

**RAMAIAH INSTITUTE OF TECHNOLOGY**

(Autonomous Institute, Affiliated to VTU)

Bangalore – 560054

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**Course:** Information Retrieval **Course Code:** CSE13

**Lab Record**

**SUBMITTED BY**

|  |  |
| --- | --- |
| **USN** | **Name** |
| 1MS16CS044 | K Sidhartha Nambiar |
| 1MS16CS052 | Lakshya Sharma |

**Term:** Jan - May 2019

****

**Ramaiah Institute of Technology**

**(Autonomous Institute, Affiliated to VTU)**

**Department of CSE**

**Course:** Information Retrieval **Term:** Jan - May 2019

**Course Code:** CSE13

|  |  |  |  |
| --- | --- | --- | --- |
| **SL. NO** | **Lab Assignment/Project/Report** | **Mark/s Assigned** | **Mark/s**  **Obtained** |
|  | Implementation of string-matching algorithms: KMP and Rabin Karp. | 1 |  |
|  | Implementation of Pre-processing of a Text Document | 1 |  |
|  | Implementation of Inverted index: Construction and searching | 1 |  |
|  | Implementation of Vector Space Model.   1. Rank 10 documents for a given query. 2. Computing Similarity between any two documents. | 1 |  |
|  | Implementation of probabilistic Model.  Rank 10 documents for a given query. | 1 |  |
|  | Implementation of various evaluation measures.   1. Calculate recall and precision values for all relevant documents and draw precision VS Recall Curve. Also calculate R-precision. 2. Compare performance of two IR algorithms for the same query q. 3. Calculate harmonic mean and E- measure (All three cases b=1, b>1, b<1). | 1 |  |
|  | Implementation of a Web Crawler. | 2 |  |
|  | Lab Project: Federated search | 10 |  |
|  | Lab Record | 02 |  |
| **Total** | | 20 |  |

INDEX

**Signature of Faculty**

1. **Implementation of KMP ad Rabin Karp string matching algorithms**

def KMPSearch(pat, txt):

M = len(pat)

N = len(txt)

lps = [0] \* M

j = 0

computeLPSArray(pat, M, lps)

i = 0

while i < N:

if pat[j] == txt[i]:

i += 1

j += 1

if j == M:

print("Found pattern at index " + str(i - j))

j = lps[j - 1]

elif i < N and pat[j] != txt[i]:

if j != 0:

j = lps[j - 1]

else:

i += 1

def computeLPSArray(pat, M, lps):

len = 0

lps[0]

i = 1

while i < M:

if pat[i] == pat[len]:

len += 1

lps[i] = len

i += 1

else:

if len != 0:

len = lps[len - 1]

else:

lps[i] = 0

i += 1

txt = "ABABDABACDABABCABAB"

pat = "ABABCABAB"

KMPSearch(pat, txt)

d=256;

def search(pat, txt, q):

M = len(pat)

N = len(txt)

i = 0

j = 0

p = 0

t = 0

h = 1

for i in range(M - 1):

h = (h \* d) % q

for i in range(M):

p = (d \* p + ord(pat[i])) % q

t = (d \* t + ord(txt[i])) % q

for i in range(N - M + 1):

if p == t:

for j in range(M):

if txt[i + j] != pat[j]:

break

j += 1

if j == M:

print ("Pattern found at index " + str(i))

if i < N - M:

t = (d \* (t - ord(txt[i]) \* h) + ord(txt[i + M])) % q

if t < 0:

t = t + q

txt = "GEEKS FOR GEEKS"

pat = "GEEK"

q = 101

search(pat, txt, q)

1. **Implementation of Pre-processing of a Text Document**

import nltk

import string

from nltk.corpus import stopwords

from nltk.stem import WordNetLemmatizer

import re

f = open("f1\\agora.txt","r")

text = f.read()

words = nltk.tokenize.word\_tokenize(text)

for i in range(0, len(words)):

words[i] = words[i].lower()

i = 0

while i < len(words):

if words[i].isdigit():

words.remove(words[i])

i -= 1

i += 1

i = 0

while i < len(words):

if words[i] in string.punctuation:

words.remove(words[i])

i -= 1

i += 1

for i in range(len(words)):

words[i] = re.sub(r'\d+|\W+|\_', '', words[i])

stop\_words = set(stopwords.words('english'))

i = 0

while i < len(words):

if words[i] in stop\_words:

words.remove(words[i])

i -= 1

i += 1

lemmatizer = WordNetLemmatizer()

lemma = []

for i in words:

lemma.append(lemmatizer.lemmatize(i))

processed\_file = ""

for i in lemma:

processed\_file+=i

processed\_file+=" "

print("\nProcessed Successfully")

1. **Implementation of Inverted index: Construction and searching**

import nltk

import string

from nltk.corpus import stopwords

from nltk.stem import WordNetLemmatizer

#UDF

def Document\_Preprocessing(readfile,writefile):

f = open(readfile, "r")

text = f.read()

f.close()

words = nltk.tokenize.word\_tokenize(text)

for i in range(0, len(words)):

words[i] = words[i].lower()

i = 0

while i < len(words):

if words[i].isdigit():

words.remove(words[i])

i -= 1

i += 1

i = 0

while i < len(words):

if words[i] in string.punctuation:

words.remove(words[i])

i -= 1

i += 1

for i in range(len(words)):

words[i] = re.sub(r'\d+|\W+|\_', '', words[i])

stop\_words = set(stopwords.words('english'))

i = 0

while i < len(words):

if words[i] in stop\_words:

words.remove(words[i])

i -= 1

i += 1

lemmatizer = WordNetLemmatizer()

lemma = []

for i in words:

lemma.append(lemmatizer.lemmatize(i))

processed\_file = ""

for i in lemma:

processed\_file += i

processed\_file += " "

f = open(writefile, "w+")

f.write(processed\_file)

f.close()

return 1

def Create\_Vocabulary():

"""Creates Vocabulary"""

Vocab = []

Docs = []

for a in new\_filenames:

f = open(a, "r")

line = f.read()

words = nltk.tokenize.word\_tokenize(line)

for i in range(len(words)):

if words[i] not in Vocab:

Vocab.append(words[i])

Docs.append([[a,1]])

elif words[i] in Vocab:

index = Vocab.index(words[i])

for j in range(len(Docs[index])):

if Docs[index][j][0] == a:

Docs[index][j][1] += 1

break

else:

Docs[index].append([a,1])

f = open("vocab.txt", "w")

for i in range(len(Vocab)):

temp = ""

temp += Vocab[i]

temp += " -> "

for j in range(len(Docs[i])):

temp += Docs[i][j][0]

temp += ":"

temp += str(Docs[i][j][1])

temp += " , "

temp += "\n"

print(temp)

f.write(temp)

#Global

old\_filenames = [

"T1.txt",

"T2.txt",

"T3.txt",

"T4.txt",

"T5.txt",

"T6.txt",

"T7.txt",

"T8.txt",

"T9.txt",

"T10.txt"

]

new\_filenames = [

"T1\_new.txt",

"T2\_new.txt",

"T3\_new.txt",

"T4\_new.txt",

"T5\_new.txt",

"T6\_new.txt",

"T7\_new.txt",

"T8\_new.txt",

"T9\_new.txt",

"T10\_new.txt"

]

#Main

for i in range(len(old\_filenames)):

Document\_Preprocessing(old\_filenames[i],new\_filenames[i])

Create\_Vocabulary()

1. **Implementation of Vector Space Model.**
2. **Rank 10 documents for a given query.**
3. **Computing Similarity between any two documents.**

import math

docs = ['new york times','new york post','los angeles times']

unique\_word=[]

for i in docs:

for j in i.split(" "):

if j not in unique\_word:

unique\_word.append(j)

print(unique\_word)

table=[]

####separating words and putting in a list

for i in range(len(docs)):

list1=[0]\*len(unique\_word)

for j in range(len(list1)):

if unique\_word[j] in docs[i].split(" "):

list1[j]+=1

print(list1)

table.append(list1)

print(table)

idf=[]

count={}

for i in unique\_word:

count[i]=0

for i in unique\_word:

for j in docs:

if i in j.split(" "):

count[i]+=1

print(count)

idf=[]

for i in range(len(unique\_word)):

temp=math.log(len(docs)/count[unique\_word[i]])/math.log(2)

idf.append(temp)

print(idf)

print(math.log(2))

temp\_list=[0]\*len(unique\_word)

for i in table:

for j in range(len(i)):

temp\_var=i[j]\*idf[j]

i[j]=temp\_var

print(table)

## doc doc similarity

compare=[]

for i in range(len(docs)):

similar=[0]\*len(docs)

for j in range(len(docs)):

temp1=table[i]

temp2=table[j]

sums=0.0

sum1=0.0

sum2=0.0

for j in range(len(temp1)):

prod=temp1[j]\*temp2[j]

sums+=prod

sum1+=temp1[j]

sum2+=temp2[j]

term=sums/(sum1\*sum2)

similar.append(term)

compare.append(similar)

print(compare)

1. **Implementation of probabilistic Model. Rank 10 documents for a given query.**

import nltk as n

new\_filenames = [

"T1\_processed.txt",

"T2\_processed.txt",

"T3\_processed.txt",

"T4\_processed.txt",

"T5\_processed.txt",

"T6\_processed.txt",

"T7\_processed.txt",

"T8\_processed.txt",

"T9\_processed.txt",

"T10\_processed.txt"

]

######getting the document######

D\_length=len(new\_filenames)

D=[]

for i in new\_filenames:

absolute\_path="D:\\NLP\\IR\\Evaluation 1\\processed\_docs\\"

path\_new=absolute\_path+i

f = open(path\_new,"r+");

data=f.read()

D.append(data)

#######finding unique words######

let=[]

for i in D:

tokens=n.tokenize.word\_tokenize(i)

for j in tokens:

if j not in let:

let.append(j)

len\_unique=len(let)

count=[0]\*len\_unique

for i in D:

consider=[]

i=n.tokenize.word\_tokenize(i)

for j in range(len(i)):

if i[j] in let and i[j] not in consider:

k=let.index(i[j])

count[k]=count[k]+1

consider.append(i[j])

print(count)

P=[.0]\*len\_unique

N=len(D)

for i in range(len(count)):

P[i]=(N-count[i]+0.5)/(count[i]+0.5)

print(P)

q="hello world and report guttenberg a new world in there"

q\_temp=n.tokenize.word\_tokenize(q)

rank=[.0]\*N

for i in range(len(D)):

temp=1

for j in n.tokenize.word\_tokenize(D[i]):

if j in q\_temp:

temp=temp\*P[let.index(j)]

rank[i]=temp

print(rank)

count\_ln=0

for i in range(len(rank)):

if rank[i]==1:

count\_ln+=1

if count\_ln!=len(rank):

temp\_rank=rank

sorted=[]

high=0

for j in range(N):

for i in range(len(temp\_rank)):

if(temp\_rank[i]>temp\_rank[high]):

high=i

sorted.append(new\_filenames[high])

temp\_rank[high]=0

print(sorted)

else:

print("no docs found")

1. **Implementation of various evaluation measures.**
2. **Calculate recall and precision values for all relevant documents and draw precision VS Recall Curve. Also calculate R-precision.**
3. **Compare performance of two IR algorithms for the same query q.**
4. **Calculate harmonic mean and E- measure (All three cases b=1, b>1, b<1).**

rQ = [3, 5, 9, 25, 39, 44, 56, 71, 89, 94, 105, 119, 124, 136, 144]

aQ = [123, 84, 56, 6, 8, 777, 511, 129, 187, 25, 38, 48, 250, 113, 44, 99, 95, 214, 136, 39, 128, 71, 14, 5]

p = {}

r = {}

relevant = 0

for i in range(len(aQ)):

doc = aQ[i]

if (doc in rQ):

relevant += 1;

p[doc] = relevant / (i + 1) \* 100

r[doc] = relevant / len(rQ) \* 100

print("Precision of relevant docs : \n", p)

print("Recall of relevant docs \n: ", r)

keysR = r.keys()

keys30 = []

keys60 = []

keys90 = []

values = []

for i in keysR:

if (r[i] <= 30.0):

keys30.append(i)

if (r[i] > 30.0 and r[i] <= 60.0):

keys60.append(i)

if (r[i] > 60.0 and r[i] <= 90.0):

keys90.append(i)

for i in keys30:

values.append(p[i])

if not values:

print("Interpolated precision at 30% is : 0")

else:

print("Interpolated precision at 30% is : ", max(values))

values.clear()

for i in keys60:

values.append(p[i])

if not values:

print("Interpolated precision at 60% is : 0")

else:

print("Interpolated precision at 60% is : ", max(values))

values.clear()

for i in keys90:

values.append(p[i])

if not values:

print("Interpolated precision at 90% is : 0")

else:

print("Interpolated precision at 90% is : ", max(values))

count=0

for i in range(len(rQ)):

if aQ[i] in rQ:

count+=1

print("R Precision : " + str(count\*100/(len(rQ))))

def harmonic\_mean(a,b):

return 2/((100/a)+(100/b))

hm={}

for doc in p:

hm[doc]=harmonic\_mean(p[doc],r[doc])

print("Harmonic Mean :")

print(hm)

b=[0.5,1,2]

def Emeasure(r,p,b):

Em=[]

for i in b:

if i<1:

Em.append(1-((1+i\*i)/((i\*i\*100)/r)+(100/p)))

if i==1:

Em.append(1-harmonic\_mean(r,p))

if i>1:

Em.append(1 - ((1 + i \* i) / ((i \* i) / r) + (1 / p)))

return Em

Em={}

for doc in p:

Em[doc]=Emeasure(r[doc],p[doc],b)

print("E measure : ")

print(Em)

1. **Implementation of a Web Crawler.**

import scrapy

class BrickSetSpider(scrapy.Spider):

name = "brickset\_spider"

start\_urls = ['http://brickset.com/sets/year-2016']

def parse(self, response):

SET\_SELECTOR = '.set'

PIECES\_SELECTOR = './/dl[dt/text() = "Pieces"]/dd/a/text()'

MINIFIGS\_SELECTOR = './/dl[dt/text() = "Minifigs"]/dd[2]/a/text()'

IMAGE\_SELECTOR = 'img ::attr(src)'

for brickset in response.css(SET\_SELECTOR):

print("exists")

NAME\_SELECTOR = 'h1 ::text'

yield {

'name': brickset.css(NAME\_SELECTOR).extract\_first(),

'pieces': brickset.xpath(PIECES\_SELECTOR).extract\_first(),

'minifigs': brickset.xpath(MINIFIGS\_SELECTOR).extract\_first(),

'image': brickset.css(IMAGE\_SELECTOR).extract\_first(),

}

NEXT\_PAGE\_SELECTOR='.next a ::attr(href)'

next\_page = response.css(NEXT\_PAGE\_SELECTOR).extract\_first()

if next\_page:

yield scrapy.Request(

response.urljoin(next\_page),

callback=self.parse

)

1. **Lab Project: Develop a federated search model which uses distributive information retrieval system.**

# -\*- coding: utf-8 -\*-

"""

Created on Tue Apr 23 09:23:37 2019

@author: Sidharth

"""

from flask import Flask

from flask import request

from flask\_cors import CORS

from flask import jsonify

import handling\_functions as hf

import time

app = Flask(\_\_name\_\_)

CORS(app)

@app.route("/")

def main():

return "Welcome!"

@app.route("/search",methods=["POST"])

def query():

print (request.is\_json)

content = request.get\_json()

print (content)

start=time.time()

res=hf.searching\_index(content['query'])

end=time.time()

data1={}

data1['time']=end-start

data1['result']=res

return jsonify(data1)

@app.route("/display/<id1>/<doc>",methods=["GET"])

def doc\_display(id1,doc):

res=hf.get\_doc\_accord\_query(id1,doc)

result={}

result["text"]=res

result["name"]=doc

result["db"]=id1

return jsonify(result)

@app.route("/suggest/<word>",methods=["GET"])

def suggest\_words(word):

res=hf.inverted\_index\_words\_to\_list(word)

if len(res)==0:

return jsonify(["none found"])

return jsonify(res)

if \_\_name\_\_ == "\_\_main\_\_":

app.run()

# Made by Lakshya

import time

import nltk

import string

from nltk.corpus import stopwords

from nltk.stem import WordNetLemmatizer

import re

import os

from multiprocessing import Process

# Global

databases = []

Vocab = []

Vocab\_Docs = []

Dict = []

Dict\_Docs = []

repository\_path = "Repository"

folders = ["f1", "f2", "f3"]

access = [0, 0, 0, 0, 0, 0]

filenames = []

# UDF

def Document\_Preprocessing(path,readfile):

"""Reads a document, processes it, and stores it as a new file"""

f = open(path+"\\"+readfile, "r")

writefile = path+"^processed^"+readfile

text = f.read()

f.close()

words = nltk.tokenize.word\_tokenize(text)

for i in range(0, len(words)):

words[i] = words[i].lower()

i = 0

while i < len(words):

if words[i].isdigit():

words.remove(words[i])

i -= 1

i += 1

i = 0

while i < len(words):

if words[i] in string.punctuation:

words.remove(words[i])

i -= 1

i += 1

for i in range(len(words)):

words[i] = re.sub(r'\d+|\W+|\_', '', words[i])

stop\_words = set(stopwords.words('english'))

i = 0

while i < len(words):

if words[i] in stop\_words:

words.remove(words[i])

i -= 1

i += 1

lemmatizer = WordNetLemmatizer()

lemma = []

for i in words:

lemma.append(lemmatizer.lemmatize(i))

processed\_file = ""

for i in lemma:

processed\_file += i

processed\_file += " "

f = open(repository\_path+"\\"+path+"\\"+writefile, "w+")

f.write(processed\_file)

f.close()

def Create\_Vocabulary(path):

"""Creates Vocabulary"""

print("Creating Vocabulary for",path)

for a in filenames:

f = open(repository\_path+"\\"+path+"\\"+path+"^processed^"+a, "r")

line = f.read()

words = nltk.tokenize.word\_tokenize(line)

for i in range(len(words)):

if words[i] not in Vocab:

Vocab.append(words[i])

Vocab\_Docs.append([[path+"\_"+a, 1]])

elif words[i] in Vocab:

index = Vocab.index(words[i])

for j in range(len(Vocab\_Docs[index])):

if Vocab\_Docs[index][j][0] == path+"^"+a:

Vocab\_Docs[index][j][1] += 1

break

else:

Vocab\_Docs[index].append([path+"\_"+a, 1])

f = open(repository\_path + "\\" + path + "\\" + "inverted\_index.txt", "w")

temp = []

for i in range(len(Vocab)):

temp.append([Vocab[i], Vocab\_Docs[i]])

f.write(str(temp))

print("Vocabulary Created for",path)

def Remove\_Redundancy(path):

"""Removes the preprocessed documents"""

print("Removing Redundant data for",path)

real\_path = repository\_path+"\\"+path

files = os.listdir(real\_path)

for i in files:

if i != "inverted\_index.txt":

os.remove(real\_path+"\\"+i)

print("Redundant data removed for",path)

def Indexing(path):

"""This function is responsible for indexing a memory location given by path"""

global filenames

filenames = os.listdir(path)

"""

print("Preprocessing Docs in",path)

for ind in range(len(filenames)):

Document\_Preprocessing(path, filenames[ind])

print("Preprocessed Docs in", path)

"""

Create\_Vocabulary(path)

#Create\_Dictionary(path)

Remove\_Redundancy(path)

def Combine\_Indexes ():

"""Combines Multiple indexes into one"""

print("Combining Indexes")

All\_Dicts = []

Dictionary = []

Index = []

f = [None]\*len(folders)

for i in range(len(folders)):

f[i] = open(repository\_path + "\\" + folders[i] + "\\" + "inverted\_index.txt", "r")

All\_Dicts.append(eval(f[i].read()))

f[i].close()

for i in range(len(folders)):

for j in range(len(All\_Dicts[i])):

if All\_Dicts[i][j][0] not in Dictionary:

Dictionary.append(All\_Dicts[i][j][0])

Index.append(All\_Dicts[i][j][1])

else:

for k in range(len(All\_Dicts[i][j][1])):

Index[Dictionary.index(All\_Dicts[i][j][0])].append(All\_Dicts[i][j][1][k])

file = open(repository\_path + "\\" + "inverted\_index.txt", "w")

temp\_dict = []

for i in range(len(Dictionary)):

temp\_dict.append([Dictionary[i], Index[i]])

file.write(str(temp\_dict))

# Main

if \_\_name\_\_ == "\_\_main\_\_":

start = time.time()

processes = [None]\*len(folders)

for i in range(len(folders)):

processes[i] = Process(target=Indexing, args=(folders[i],))

for i in range(len(processes)):

processes[i].start()

for i in range(len(processes)):

processes[i].join()

Combine\_Indexes\_New()

end = time.time()

print("\nTotal time Taken =", (end-start)-((end-start)%0.01), "seconds")

# -\*- coding: utf-8 -\*-

"""

Created on Thu May 2 22:20:09 2019

@author: Sidharth

"""

import nltk

import string

from nltk.corpus import stopwords

from nltk.stem import WordNetLemmatizer

from difflib import SequenceMatcher

import re

def sort\_results(result):

count=[0]\*len(result)

new\_result=[]

for i in range(len(result)):

\_temp\_keys=result[i][1].keys()

count[i]=len(\_temp\_keys)

max\_occurence = max(count)

for i in range(max\_occurence,0,-1):

for j in range(len(result)):

if count[j]==i:

new\_result.append(result[j])

return new\_result

def inverted\_index\_words\_to\_list(query):

f=open(".\\Repository\\inverted\_index.txt","r")

d = f.read()

data=eval(d)

result=[]

f.close()

for i in data:

if query in i[0]:

result.append(i[0])

return result

def get\_query\_presence(query):

f=open(".\\Repository\\inverted\_index.txt","r")

d = f.read()

data=eval(d)

result=[]

f.close()

if query in data.keys():

return data[query]

else:

return []

def preprocess1(text):

words = nltk.tokenize.word\_tokenize(text)

for i in range(0, len(words)):

words[i] = words[i].lower()

i = 0

while i < len(words):

if words[i].isdigit():

words.remove(words[i])

i -= 1

i += 1

i = 0

while i < len(words):

if words[i] in string.punctuation:

words.remove(words[i])

i -= 1

i += 1

for i in range(len(words)):

words[i] = re.sub(r'\d+|\W+|\_', '', words[i])

stop\_words = set(stopwords.words('english'))

i = 0

while i < len(words):

if words[i] in stop\_words:

words.remove(words[i])

i -= 1

i += 1

lemmatizer = WordNetLemmatizer()

lemma = []

for i in words:

lemma.append(lemmatizer.lemmatize(i))

return lemma

def line\_occurence(file,query\_list):

path\_list=file.split("\_")

db=path\_list[0]

path\_list.remove(path\_list[0])

file=""

if(len(path\_list)>1):

file="\_".join(path\_list)

else:

file=path\_list[0]

r = open(".\\"+db+"\\"+file,"r")

content\_list=r.read().split("\n")

result1=re.sub(r'\W+|\_',' ',content\_list[0])

result1=result1.strip()

if result1=='':

result1=file

r.close()

return result1

def searching\_index(query):

f = open(".\\Repository\\inverted\_index.txt","r")

data=eval(f.read())

res1=preprocess1(query)

result=[]

occurence={}

for j in res1:

for i in range(len(data)):

if j==data[i][0]:

for k in data[i][1]:

if k[0] in occurence.keys():

occurence[k[0]][j]=k[1]

else:

occurence[k[0]]={}

occurence[k[0]][j]=k[1]

else:

continue

match={}

for j in occurence.keys():

match[j]=line\_occurence(j,res1)

for i in occurence.keys():

result.append([i,occurence[i],match[i]])

if len(result)!=0:

result=sort\_results(result)

f.close()

return result

def get\_doc\_accord\_query(folder, name):

f = open(".\\"+folder+"\\"+name, "r")

data = f.read()

return data

**Explanation:**

Federated search is an information retrieval technology that allows the simultaneous search of multiple searchable resources. A user makes a single query request which is distributed to the search engines, databases or other query engines participating in the federation. The federated search then aggregates the results that are received from the search engines for presentation to the user. Federated search is an information retrieval technology that allows the simultaneous search of multiple searchable resources. A user makes a single query request which is distributed to the search engines, databases or other query engines participating in the federation. In this case federation may consist of various water communities like India Water Portal, Wateraware and some government projects like Water Resource Information System: India-WRIS , Central Ground Water Board (CGWB), India Metrological department and Bhuvan. Also other datasets are available from Open Government Data (OGD) Platform India, knoema, World Resources Institute.

1. The federated search then aggregates the results that are received from the search engines for presentation to the user. Deep learning approach will be used to combine the results obtained from various sources and to remove the duplicates.

2. Results can be shown to the user as ranked documents as per the relevance of the query. Deep learning approach will be used to rank the results obtained from various sources.