None else int(1) for e in array_to_list],
T.ArrayType(T.IntegerType()))
sample_300_df2=sample_300_df.withColumn('chordRatioMinHash', myFunc('chordRatio'))
df0=sample_300_df2.select(["_id", "chordRatio", "chordRatioMinHash"])

from pyspark.sql import Row
from pyspark.sql.functions import col
from sparkaid import flatten
df0_flat=flatten(df0)
columns_list1=df0_flat.columns[1:-1]
array_df=df0_flat.select('_id', 'chordRatioMinHash',array(columns_list1).alias('chordRatioJS'))

#fill NaNs with zeros in the array column
df2_flat=df0_flat.na.fill(float(0))
columns_list2=df2_flat.columns[1:-1]
array_df2=df2_flat.select('_id', 'chordRatioMinHash',array(columns_list2).alias('chordRatioJS_no_Nulls'))

to_vector = udf(lambda a: Vectors.dense(a), VectorUDT())
```

```
data = array_df2.select('_id', 'chordRatioMinHash', "chordRatioJS_no_Nulls",
to_vector("chordRatioJS_no_Nulls").alias("chordRatioWJS"))


import scipy.sparse
from pyspark.ml.linalg import Vectors, _convert_to_vector, VectorUDT
from pyspark.sql.functions import udf, col
## from dense to sparse array
def dense_to_sparse(vector):
    return _convert_to_vector(scipy.sparse.csc_matrix(vector.toArray()).T)

to_sparse = udf(dense_to_sparse, VectorUDT())
data_sparse=data.withColumn("sparseChordRatioJS", to_sparse(col("chordRatioWJS")))
#data_sparse2=data_sparse.select('_id', 'chordRatio_for_minHash', 'sparseChordRatioJS')

indices_udf = udf(lambda vector: vector.indices.tolist(), ArrayType(IntegerType()))
values_udf = udf(lambda vector: vector.toArray().tolist(), ArrayType(DoubleType()))
data_sparse3=data_sparse.withColumn('indicesJS', indices_udf(F.col('sparseChordRatioJS')))\
.withColumn('values', values_udf(F.col('sparseChordRatioJS')))

## renaming columns for the following steps
simil_DF=data_sparse3.select('_id', 'chordRatioMinHash', 'chordRatioJS_no_Nulls','indicesJS')
simil_DF = simil_DF.select(col("_id").alias("_id"),
                           col("chordRatioMinHash").alias("chordRatioMinHash"))

minHashDF1=simil_DF.select('_id', 'chordRatioMinHash'). \
withColumnRenamed('_id','id1'). \
withColumnRenamed('chordRatioMinHash', 'chordRatioMinHash1')

data1=np.array(minHashDF1.select('chordRatioMinHash1').collect())
#print('rows: '+str(len(data1)), 'columns: '+str(len(data1[0])))
flattened = np.array([val.tolist() for sublist in data1 for val in sublist])

def make_random_hash_fn():
    a =randrange(1, p-1, 2) #odd numbers
    b =randrange(0, p-1)  #all numbers even and odd
    if a!=b:
        return lambda x: (a * x + b) % m

def generate_N_hashFunctions(N):
    hashfuncs=[make_random_hash_fn() for i in range(N)]
    return hashfuncs

### minhash the matrix
### Adapted from ref
#https://www.bogotobogo.com/Algorithms/minHash_Jaccard_Similarity_Locality_sensitive_hashing_LSH.php

def minhash1(data, hashfuncs):
    '''
    Returns the minhash signature matrix for the songs using a number of
    hash functions randomly generated
    '''
    '''songs x hashfuncs matrix with values=infinite to start with'''
    sigmat1= [[sys.maxsize for x in range(len(hashfuncs))] for x in range(len(data))]

    for c in range(len(data[0])):
        hashvalue = list(map(lambda x: x(c), hashfuncs))

        for r in range(len(data)):
            if data[r][c] == 0:
                continue
            for i in range(len(hashfuncs)):
                if sigmat1[r][i] > hashvalue[i]:
                    sigmat1[r][i] = hashvalue[i]

    return sigmat1
```

```
hash_fnc_list1=generate_N_hashFunctions(num_minhashfuncs)
sign_matrix_rf=minhash1(flattened , hash_fnc_list1)

signatures_df=spark.createDataFrame(sign_matrix_rf)
signatures_df.columns

#from minhash signature matrix to pyspark dataframe
def columns_to_array_column(N):
    '"N MH functions"'
    return array([col("_"+str(N)) for N in range(1, N+1)])

a=columns_to_array_column(num_minhashfuncs)
signatures_df2 = signatures_df.withColumn("signature1", a)
signatures_df3=signatures_df2.select('signature1')

# joining the minhash dataframe with the mean one
df1 = minHashDF1. \
withColumn('id', row_number().over(Window.orderBy(monotonically_increasing_id())) - 1)

df2 = signatures_df3. \
withColumn('id', row_number().over(Window.orderBy(monotonically_increasing_id())) - 1)

df3=df1.join(df2,on='id',how='left').orderBy('id',ascending=True)  #this is correct way to do it

##LSH
#df4=df3.select('id1', 'signature1')
df3=df3.drop('id')

#  NB items_per_band=int(N_hashfuncs/N_bands) needs to be defined before the get_LHS_buckets
# m is a large prime number 1.3 x hashtable

def get_LHS_buckets(list1, number_bands, number_hashfuncs, P, M):
    MYRDN=random.Random(13)
    items_per_band=int(len(list1)/number_bands)
    coeff=[[MYRDN.randrange(1, P-1, 2) for _ in range(items_per_band)] for _ in range(number_hashfuncs)]
    list_of_tuples=list(zip(*([iter(list1)] * items_per_band)))
    polynomial_results=[[builtins.sum(x*c for x, c in zip(t, cf))%m] for t in list_of_tuples for cf in coeff]
    return polynomial_results


##LSH UDF
get_LHS_buckets_udf = udf(lambda x, y, w, z1, z2: get_LHS_buckets(x, y, w, z1, z2),
ArrayType(ArrayType(IntegerType())))
# these are the parameters for get_LHS_buckets(list1, number_bands, number_hashfuncs, P, M)
df4=df3. \
withColumn('LSH1', get_LHS_buckets_udf("signature1", lit(num_bands), lit(num_LSHfuncs), lit(p), lit(m1)))

LSH_DF1=df4

LSH_DF2=df4.select('*'). \
withColumnRenamed('signature1','signature2'). \
withColumnRenamed('id1','id2'). \
withColumnRenamed("LSH1","LSH2")

# crossjoin without doubles and swapped ids
spark.conf.set("spark.sql.crossJoin.enabled", True)
crossjoin_LSH_DF=LSH_DF1.join(LSH_DF2,LSH_DF1.id1>LSH_DF2.id2)
#select('id1', 'id2','LSH1', 'LSH2')

### LSH candidate pairs
#minhash similarity function
import builtins
import time
from pyspark.sql.functions import size

import builtins
def LHSSimilarpairs(l1,l2,n):
```

```
    L3=[1 if x==y else 0 for s1, s2 in zip(l1,l2) for x, y in zip(s1,s2)]
    L4=[builtins.sum(L3[i:i+n]) for i in range(0, len(L3),n)]
    LSH=len([x for x in L4 if x>=n])
    return LSH

#minhash similarity udf
num_rows=int(num_minhashfuncs/num_bands)
LHSSimilarity_udf = udf(lambda x, y, z: LHSSimilarpairs(x, y, z), IntegerType())

df5=crossjoin_LSH_DF. \
withColumn('SHARED_BUCKETS', LHSSimilarity_udf('LSH1', 'LSH2', lit(num_rows)))

#num_candidates = df5.filter(df5.SHARED_BUCKETS>0).count()
df6=df5.filter(df5.SHARED_BUCKETS>0)
df6.show()


## computing the MinHash similarity for only the candidate pairs obtained from LSH banding technique
import builtins
def minHashSimilarity(m1, m2):
    size=len(m1)
    if size != len(m2): raise Exception("Signature lengths do not match")
    if size == 0: raise Exception("Signature length is zero")
    if size == len(m2):
        try:
            count=builtins.sum(1 for x, y in zip(m1, m2) if x==y)   #ok in one liner
            SIM=count/size

        except ZeroDivisionError:
            pass

        #print('elapsed'.format(elapsed))
        return SIM

#minhash similarity udf
minHashSimilarity_udf = udf(lambda x, y: minHashSimilarity(x, y), FloatType())

import time
import os
import psutil
tt0=time.time()

df7=df6.withColumn('MH_SIM', minHashSimilarity_udf('signature1', 'signature2')).collect()

process = psutil.Process(os.getpid())
memory=process.memory_info().rss # in bytes
time_LSH_sign=time.time()-tt0
print("Total time to calculate MinHash SIM after LSH: "+str(time.time()-tt0) + " secs;", "Memory:
"+str(memory*0.000001) +" Mb")
print("Total time to calculate MinHash SIM after LSH: "+str(time.time()-tt0) + " secs;")

print('TASKS DONE AND DUSTED')
spark.stop()
```

WJS_JS.py

```
#importing main libraries
import sys
import socket
import os
import time
import psutil
import random
from random import randrange
import builtins
import functools
import operator
```

```
import itertools

import findspark
findspark.init()
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
from pyspark.sql.types import IntegerType
from pyspark.sql.functions import desc
from pyspark.sql.functions import asc
#from pyspark.sql.functions import sum as Fsum
from pyspark.sql import Row
from sparkaid import flatten

import pyspark.sql.functions as F
from pyspark.sql.functions import array, udf, col, lit, mean, min, max, concat, desc, row_number,
monotonically_increasing_id
from pyspark.sql.functions import *
from pyspark.ml.linalg import Vectors, VectorUDT, _convert_to_vector
from pyspark.sql.functions import desc, row_number, monotonically_increasing_id

from pyspark.sql.window import Window
import scipy.sparse

from pyspark.sql.types import ArrayType, StringType, DoubleType, IntegerType, FloatType
import pyspark.sql.functions as f
import pyspark.sql.types as T
import pyspark

import numpy as np
import pandas as pd
from matplotlib import pyplot

import builtins

##### input variables ###########
songs=10000              ### enter number of songs to compare (pairwise)

#############################################################
## MongoDB database owned by Dr Johan Pauwels.
## Connector to the MongoDB provided by Dr Johan Pauwels

sc = pyspark.SparkContext.getOrCreate()
spark = pyspark.sql.SparkSession.builder \
    .config("spark.mongodb.input.uri", "mongodb://city-projects:dbsZPUMXm7QB8WgP@c4dm-xenserv-
virt5.eecs.qmul.ac.uk/admin")\
    .getOrCreate()

df0 = spark.read.format("mongo").option('database', 'jamendo').option('collection', 'chords').load()
sample_300_df=df0.select(["_id","chordRatio"]).limit(songs)

myFunc = f.udf(lambda array_to_list: [int(0) if e is None else int(1) for e in array_to_list],
T.ArrayType(T.IntegerType()))
sample_300_df2=sample_300_df.withColumn('chordRatioMinHash', myFunc('chordRatio'))
df0=sample_300_df2.select(["_id", "chordRatio", "chordRatioMinHash"])

from pyspark.sql import Row
from pyspark.sql.functions import col
from sparkaid import flatten
df0_flat=flatten(df0)
columns_list1=df0_flat.columns[1:-1]
array_df=df0_flat.select('_id', 'chordRatioMinHash',array(columns_list1).alias('chordRatioJS'))

#fill NaNs with zeros in the array column
df2_flat=df0_flat.na.fill(float(0))
```

```
columns_list2=df2_flat.columns[1:-1]
array_df2=df2_flat.select('_id', 'chordRatioMinHash',array(columns_list2).alias('chordRatioJS_no_Nulls'))

###
to_vector = udf(lambda a: Vectors.dense(a), VectorUDT())
data = array_df2.select('_id', 'chordRatioMinHash', "chordRatioJS_no_Nulls",
to_vector("chordRatioJS_no_Nulls").alias("chordRatioWJS"))
data.show(1, truncate=False)

import scipy.sparse
from pyspark.ml.linalg import Vectors, _convert_to_vector, VectorUDT
from pyspark.sql.functions import udf, col

def dense_to_sparse(vector):
   return _convert_to_vector(scipy.sparse.csc_matrix(vector.toArray()).T)

to_sparse = udf(dense_to_sparse, VectorUDT())
data_sparse=data.withColumn("sparseChordRatioJS", to_sparse(col("chordRatioWJS")))
#data_sparse2=data_sparse.select('_id', 'chordRatio_for_minHash', 'sparseChordRatioJS')

indices_udf = udf(lambda vector: vector.indices.tolist(), ArrayType(IntegerType()))
values_udf = udf(lambda vector: vector.toArray().tolist(), ArrayType(DoubleType()))
data_sparse3=data_sparse.withColumn('indicesJS', indices_udf(F.col('sparseChordRatioJS')))\
.withColumn('values', values_udf(F.col('sparseChordRatioJS')))

## renaming columns for the following steps
simil_DF=data_sparse3.select('_id', 'chordRatioJS_no_Nulls','indicesJS')
simil_DF = simil_DF.select(col("_id").alias("_id"),
                        col("chordRatioJS_no_Nulls").alias("chordRatioWeightJS"),
                        col("indicesJS").alias("indicesJS"))

SIMDF1=simil_DF.select('_id', "chordRatioWeightJS", "indicesJS"). \
withColumnRenamed('_id','id1'). \
withColumnRenamed("chordRatioWeightJS","chordRatioWeightJS1"). \
withColumnRenamed("indicesJS", "indicesJS1")


SIMDF2=simil_DF.select('_id', "chordRatioWeightJS", "indicesJS"). \
withColumnRenamed('_id','id2'). \
withColumnRenamed("chordRatioWeightJS","chordRatioWeightJS2"). \
withColumnRenamed("indicesJS", "indicesJS2")

# joining SIMDF1 and SIMDF2
spark.conf.set("spark.sql.crossJoin.enabled", True)
crossjoinDF=SIMDF1.join(SIMDF2,SIMDF1.id1>SIMDF2.id2)
crossjoinDF.show()

#Jaccard similarity
def jaccard_similarity_handling_zero_error(list1, list2):
            ''' This function takes two arrays list1 and list2
            and returns the simple Jaccard similarity '''
   intersection = len(set.intersection(set(list1), set(list2)))
   union = len(set.union(set(list1), set(list2)))
   try:
      sim = intersection/union
   except ZeroDivisionError:
      pass
   return sim

#Jaccard similarity UDF
jaccard_HZE_udf=udf(lambda x, y: jaccard_similarity_handling_zero_error(x, y), FloatType())

df = crossjoinDF.withColumn('JS', jaccard_HZE_udf('indicesJS1', 'indicesJS2'))  ##comment out for checking just
JS

import builtins
def weighted_JS(l1, l2):
```

```
        ''' This functions takes two arrays list1 and list2
        and returns the weighted Jaccard similarity '''

    A=builtins.sum([builtins.min(x, y) for x, y in zip(l1, l2)])
    zip_list=zip(l1, l2)
    B=builtins.sum([builtins.max(x, y) for x, y in zip(l1, l2)])
    try:
        WJS=A/B
    except ZeroDivisionError:
        #return 0.0
        pass
    return WJS


weighted_JS_UDF= F.udf(lambda x, y: weighted_JS(x, y) , DoubleType())

df=crossjoinDF.withColumn('WJS', weighted_JS_UDF('chordRatioWeightJS1', 'chordRatioWeightJS2'))

#################  Visualization  ##################
############### histograms JS and WJS ############
df_pandas = df.toPandas()
z1=df_pandas.JS
z2=df_pandas.WJS

import matplotlib
from matplotlib import pyplot
from matplotlib.pyplot import figure


bins=np.linspace(0,1,100)
pyplot.hist(z1,bins,alpha=0.5,color='g',label='JS')
pyplot.hist(z2,bins,alpha=0.5,color='r',label='WEIGHTED_SIM')
pyplot.legend(loc='upper right')
pyplot.xlabel('similarity')
pyplot.ylabel('count')

matplotlib.pyplot.savefig('./slurm-logs/hists_JS_WJS_songs'+str(songs)+'.png')
pyplot.show()

df.describe('JS', 'WJS').show()



df.describe('JS').show() ##comment our if checking statistics for JS

df.describe('WJS').show() ##comment out if checking statistics for WJS

print('TASKS DONE AND DUSTED')
spark.stop()
```