

**WEB MINING**

**CSE - 3024**

**SLOT: L5+L6**

**LAB CAT**

***16BCE0240***

***ISHAN JAIN***

1. Write a program to extract the textual content (not the html tags) from a website which should be getting from command line and display the number of words and lines present in them without applying stop word removal. Also, find the term frequency of each word present in that document.

**CODE:**

import urllib.request

from bs4 import BeautifulSoup

import nltk

urlc = urllib.request.urlopen("http://vit.ac.in")

html = urlc.read()

soup = BeautifulSoup(html,"html5lib")

for script in soup(["script", "style"]):

script.extract()

text = soup.get\_text()

print(text)

num\_lines = 0

for line in text:

num\_lines += 1

print("Number of lines:")

print(num\_lines)

num\_words=0

for line in text:

words = line.split()

num\_words += len(words)

print("Number of words:")

print(num\_words)

tokens=[t for t in text.split()]

freq = nltk.FreqDist(tokens)

for key,val in freq.items():

print (str(key) + ':' + str(val))

1. Write a program to include a collection of new words into stop\_word list and apply the stop\_word removal procedure. Besides, also apply the stemming over the resultant terms.

CODE:

from bs4 import BeautifulSoup

import urllib.request

import nltk

from nltk.corpus import stopwords

from nltk.stem import PorterStemmer

from nltk.tokenize import sent\_tokenize,word\_tokenize

stop\_words = nltk.corpus.stopwords.words('english')

newStopWords = ['stopWord1','stopWord2']

stop\_words.extend(newStopWords)

response = urllib.request.urlopen('http://vit.ac.in/')

html = response.read()

soup = BeautifulSoup(html,"html5lib")

for script in soup(["script", "style"]):

script.extract()

text = soup.get\_text(strip=True)

tokens = [t for t in text.split()]

clean\_tokens = tokens[:]

sr = stopwords.words('english')

for token in tokens:

if token in stopwords.words('english'):

clean\_tokens.remove(token)

print(clean\_tokens)

print('\n\n\n\n\n')

ps = PorterStemmer()

#words = word\_tokenize(clean\_tokens)

for w in clean\_tokens:

print(ps.stem(w))

1. Write a crawler to extract the textual content (not the html tags) from a website, apply the tokenization, stop word removal to create a text corpus. Finally search for a word inside the text corpus.

CODE:

import urllib.request

from bs4 import BeautifulSoup

import nltk

from nltk.corpus import stopwords

from nltk.stem import PorterStemmer

from nltk.tokenize import word\_tokenize

urlc = urllib.request.urlopen("http://vit.ac.in")

html = urlc.read()

soup = BeautifulSoup(html,"html5lib")

for script in soup(["script", "style"]):

script.extract()

text = soup.get\_text()

print(text)

##### Till scraping the html tags

tokens = [t for t in text.split()]

clean\_tokens = tokens[:]

sr = stopwords.words('english')

for token in tokens:

if token in stopwords.words('english'):

clean\_tokens.remove(token)

print(clean\_tokens)

ps = PorterStemmer()

x1=[]

for w in clean\_tokens:

x1.append(ps.stem(w))

for i in x1:

print(i)

x=input("Enter the word: ")

if x in x1:

print("\n\n\nFound WORD\n\n\n")

else:

print('\n\n\nWord NOT FOUND\n\n')

LAB 2

1. Write a program to implement a concurrent crawler.

CODE:

import requests

import multiprocessing

from queue import Queue

import urllib.request

from bs4 import BeautifulSoup

from urllib.parse import urljoin,urlparse

from concurrent.futures import ThreadPoolExecutor

class MultiThreadScraper:

def \_\_init\_\_(self,base\_url):

self.base\_url = base\_url

self.root\_url = '{}://{}'.format(urlparse(self.base\_url).scheme,urlparse(self.base\_url).netloc)

self.pool = ThreadPoolExecutor(max\_workers=5)

self.scraped\_pages = set([])

self.to\_crawl = Queue()

self.to\_crawl.put(self.base\_url)

def parse\_links(self,html):

soup = BeautifulSoup(html,'html.parser')

links = soup.find\_all('a',href=True)

for link in links:

url = link['href']

if url.startswith('/') or url.startswith(self.root\_url):

url = urljoin(self.base\_url,url)

if url not in self.scraped\_pages:

self.to\_crawl.put(url)

def scrape\_page(self,url):

res = requests.get(url,timeout = (3,30))

return res

def post\_scrap\_callback(self,res):

result = res.result()

self.parse\_links(result.text)

def run\_scraper(self):

while True:

print(multiprocessing.current\_process().name)

target\_url = self.to\_crawl.get(timeout = 60)

if target\_url not in self.scraped\_pages:

print("Scraping URL: {}".format(target\_url))

response = urllib.request.urlopen(target\_url)

html = response.read()

print(html)

self.scraped\_pages.add(target\_url)

job = self.pool.submit(self.scrape\_page,target\_url)

job.add\_done\_callback(self.post\_scrap\_callback)

if \_\_name\_\_ == '\_\_main\_\_':

s1 = MultiThreadScraper("http://www.vit.ac.in")

s1.run\_scraper()

1. Write a program to calculate TF-IDF of the text corpus and return a set of URLs that matches the query string of your choice.

CODE:

import math

from textblob import TextBlob as tb

def tf(word, blob):

return blob.words.count(word) / len(blob.words)

def n\_containing(word, bloblist):

return sum(1 for blob in bloblist if word in blob.words)

def idf(word, bloblist):

return math.log(len(bloblist) / (1 + n\_containing(word, bloblist)))

def tfidf(word, blob, bloblist):

return tf(word, blob) \* idf(word, bloblist)

document1 = tb(open("/Users/garvitkhurana/Desktop/doc\_1.txt","r").read())

document2 = tb(open("/Users/garvitkhurana/Desktop/doc\_2.txt","r").read())

document3 = tb(open("/Users/garvitkhurana/Desktop/doc\_3.txt","r").read())

bloblist = [document1, document2, document3]

for i, blob in enumerate(bloblist):

print("Top words in document {}".format(i + 1))

scores = {word: tfidf(word, blob, bloblist) for word in blob.words}

sorted\_words = sorted(scores.items(), key=lambda x: x[1], reverse=True)

for word, score in sorted\_words:

print("\tWord: {}, TF-IDF: {}".format(word, round(score, 5)))

1. Write a program that returns the unary, elias delta, elias gamma and Golomb coding of a given decimal number.

CODE:

from math import log,floor

log2 = lambda x: log(x,2)

def binary(x,l=1):

fmt = '{0:0%db}' % l

x1=fmt.format(x)

return x1

def unary(x):

return ((x-1)\*'0')+'1'

def elias\_generic(lencoding, x):

if x == 0:

return '0'

l = 1+int(log2(x))

a = x - 2\*\*(int(log2(x)))

k = int(log2(x))

return lencoding(l) + binary(a,k)

def golomb(b, x):

q = int((x) / b)

r = int((x) % b)

l = int(floor(log2(b)))

d=((2\*\*(l+1))-b)

if(r<d):

return (unary(q+1)+binary(r, l))

if(r>=d):

print(q,r,l,d)

return unary(q+1) + binary(d+r,l)

def elias\_gamma(x):

return elias\_generic(unary, x)

def elias\_delta(x):

return elias\_generic(elias\_gamma,x)

i=7

print(i,unary(i), elias\_gamma(i),elias\_delta(i), golomb(10,i))

LAB 3

1. Write a program to implement document indexing which can return a set of resultant documents for any arbitrary search word.

import re

def process\_files(filenames):

file\_to\_terms = {}

for file in filenames:

pattern = re.compile('[\W\_]+')

file\_to\_terms[file] = open(file, 'r').read().lower();

file\_to\_terms[file] = pattern.sub(' ',file\_to\_terms[file])

re.sub(r'[\W\_]+','', file\_to\_terms[file])

file\_to\_terms[file] = file\_to\_terms[file].split()

return file\_to\_terms

def index\_one\_file(termlist):

fileIndex = {}

for index, word in enumerate(termlist):

if word in fileIndex.keys():

fileIndex[word].append(index)

else:

fileIndex[word] = [index]

return fileIndex

def make\_indices(termlists):

total = {}

for filename in termlists.keys():

total[filename] = index\_one\_file(termlists[filename])

return total

def fullIndex(regdex):

total\_index = {}

for filename in regdex.keys():

for word in regdex[filename].keys():

if word in total\_index.keys():

if filename in total\_index[word].keys():

total\_index[word][filename].extend(regdex[filename][word][:])

else:

total\_index[word][filename] = regdex[filename][word]

else:

total\_index[word] = {filename: regdex[filename][word]}

return total\_index

filenames=['C:\Python34\input\doc2.txt','C:\Python34\input\doc3.txt']

termslist=process\_files(filenames)

print('\nTerm List \n')

for i in termslist:

print(i,termslist[i],'\n\n')

print('\n')

totaldict=make\_indices(termslist)

print('TERM FREQUENCY DOCUMENT INDEXING\n')

for i in totaldict:

print(i,totaldict[i],'\n\n')

print('\n')

index=fullIndex(totaldict)

print('TOTAL INDEXING\n')

for j in index:

print (j,end=": ")

for k in index[j]:

print (k,index[j][k],end=" ")

qu='mining'

if qu in index:

print(qu,"present in ",index[qu])

1. Write a program to construct AT matrix by randomly surfing the first 10 URLs of the seed page. From that, calculate the final steady state page rank of those 10 pages where the damping factor d=0.85.

import numpy as np

def pagerank(M, eps=1.0e-8, d=0.85):

N = M.shape[1]

v = np.random.rand(N, 1)

v = v / np.linalg.norm(v, 1)

last\_v = np.ones((N, 1), dtype=np.float32) \* np.inf

M\_hat = (d \* M) + (((1 - d) / N) \* np.ones((N, N), dtype=np.float32))

while np.linalg.norm(v - last\_v, 2) > eps:

last\_v = v

v = np.matmul(M\_hat, v)

return v

M = np.array([[0, 0, 0, 0, 1],

[0.5, 0, 0, 0, 0],

[0.5, 0, 0, 0, 0],

[0, 1, 0.5, 0, 0],

[0, 0, 0.5, 1, 0]])

v = pagerank(M, 0.001, 0.85)

print (v)

1. Write a program to train the classifier using five given documents. Further, classify the target pages using the trained model.

LAB 4

1. Write a program to implement apriori algorithm from web log data.

**CODE:**

import pandas as pd

from itertools import combinations

data = [

{'1': 'a' , '2':'b' , '3':'c'},

{'1': 'b' , '2':'c' , '3':'d'},

{'1': 'a' , '2':'b' , '3':'c' , '4':'d'},

{'1': 'b' , '2':'c'},

{'1': 'a' , '2':'b' , '3':'d'},

{'1': 'd' , '2':'e'},

{'1': 'a' , '2':'b' , '3':'c'},

{'1': 'c' , '2':'d' , '3':'e'},

{'1': 'a' , '2':'b' , '3':'c'}]

df = pd.DataFrame(data)

def apriori(trans, support=0.2, minlen=1):

data=pd.get\_dummies(trans.unstack().dropna()).groupby(level=1).sum()

collen, rowlen =data.shape

pattern = []

for cnum in range(minlen, rowlen+1):

for cols in combinations(data, cnum):

patsup = data[list(cols)].all(axis=1).sum()

patsup=float(patsup)/collen

pattern.append([",".join(cols), patsup])

sdf = pd.DataFrame(pattern, columns=["Pattern", "Support"])

results=sdf[sdf.Support >= support]

return results

Ap = apriori(df)

print(Ap)

1. Write a program to demonstrate hierarchical and K-means clustering.

**CODE FOR HIERARCHIAL CLUSTERING:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.cluster.hierarchy import dendrogram, linkage

#from matplotlib import pyplot as plt

s1 = 'name python computer'.split()

s2 = 'alpha python'.split()

s3 = 'alpha work computer'.split()

s4 = 'name office linkage'.split()

s5 = 'linkage study'.split()

terms = sorted(list(set(s1+s2+s3+s4+s5)))

term\_doc = [[0 for i in range(len(terms))] for j in range(len([s1,s2,s3,s4,s5]))]

c = 0

for i in [s1,s2,s3,s4,s5]:

term\_doc[c] = [i.count(terms[j]) for j in range(len(terms))]

c+=1

A = term\_doc

X = np.array(A)

linked = linkage(X, 'single')

lo = 1

hi = np.shape(X)[0]+1

labelList = range(lo, hi)

plt.figure(figsize = (hi-lo, 5))

dendrogram(linked,

orientation = 'top',

labels=labelList,

distance\_sort = 'descending',

show\_leaf\_counts=True)

plt.show()

**CODE FOR k–Means CLUSTERING:**

from copy import deepcopy

import numpy as np

import pandas as pd

from matplotlib import pyplot as plt

data = pd.read\_csv('xclara.csv')

f1 = data['V1'].values

f2 = data['V2'].values

X = np.array(list(zip(f1, f2)))

def dist(a, b, ax=1):

return np.linalg.norm(a - b, axis=ax)

k = 3

C\_x = np.random.randint(0, np.max(X)-20, size=k)

C\_y = np.random.randint(0, np.max(X)-20, size=k)

C = np.array(list(zip(C\_x, C\_y)), dtype=np.float32)

print("Initial Centroids")

print(C)

C\_old = np.zeros(C.shape)

clusters = np.zeros(len(X))

error = dist(C, C\_old, None)

while error!=0:

for i in range(len(X)):

distances = dist(X[i], C)

cluster = np.argmin(distances)

clusters[i] = cluster

C\_old = deepcopy(C)

for i in range(k):

points = [X[j] for j in range(len(X)) if clusters[j] == i]

C[i] = np.mean(points, axis=0)

error = dist(C, C\_old, None)

colors = ['r', 'g', 'b', 'y', 'c', 'm']

fig, ax = plt.subplots()

for i in range(k):

points = np.array([X[j] for j in range(len(X)) if clusters[j] == i])

ax.scatter(points[:, 0], points[:, 1], s=7, c=colors[i])

1. Write a program to reformulate the original query using Rocchio method.

**CODE:**

q0 = [float(i) for i in input('Enter query matrix: ').split()]

n = int(input('Enter number of relevant docs: '))

relevant = []

for i in range(n):

ai = [float(i) for i in input('Enter relevant matrix ' + str(i+1) + ' out of ' + str(n) + ': ').split()]

relevant.append(ai)

m = int(input('Enter number of irrelevant docs: '))

irrelevant = []

for i in range(n):

ai = [float(i) for i in input('Enter relevant matrix ' + str(i+1) + ' out of ' + str(n) + ': ').split()]

irrelevant.append(ai)

relevant\_centroid = [0 for i in range(len(q0))]

for i in range(len(q0)):

x = 0

for j in range(n):

x+=relevant[j][i]

relevant\_centroid[i] = x/n

irrelevant\_centroid = [0 for i in range(len(q0))]

for i in range(len(q0)):

x = 0

for j in range(n):

x+=irrelevant[j][i]

irrelevant\_centroid[i] = x/n

alpha = float(input('Enter alpha: '))

beta = float(input('Enter beta: '))

gamma = float(input('Enter gamma: '))

qf = [0 for i in range(len(q0))]

for i in range(len(q0)):

qf[i] = alpha\*q0[i] + beta\*relevant\_centroid[i] - gamma\*irrelevant\_centroid[i]

print('Reformulated query matrix: ', end='')

print(\*qf)