Design Document

CS F469 IR ASSIGNMENT - 1

# Domain Specific Information Retrieval System

# Museek – Lyric Based Information Retrieval

…………………………………………………………………………………………………………………………………………………………….

2016A7PS0006H R MONITH SOURYA

2016A7PS0033H SANJAY DEVPRASAD

2016A7PS0049H NIKHIL SREEKUMAR NAIR

2016A7PS0086H N VISHNU TEJA

# Goal

The application is designed to retrieve information about a song given the user knows more than 2 words from its lyrics. Retrieved information includes song name, artist name, lyrics, and a link to the page from which the lyrics were drawn from.

# Technologies Used

* **NLTK (natural language toolkit):** used for tokenising queries and lyrics.
* **PyQT:** front-end and UI development.
* **Pandas:** fast data processing and manipulation (corpus/dataset related).
* **Beautiful Soup:** web scraping to collect corpus/dataset for information retrieval system.

# Application Architecture

On the highest level, the application can be broken down into an Indexer, a Search operator, a Ranking mechanism, and the basic user interface.

# Indexer

The first step towards indexing the corpus is tokenising the lyrics of all the songs in the dataset. Besides the lyrics, the dataset includes the song id, song name, artist, artist gender, and a link to the source of the lyrics. The lyrics are tokenised into singleton terms, and then indexed using positional indexing. In general, for any term ti, doc ids di, and occurrences pi (for n occurrences),

ti : di : p1, p2, p3, ….., pi, ….., pn

denotes the method the indexer stores the positional indexes for every term’s occurrences.

The final results of the positional indexing operation are stored as a JSON file.

# Search

The query is split into biwords and searched with a proximity constant k, which denotes how far apart the constituents of the biword are from each other. In other words, the smaller the k value, the closer the phrase is to a potential search result. Also, the application processes a string through a spell check and phonetic mapping to minimise user error.

## Spell Check using Edit Distance

Each word of the user query is first run through an edit distance module that first checks if that particular word is a case of poor spelling. If it isn’t, it returns the same term and checks the next. If it is a case of poor spelling, then it first checks for the term in the index, and returns the same term if found within the index. If not even found within the index, then the application runs the edit distance algorithm (each edit corresponds to a replacement, deletion, or addition):

E\_dist1, E\_dist2 (functions within the spell check module) consecutively check for edit distance corrections and replace the user query term with a term that it finds in the index. If E\_dist1 (i.e. using one edit) is able to find any such term, then the spell check for that term terminates right there. Only in the case of an event where E\_dist1 finds no words to replace the current term, will it pass on to E\_dist2 which keeps a target of finding a suitable replacement in two edits.

The final choice for the word is made based on the probability of occurrence in the index. E\_dist1 and E\_dist2 don’t directly replace words but add “candidate” replacements to a list, and then a pick one among them based on higher probability of occurrences.

## Phonetic tolerance using Soundex Codes

On the chance that even E\_dist1 and E\_dist2 are not able to find any replacements from within the index, then it is probable that the user’s mistake lies with his/her phonetic interpretation of the lyric and not the spelling. For this, we compare the Soundex code for that particular term to those of all the indexed terms and see if it can return any.

Similar to the edit distance module, this too chooses the replacement term that has the highest probability of occurrence.

# Merge and Ranking

Positional indices of each term in the biword are observed and recorded if occurring consecutively. Biword/triword/simultaneous occurrences are recorded as lists with position elements.

e.g. query: “in the cloud”

Biwords here would be “in the” and “the cloud”. Let’s say positions recorder are [71, 72], and [72, 73] among many others. Then these would be merged to an inner list [71, 72, 73], thus, a list of lists would constitute the value part of a key-value dictionary. Ranking is based on two metrics, first on the length of the inner strings, and secondly on the number of such lists. After empirical trials, it was found that the equation

ri = (lengthi)4 x n(occurencei)

returned the better rank. Here, ri  = contribution to rank of song from the ith inner list, lengthi = length of inner list i, n(occurencei) = number of times an inner list of the same length occurs. Based on this, the songs are ranked and returned accordingly to the front end.

# Basic UI (front-end)

Once the indexing is complete and the JSON file is prepared, the UI can be launched. The UI is minimalist in the sense that it encompasses a typical search bar and an output pane to display all the returned results. In particular, it displays the song name, artist name, lyrics excerpt with the query text highlighted, and also a link to the page from which the dataset drew the lyrics from.

# Project Specifics

# Corpus Description

The corpus in its raw form is a .csv file with over 55,000 songs, with headers: id, song\_name, lyric\_link, lyric\_text, and artist\_gender.

# Data Structures

* **List:** to work with many related values, and perform the same methods and operations on multiple values at once. Implemented for storing positions, candidate substitutes for edit-distance, etc.
* **Dictionary:** (internally implements a hash-table)can be treated as a database of sorts for storing, organising, and retrieving data. Implemented for storing list of positions, Soundex codes, etc.
* **Skip List:** to traversefaster through the positional indices. This is very applicable in the scenario where the corpus is big and term incidences are very high.
* **Dataframe (Pandas):** (internally implements a hash table) helps handle huge datasets in a fast and efficient manner. Implemented during data scraping and manipulation.

# Pre-processing

Pre-processing involves curating data from the corpus for efficient indexing. For this, the text was regularised into lower case, all punctuation marks and articles were also removed.

# Run times for Search & Retrieval

Run times for search and retrieval were averaged for 20 searches, where each query was between 3 to 5 words long and yielded a mean value of 0.025 seconds.

……………………………………………………………………………………………………