Design Document For  
Domain-Specific IR System

short line

# Team:

1. Snehit Reddy (2017A7PS0868H)
2. Dhanush Karupakula (2017B3A71011H)
3. Sai Satvik Vuppala (2017B4A71449H)
4. Amit Raj Reddy D (2018B4A70813H)

# Topic

Implementing a search engine to find the song lyrics.

# Introduction

Domain-Specific Retrieval System is the extraction of information from a collection (resources) specific to a domain ( can be health, finance, songs, etc.). Searches in this retrieval system can be full-text based or can be a part of it. In this project, we are trying to implement a search engine based on music lyrics data that takes a query as an input and searches the query in the dataset, and gives back 10 relevant matches which might seem similar to the query processed.

# Libraries Used

These are the python libraries used in this project :

1. **math:** This library of python is used generally for all the mathematical functions.
2. **numpy:** This library is used to store the data in the form of an array and is used for easier array computations.
3. **pickle:** This is used to serialize the objects in python, here it is used to serialize the dictionaries which store our data.
4. **nlargest ( heapq ):** This library function of python is used to return the k largest elements from the iteration which satisfies the key, where k is specified as a parameter.
5. **time:** This was used to keep track of the amount of time that each process was taking.
6. **spatial ( scipy ):** This was used to calculate the cosine distance between two 1D arrays during the query processing.
7. **tabulate (tabulate):** We used this particular function to tabulate the obtained results and present them in the output.

# Implementation

# **Stemming and Tokenizing**

The original lyrics data is protected by copyright. So the dataset we use here is already a stemmed dataset in the bag of words format. The bag of words format will be explained later. In the stemming process there were some additional stemming rules that were applied in addition to the Porter2 algorithm. We use this same stemming algorithm for query processing.

One such rule of removing punctuation is “I’m“ is converted into “I am”. Similarly “can’t” become “can not”. This is because of the semantic similarity of these terms. Some of the lyrics also contain words such as ‘<’ ‘>’, these were considered fake words and removed from the list of terms.

**The Datasets**

The dataset used in this project is not in the form of a sparse data but is in the bag of words format. The first part of the dataset has all the terms from all the 210,519 songs. There are 5000 terms which are stemmed and tokenized. Each line in the second part of the dataset contains the unique track id, unique mxm id and the word count for each of the top words that are common in both the song and the first part and all these are comma-separated. This dataset which is in the bag of words format is used for further processing and creating the vector space model.

An instance of bag of words format from the dataset and it’s breakdown

**TRAAAAV128F421A322,4623710,1:6,2:4,3:2,4:2,5:5,6:3,7:1**

**TRAAAAV128F421A322 :** Track id

**4623710 :** mxm\_track id

**1:6 :** (Term frequency, tf) First word from the list of 5000 terms occurs 6 times in this song.

Along with this dataset we also have an additional dataset which maps the Track IDs to the name of the song and the Artist

**Corpus Preprocessing**

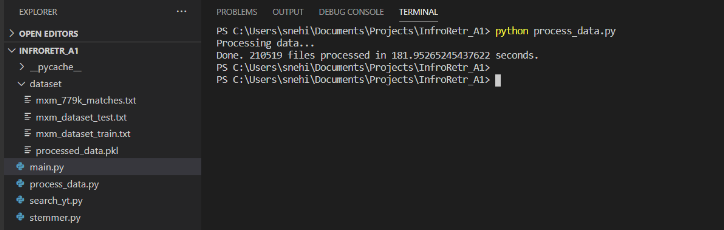
The corpus which is in the bag of words format is now used to create the vector space by calculating the tf-idf values. We create a list that contains all the 5000 terms. We then create a dictionary in which the key is space separate track\_id and mxm\_track\_id. The value is a list of size 5000 where each cell corresponds to the weight of each term. We also use this dictionary to calculate the document frequency (number of documents each term appears in) of each term and store it as well. All three data items are dumped into a pickle file to be used during query processing. The bag of words are then used to calculate the tf-idf score for each pair of (term, document) using a specific formula:

**tf**: is the term frequency

**idf**: the inverse document frequency

**df**: the document frequency.

It can be seen as obtained below that the preprocessing of the data and converting it into a pickle file takes around 180 seconds.



# **Query Processing in the Vector Space Model**

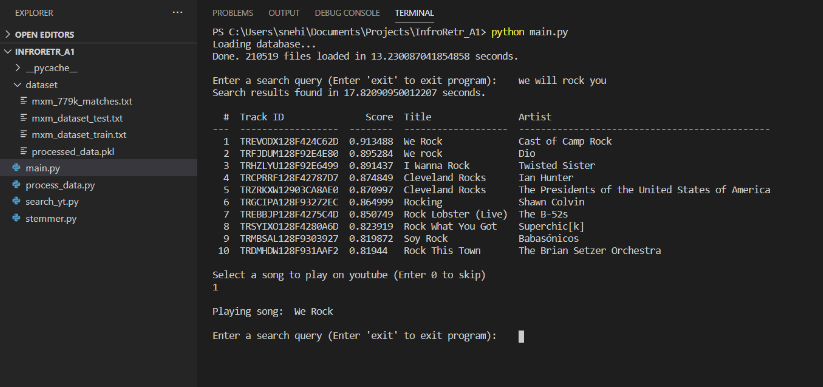
After creating the tf-idf dictionary, we use the pickle file, load the pre-processed dataset, and reuse it. Now, once the query is given, we preprocess the query by stemming and tokenizing it using the previously mentioned specific stemmer and tokenizer. We then create the vector corresponding to the query using the same formulae. Now using the cosine similarity technique we calculate the similarity index between the query and each document (here song). Using the nlargest function in the heapq, we generate the top 10 related documents by their similarity index scores. It takes around 15-20 seconds to give the result.

# Limitations of this Model

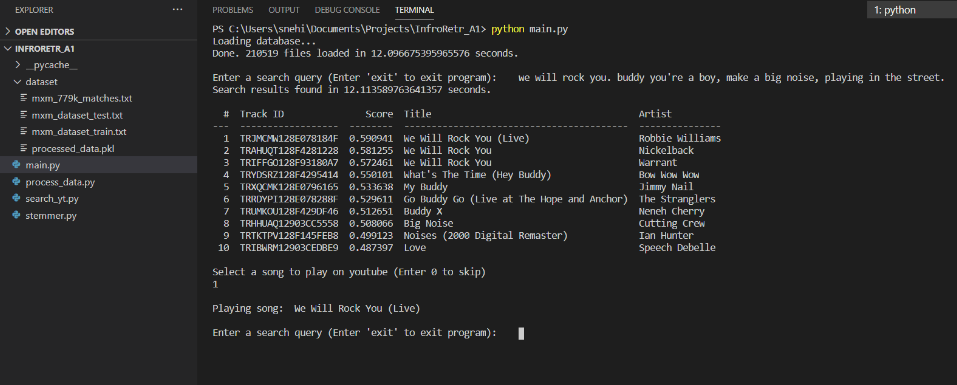
It is noticed on experimenting with different input queries that for smaller queries the result is often not as expected because the same words might be present more frequently in other songs even if not in the same order. Thus tf-idf doesn’t cater to the relative position of words.

For example, when the input "we will rock you" is given, we expect that the model would return the original song “We will rock you” or some other versions of it. But it resulted in giving some random songs in which our expected song might not even be present. This is because the model returns the songs which might contain the words of the query abundantly used resulting in a high similarity score.

The following screenshot shows the result when the input is “we will rock you”, and we can see that the output is as explained above.



Now in order to get the desired original song, queries of longer length work. When we search "we will rock you. buddy you're a boy, make a big noise, playing in the street" i.e. part of the lyrics, we obtained the desired song as our top result.



# Additional feature

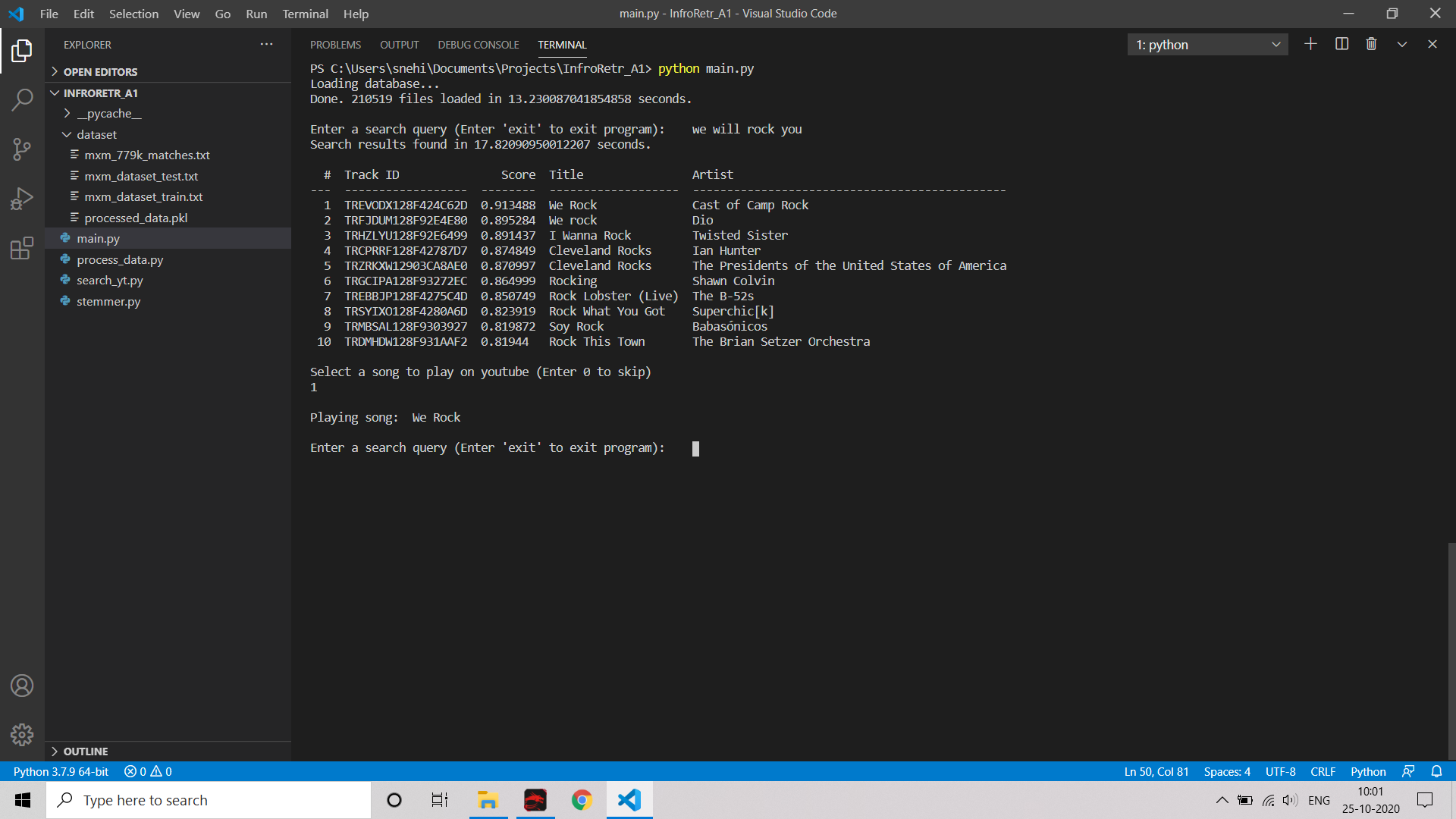
# **Playing a song from the output on Youtube**

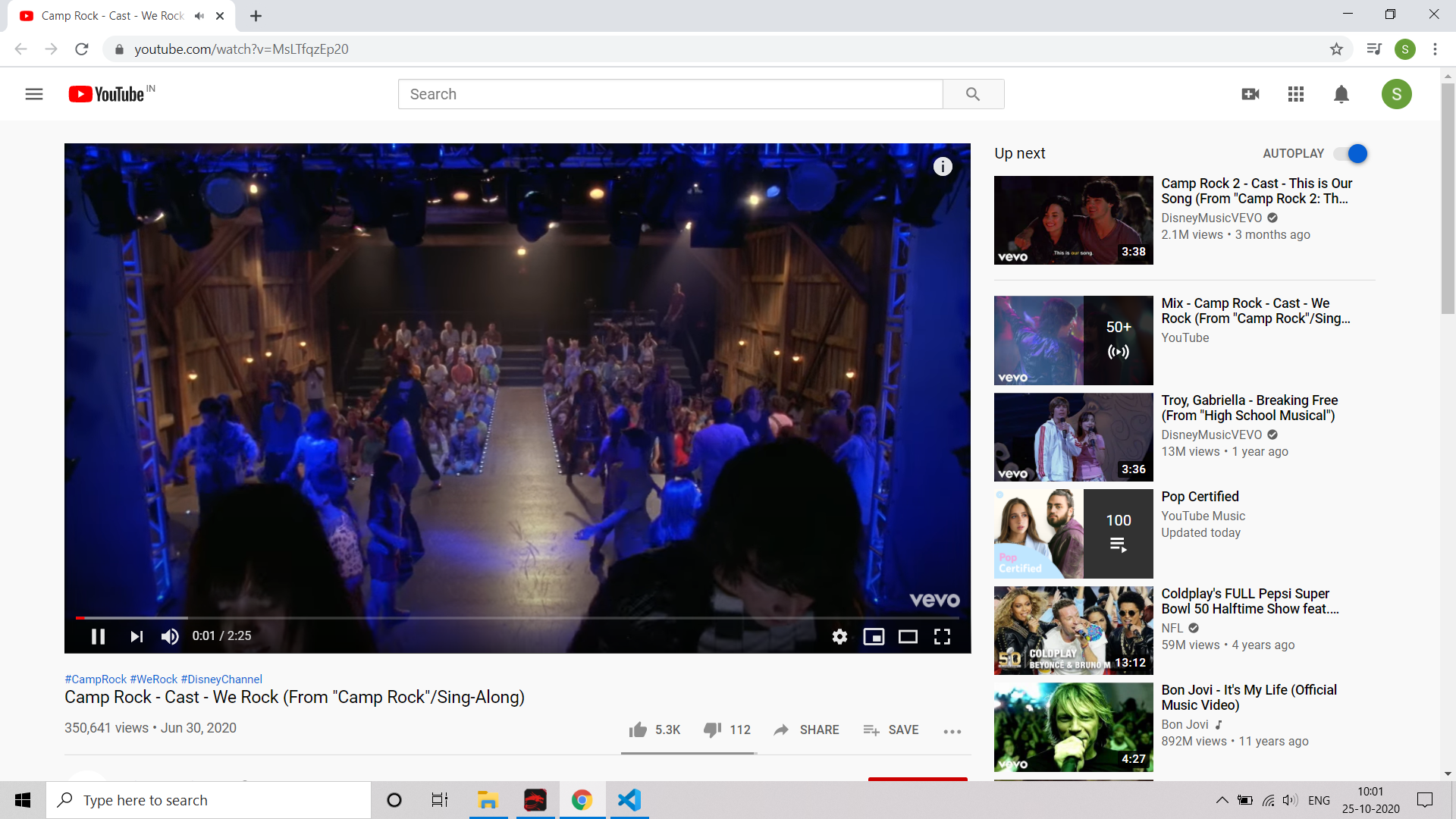
**Additional Libraries**

1. **urllib.request:** The functions provided by this module are used to open urls.
2. **urllib.parser:** This is used to convert a relative url to an absolute url which.
3. **re:** Regular expression module used to find all the results obtained
4. **webbrowser:** The first result is then played by simply calling the open() function from the webbrowser module.

We have provided an additional feature, keeping in mind that, more often people fail to recognize the song with their names but recognize them when they listen to it. This feature helps the user to verify whether one of the top 10 results is the one that the user was expecting by listening to them. A user can select one song from the top 10 results by giving the Serial Number in the column named ‘#’. Then the program searches for the song on Youtube and plays the first result.

When the user enters ‘1’ for the following query and its corresponding output. The first song (here, We rock) is played on youtube.



****

**A flowchart explaining the process in brief:**



# References

1. C. D. Manning, P. Raghavan, and H. Schutze. Introduction to Information Retrieval, Cambridge University Press, 2008.
2. Thierry Bertin-Mahieux and Daniel P.W. Ellis and Brian Whitman and Paul Lamere, The Million Song Dataset, Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011), 2011. ( [http://millionsongdataset.com/musixmatch](http://millionsongdataset.com/musixmatch/index.html) )
3. The dataset to create the model (bag of words format) : <http://millionsongdataset.com/sites/default/files/AdditionalFiles/mxm_dataset_train.txt.zip>
4. The dataset containing track ids mapped to song name and the artist name : <http://millionsongdataset.com/sites/default/files/AdditionalFiles/mxm_779k_matches.txt.zip>