Italian Lyrics Retrieval System

Carlo De Nardin - carlo.denardin@studenti.units.it

A.A. 2022-2023 - Information Retrieval

Abstract

The *Italian Lyrics Retrieval System* project originated as an exam for *Information Retrieval* course taught by Professor Manzoni at the University of Trieste. Various aspects that contribute to the development of this system are examined and explored in the following text. Highlighted among the implemented concepts are methods that include document pre-processing, duplicate removal, compression, results classification, handling of phrase queries, and a brief analysis of the phonetic algorithms utilized.

Disclaimer: The following report is not intended as a detailed guide on algorithm implementations; for these details, please refer to the book *Introduction to Information Retrieval* [1]. Alongside the text, a simple implementation of the system in Python is provided [2].

keywords: *Pre-processing*, *SimHash*, *Gamma encoding*, *Phrase queries*, *Okapi-BM25*, *phonetic algorithm*

1 Introduction

The aim of this report is to examine and implement methods for creating an Information Retrieval system that is focused on a collection of Italian-language musical lyrics. A system like this could be utilized in a variety of scenarios, including:

- Retrieving the lyrics of a song given its title.
- Providing a list of songs given a specified author.
- Retrieving the lyrics of a song given some keywords or a phrase.

These examples demonstrate how an Information Retrieval system that can meet these user information needs requires the development and adoption of different methodologies.

A methodology for removing duplicates and similar documents is conducted in the following chapter after analyzing the dataset and various preprocessing techniques. The third chapter outlines various techniques for creating indexes. Compression techniques for the indices are described in the fourth chapter, followed by a size comparison with uncompressed indices. In Chapter 5, query techniques are analyzed, and the final chapter provides an analysis (history) of phonetic techniques.

2 Dataset

For the realization of the project, a dataset containing Italian song lyrics was sought. Initially, the option of Musixmatch APIs was examined, which, however, only provide 30% of the lyrics of individual songs for free and currently do not make data available for academic purposes. To obtain a sufficiently large and accurate dataset, lyricsgenius APIs [3] were utilized. These APIs interface with the well-known lyrics portal Genius and perform scraping of the necessary information. In order to do this, another portal was used to obtain the names of 341 famous Italian artists, and then the song lyrics were retrieved using the APIs. A collection of about 28,000 Italian song lyrics was produced, which became 26,214 after preprocessing.

N° Documents	Length (avg)	Tokens (approx.)
26214	130.34	3500000

Table 1: Dataset information (number of documents, average length, total tokens).

2.1 Preprocessing

Initially, an analysis of the collection's documents was conducted to identify any anomalies, such as empty texts, duplicates, strange characters, or foreign languages. The indexing of terms for the subsequent documents was limited to Italian and English. Therefore, different preprocessing stages were implemented for the documents.

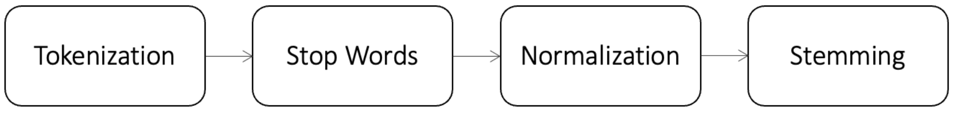

Figure 1: Pre-processing pipeline

- **Tokenization**: Process that splits documents into individual units called tokens (a single-word granularity was chosen).
- **Stop words**: Removal of words not useful for building the system in Italian and English, obtained through the use of the nltk library.
- **Normalization**: Normalization of tokens through various techniques that are listed in the below table.

Technique	Description
Short tokens	Removal of words with a length fewer than three characters
Digits	Removal of numbers or words containing numbers (documents entered through speech recognition)
Patterns	Removal of multiple patterns found in various texts ('ah', 'uh', 'oh', 'oioi')
Foreign tokens	Removal of tokens in foreign languages, excluding English, using the *lingua* [4] library

Table 2: Normalization techniques

- **Stemming**: Transformation of tokens into a base form using the Porter Stemmer [5] for both Italian and English languages.

2.2 Near duplicates removal

Several duplicates or near duplicates were discovered during further analysis of the collection's documents.

ID	Author	Title
1166	Andrea Bocelli	Nessun dorma
1167	Andrea Bocelli	Nessun dorma - Live At Central Park

Table 3: Example of duplicate documents

Indexing two documents of this type does not provide any added value to our system. Obtaining the title *Live At Central Park* is not a user need in almost any scenario. The *simhash* [6] technique is introduced to effectively eliminate similar documents due to this reason.

2.2.1 SimHash

Based on the terms and frequency of a document, the SimHash technique generates a hash (identifier) for it. Each word is connected to its frequency in the document. Subsequently, the document is transformed into a unique hash of a fixed length b. Starting from the most significant bit, weights are summed if the bit is 1, while they are subtracted if the bit is 0. In the end, each value is converted to 1 if it is greater than 0, or to 0 if it is less than 0. The hash of the document is represented by this final sequence of bits.

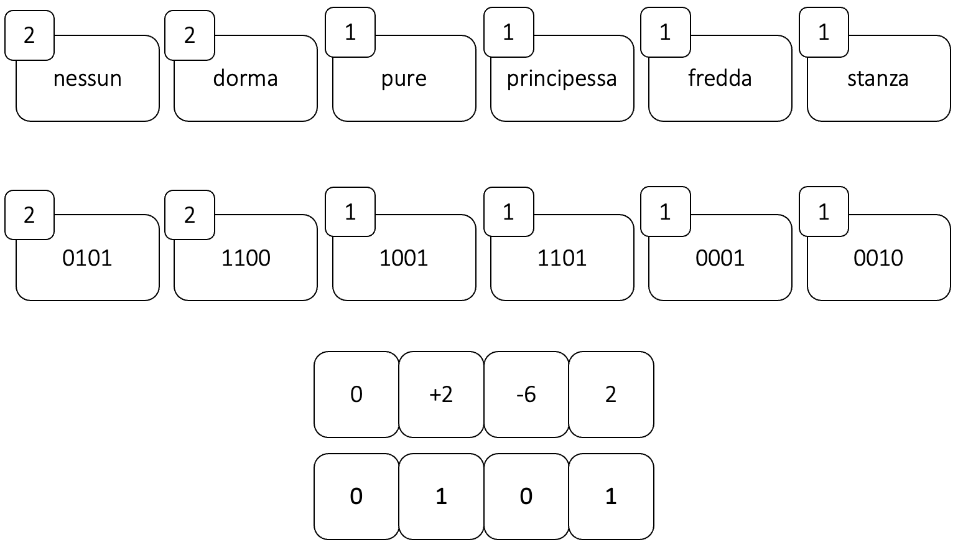

Figure 2: SimHash application

The purpose of this method is to identify duplicate documents belonging to individual artists. To preserve information related to remastered songs, duplicate documents are not compared across different artists.

This process is carried out prior to indexing, which results in a final dataset of the following dimensions.

N° Documents	Length (avg)	Tokens (approx.)
25597	139.39	3500000

Table 4: Dataset information (number of documents, average length, total tokens) after performing the SimHash algorithm

3 Index construction

As previously mentioned, the system in question needs to answer various queries:

- Retrieving the lyrics of a song given its title.
- Providing a list of songs given a specified author.
- Retrieving the lyrics of a song given some keywords or a phrase.

The development of three separate indexes was deemed necessary due to these potential queries:

- **Title**: an index containing song titles.
- **Author**: an index containing song authors.
- **Lyrics**: an index containing song lyrics.

Efficiency and precision are the basis for the decision to divide the document into separate indexes. In cases in which a user wants to search for songs by a specific author, searching within a single index might result in incorrect results (for instance, the user's name might be present in the lyrics of some songs).

3.1 Authors and titles indexes

The indexes for titles and authors are handled in the same manner, taking into account scenarios when these entities consist of single or multiple words. Due to its small size, two boolean indices are established with granularity at the level of individual word tokens.

The decision to use an index composed of single terms rather than multiple terms reflects the intention to respond to queries composed by single words, for example, *"Claudio Baglioni"* will probably retrive results equal to *"Baglioni"* since the user may search for both with the same need.

In details, for a query like *"Claudio Baglioni"* a logical **AND** operator is applied between the words as we do not want to return songs by other authors with the name *"Claudio"*. Without the use of the AND operator, documents related to songs by *"Claudio Villa"*, for example, would also be retrieved.

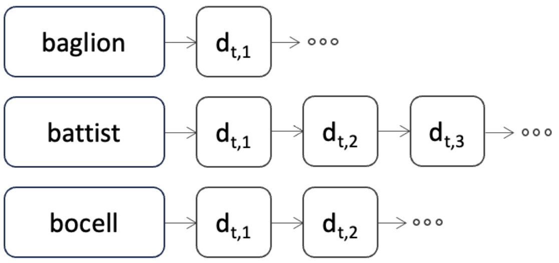

Figure 3: Boolean index

In scenarios such as searching to find songs by *"Baglioni"*, all documents associated with the term are returned. Further enhancements could be made by associating a precomputed score with each entry in the posting lists based on various factors, such as the song's popularity or the number of listens. In the absence of this information, It was decided to retrieve all songs found for a specific autorh or title.

In these indixes a more flexible Pre-processing technique was implemented removing the stemming operation. In the case of an author query like*"Laura"* or *"Lauro" (Achille Lauro)*, stemming would have transformed the words into *"Laur"*, yielding incorrect results. In situations where specific terms are entered, the use of stemming is not always recommended.

3.2 Lyrics index

Additional statistics are used to process the song lyrics index to make it easier to query the system with more complex queries like:

- A user wants to find songs related to specific topics, so He/She performs a search by entering specific keywords.
- A user has some keywords of a song in mind or a phrase and want to identify the song using this information.

Due to the volume of documents and their size, using a boolean index is not the most appropriate choice as it does not allow for results ranking. There is a risk of returning many results that include even just a repetition of a single keyword. In addition, a positional index is necessary to handle queries about specific phrases in songs.

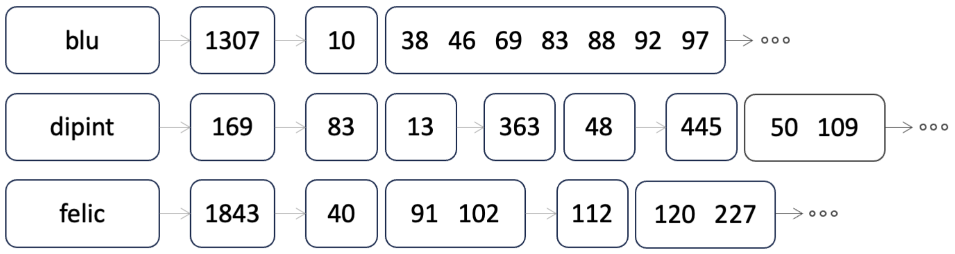

Figure 4: Positional index

To answer queries in a more detailed manner, we can store various pre-computed statistics using an index of this type, as is possible to observe in the above example.

- Regarding queries where a user searches for keywords, we can associate some scores using a ranking function (Okapi BM25 or tf-idf) to provide to the user a better representation of the results.
- The positional index allows us to respond to phrase queries by using the positions of various words within the documents.

These techniques will be further explored in Chapter 5 to respond to different user queries.

4 Compression

The *Italian Lyrics Retrieval System* utilizes compression algorithms that have been analyzed and implemented. The efficiency of using and loading compressed and uncompressed indices is not compared due to the small dataset used to notice these aspects. The analysis will focus on examining the reduction in used space, with only the adopted compression techniques being described.

4.1 Dictionary compression

A dictionary is a data structure that contains a list of all unique words in a set of documents. Using a fixed size for each term is the simplest and most expensive way to save terms in a dictionary. This solution is inefficient because it causes a significant amount of memory space to be wasted. English vocabulary requires a fixed size of approximately 20 bytes for each term, yet the average word length only occupies 8 bytes [1].

The solution adopted and implemented in this system involves representing the dictionary as a single string divided into blocks of k terms. Next, the front-coding technique is applied to every block, reducing the terms based on the use of prefixes.

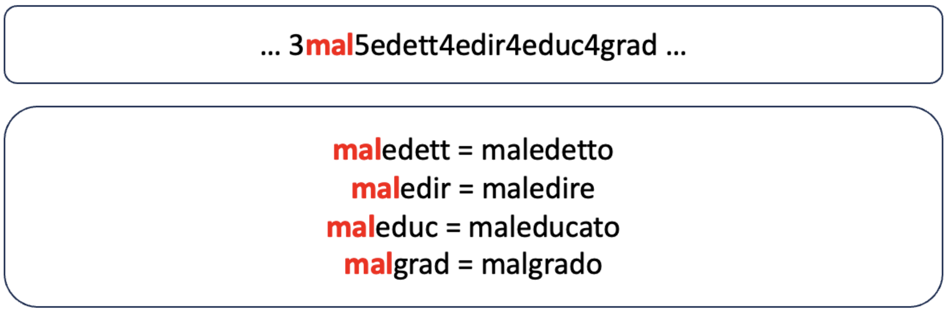

Figure 5: Single string dictionary compression

The compression of a block with a size equal to four can be seen in the example above. The first term corresponds to the prefix for the other terms. By using a block-wise approach, there is a memory-saving advantage in storing pointers. In this scenario, only one pointer per block needs to be saved and since the size of our compressed dictionary is less than 65536 it is feasible to use 2 bytes for each pointer to a block.

4.2 Posting list compression

The posting list is a sorted list that contains the identifiers of documents that contain the associated term. To compress the posting list, we can leverage the fact that the posting lists are ordered in ascending order. We can obtain the gaps between the documents and encode them using gamma encoding in this manner.

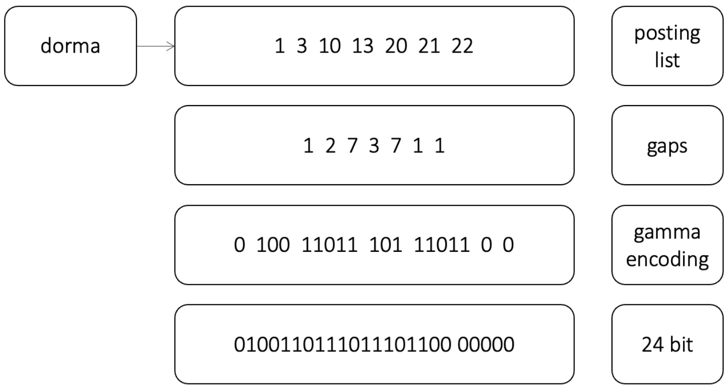

Figure 6: Posting list compression with gaps and gamma encoding

Extra trailing zeros are added to achieve a multiple of 8 bits, as shown in the gamma compression example. To handle posting lists that were divided into bytes, this was essential. However, this technique requires storing information in the pointers file regarding the length of the posting list to determine its termination point. If you don't take this precaution, a sequence of zeros using gamma encoding would be translated into a sequence of ones.

4.3 Pointers

It is necessary to maintain references to the compressed dictionary and the compressed posting list. In this implementation, it was required to save the length of the posting list, the pointer to the posting list, and the pointer to the first term of the dictionary block.

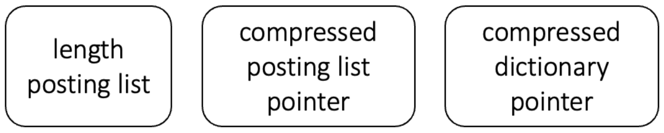
length posting list
compressed posting list pointer
compressed dictionary pointer

Figure 7: Pointers to achieve the compression

4.4 Consideration

The size differences between various indices are depicted in the table below. The uncompressed lyrics index with no additional statistics (frequency, positions) is compared to the same index in a compressed version.

Index	Dimension
Lyrics (lite)	12.6 MB
Dictionary compressed	287 KB
Posting list compressed	3.2 MB
Pointers	623 KB
Lyrics (compressed)	4.1 MB approx.

Table 5: Dimension of the lite lyrics index and the compressed index

The data above shows that there is a significant savings because the uncompressed index weighs three times as much as the compressed index.

5 Data structure and Query execution

In this section, decisions regarding data structures and methodologies adopted to optimize user queries are outlined. Concerning data structures, the *Trie* was chosen, a type of prefix tree structure that offers advantages in the efficient management of textual data.

In the context of queries, various methodologies have been implemented to meet user needs. In particular, the report outlines the use of boolean queries to answer queries about authors and titles. Additionally, the description of techniques for phrase queries and relevant document classification are achieved on the lyrics index.

5.1 Trie

The *Trie* or prefix tree is a data structure that enables searching for terms with a complexity of *O(k)*, where k is the size of the term. This structure leverages common prefixes among various terms to create a compact tree for efficient searching.

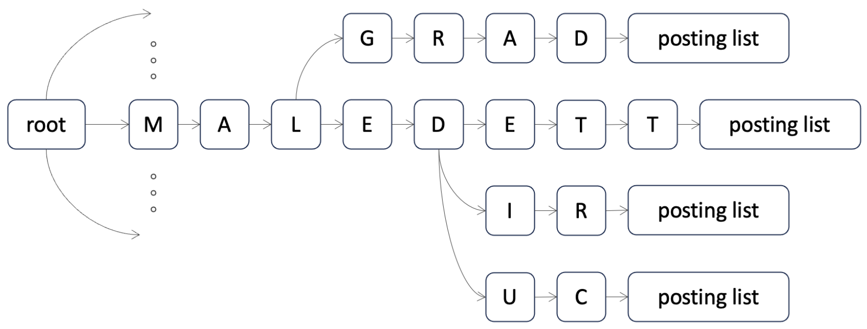

Figure 8: Trie data structure

Multiple instance of this structure are employed to load all the presented indices. Generally, the structure was designed to work with positional indexes but is also utilized to manage simpler indexes, such as those presented for authors and titles. In the following section, we explore the system's modes for generating results based on a given query.

5.2 Query execution

As mentioned earlier, the queries for which we seek answers are of different types:

- Retrieving the lyrics of a song given its title.
- Providing a list of songs given a specified author.
- Retrieving the lyrics of a song given some keywords or a phrase.

5.2.1 Author and title queries

The first two queries that involve the search for documents by author and title, are handled using boolean operations. Specifically, the posting lists of terms are identified after a search in the Trie, and the boolean **AND** operation is performed between the posting lists. This type of queries management allows us to address queries in which the user is looking for a single author, perhaps composed of multiple words. For example, the query *Claudio Baglioni* will return songs related to the full author name and not songs associated with *Claudio* combined with songs associated with *Baglioni*.

The responses to queries for author and title do not include a ranking of results, but only the set of documents containing the specified author or title is retrieved.

5.2.2 Lyrics and phrase queries

Searching within song lyrics is one of the additional queries that the system handles. Two searches have been implemented in particular:

- **Simple queries**: Keywords or words found in the documents are specified by users in this type and to retrieve the top k results, the Okapi BM25 ranking function is employed.

$$RSV_d = \sum_{t \in q} idf_t \cdot \frac{(k_1 + 1)tf_{t,d}}{k_1((1-b) + b\frac{L_d}{L_{avg}}) + tf_{t,d}}$$

- **Phrase queries**: The system returns the top k results after users specify a phrase contained within a song. Searching by phrase in a system like this is crucial because users often have phrases from songs in mind but may have trouble remembering the title. This issue is addressed by query management. Double quotes are used to specify phrase queries in the system.

6 Sound of words

Various algorithms have been created over time to convert a word into its corresponding phonetic representation. This can be highly beneficial in a system like this, as often we might not remember a particular song's detail precisely and may confuse a word with another that sounds similar. Here is a list of advancements made in algorithms of this nature:

- **Soundex**: An algorithm that encodes terms to represent their phonetic pronunciation. Created in 1930 with the primary objective of recognizing words that sound similar but are spelled differently. Specifically, the algorithm converts terms into a sequence of numerical characters, keeping the first letter unchanged and encoding subsequent consonants according to a specific mapping.
- **Metaphone**: A phonetic algorithm was devised by Lawrence Philips in 1990 to index words based on their English pronunciation. This algorithm was developed to improve Soundex, taking into account the variations and inconsistencies in English spelling and pronunciation. This led to a more accurate encoding.
- **Double Metaphone**: This algorithm is a more advanced version than the Metaphone algorithm. The aim is to address the ambiguous cases that were present in the previous version and introduce the handling of irregularities in different languages, including Italian.

A potential future enhancement for the system could involve integrating the *Double Metaphone* algorithm to make the system more accurate, taking into account the thematic content.

6.1 Conclusion

This report describes and implements [2] an information retrieval system applied to a real-world context, such as music search. In summary, it gave a general overview of various techniques which can be utilized in building such a system. There is a possibility of improving efficiency in the implementations discussed. This report serves as a foundational guide that presents potential implementations in a high-level language like Python.

References

[1] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008.

[2] Carlo De Nardin. https://github.com/carlodenardin-units/italian-lyrics-retrieval-system/tree/main.

[3] Lyrics Geniurs. https://lyricsgenius.readthedocs.io/en/master/contributing.html.

[4] Lingua. https://github.com/pemistahl/lingua-py.

[5] Porter. https://tartarus.org/martin/porterstemmer/.

[6] Simhash. https://algonotes.readthedocs.io/en/latest/simhash.html.