A technique for extracting song lyrics from web pages without knowing their structure

Dhruv Matani

October 20, 2006

Abstract

A search result for many song lyrics on popular search engines returns many mostly relevant results. However, the target pages are filled with ads, videos and images. Anyone searching for the lyric text would not be interested in all that paraphernalia. liblyric is an attempt to automate the process of scanning these individual result pages and extract the common textual content from them in the hope that the common parts will definitely be just the song's lyric text. The techniques that liblyric employs have turned out to give accurate results more than 90% of the time.

1 Introduction

There are many web sites that source and display song lyrics on web-pages for end users. These pages usually are bloated with advertizements, videos and other images. As an end user, it is more desirable to have the display of song lyrics integrated with your music player. Music players generally fetch song lyrics by parsing the content of web-pages of one or two well known sites for which they have built parsers and scrapers. However, these scrapers become useless since web-sites readily change their page structure and

- The music player writers may not be able to keep up with the rate of these changes
- It may not always be possible to push out updates to the music player at the same frequency at which the page structures change
- It may not always be possible to update any part of the already installed music player due to various reasons such as the user having suppressed it, etc... This drawback can be addressed by always making a lyric-fetch call to a local service which may be updated at any time. This however may lead to this service being abused by other music players.

A page-structure independent technique for fetching the song lyrics is needed if these shortcoming are to be addressed. We present one such technique that will almost always find the lyrics of a song if the text exists in more than two places on the web.

In the first stage, a search engine is used to fetch potential sources of information. These information sources are cleansed (stripped of everything but textual content) and a document similarity algorithm is applied to each pair of candidate sources. The document similarity algorithm not only quantitatively determines the similarity between the two documents but also returns the exact text from both documents that it deems to be the largest (approximately) intersecting block of text.

2 Data Sources & Cleansing

Any search engine such as Google[1] or Dogpile[2] or Yahoo![3] may be used to fetch the initial set[4] of documents.

These candidate documents are HTML files that are stripped of everything except for the actual textual content. This can be done using any tool[5] that strips the document of HTML tags and preserves the textual content of the pages.

[1] http://www.google.com/
[2] http://www.dogpile.com/
[3] http://www.yahoo.com/
[4] Currently, liblyric fetches about 8-10 documents
[5] Currently, liblyric uses unhtml for doing this

3 Document Similarity

The document similarity algorithm that is used to extract similar blocks of text needs to be fuzzy since different sources may have slightly mutated versions of the same song lyrics. This happens because:

- Different lyric contributors may use punctuations differently and spell certain words differently (for example, color and colour). This can be mitigated to a certain extent by ignoring punctuations and trying to stem words. This is language dependent and will work only for a specific set of languages for which stemming has been implemented.
- Contributors may get lazy and instead of copying a line that appears consecutively four times, they might just add *(Four times)* after the first time it occurs.
- Some contributors may add text such as *(solo)* and *(chorus)* and specify the singer more accurately than others.
- Some providers insert textual advertizements between two lyric verses.
- Some providers include links and text for downloading ringtones that have the same tune as the song whose lyrics are being displayed.

Even with all these problems, some of these documents are equally accurate and it would be reasonable to show any of them (without the advertizements) to the end user.

The fuzzy document similarity algorithm that is used to find lyric text in web pages has these characteristics to let it work even in the face of noise in the content as mentioned above:

- It should be permissive enough to detect a match even if a certain contiguous blocks of text differ at a few places.
- It should not be so permissive as to wrongly detect uncommon blocks of text as common blocks of text.
- There should be a limit on how many mismatched sub-blocks of text (of a certain size) are allowed in a block of text that is deemed to be similar to another block of text in another document.

The fuzzy document similarity algorithm that is used in liblyric takes into account all the above requirements and returns the largest blocks of text from both documents that are deemed to be similar. The ranking algorithm (explained below) uses all these *matched* blocks and tries to rank the best match and produce a final result.

3.1 The Algorithm for fuzzy similarity

A fuzzy match between two documents is achieved by trying to match blocks of text and returning the largest block of text that is *fuzzily* common to both the documents.

For doing so, we maintain a candidate list of blocks for one of the documents. A candidate block is a block of text represented by the tuple *(documentID, start offset, end offset)* meaning that this is a block of text starting at offset *start offset* and ending at *end offset* in the document *documentID*.

If some candidate block in the 1^{st} document can not be extended by any text in the 2^{nd} document, a new block starting at that position in the 1^{st} document is created. Otherwise, all existing blocks that can be extended by the text found at the current offset in the $2nd$ document are extended and their end markers are appropriately updated. This process is continued till the end of the document has been reached at which point the largest candidate block is selected and returned as *the* common representative similar block of text in both the documents.

Appropriate fuzzy measures as detailed in the section above are incorporated while performing the activity of candidate block extension.

4 Ranking

The ranking algorithm is a fairly trivial idea. It just sorts all the candidate blocks based on the size (number of characters) of the block and chooses the 2^{nd} largest block as the final intersecting block. The choice of using the 2^{nd} block is purely empirical since I noticed that it gives a faithful representation across many tests.