Song Lyrics Search Engine

Chenfeng Fan, Shi Qiu

December 2019

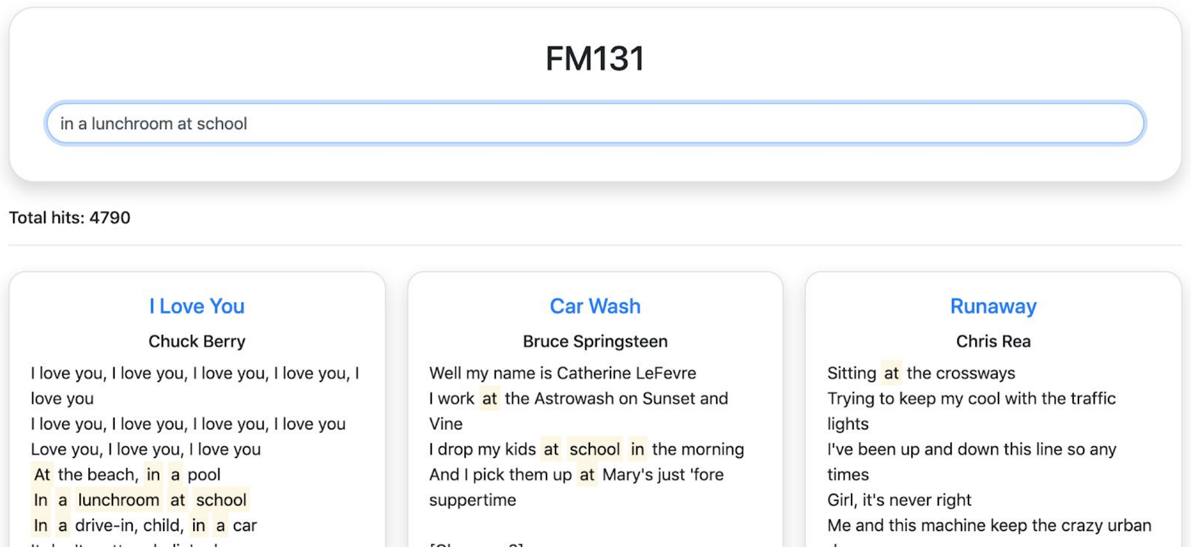
Abstract

We build a lyrics search engine based on vector space model and TF-IDF values. The search engine requires a piece of input lyrics, and returns all songs with relevant lyrics and artist in descending similarity order. Besides model building, we also implement util package and user-friendly interface that helps to accomplish our desired functions.

1. Introduction

We all had such experience that a piece of lyrics was lingering in your mind, but you cannot recall from which song it came. Or maybe they are just some certain words from a song, yet we cannot puzzle the whole lyrics from our memories. Such experience provides us the motivation behind this project.

The goal of our project is designing and building a lyrics search engine from data of some existing pop songs. The search engine returns the most relevant song based on the keyword, which requires a user's input a piece of lyrics. Our implementation follows the steps of util package building, vector model building, search function building and user interface implementation. (Figure1)



FM131

in a lunchroom at school

Total hits: 4790

I Love You
Chuck Berry
I love you, I love you, I love you, I love you, I love you
I love you, I love you, I love you, I love you
Love you, I love you, I love you
At the beach, in a pool
In a lunchroom at school
In a drive-in, child, in a car

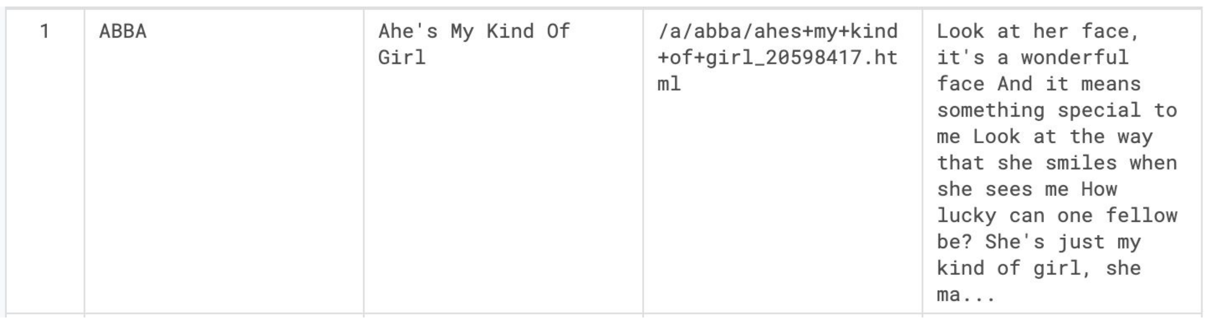
Car Wash
Bruce Springsteen
Well my name is Catherine LeFevre
I work at the Astrowash on Sunset and Vine
I drop my kids at school in the morning
And I pick them up at Mary's just 'fore suppertime

Runaway
Chris Rea
Sitting at the crossways
Trying to keep my cool with the traffic lights
I've been up and down this line so any times
Girl, it's never right
Me and this machine keep the crazy urban

Figure1. Search result

To deepen the linguistic processing procedure, we build our own tokenizer, lemmatizer and stemmer instead of using the ones from NLTK package. Using a vector space model and TF-IDF values, our search engine will compare input keywords with the existing lyrics to find the songs that have high similarity with keywords. The search results are listed from the most relevant song to the least relevant one. In order to improve user experience, we design and implement a user interface for search engine along with AJAX auto refresh our result web page.

The song data we used can be found at kaggle
https://www.kaggle.com/mousehead/songlyrics
Acknowledgement to Sergey Kuznetsov. The dataset includes four categories: Artist, Title, Text(lyrics) and Links.(Figure 2) All three categories but the links weighs in our search results.



1	ABBA	Ahe's My Kind Of Girl	/a/abba/ahes+my+kind+of+girl_20598417.html	Look at her face, it's a wonderful face And it means something special to me Look at the way that she smiles when she sees me How lucky can one fellow be? She's just my kind of girl, she ma...

Figure2. Dataset Sample

Running requirement:
Click==7.0
Flask==1.1.1
itsdangerous==1.1.0
Jinja2==2.10.3
MarkupSafe==1.1.1
nltk==3.4.5
six==1.13.0
Werkzeug==0.16.0

Work split:
Chenfeng Fan - search algorithm, vector model build, user interface build, report draft
Shi Qiu - util package build, vector model build, report draft

2. Implement and Mechanisms

2.1 Tokenizer, Stemmer and Lemmatizer

At first, we build a util package: util.py which includes some helper functions for vector space model analyzing. The goal of implementing this util package is to provide us any needed functions, while we could avoid using nltk tokenizer and nltk stemmer. This part also aims to deepen our practice in language processing.

Tokenizer

```
-- util.tokenize(String): list of all tokens in the string
```

The tokenizer was built using regular expressions. We use nltk regular expression findall to scan through the string for each pattern of word, ellipsis, white lines, dashes, brackets, punctuation, quotes and fall through patterns.

Lemmatizer

```
-- util.lemmatize(String, POS): lemmatized form of this word
```

The implementation of lemmatizer requires to pass a word and its part of speech. If no POS passed, it will assume this word is a noun. It uses nltk wordnet morphy as a helper. Upon taking a word and its POS, it will delete the word's suffix and reform it to its closest form. However, the stepback of this implementation is obvious. Firstly, it requires the POS to perform accurately. Secondly, for some more complex words, such as "Internationalize" will not be lemmatized with this lemmatizer. We then decide to build a more functional stemmer to overcome the shortages.

Stemmer

```
-- util.stem(String): stemmed form of this word
```

We design this stemmer in a way that it counts for some more complex cases rather than just removing the last letters of current suffix and returning the result. The coding skill is not complicate in this part, yet the challenges are in generating the linguistic rules for suffixes removing. As the outcomes, this implementation works better than wordnet.morphy lemmatizer. The stemmer also does not require to input the POS of target word. For example, the word "Internationalize" is stemmed into "intern"; the long word "pneumoencephalographically" is stemmed into "pneumoencephalograph".

The algorithm for the stemmer is consists of 6 main steps, each step handles a set of some suffixes. Before starting step 1, the word is also preprocessed with "y's" which could be considered as vowels and consonants. Step 1 handles the situation such as "-ied, ies, etc."; step 2 removes suffixes "eedly, ingly, edly, etc."; step 3 transform any "y" at the end of the word to "i"; step 4 handles "ization, ational, fulness, etc" and restores the word into its next closest form; step 5 handles "alize, icate, etc"; and step 6 removes "able, ible, ment, etc". A word has to go through all these steps in order to be returned as a stemmed form. Due to the multiple filters, our implementation of stemmer works better than the lemmatizer mentioned above.

Other functions

```
-- read_json(file): loaded file
-- write_json(file): write json file into directory
-- cvs2dict(file): dictionary(key=index of a song, value =lyrics
of this song)
```

2.2 Vector Space Model

Theory

The basic idea in vector space model is that we extract information from raw data text, generate a vector of all tokens in one song's text and another vector from query, and calculate the cosθ for the two vectors to get the similarity between these two documents. The mathematical model for vector space model shows as following:

$$\cos(d_j, q) = \frac{\mathbf{d_j} \cdot \mathbf{q}}{\|\mathbf{d_j}\| \, \|\mathbf{q}\|} = \frac{\sum_{i=1}^{N} w_{i,j} w_{i,q}}{\sqrt{\sum_{i=1}^{N} w_{i,j}^2} \sqrt{\sum_{i=1}^{N} w_{i,q}^2}}$$

To get each vector, it also requires to obtain tf-idf value for each token in a document, which evaluates how important one word is to a document in a corpus. The mathematical formula for ti-idf shows as following:

$$(1 + \log f_{t,d}) \cdot \log \frac{N}{n_t}$$

where $f_{t,d}$ is term frequency of this word, N is number of documents in this corpus, n_t is number of documents that contains target word.

Vector Generation

All vector generating functions can be found at helper.py.

1. Rawdata gather:
   ```
   -- get_song_corpus(): return a dictionary(song id, song
   corpus)
   ```
   Our raw data was gathered and returned as a json file which is a dictionary whose key is song id and values are artist, title and text of lyrics, and link.
2. Stemmed corpus gather:
   ```
   -- get_stemmed_song_corpus(): return a dictionary(song id,
   each row data of this song)
   ```
   Given raw data in step 1, we then stemmed all tokens from song text so that all cognate would be counted for the same word in tf-idf calculation.
3. Generate posting list, and invert posting list:
   ```
   -- get_postings_list(stemmed_corpus): returns two
   dictionaries
   ```
   Posting list is a dictionary of list, where key is a token and value is the list that all songs ID's for all songs containing the key token. In this way, we get the number of documents that contains one certain token. That is used in later tf-idf calculation.

Invert posting list is a dictionary of set, where key is a song ID and value is a set of all tokens this song contains. We used a set as value to expedite the running time, since we have a large song data with repetition of words, using a list will reduce the running speed.

4. Generate a term frequency dict:
   `-- get_tf_dict(stemmed_corpus): returns a dictionary of term frequency in each document`
   To get the tf_dict, we iterate every token in stemmed corpus. As a result, we have a dictionary whose key is a term, and the value is another dictionary which records the token frequency in each of the songs. This result is later used in tf-idf calculation, for the term frequency value.
5. Generate TF-IDF dictionary
   `-- get_tf_idf_dict(stemmed_corpus): returns a dictionary of tf-idf of every token in every document`
   In step 3 and 4, we have already obtained the term frequency and the number of documents that contains this term. Using the formula mentioned in theory, we now calculate the tf-idf value for each term in every of our documents. These tf-idf values will later be used to generate the vector model of for our corpus.
6. Generate normalize factor for vector model:
   `-- get_cos_norm(stemmed_corpus): returns a normalization factor for vector building`
   We still miss the item to normalize our vectors. In our case, we use the sum of square for each term's tf-idf value, and then take the square root of this value. The goal of normalize our tf-idf value is that we want to compare the term frequency in a relative percentage value rather than an absolute value. For instance, a word that appears 10 times in a 100-word-document should weigh more if it appears 10 times in a 1000-word-document. This normalizing also has a mathematical meaning, which is the length of our vector. We now obtain a normalizing factor for every song in our dataset.
7. Generate normalized tf-idf dictionary:
   `-- get_tf_idf_norm_dict(stemmed_corpus): returns a normalized value of our existing tf_idf dictionary`
   We have obtained every factor for vector model. In this step, we just generate the normalized tf-idf value using the normalizing factor in step 6. These values will be used for our searching algorithm.

Search Algorithm and Results

Implementation can be found at query.py.
We implement our search algorithm in this part, which includes query processing, searching keywords in dataset, and results returning. Our results depend on the normalized tf-idf values. We scan every term in query, and sums each terms' normalized tf-idf values for every song in dataset. The song that has the highest summation value is the most relevant one between

query and the song's text. Results are returned in descending order, so we can display the results from the most relevant one to the least one.

1. Query data processing:
   `-- parse_query_str(query_str): returns a list of query strings`
2. Search algorithm:
   `-- get_largest_score_doc(query_terms, page_idx, largest=10): returns a dictionary which key is the song ID, values are normalized tf-idf scores and matched terms from query in descending order`
   We iterate through all terms in the query, and look for every song in the dataset that contains this term. Then we sums the tf-idf value for this term in those songs, and save them to a dictionary where key is song ID and value contains the summation of tf-idf values. After we scan through all terms in query and all songs containing them, we return the dictionary in descending order of tf-idf values summation. So the first one should have the largest score, which means that it has the largest similarity with query.
3. Search result return:
   `-- get_doc_snippet(doc): returns a snippet for result page`
   For every song ID in the dictionary we got in step 2, we now return a snippet that contains song title, artist name and its lyrics. The snippet will be displayed on result result web page.

3. User Interface

3.1 Search Engine Interface

In order to give user an efficient query experience, we used Flask as our backend server to receive and process query requests from user input. For the frontend part, we used Jinja2 as the html template engine, and used Bootstrap UI framework to present an user-friendly interface.

Functions:

`-- results(): returns a results for query and represents results`
`-- movie_data(song_id): Given the doc_id for a song, present the title, artist and lyrics`

3.2 Webpage Autorefresh functionality

In order to bring the user experience to another level, we used Ajax to achieve the auto-refresh feature. This feature will display the search result changes while input keywords changes. It allows the user to see any possible result instantly without changing the input every time to see the one different outcome.

4. Conclusion

Our project completes the function to search for a piece of lyrics and return the result from the most relevant one to the least relevant one. We build our util package, vector model for song dataset, search model and frontend web page display to accomplish the desired functions. Yet, there are also obstacles and problems that we want to fix in further works. For example, in our project the stopwords are hard to deal with, since most of the lyrics consists of lots of stopwords. In this implementation, we did not remove the stopwords. Removing all stopwords might lead to lack of terms in search query, which make the search results less accurate. In later versions, we look forward to solving this problem with a better solution.