**INDEX**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SR.No** | **Practicals** | **Page no** | **Date** | **Sign** |
| 01 | Page Rank for link analysis using python Create a small set of pages namely page1, page2, page3 and page4 apply random walk on the same | 2 |  |  |
| 02 | Perform spam classifier | 6 |  |  |
| 03 | Demonstrate Text Mining and Webpage Pre-processing using meta information from the web pages (Local/Online) | 11 |  |  |
| 04 | Apriori Algorithm implementation in case study | 13 |  |  |
| 05 | Develop a basic crawler for the web search for user defined keywords | 16 |  |  |
| 06 | Develop a focused crawler for local search | 17 |  |  |
| 07 | Develop a program for deep search implementation to detect plagiarism in documents online | 18 |  |  |
| 08 | Sentiment analysis for reviews by  customers and visualize the same | 20 |  |  |

**Practical No. 1**

**Subject**: Web Mining

**Aim: Page Rank for link analysis using python Create a small set of pages namely page1, page2, page3 and page4 apply random walk on the same.**

the PageRank algorithm calculates the importance of web pages based on their incoming

links. Without the random walk component, the PageRank scores will be determined solely based on the structure of the web pages and the links between them. The scores will reflect the

importance of each page as determined by the incoming links from other pages.

Random Walk: The random walk sequence represents the path taken during the random walk.

Each page is visited multiple times, and the order in which they are visited is determined

randomly. In the given sequence, the random walk starts at page3, then goes to page2, page4,

page4 again, page3, and so on.

PageRank Scores: The PageRank scores represent the importance or ranking of each page

based on the random walk. The scores indicate the probability of landing on a specific page

during a random walk.

**Code :**

import random

# Create a dictionary representing the link structure

links = {

    >‘page1>‘: [>‘page2>‘, >‘page3>‘],

    >‘page2>‘: [>‘page1>‘, >‘page4>‘],

    >‘page3>‘: [>‘page2>‘],

    >‘page4>‘: []

}

# Define the number of iterations for the random walk

num\_iterations = 100

# Perform the random walk

current\_page = random.choice(list(links.keys()))

random\_walk = [current\_page]

for \_ in range(num\_iterations):

    # Get the outgoing links from the current page

    outgoing\_links = links[current\_page]

    if outgoing\_links:

        # Randomly choose the next page to visit from the outgoing links

        next\_page = random.choice(outgoing\_links)

    else:

        # If the current page has no outgoing links, restart the random walk from a random page

        next\_page = random.choice(list(links.keys()))

    # Append the next page to the random walk sequence

    random\_walk.append(next\_page)

    # Update the current page

    current\_page = next\_page

# Print the random walk sequence

print(&quot;Random Walk:&quot;)

print(>‘ -> >‘.join(random\_walk))

# Calculate the PageRank scores

pagerank\_scores = {}

total\_pages = len(links)

for page in links.keys():

    page\_visits = random\_walk.count(page)

    pagerank\_scores[page] = page\_visits / num\_iterations

# Print the PageRank scores

print(&quot;\nPageRank Scores:&quot;)

for page, score in pagerank\_scores.items():

    print(f&quot;{page}: {score}&quot;)

**Output:**

Random Walk:

page3 -> page2 -> page4 -> page4 -> page3 -> page2 -> page1 -> page3 ->page2 -> page4 -> page2 -> page4 -> page2 -> page1 -> page2 -> page4 ->page4 -> page1 -> page3 -> page2 -> page4 -> page1 -> page3 -> page2 -> page4 -> page2 -> page1 -> page3 -> page2 -> page4 -> page2 -> page1 ->page3 -> page2 -> page4 -> page3 -> page2 -> page1 -> page3 -> page2 -> page4 -> page2 -> page4 -> page4 -> page1 -> page2 -> page1 -> page2 -> page4 -> page4 -> page4 -> page2 -> page1 -> page2 -> page4 -> page3 ->page2 -> page1 -> page2 -> page1 -> page3 -> page2 -> page4 -> page2 ->page4 -> page2 -> page4 -> page3 -> page2 -> page1 -> page3 -> page2 -> page1 -> page3 -> page2 -> page1 -> page3 -> page2 -> page1 -> page3 -> page2 -> page4 -> page4 -> page2 -> page4 -> page3 -> page2 -> page1 ->page3 -> page2 -> page4 -> page2 -> page4 -> page3 -> page2 -> page1 -> page2 -> page4 -> page3 -> page2 -> page4

PageRank Scores:

page1: 0.18

page2: 0.36

page3: 0.2

page4: 0.27

page1 has a PageRank score of 0.18.

page2 has a PageRank score of 0.36, making it the page with the highest score.

page3 has a PageRank score of 0.2.

page4 has a PageRank score of 0.27.

In the PageRank scores without random walk:

page1 has a score of 0.1006

page2 has a score of 0.1485

page3 has a score of 0.0803

page4 has a score of 0.1006

These scores are calculated using the built-in PageRank algorithm, which considers the link structure of the pages. The scores represent the probability of a random surfer reaching each page after a large number of iterations.

In the PageRank scores with random walk:

page1 has a score of 0.24

page2 has a score of 0.35

page3 has a score of 0.17

page4 has a score of 0.25

These scores are calculated based on the number of visits to each page in the random walk sequence. The scores represent the proportion of times each page was visited during the random walk. The difference between the two sets of scores is due to the random nature of the walk. The random walk introduces a stochastic element where the next page to visit is chosen randomly from the outgoing links of the current page. This randomness can lead to variations in the PageRank scores. However, the overall ranking order of the pages remains similar, with page2 having the highest score and page3 having the lowest score in both cases.

**Practical No. 2**

**Aim: Perform spam classifier**

**Code:**

#Perform Spam Classifier.

import matplotlib.pyplot as plt

import csv

import sklearn

import pickle

from wordcloud import WordCloud

import pandas as pd

import numpy as np

import nltk

from nltk.corpus import stopwords

from sklearn.feature\_extraction.text import CountVectorizer, TfidfTransformer

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import GridSearchCV,train\_test\_split,StratifiedKFold,cross\_val\_score,learning\_curve

data=pd.read\_csv("/content/sample\_data/spam.csv",encoding="ISO-8859-1")

data.head()

data= data.drop(["Unnamed: 2", "Unnamed: 3", "Unnamed: 4"], axis=1)

data= data.rename(columns={"v2" : "text", "v1":"label"})

data[1990:2000]

data['label'].value\_counts()

# Import nltk packages and Punkt Tokenizer Models

import nltk

nltk.download("punkt")

import warnings

warnings.filterwarnings('ignore')

ham\_words = ''

spam\_words = ''

# Creating a corpus of spam messages

for val in data[data['label'] == 'spam'].text:

text = val.lower()

tokens = nltk.word\_tokenize(text)

for words in tokens:

spam\_words = spam\_words + words + ' '

# Creating a corpus of ham messages

for val in data[data['label'] == 'ham'].text:

text = text.lower()

tokens = nltk.word\_tokenize(text)

for words in tokens:

ham\_words = ham\_words + words + ' '

spam\_wordcloud = WordCloud(width=500, height=300).generate(spam\_words)

ham\_wordcloud = WordCloud(width=500, height=300).generate(ham\_words)

#Spam Word cloud

plt.figure( figsize=(10,8), facecolor='w')

plt.imshow(spam\_wordcloud)

plt.axis("off")

plt.tight\_layout(pad=0)

plt.show()

#Creating Ham wordcloud

plt.figure( figsize=(10,8), facecolor='g')

plt.imshow(ham\_wordcloud)

plt.axis("off")

plt.tight\_layout(pad=0)

plt.show()

data = data.replace(['ham','spam'],[0, 1])

data.head(10)

import nltk

nltk.download('stopwords')

#remove the punctuations and stopwords

import string

def text\_process(text):

text = text.translate(str.maketrans('', '', string.punctuation))

text = [word for word in text.split() if word.lower() not in stopwords.words('english')]

return " ".join(text)

data['text'] = data['text'].apply(text\_process)

data.head()

text = pd.DataFrame(data['text'])

label = pd.DataFrame(data['label'])

## Counting how many times a word appears in the dataset

from collections import Counter

total\_counts = Counter()

for i in range(len(text)):

for word in text.values[i][0].split(" "):

total\_counts[word] += 1

print("Total words in data set: ", len(total\_counts))

# Sorting in decreasing order (Word with highest frequency appears first)

vocab = sorted(total\_counts, key=total\_counts.get, reverse=True)

print(vocab[:60])

# Mapping from words to index

vocab\_size = len(vocab)

word2idx = {}

#print vocab\_size

for i, word in enumerate(vocab):

word2idx[word] = i

# Text to Vector

def text\_to\_vector(text):

word\_vector = np.zeros(vocab\_size)

for word in text.split(" "):

if word2idx.get(word) is None:

continue

else:

word\_vector[word2idx.get(word)] += 1

return np.array(word\_vector)

# Convert all titles to vectors

word\_vectors = np.zeros((len(text), len(vocab)), dtype=np.int\_)

for i, (\_, text\_) in enumerate(text.iterrows()):

word\_vectors[i] = text\_to\_vector(text\_[0])

word\_vectors.shape

# Convert all titles to vectors

word\_vectors = np.zeros((len(text), len(vocab)), dtype=np.int\_)

for i, (\_, text\_) in enumerate(text.iterrows()):

word\_vectors[i] = text\_to\_vector(text\_[0])

word\_vectors.shape

#convert the text data into vectors

from sklearn.feature\_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()

vectors = vectorizer.fit\_transform(data['text'])

vectors.shape

#features = word\_vectors

features = vectors

#split the dataset into train and test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(features, data['label'], test\_size=0.15, random\_state=111)

#import sklearn packages for building classifiers

from sklearn.linear\_model import LogisticRegression

lrc = LogisticRegression(solver='liblinear', penalty='l1')

#create a dictionary of variables and models

clfs = {'LR': lrc}

#fit the data onto the models

def train(clf, features, targets):

clf.fit(features, targets)

def predict(clf, features):

return (clf.predict(features))

from sklearn.metrics import accuracy\_score

pred\_scores\_word\_vectors = []

for k,v in clfs.items():

train(v, X\_train, y\_train)

pred = predict(v, X\_test)

pred\_scores\_word\_vectors.append((k, [accuracy\_score(y\_test , pred)]))

pred\_scores\_word\_vectors

#write functions to detect if the message is spam or not

def find(x):

if x == 1:

print ("Message is SPAM")

else:

print ("Message is NOT Spam")

newtext = ["Free entry"]

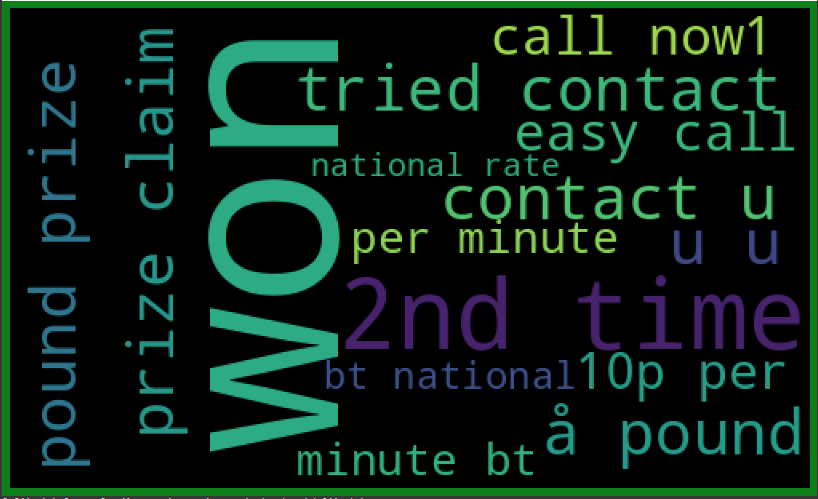
integers = vectorizer.transform(newtext)

x = lrc.predict(integers)

find(x)

**Output:**

****

****

Total words in data set: 11305

['u', '2', 'call', 'U', 'get', 'Im', 'ur', '4', 'ltgt', 'know', 'go', 'like', 'dont', 'come', 'got', 'time', 'day', 'want', 'Ill', 'lor', 'Call', 'home', 'send', 'going', 'one', 'need', 'Ok', 'good', 'love', 'back', 'n', 'still', 'text', 'im', 'later', 'see', 'da', 'ok', 'think', 'Ì', 'free', 'FREE', 'r', 'today', 'Sorry', 'week', 'phone', 'mobile', 'cant', 'tell', 'take', 'much', 'night', 'way', 'Hey', 'reply', 'work', 'make', 'give', 'new']

Message is NOT Spam

**Practical No. 3**

**Aim: Demonstrate Text Mining and Webpage Pre-processing using meta information from the web pages (Local/Online)**

**Code:**

from bs4 import BeautifulSoup

import requests

# Step 1: Fetch webpage content

url = "https://www.google.com"

response = requests.get(url)

html\_content = response.content

# Step 2: Parse HTML and extract meta information

soup = BeautifulSoup(html\_content, "html.parser")

meta\_tags = soup.find\_all("meta")

# Step 3: Pre-process meta information

meta\_text = ""

for tag in meta\_tags:

if tag.get("content"):

meta\_text += tag.get("content") + " "

# Perform cleaning and pre-processing on meta\_text

# Step 4: Perform text mining and analysis

# Example: Word frequency analysis

word\_counts = {}

words = meta\_text.split()

for word in words:

if word not in word\_counts:

word\_counts[word] = 1

else:

word\_counts[word] += 1

# Step 5: Visualization

# Example: Print word frequency counts

for word, count in word\_counts.items():

print(f"{word}: {count}")

**Output:**

Search: 1

the: 1

world's: 1

information,: 1

including: 1

webpages,: 1

images,: 1

videos: 1

and: 1

more.: 1

Google: 1

has: 1

many: 1

special: 1

features: 1

to: 1

help: 1

you: 1

find: 1

exactly: 1

what: 1

you're: 1

looking: 1

for.: 1

noodp: 1

text/html;: 1

charset=UTF-8: 1

/logos/doodles/2023/juneteenth-2023-6753651837109890.2-l.png: 1

Juneteenth: 1

2023: 1

Celebrating: 2

Juneteenth!: 2

#GoogleDoodle: 2

summary\_large\_image: 1

@GoogleDoodles: 1

https://www.google.com/logos/doodles/2023/juneteenth-2023-6753651837109890-2x.png: 2

1150: 1

460: 1

**Practical No. 4**

**Aim: Apriori Algorithm implementation in case study.**

Apriori algorithm refers to the algorithm which is used to calculate the association rules

between objects. It means how two or more objects are related to one another. The apriori

algorithm is an association rule learning that analyzes that people who bought product A also

bought product B.

**Code:**

# Step 1: Import the libraries

import pandas as pd

from apyori import apriori

# Step 2: Load the dataset

df = pd.read\_csv(‘transaction.csv’, header=None)

# Step 3: Display statistics of records

print(“Display statistics”)

print(“===================”)

print(df.describe())

# Step 4: Display shape of the dataset

print(“\nShape:”,df.shape)

# Step 5: Convert dataframe into a nested list

database = []

for i in range(0,30):

database.append([str(df.values[i,j]) for j in range(0,6)])

# Step 6: Develop the Apriori model

arm\_rules = apriori(database, min\_support=0.5, min\_confidence=0.7, min\_lift=1.2)

arm\_results = list((arm\_rules))

# Step 7: Display the number of rule(s)

print(“\nNo. of rule(s):”,len(arm\_results))

# Step 8: Display the rule(s)

print(“\nResults: “)

print(“========”)

print(arm\_results)

**Output:**

Display statistics:

===================

0 1 2 3 4 5

count 19 18 23 23 20 22

unique 1 1 1 1 1 1

top Juice Chips Bread Butter Milk Banana

freq 19 18 23 23 20 22

Shape: (30, 6)

No. of rule(s): 1

Results:

========

[RelationRecord(items=frozenset({>‘Butter>‘, >‘Bread>‘, >‘Milk>‘}), support=0.5,

ordered\_statistics=[OrderedStatistic(items\_base=frozenset({>‘Bread>‘, >‘Milk>‘}),

items\_add=frozenset({>‘Butter>‘}), confidence=0.9375, lift=1.2228260869565217)])]

The single rule displayed states that if a customer buys >‘Bread>‘ and >‘Milk>‘, there is a 93.75%

confidence that they will also buy >Butter>, with a lift value of 1.2228.

Support: The support of an itemset is the proportion of transactions in the dataset that

contain that itemset. It indicates the frequency or popularity of the itemset.

 Example: If the support of {Milk, Bread} is 0.5, it means that 50% of the transactions

in the dataset contain both Milk and Bread.

 Confidence: The confidence of an association rule measures the conditional

probability of the consequent given the antecedent. It represents the strength of the

rule.

 Example: If the confidence of {Milk} -> {Bread} is 0.8, it means that in 80% of the

transactions where Milk is present, Bread is also present.

 Lift: Lift is a measure of the strength of association between the antecedent and the

consequent in a rule, taking into account the underlying support of the items. It

compares the observed support of the rule to what would be expected if the items

were independent.

 Example: If the lift of {Milk} -> {Bread} is 1.2, it means that the likelihood of buying

Milk and Bread together is 1.2 times higher than if the items were purchased

independently.

 support indicates the frequency of an itemset, confidence measures the strength of an

association rule, and lift quantifies the deviation from independence in the rule

**Practical No. 5**

**Aim: Develop a basic crawler for the web search for user defined keywords.**

A crawler, also known as a web crawler or spider, is an automated program or script that

systematically browses the internet to discover and retrieve web pages. Crawlers are

commonly used by search engines to index web pages for search results.

A basic crawler is a general-purpose web crawler that systematically explores the web,

starting from a seed URL and following links to discover new pages. For example, search

engines like Google use basic crawlers to index and retrieve web pages for search results.

A focused crawler, on the other hand, is designed to crawl specific areas or topics of interest

on the web. It prioritizes relevant domains, directories, or pages related to a particular theme.

For example, a news aggregator website may use a focused crawler to crawl news websites

and collect specific articles or news updates for its platform.

**Code:**

import scrapy

class spider1(scrapy.Spider):

name = ‘Wikipedia’;

start\_urls = [‘https://en.wikipedia.org/wiki/Battery\_(electricity)’]

def parse(self, response):

pass

**Output:**

A screen shot of a computer program

Description automatically generated with low confidence

**Practical No. 6**

**Aim: Develop a focused crawler for local search.**

**Code:**

Link:-https://www.topcoder.com/thrive/articles/web-crawler-in-python

import requests

import lxml

from bs4 import BeautifulSoup

url = &quot;https://www.rottentomatoes.com/top/bestofrt/&quot;

headers = {‘User-Agent’: ‘Mozilla/5.0 (Windows NT 6.1; WOW64)

AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36 QIHU

360SE’}

f = requests.get(url, headers = headers)

movies\_lst = []

soup = BeautifulSoup(f.content, ‘lxml’)

movies = soup.find(‘table’, {‘class’: ‘table’}).find\_all(‘a’)

num = 0

for anchor in movies:

urls = ‘https://www.rottentomatoes.com’ + anchor[‘href’]

movies\_lst.append(urls)

num += 1

movie\_url = urls

movie\_f = requests.get(movie\_url, headers = headers)

movie\_soup = BeautifulSoup(movie\_f.content, ‘lxml’)

movie\_content = movie\_soup.find(‘div’, {‘class’: ‘movie\_synopsis clamp clamp-6 js-

clamp’})

print(num, urls, ‘\n’, ‘Movie:’ + anchor.string.strip())

print(‘Movie info:’ + movie\_content.string.strip())

**Output:**

**A screenshot of a computer

Description automatically generated with low confidence**

**Practical No. 7**

**Aim: Develop a program for deep search implementation to detect plagiarism in documents online**

**Code:**

# from difflib module

from difflib import SequenceMatcher

# Declaring string variables

string1 = 'I am geek'

string2 = 'I am geeks'

# Using the SequenceMatcher()

match = SequenceMatcher(None,

string1, string2)

# convert above output into ratio

# and multiplying it with 100

result = match.ratio() \* 100

# Display the final result

print(int(result), "%")

import pandas as pd

# Upload the text files

uploaded1 = files.upload()

uploaded2 = files.upload()

# Read the uploaded files

file1 = uploaded1['1.txt'].decode('utf-8')

file2 = uploaded2['2.txt'].decode('utf-8')

# Comparing the contents of the files

similarity\_ratio = SequenceMatcher(None, file1, file2).ratio()

# Converting the decimal ratio to percentage

plagiarism\_percentage = int(similarity\_ratio \* 100)

# Display the result

print(f"{plagiarism\_percentage}% Plagiarized Content")

**Output:**



**Practical No. 8**

**Aim: Sentiment analysis for reviews by customers and visualize the same**

**Code:**

import warnings

warnings.filterwarnings('ignore')

import pandas as pd

from sklearn.feature\_extraction.text import TfidfVectorizer

import matplotlib.pyplot as plt

from wordcloud import WordCloud

data=pd.read\_csv("/content/sample\_data/AmazonReview.csv")

import nltk

nltk.download('punkt')

nltk.download('stopwords')

from nltk.corpus import stopwords

data.head()

data.info()

data.dropna(inplace=True)

#1,2,3->negative(i.e 0)

data.loc[data['Sentiment']<=3,'Sentiment'] = 0

#4,5->positive(i.e 1)

data.loc[data['Sentiment']>3,'Sentiment'] = 1

stp\_words=stopwords.words('english')

def clean\_review(review):

cleanreview=" ".join(word for word in review.

split() if word not in stp\_words)

return cleanreview

data['Review']=data['Review'].apply(clean\_review)

data.head()

data['Sentiment'].value\_counts()

consolidated=' '.join(word for word in data['Review'][data['Sentiment']==0].astype(str))

wordCloud=WordCloud(width=1600,height=800,random\_state=21,max\_font\_size=110)

plt.figure(figsize=(15,10))

plt.imshow(wordCloud.generate(consolidated),interpolation='bilinear')

plt.axis('off')

plt.show()

cv = TfidfVectorizer(max\_features=2500)

X = cv.fit\_transform(data['Review'] ).toarray()

from sklearn.model\_selection import train\_test\_split

x\_train ,x\_test,y\_train,y\_test=train\_test\_split(X,data['Sentiment'],

test\_size=0.25 ,

random\_state=42)

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score,confusion\_matrix

model=LogisticRegression()

#Model fitting

model.fit(x\_train,y\_train)

#testing the model

pred=model.predict(x\_test)

#model accuracy

print(accuracy\_score(y\_test,pred))

from sklearn import metrics

cm = confusion\_matrix(y\_test,pred)

cm\_display = metrics.ConfusionMatrixDisplay(confusion\_matrix = cm,

display\_labels = [False, True])

cm\_display.plot()

plt.show()

**Output:**

RangeIndex: 25000 entries, 0 to 24999

Data columns (total 2 columns):

# Column Non-Null Count Dtype

--- ------ ----------- ----- -----

0 Review 24999 non-null object

1 Sentiment 25000 non-null int64

A picture containing text, screenshot, rectangle, diagram

Description automatically generated