Evaluation of n-gram language models for lyrics retrieval

Thijs van den Hout
Radboud University
Nijmegen
t.vandenhout@student.ru.nl

Luke Peters
Radboud University
Nijmegen
l.peters@student.ru.nl

Lisette Boeijenk
Radboud University
Nijmegen
e.boeijenk@student.ru.nl

ABSTRACT

This report introduces a probabilistic information retrieval model for song lyrics. Unigram and bigram language models were created on the basis of an open-source data set containing 266,556 songs with corresponding lyrics. Users can issue a query to the program to receive a list of songs ranked by relevance, much like a traditional search engine. Results can also be re-ranked on bigram or trigram presence. We evaluate the performance of our models with a user study, and assess computation times. In this report we reflect on the process of creating our retrieval system, its evaluation, and the challenges we faced.

1 INTRODUCTION

With the vast amount of music available these days, finding different ways of searching for music is increasingly relevant. Much research has been done in the field of music information retrieval; retrieval systems on the basis of all musical aspects have been introduced. Research by J Steven Downie in 2013 claimed music information retrieval is inherently difficult due to its multicultural, multi-experiental and multidisciplinary aspects [3]. A study by Brochu et al. introduced a probabilistic method of jointly modelling text and music for music retrieval [1]. This resulted in the possibility to search for music by humming or singing, alongside playing a song fragment. Closely related to their research is the field of language modelling. In language modelling, probabilistic models are created to rank the likelihood of a query being constructed with a particular document in mind.

In this work, an implementation of probabilistic language modelling will be introduced in the domain of music information retrieval. With the broad availability of textual music data (e.g. lyrics) on the web, language models can be created that allow a precise and efficient way of song retrieval, In comparison, MIR systems using audio signals require more processing time, due to their larger file size and representation complexity. Therefore, this research will target the symbolic representation of music, leaving out the auditory side of the music information.

In this report, we describe our implementation of an information retrieval system designed to retrieve song lyrics from queries related to the song, like a sentence of the lyrics. For our approach we introduced a unigram and bigram model, which were later evaluated on mean first rank (MFR) and success at k (S@k). Moreover, to test which model worked best, we formulated the following research question:

What is the effect of using different n-grams in a probabilistic language model on music information retrieval?

By answering this question, we introduce an alternative method of music information retrieval through probabilistic language modelling, which could inspire future research to use similar natural language processing methods for MIR. Furthermore, by analyzing different language models for lyrics retrieval, we can provide insights on how and why these models are beneficial for information retrieval problems.

2 DATA

The data set used in this research project was obtained from Kaggle [5], a website on which data sets can easily be created and shared with the public. This data set contains 266,556 songs[1] with the following information: artist name, song name, genre, year and lyrics. For the purpose of this study we only require the song name, artist name and lyrics.

2.1 Preprocessing

Inevitably, the data set must be cleaned before it can be turned into a language model. In this section, some preprocessing steps that we used are explained. Firstly, the artist and song names were split by a hyphen instead of a space, we changed this for the representation of the results. We cleared the data set of any empty values in the artist name, song name or lyrics. Irrelevant columns were dropped (year, genre). We made the decision to replace all punctuation by white space, to aid in the tokenization of the lyrics at a later stage. We then applied the tokenization function of the NLTK Python package[2] to the lyrics and subsequently stemmed the tokens. Stemming is done to increase the term overlap between query and document: light, lights and lighting will all be mapped to 'light', so a query with any form of 'light' can find documents containing any form of 'light'.

3 IMPLEMENTATION

All code and other relevant data of this research project can be found on the GitHub page of one of the authors at https://github.com/tvdhout/lyrics-retrieval.

The implementation of all data preprocessing and language models was done in Python 3 (Jupyter Notebook). We created a unigram language model and a bigram language model, with the option to re-rank results on the presence of bigrams and trigrams.

3.1 Unigram model

The preprocessing steps left us with a list of stemmed tokens for each song. We use the FreqDist function from the nltk.probability package to get the frequency of each term in the song's lyrics, this can be interpreted as a dictionary. At this time, such a model is also created for the complete collection of lyrics to allow smoothing

Information Retrieval, December 2018, Nijmegen, the Netherlands

[1] After preprocessing
[2] https://www.nltk.org/

in the retrieval methods later. To greatly improve computation time we created an inverted file index from these dictionaries with term:count pairs, i.e. transform the model to have the unique terms on the first axis instead of the documents. This transformation reduced the computation time in the bigram model from 15 seconds to 0.2 seconds on average. After adding the total count of a term to the resulting dictionary we have the following data structure.

```
{term : [total_count, {index:count, ...}],
 term : ...}
```

Where `index` is the index of a song in the data and `count` is the count of `term` in its lyrics, for each term in the collection and for all song indices containing that term. The total number of occurrences of `term` in the collection is captured in `total_count`. This inverted index contains 354,879 unique, stemmed terms.

3.2 Bigram model

The bigram language model is very similar to its unigram counterpart. From the list of stemmed terms bigrams are extracted. Bigrams are pairs of two consecutive terms. Now we have a list of bigrams for each song, from which the counts are extracted and again an inverted version of this dictionary is constructed. Unmistakably, the number of unique bigrams is much larger than the number of unique terms. The inverted index for the bigram model contains 5,092,535 unique bigrams. At the same time, each bigram occurs in fewer songs than terms do. This is an advantage for the inverted file system because dictionary lookups are computationally cheap, while for each lookup, the total list of songs in which it occurs must be visited. If this list is short, computation is faster.

3.3 Retrieval

In language model retrieval systems, we want to simulate the probability that the query is constructed with a particular document in mind. On the basis of this assumption, we rank the documents according to their likelihood of the query being constructed from that document. The relevance of a document to a query is defined by the user's information need, which is often restricted knowledge. In the case of our system however, the information need is clear: find the lyrics of a song from which you only know a small fragment. For this information need, a relevant document is one which is likely to be the song that the user is searching for with their query. Since we assume the user constructed the query with this song in mind, we may assume the terms in the query occur in the sought song. We denote this quality with term frequency (TF), the number of occurrences of query terms in a document. Furthermore, we can assume a term is extra relevant to the query if it occurs only a few times in the collection, as it increases the chance that this term is distinguishable in the song. To achieve this quality we can take the inverse of the document frequency of the term (IDF), one divided by the number of documents in which the terms occurs. To avoid the probability of a query being constructed from a song to be 0 if any $TF = 0$, a smoothing factor λ is introduced. This factor interpolates the frequency of a term in a document with the frequency of that term in the collection. A larger smoothing factor brings all probabilities closer together, to the point where they are all equal at $\lambda = 1$. Often, a large smoothing factor works best. In our models we employ $\lambda = 0.85$. When we bring all these retrieval aspects together, we arrive at the following formula.

$$\log P(Q|D) = \sum_{i=1}^{n} \log((1-\lambda)\frac{f_{qi,D}}{|D|} + \lambda\frac{C_{qi}}{|C|})$$

Which we can rewrite for performance, while retaining equality in rank to the following.

$$\sum_{i:f_{qi},D>0} \log\frac{(1-\lambda)\frac{f_{qi},D}{|D|}}{\lambda\frac{C_{qi}}{|C|}} + \frac{\log|D|}{\log|C|} + 1$$

Where f_{qi}, D is the frequency of query term i in document D, $|D|$ being the length of document D and C being the collection of documents. Adding $\frac{\log|D|}{\log|C|}$ to the score acts as a length prior and prevents documents with very short lyrics to have an inherent advantage over longer documents. This retrieval formula is implemented in our Retrieval notebook in two separate ways: one employing the unigram language model (see Appendix A.1) and the other the bigram language model (see Appendix A.2). Both methods results in a ranked list of documents in decreasing order of relevance to the query.

3.4 Re-ranking

To improve the accuracy of our ranked results page, songs can be re-ranked based on the occurrence of query n-grams in their lyrics. N-grams give us more information about the order of words in the song, which is especially useful in our case, as queries will often contain partial lyrics which are consecutive terms. If a document contains a high number of consecutive query terms, it is likely to be more relevant than when a document simply contains many query terms. This idea is expressed in our `rerank_ngrams` function which can re-rank a list of (already relevant) results based on the number of query n-grams that are present (see Appendix A.3). n, the number of consecutive terms, is variable and can be passed on as a parameter. N-grams are computed on the fly from the list of tokenized lyrics, as the n-grams are variable and only the songs that have already been retrieved must be parsed. In our experiments we re-ranked the top 100 results obtained from the language models. Re-ranking 100 results on n-gram presence takes 0.05 second on average.

4 METHODS

To test the music information retrieval system and to evaluate the effects of the different n-gram models, a set of representative queries were needed. Generating representative queries to evaluate the system on is quite a challenge. Our approach will be described below, as well as the chosen evaluation techniques. First we will discuss the proxy queries we did initial tests with.

4.1 Proxy queries

Before we obtained queries from the participants of our user study we needed a proxy to test the retrieval system. For this purpose we created a function which returned a proxy query from a random song in the data set (see Appendix A.4). The function extracts a random sequence of consecutive words from a song. The length of the sequence is variable, but we generally used 6 terms as we thought it representative of actual lyric queries. These proxies

are not completely representative as they do not contain typos or misheard lyrics, which noticeably favoured the bigram model with trigram re-rank, but they were good enough to work with.

4.2 Queries

For the final evaluation of the different models, we needed to have a reasonable amount of representative queries, queries which a real user might use for the system. We assumed that one would use such a retrieval system when looking for a song they previously heard. This might happen when listening to the radio in the car, or hearing a song at a party or a movie. In all of these situations, attention is paid to the music and lyrics only once someone recognizes their appreciation for it. Therefore, one might only hear a small part of a song actively, listening for key parts in the lyrics to remember and later search on. This is what we aimed to recreate in our user study.

The user study we developed to simulate this real life usage consists of the following components:

Instruction The participant was told that they would have to attentively listen to a short fragment of a song, with the task to think of a sentence (max. 10 words) that they would use when looking for that specific song on the web.

Randomized song The interviewer would use a python script to extract a random song from the database, if it was an English song, the interviewer looked up the song on Youtube, and picked a random time to start playing the song from.

Listening The subject attentively listened to the song fragment for about 30 to 45 seconds.

Mental task To simulate the person not being able to search for the song immediately in real situations, they were distracted with a mental task. This mental task consisted of solving a math problem such as $341 - 124$ or $24 * 13$, which required the subject to actively think for a short period of time.

Query formulation The subject is asked to formulate the query they would use to search for the song.

Steps **Randomized song** to **Query formulation** was repeated five times for each participant to counteract query bias. All queries that were generated by participants can be found in Appendix B.

4.3 Participants

The test subjects were mostly students or other persons in the same age range (median age 22.5) and were proficient in the English language. In total the user study was done with 10 participants (3 female, 7 male) to counteract user bias, resulting in a total of 50 queries.

5 EVALUATION

The mean first rank (MFR) score is used to evaluate the models and is computed as the mean over the query ranks of the instances where the target song occurs in the results list (rank 20 or better). Furthermore, the "success at k" (S@k) score will be used for evaluation of our models. For the latter measure, the number of query results in the top k is divided by the total amount of queries. This represents the percentage of query results being in top ranks. In known item search this boils down to whether the target song is retrieved in the top k ranks. For this project, we decided to use S@20, S@10, S@3 and S@1; s@20 to obtain the percentage of queries for which its target is in the ranking at all, s@10 to get a good halfway measure and s@3 and s@1 because many people solely look at the top results when issuing a query. Additionally, we compared the computation time between models.

6 RESULTS

The ranking of each query for each model can be found in Appendix C. The results of our evaluation methods are displayed in table 1.

Model	Success at rank				MFR
	20	10	3	1	
Unigram	0.24	0.18	0.14	0.1	5.00
+ Bigram Rerank	0.48	0.48	0.48	0.46	1.04
Bigram	0.74	0.7	0.68	0.6	2.08
+ Trigram Rerank	0.74	0.72	0.64	0.6	1.76

Table 1: Success at rank and MFR results for the different models. MFR is computed over the results in the top 20 ranks to avoid a skewed mean.

In figure 1, we show the results of our timing studies. both the unigram and bigram model were tested on their computational time in relation to the number of query terms. The re-ranking method consistently took no longer than 0.1 second, indifferent to bigrams or trigrams re-rank.

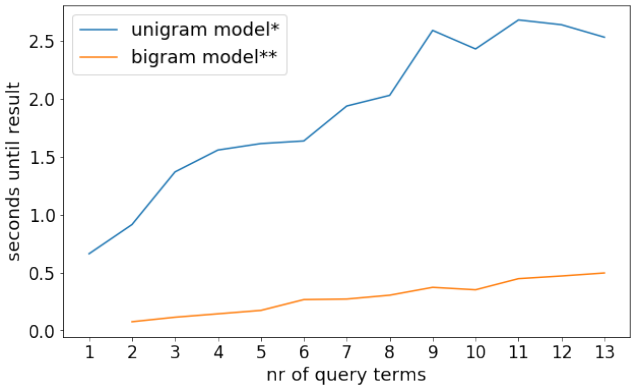

Figure 1: Timing studies: number of query terms vs. computation time

* average over 50 queries
** average over 100 queries

7 CONCLUSION

From these results we can derive a few conclusions. It is clear that the unigram model on its own fails to perform effectively: merely 24% of queries resulted in their target song appearing in the ranked list, and only 14% appeared in the top 3 results. A bag-of words approach is not discriminative enough. Re-ranking the top 100 results on bigram presence improved the result list (top 20 ranks) quite drastically. If the target song appeared in the top 100, the

re-rank method gave it a boost to higher ranks because bigrams are more discriminative.

Initially retrieving results by bigrams improved results even further. Relevant bigrams occur in fewer songs, increasing the chance of the target song appearing at a high rank. 74% of all queries had their target song in the result list, many of which on a high rank. Re-ranking these results further on trigrams improved the average rank slightly, but not by much. The bottleneck to n-grams came from the user's ability to write a query with consecutive terms exactly as it is represented in the lyrics. This is why we tested language models up to trigrams, any larger n-grams would result in a decrease of accuracy.

The MFR evaluation may not perfectly reflect the trend of a model; large values skew an arithmetic mean. The S@k results give the best insight into the models.

With respect to the timing study, we can see the bigram model is noticeably faster. This is because for each bigram (or term) in the query, each document in which that bigram occurs must be visited. As mentioned before, bigrams occur in fewer documents which has a positive effect on computation time.

8 DISCUSSION

In this section we will reflect on the processes of this research, interesting findings, and what we would do differently if we were do this research project again.

8.1 Data set limitations

Every data set not tailored to a research is bound to have some limitations. Firstly, the Kaggle user who created this data set ran out of storage for their web crawler, so the resulting data set contains only artists with leading letter A through G. This could be seen as a bias in the data set, but with this many songs we assume it is a representative subset of all songs. Secondly, the data contains many duplicate lyrics. Multiple versions of songs are often included, such as its original, acoustic version and covers, resulting in duplicate lyrics in our data set. This made it difficult to automate the process of query testing. If we could reliably automate the process of evaluation, we could optimize hyper-parameters such as smoothing factor and number of results to re-rank on n-grams using the proxy queries. If we were to do the research project again we would spend more time on either finding a better data set or preprocessing such that (almost) duplicate lyrics are removed before creating the language models.

We also chose not to remove stop words from the lyrics, as they are often included in queries and provide too much information to discard. Searching for "Flowers in her hair" would be equal to searching for "Flowers and his hair", because the middle two words are discarded. Since users often search for (part of) a sentence from the lyrics, we need these words to find more n-grams in the lyrics.

8.2 Genius

At first we wanted to compare our retrieval system with a state of the art retrieval system such as the search API of Genius[3], a renowned lyrics website. However, we soon realized this was unfeasible as we did not have access to the same data set or have any insight in their retrieval methods, which would make comparison of results superficial. This is why we decided to compare our retrieval systems among themselves using the measures that gave the most insight in the ability of our systems.

[3]https://genius.com/

8.3 Evaluation techniques

We treated the retrieval of a song based on the lyrics as a *known item search*, where there is only one target song for a query. To evaluate a known item search, the mean reciprocal rank (MRR) is a popular measure [6, p. 180]. However, as [4] describes, RR is an ordinal scale, over which one should not compute a mean. As an alternative, [4] proposes the mean first relevant (MFR) measure, the mean rank at which the first relevant document is found. The "success at k" (S@k) measure is another popular measure in the field of information retrieval, also often used at TREC [2]. We decided to also use this measurement to obtain a better insight in the ranking of our models. Besides S@k and MFR, we also looked at rank biased precision, expected reciprocal rank and expected first rank. These all proved more disadvantageous, which is why we decided on S@k and MFR.

8.4 Query limitations

We found that some queries do not lead to retrieving the correct document in any of the models. One of the problems concerning those queries is that they are too general to distinguish the target song. For example the query "*hold you every day*" retrieves a lot of perfect matches, making it random whether the desired result is displayed. We had quite a large number of these queries with no results (11/50) split over 6 participants. In future research, we would ask more participants and have each participant create fewer queries to counter user-bias. Another problem concerning language is the use of slang and contractions. For example, the use of *"gimme"* instead of *"give me"* in either lyrics or query made it extremely unlikely to find the target song. In future research we advise to counteract this problem by integrating some sort of slang dictionary to map these instances to each other.

8.5 Scientific Relevance

Our research showed that unigram language models for MIR are ineffective, while bigram and trigram models showed to be the better alternative. Future research could be done to investigate how user queries can be optimally matched with song lyrics, using our work as a means of comparison.

REFERENCES

[1] Eric Brochu and Nando De Freitas. 2003. " Name That Song!" A Probabilistic Approach to Querying on Music and Text. In *Advances in neural information processing systems*. 1529–1536.
[2] Nick Craswell, Arjen P de Vries, and Ian Soboroff. 2005. Overview of the TREC 2005 Enterprise Track.. In *Trec*, Vol. 5. 199–205.
[3] J Stephen Downie. 2003. Music information retrieval. *Annual review of information science and technology* 37, 1 (2003), 295–340.
[4] Norbert Fuhr. 2018. Some Common Mistakes In IR Evaluation, And How They Can Be Avoided. In *ACM SIGIR Forum*, Vol. 51. ACM, 32–41.
[5] Mishra Gyanendra. 2016. 380,000+ lyrics from MetroLyrics, by *GyanendraMishra*. https://www.kaggle.com/gyani95/380000-lyrics-from-metrolyrics. Accessed: 2018-10-10.
[6] ChengXiang Zhai and Sean Massung. 2016. *Text data management and analysis: a practical introduction to information retrieval and text mining*. Morgan & Claypool.

Appendix

A CODE SNIPPETS

A.1 Unigram retrieval funtion

inverted_terms is the inverted index of unigrams, its data structure is mentioned in **3.1 Unigram model**. The formula used for the computation of *score* is mentioned in **3.3 Retrieval**.

```
def retrieve_1(query, max_heap_size=100, smoothing=.85):
    '''
    base: inverted unigram language model
    Retrieves 'max_heap_size' indices of the lyrics in the lyrics DataFrame in decreasing
        order of relevance to the query
    uses an inverted unigram language model defined in the preprocessing notebook
    applies smoothing with smoothing factor defined by 'smoothing'
    uses only unigrams
    '''
    stemmer = PorterStemmer() #stemmer
    #max_heap is a list with (score, index) tuples, where the index is the index of the song
        in the dataframe
    max_heap = list(zip(np.repeat(0, max_heap_size), np.repeat(-1, max_heap_size))) #
        initialise max heap with (0, -1)
    query = re.sub('\W', '␣', query) #strip query from punctuation
    terms = [stemmer.stem(t) for t in word_tokenize(query.lower())] #stemmed, tokenized query
         terms

    scores = {} #{song index : score}
    for term in terms: #sum over query terms:
        if term in inverted_terms: #if term in the collection:
            collection_count = inverted_terms[term][0]
            for key in inverted_terms[term][1]: #for each document in which term occurs:
                #add term score (proportional to TF, inversely proportional to IDF) to total
                    score of document
                score = np.log(1+ ((1-smoothing)*inverted_terms[term][1][key]/
                    song_total_terms[key])/((smoothing*collection_count)/collection_N))
                if key in scores:
                    scores[key] += score
                else:
                    scores[key] = score
    for key in scores:
        scores[key] += np.log(song_total_terms[key])/np.log(collection_N) #length prior
        if scores[key] > min(max_heap)[0]: #update max heap
            heapq.heappop(max_heap)
            heapq.heappush(max_heap, (scores[key],key))

    return [x[1] for x in sorted(max_heap, reverse=True)] #return indices of songs in
        decreasing order of relevance
```

A.2 Bigram retrieval function

This function is nearly identical to its unigram counterpart. It uses *inverted_bigrams* as inverted file index

```
def retrieve_2(query, max_heap_size=100, smoothing=.85):
    '''
    base: bigram language model with inverted file index
    same as retrieve_1 but uses bigrams
    '''
    stemmer = PorterStemmer() #stemmer
    #max_heap is a list with (score, index) tuples, where the index is the index of the song
        in the dataframe
    max_heap = list(zip(np.repeat(0, max_heap_size), np.repeat(-1, max_heap_size))) #
        initialise max heap with (0, -1)
    query = re.sub('\W', '␣', query) #strip query from punctuation
    terms = [stemmer.stem(t) for t in word_tokenize(query.lower())] #stemmed, tokenized query
         terms
    query_bigrams = list(nltk.bigrams(terms))

    scores = {} #{song index : score}
    for bigram in query_bigrams: #sum over query bigrams:
        if bigram in inverted_bigrams: #if bigram in the collection:
            collection_count = inverted_bigrams[bigram][0]
            for key in inverted_bigrams[bigram][1]: #for each document in which bigram occurs
                :
                #add bigram score (proportional to TF, inversely proportional to IDF) to
                    total score of document
                score = np.log(1+ ((1-smoothing)*inverted_bigrams[bigram][1][key]/
                    song_total_bigrams[key])/((smoothing*collection_count)/bigram_collection_N
                    ))
                if key in scores:
                    scores[key] += score
                else:
                    scores[key] = score
    for key in scores:
        scores[key] += np.log(song_total_bigrams[key])/np.log(bigram_collection_N) #length
            prior
        if scores[key] > min(max_heap)[0]: #update max heap
            heapq.heappop(max_heap)
            heapq.heappush(max_heap, (scores[key],key))

    return [x[1] for x in sorted(max_heap, reverse=True)] #return indices of songs in
        decreasing order of relevance
```

A.3 Re-rank function

This function uses *tokenized_lyrics*, a list of stemmed, tokenized words for each song.

```
def rerank_ngrams(indices, query, n=2):
    '''
    rerank indices on n-gram presence
    defines n-grams on the fly using nltk.ngrams with n = n
    this is rather quick since we have the list with terms for each song (tokenized_lyrics)
        and we only need to re-rank
    max_heap_size indices (from retrieval method)
    '''
    stemmer = PorterStemmer()
    query = re.sub('\W', '␣', query) #strip query from punctuation
    terms = [stemmer.stem(t) for t in word_tokenize(query.lower())] #stemmed, tokenized query
         terms
    query_ngrams = list(nltk.ngrams(terms, n)) #all n-grams generated from query

    ngram_counts = [] #for each index in indices, (-) the number of query ngrams that occur
        in those lyrics
    for index in indices:
        if index == -1: # empty result
            ngram_counts.append(1) #this is worse than 0
            break
        #list of lists of ngrams for each lyrics in the top results
        all_ngrams = list(nltk.ngrams(tokenized_lyrics[index], n)) #all ngrams for lyrics at
            index
        count = 0
        for ngram in query_ngrams: #for every n-gram from the query
            if ngram in all_ngrams: #if the n-gram is present in the lyrics: improve score
                count -= 1 #minus for sorting to remain order within the same counts (reverse
                     sort reverses this intra-score order)

        ngram_counts.append(count)

    #sort indices based on ngram_counts
    new_ranking = [index for _,index in sorted(zip(ngram_counts,indices), key=lambda x:x[0])]
    return new_ranking #indices from the lyrics dataframe in decreasing order of relevance
```

A.4 Proxy queries

```
'''
Get a query proxy: a string of k consecutive words extracted randomly from a random song
these queries are used to test the models and tweak hyperparameters
'''
def query_proxy(k=6):
    words=[]
    while(len(words)<k): #make sure the song has at least k words
        sample = lyrics_original.sample()
        words = sample['lyrics'].iloc[0].split() #split words

    r = random.randint(0,len(words)-k) #random location in the song lyrics
    query = '␣'.join(words[r:r+k]) #slice k words from lyrics
    print('Query:␣', query)
    print("Index:␣{}\nSong:␣{}\nArtist:␣{}".format(sample['index'].iloc[0], sample['song'].
        iloc[0],    sample['artist'].iloc[0]))
    return query
```

B QUERIES

Here we present all the queries that have been generated by participants, alongside the artist and song name from which they had to extract the query.

ID	Artist	Song	Query
1	golden smog	strangers	All the things i do ,i will share with you, we're one
2	daniel akakpo	glory and splendour	holy and faithfull, are you here
3	agnes	right here right now my heart belongs to you	right here, right now my heart belongs to you
4	christopher	mama	got a education paid for, aint no worry about the lease
5	fergie	it ain t nothing	like it aint nothing, nothing nothing nothing nothing nothing nothing
6	garzilli enrico	the world s greatest lover	I know the game much better than the rest
7	dean wareham	heartless people	Which way the power lies
8	champion jack dupree	can t kick the habit	I went to the docter and he shook his head
9	angie stone	green grass vapours	He looked at my eyes at night
10	gabriella cilmi	love me cos	You can't bring back yesterday
11	carly rae jepsen	picture	Tell me that you feel it, you and me together
12	diamond rugs	blue mountains	what you want I got it what you need
13	big l	fed up with the bullshit	stop em by any means the big I won't hesitate
14	company flow	last good sleep	now you all up in the family broke and nuclear
15	alberta cross	ghost of santa fe	I will get past it after all I...
16	everclear	sin city	I'm gonna win in sin city
17	celine dion	love is on the way	I'm coming as sure as the heavens
18	belinda	takes one to know one	liar, cheater, loser, takes one to know
19	boomkat	rip her to shreds blondie cover	she's so dull come on rip her to shreds
20	cowboy mouth	get out of my way	heart and soul are on fire get out of my
21	bonnie tyler	give me your love	Gimme your love, gimme enough
22	bun b	untitled flow	you call a nigga grandma, you better bring a hammer
23	floorpunch	shottsie	hold you every day
24	elliott yamin	3 words	to say it's over
25	elton john	the last song	i misjudged love between a father and a son
26	bee gees	love never dies	love never dies, love goes on
27	dolly parton	life s like poetry	life is to short to think about wrong or right
28	ayreon	day four mystery	moment of destruction
29	the buck owen s buckaroos	christmas shopping	I've got ten toy soldiers
30	dj bobo	try try try	Everything you have to do, try try try
31	foghat	love zone	closer and closer never let me go
32	austra	darken her horse	it's all insane it's all insane
33	the crowns	yeeeaaahhh	whenever we're around the place is going
34	del shannon	it's my feeling	no no no no I can't be wrong
35	craig david	insomnia	how this will go maybe in time
36	frank zappa	conehead	kind of a girl thing
37	departed	prayer for the lonely	I don't wanna be with this misery
38	drake	friends with money	flow is heavy like a wet sponge
39	boo radleys	stuck on amber	I stare at my face
40	alkaline trio	cringe	The last thing I ever saw
41	far from heroes	the tallest tale	There's room for many more
42	association	the time it is today	All I know is what I feel
43	backyard babies	stars	Don't talk to me misery
44	armin van buuren	wasting hidden logic presents luminary	I'll play the game
45	ace hood	knock knock	knock knock bang bang
46	bad religion	get off	get off
47	the coup	busterismology	they want a revolution
48	al green	together again	i want to hold your hand
49	boney m	new york city	new york city
50	buddy guy	too many tears	now things have changed and I've got to go

C RESULTS

Here, the ranks obtained by each query for each model are presented. *uni* is our unigram model, *bi* is our bigram model, and $+bi$, $+tri$ mean the results are re-ranked on bigrams and trigrams respectively. Blank cells indicate the model did not rank the target song in the top 20 ranks.

ID	uni	uni+bi	bi	bi+tri
1			2	2
2		2	1	4
3		1	1	1
4				
5				
6		1	1	1
7		1	1	1
8			16	4
9			1	1
10			16	
11			1	1
12				2
13		1	1	1
14				
15		1	1	1
16		1	1	1
17			1	1
18		1	1	1
19		1	1	1
20		1	1	1
21				
22			3	1
23				
24		1	2	1
25	1	1	1	1
26	3	1	1	1
27	1	1		
28				
29	1	1	1	1
30	15	1	1	1
31			1	1
32			1	14
33	11	1	1	1
34			1	1
35			1	1
36				
37	12	1	1	1
38	5	1	1	1
39	6	1	1	1
40		1	2	5
41			1	1
42			6	4
43				
44				
45	3	1	1	1
46				
47				
48	1	1	1	1
49	1	1	1	1
50			1	1