Assignment 6 (Final Project)
Nadine Amsel
Nathanael Lenart

PhotoLyrics

Abstract:
PhotoLyrics is a new music visualization system designed to merge stimuli from different senses, particularly vision and audio. PhotoLyrics allows a user to actually visualize the lyrics in the song playing. The only input required from a user is submitting an mp3 file and the system takes care of the rest, allowing the user to sit back and enjoy the sensory overload. In this paper, we will discuss the implementation details of our system in depth, all the way from initial audio submission to final output presentation.

Overview:
Our system is composed of six different modules which are put together and run sequentially for a user-initiated search. These six modules are as follows:

1. Audio submission by the user,
2. ID3 tag extraction from the submission,
3. Retrieval of lyrics based on the artist and title information obtained from ID3 tags
4. Extraction of keywords from lyrics
5. Image retrieval based on keyword tags extracted from lyrics
6. Presentation of images to the user

In order to build our system, we depended on a number of technologies, the first of which are those necessary to run a web application. We used JRuby on Rails for this. Most of the processing was done in Java, however, which was useful since we were able to repurpose code written for other projects. Some of our modules implemented local search techniques while other modules implemented remote search techniques; namely, our keyword extracting module searched against a posting list we had already created during a large Wikipedia crawl, and our lyric finding module and image finding module used ChartLyrics and Flickr APIs, respectively. To facilitate the processes of extracting ID3 tags from mp3 files and obtaining URLs of images on Flickr, we utilized two open-source Java packages that can be found easily on the web at http://www.sourceforge.net: jid3lib and flickrj.

Discussion of main components:
Audio Submission:
Using JRuby on Rails, we created a few webpages for easy interaction with the underlying software. The landing page is the starting point to our software, presenting an upload form to the user. The server would then accept the provided upload, place it in the correct directory, update the database to point to the newly added file, and start the search process. As a side note, we also created a page that listed out all the files that had been previously uploaded so the search could be repeated without having to re-upload the file.

Getting artist and song name:
Once the mp3 has been uploaded, the Java program gets the artist and song name from the mp3's ID3 tags. This is done using the Java ID3 Tag Library (http://javamusictag.sourceforge.net). The Java ID3 Tag Library has a class called MP3File which creates an MP3File from a source file. The values in the ID3 tags can be obtained from the mp3file and the values are returned in the form of a Collection. Within this Collection is all of the information that is encoded in the mp3, including artist name, song title, album name, etc. Each of these values are in different tags. The song name is in a tag called "TIT2" while the artist name is another tag called "TPE1." Within the Collection, a Java function searches for these strings and does some String parsing to extract the artist name and song title. Most of the time the ID3 tags are encoded in ISO-8859-1 (Latin-1), which works great. However sometimes the ID3 tags are encoded in UTF-16, which turns out to be a jumbled mess of non-ASCII characters. We attempted to transform the UTF-16 tags into ISO-8859-1 but were unsuccessful. Luckily each ID3 tag specifies the encoding (UTF-16 or ISO-8859-1) so we can just ignore mp3s that have the ID3 tags encoded in UTF-16 and our web interface simply reports to the user that there was a problem getting the lyrics.

Clearly, this is not an ideal scenario, so we would like to implement a fall-back mechanism, other than just forcing the user to manually edit the ID3 tags of the mp3 and retry. One simple idea for this is to put artist and title text fields in the form along with the upload, allowing the user to specify the correct artist and title. This has the benefit of simplicity, but the drawback of incorrect submission. A more difficult solution would be to fingerprint audio files in advance using a custom fingerprinting routine and indexing the fingerprints in a database. Then, a simple fingerprint of the uploaded file followed by a database query could easily return the correct artist and title. An implementation of all three mechanisms should theoretically lead to a very high accuracy, but the immediate question to draw itself out is, "which method is most likely to be correct?" Ideally we would implement all three methods and attempt to find out the answer to this question, but unfortunately we did not have enough time.

Obtaining lyrics:
The lyrics are obtained by using the ChartLyrics API. ChartLyrics supports both the SOAP and GET protocol, so for simplicity and ease of use we chose to use the GET protocol. This involves constructing a URL based on the artist name and song title and performing an HTTP GET request on the URL. The first URL is a "SearchLyric" URL, which returns an XML file containing two important values: the LyricCheckSum and LyricId of the song (example: http://api.chartlyrics.com/apiv1.asmx/SearchLyric?artist=michael%20jackson&song=bad). The resulting XML is parsed to obtain these two values and then based on the LyricCheckSum and LyricId, a new URL is constructed. This is a "GetLyric" URL which returns an XML file containing the song lyrics (example: http://api.chartlyrics.com/apiv1.asmx/GetLyric?lyricId=1710&lyricCheckSum=2a3ea713422cbc97470b0c38c6e5a552). If any non-ASCII characters appear in the lyrics we remove them before passing the lyrics on to the next step. In our Java program the XML was parsed using Java DOM API for XML (included in Java). If the lyrics do not appear in the ChartLyrics database, our program returns and our web interface renders this error to explain this to the user.

One problem we noticed with obtaining the lyrics is that occasionally ChartLyrics returns the wrong lyrics based on the artist and song title input. For example, many rap/hip-hop songs have more than one artist (e.g. the song "What Them Girls Like" by Ludacris featuring Chris Brown & Sean Garrett). If all the featured artists are included in the "artist" tag, sometimes ChartLyrics won't be able to find it. For songs that feature more than one artist it seems that ChartLyrics includes this in the song title rather than the artist name (e.g. "What Them Girls Like (featuring Chris Brown & Sean Garrett)" by Ludacris). The XML returned from ChartLyrics sometimes has more than one choice of songs that fit the query, but we are just taking the first entry and assuming that it is correct (which it is most of the time.) We noticed that with this particular song, ChartLyrics did have the correct lyrics in the returned XML file but it wasn't the first song.

Extracting tags from lyrics:
Tag extraction from lyrics was arguably the most interesting and difficult portion of the assignment. After obtaining a string representation of lyrics for the submitted song, we tried to determine which words were the best words to search for. Ideally, these would be the most representative nouns of the song so that image searches on them return the most interesting and relevant results. As it is very difficult to do natural language processing to filter our results, we were unable to discover whether we were getting nouns or not. Our first step was to take the lyrics we were given and attempt to "fix" them, removing beginning and ending punctuation, stripping out non-ascii characters, and replacing newlines with spaces. This way, the lyrics became a single string with each word separated by a space, making it easy to parse and check individual terms.

One first strategy was to use a stop list to quickly eliminate meaningless words, such as prepositions and articles, before further mathematical processing. Next, we created a HashMap where each term was mapped to its corresponding count in the lyrics. This was used to determine the weight of the term in the query, assuming that a higher weight in a query meant a higher importance. For each term we found in the lyrics, we looked up its collection frequency in our Wikipedia posting list, figuring that the posting list would be a representative sample of the relative frequency of the term on the web. It seems intuitive that the more often the term is used, the more important the term is. This seems to be reasonable between the fact that we eliminated common words with the stop list and that our posting list only contains terms below a certain threshold (125,000 counts). We also tested the document frequency of each term in our posting list. However, we assumed the reverse of the collection frequency, namely, that if a term appeared in more documents, it was probably less important than a term that only shows up in a few specific documents. One more metric we used was the length of the term itself as it seems intuitive that longer words are less frequent than shorter words and are thus, more important. A small tweak we added that dramatically improved results was to include the title in the search. Instead of modifying our code to accept the title as well and create a new weighted scoring system, we found it was much simpler to inject the title 3-4 times into the beginning of the lyrics before submitting them to query program. This ensured that its weight in the query would be high, thus biasing the scores of the terms in the title up.

For each term in the query (lyrics), we calculated several parameters. Note that we will use the term "log" as follows to mean "log base 2." First, the term's weight in the query was calculated

by dividing the corpus size by the document frequency of the term and taking the log of the result. This was multiplied by the log of the frequency of the term in the query. In mathematical notation, *alpha = log(tf) * log(corpus_size/df)*. Second, we set a variable *beta* equal to the log of the collection frequency of the term. This is noticeably higher than the other variables, and thus, more dominant. Again, in notation, *beta = log(cf)*. Third, we set a variable *gamma* equal to the log of the division of the corpus size by the document frequency, or *gamma = log(corpus_size/df)*. Lastly, we took into account the log of the length of the term. Assuming most words are between 3 and 10 characters, this factor ranged from about 1.58 to about 3.32, so it was not a large factor, but it did seem to help empirically. These factors were all multiplied together to determine a final score.

It was difficult to come up with a better metric as the collection frequency dominated most of the terms -- using any kind of sum would not have been effective at distinguishing scores. After the score for a term was calculated it was placed into a HashMap with the key being the term and value being the score. We wanted to return only the tags with the highest score, but we realized it was not necessary to sort the entire collection if we used a max heap. We input the scores into the heap one at a time, allowing the heap to figure out the order. Then, after all the scores had been input, we simply pop the first 5 or so tags off the top of the heap and use these to image search with. Originally, we looped through each term in the lyrics in the order they appeared, meaning there was possibility for duplicates, necessitating the need for a HashMap so we could modify the score. However, we re-wrote the code to determine the counts of each word in the query beforehand, so looping meant no possibility for duplicates. This leads to a missed optimization step of placing the score directly into the heap and skipping the intermediate HashMap altogether.

Fetching images from tags:
Fetching images from Flickr was easy using the tags we had extracted from the last step. Flickr has a public API that anyone can use, provided you sign up and explain your purpose beforehand. After signing up, we were given secret API keys that we had to use to connect to the Flickr servers; these were simply programmed into our code. As we mentioned above, we didn't connect to the Flickr API directly; we used the open source flickrj library to manage the connection and make the calls for us. We set a data structure called SearchParameters with our tags and then set the output order. This was rather interesting as there were a number of sorting methods, such as date posted, date taken, interestingness, and relevance. We didn't care much about the date sorts, but we did try interestingness and relevance. Documentation on the difference in ranking methods is weak to non-existant so we resorted to trial-and-error methods to look for differences. Both methods appeared to match the submitted tags well, but it seemed like the photos found by "interestingness" tended to be more on the abstract or comedic side, where those found by "relevance" seemed to be a more conservative match. After setting the search parameters, we then submitted them to the photos interface, which queried Flickr and returned a list of photo urls. We could then iterate over this list to present them to the user in JRuby on Rails.

Presentation/Implementation
We used a couple of different languages to implement our application, the most major of which

is Java. Java is responsible for the bulk of our processing; it opens the mp3 file, gets the lyrics, extracts the tags, and fetches the image urls. This is accomplished by three Java classes: WikiQuery, LyricsQuery, and FlickrQuery. WikiQuery is started first, independent of the web application. It reads our posting list into memory and opens up a socket to listen on, spawning new threads and passing off the connection on every incoming connection. LyricsQuery is used to query ChartLyrics to get the lyrics for the submitted mp3 file. FlickrQuery accepts the mp3 file, calls LyricsQuery to get the lyrics, opens a socket to WikiQuery to get the tags for the lyrics, and then fetches images from Flickr based on the tags. In our original code, FlickrQuery's main function did all of the work, outputting intermediate status messages to a user through System.out.println() commands, but we had to refactor everything when we converted our command-line app to be a queriable interface from JRuby. This meant a couple of changes: splitting up the main() function into function pieces (which we probably should have done anyway) and converting all System.out.println() messages to be returns instead, so that our JRuby controller could get access to the intermediate variables and output them on the webpages.

We used JRuby on Rails as the interface between the user and our Java application. JRuby is a compilation of Ruby in Java, after being ported over from C. Since JRuby is underlying Java, this allows us to easily import and use Java applications in our JRuby code. It is simple as including Java support, requiring the proper Java .jar files, and importing and instantiating Java classes -- it even converts Java style to Ruby style, such as converting Java packages like "edu.uci.ics.flickr" to "Edu::Uci::Ics::Flickr," or wrapping the 'getConnection()' function in a Ruby function called 'get_connection()' instead. Installing the Rails framework on top of JRuby gave us exactly the interface we were looking for -- interfaces to our Java programs and easy webpage creation for simple interaction with the user. In addition, the Rails framework is easily extend-able through the use of downloadable plugins; one such plugin we used was a rails wrapper around SoundManager2, a hybrid JavaScript/Flash MP3 player. This plugin allowed us to play the uploaded MP3 files with only 4 lines of code.

Future Work:
The following is a discussion of various tweaks or additional components that would potentially improve PhotoLyrics:

1. Better ways of dealing with improper id3 tags (this includes tags that are non-existent and tags encoded in UTF-16). Potential fixes include: allowing the user to manually enter the artist and song title (and even updating the id3 tags for future use), fingerprinting audio files in advance using a custom fingerprinting routine and indexing the fingerprints in a database, or having the user manually edit the id3 tags and re-upload.
2. Implementing a method to get lyrics that aren't on ChartLyrics. Potential fixes include: having a back-up database for lyrics that aren't on ChartLyrics, allowing users to manually input the song lyrics, or even scraping other lyrics websites without an API.
3. Implementing a feedback strategy to notify our system when the incorrect lyrics are returned from ChartLyrics. This could involve allowing the user to edit the id3 tags so the artist and song titled is interpreted differently by ChartLyrics or have them input the

correct artist and song title, thus allowing us to construct a different query on ChartLyrics and get the correct lyrics.

4. Improving our tag extracting strategy. This would probably be the hardest to implement as we have already tried several different strategies and the automatic tag extraction still isn't as good as a human could do. A major reason why a human can do this better than a computer is because the important words have more to do with the interpretation of the song than the actual lyrics. This was quite improved after including the title in the search as well, but it could still be better. A possible solution would be implementing a natural language parser that attempts to retrieve only nouns and verbs (i.e. "frisbee" and "jumping" as opposed to "between" and "light"). Another way we could improve tag extraction would be to utilize stemming in both the posting list and in the query. For example, without stemming, the posting may have an entry for "allen" that occurs x times and an entry for "allen's" that occurs y times, but it would be helpful if they existed in the posting together as "allen" with a count of $x+y$. Then, employing the same stemming technique to the query, we could in theory get better counts and matches. This is quite difficult, though, as there a number of cases to take care of, such as possessive or past tense (e.g. stripping off "'s" and "ed"), as well irregular stem changes (e.g. "swim" to "swam").
5. Improving the speed of our implementation. It usually takes about 20 seconds for the final result to load. It would of course be ideal if this took more like 5 seconds or less. One potential improvement is to cache the results of mp3 files that users have already uploaded instead of repeating the entire process each time. Everything but the images can be stored in a database (success/error, artist, title, lyrics, and tags). Another improvement would be optimization of our Java code in general. For the most part, we moved on to the next piece of the project whenever something worked correctly without further thoughts of optimization. For instance, removing the unnecessary HashMap as mentioned above would remove one entire loop from the processing. Another optimization step would be to re-factor as much of the intermediate code as possible. One example of this is calculating the log of 2 using Math.log(2) in a number of places could easily have been stored in a variable first.

Conclusion:

PhotoLyrics is our attempt at creating a new type of music visualization system. Most music visualization systems consist of generating shapes and colors, either randomly generated or based on the beats in the music being played. We wanted to take this a step further and display images that were relevant to the song lyrics. We are excited about this project because to our knowledge no other music visualization techniques actually take into account song lyrics. We could see this eventually being a cool plug-in to iTunes or other media player software systems since the only required input is an mp3 file. (Of course we would first have to deal with potential copyright issues with the images from flickr but there might be a way to work around that.)