**Practical No. 1**

**Aim :-** Write a program to demonstrate bitwise operation.

**Code :-**

1. **Python:**

import pandas as pd

from sklearn.feature\_extraction.text import CountVectorizer

corpus=['this is the first document','this is the Second document','And this is the third one','Fourth Document']

print('Dataset (corpus) is : \t',corpus)

vectorizer = CountVectorizer()

X = vectorizer.fit\_transform(corpus)

print("Fit Transform is : ",X.toarray())

df = pd.DataFrame(X.toarray(),columns=vectorizer.get\_feature\_names\_out())

print("The generated data frame is : \n",df.to\_string(index=False, justify='center'))

alldata = df[(df['this']==1)&(df['first']==1)]

print("Indices where \'this\' and \'first\' terms are present are : ",alldata.index.tolist())

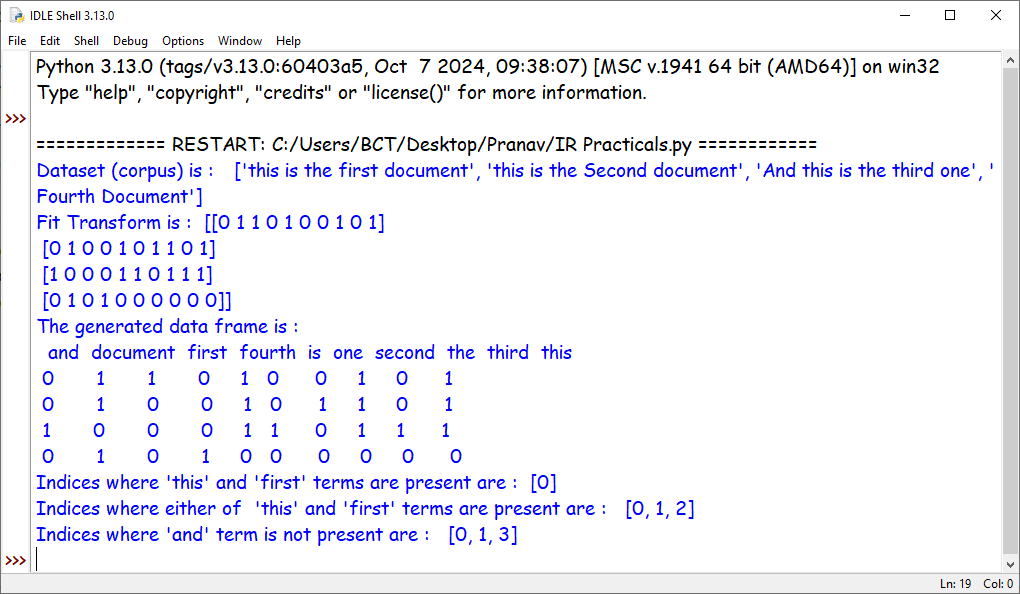
ordata = df[(df['this']==1)|(df['first']==1)]

print('Indices where either of \'this\' and \'first\' terms are present are : ',ordata.index.tolist())

notdata = df[(df['and']!=1)]

print('Indices where \'and\' term is not present are : ',notdata.index.tolist())

**Output :-**

****

**Practical No. 2**

**Aim :-** Write a python program to perform N-Gram analysis specifically based on unigram, bigram and trigram. Using NLTK.

**Code :-**

import nltk

from nltk import word\_tokenize

from nltk.util import ngrams

#Sample text

text = 'This is a sample text for unigram, bigram, and trigram extraction using NLTK.'

#Tokenize the text

tokens = word\_tokenize(text.lower()) #Converting to lowercase for consistency

#Unigrams

unigrams = list(ngrams(tokens,1))

#Bigrams

bigrams = list(ngrams(tokens,2))

#Trigrams

trigrams = list(ngrams(tokens,3))

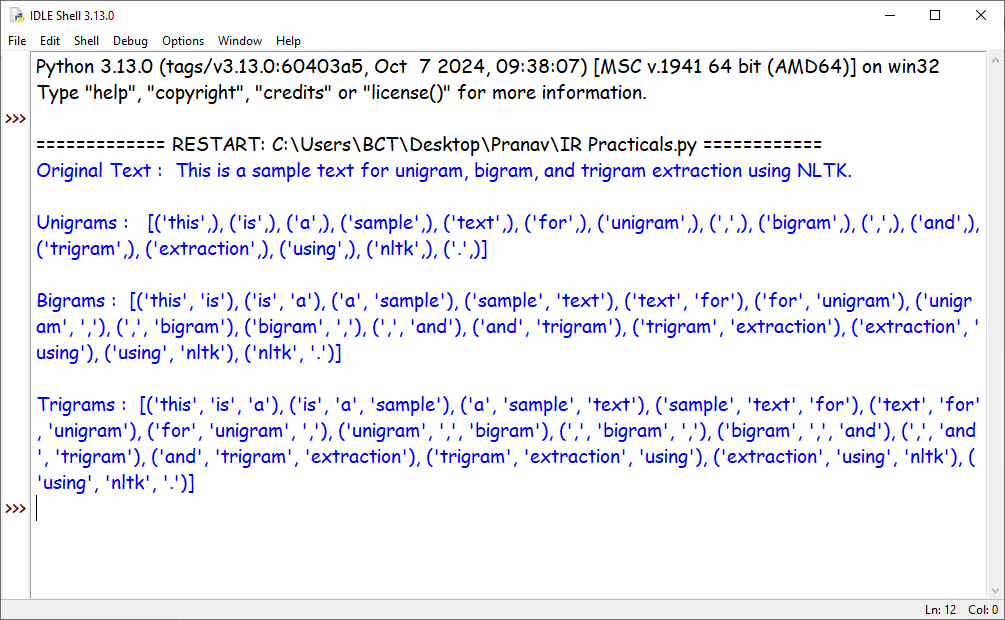
print('Original Text : ',text)

print('\nUnigrams : ',unigrams)

print('\nBigrams : ',bigrams)

print('\nTrigrams : ',trigrams)

**Output :-**



**Practical No. 3**

**Aim :-** Write a python program to evaluate the performance of an IR model using standard evaluation metrics.

**Code :-**

from sklearn.metrics import precision\_score,recall\_score,f1\_score

#Sample data (ground truth and predicated relevance)

ground\_truth = [1,0,1,1,0,0,0,1,1,1,0,0,1,0,1]

predicated\_relevance = [1,1,0,0,0,0,0,1,1,0,1,0,1,1,1]

print('Ground Truth = ',ground\_truth,'\nPredicated Relevance = ',predicated\_relevance)

#Calculate evalution metrics

precision = precision\_score(ground\_truth,predicated\_relevance)

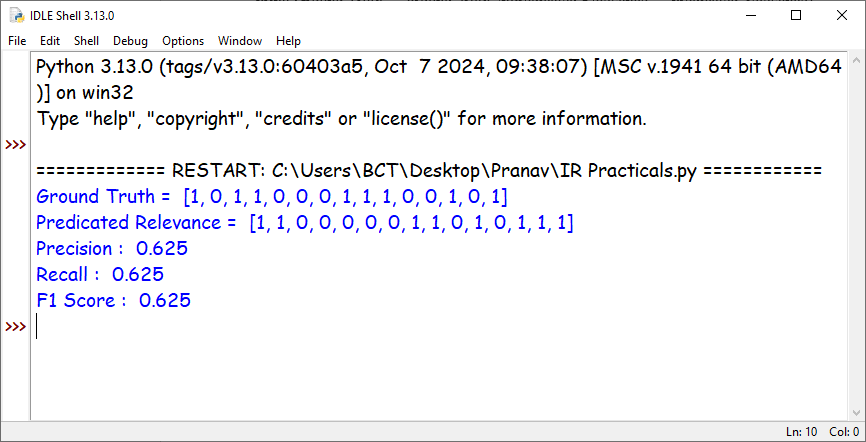
recall = recall\_score(ground\_truth,predicated\_relevance)

f1 = f1\_score(ground\_truth,predicated\_relevance)

#Print the results

print('Precision : ',precision,'\nRecall : ',recall,'\nF1 Score : ',f1)

**Output :-**



**Practical No. 4**

**Aim :-** Write a program to compute similarity between two text documents.

**Code :-**

import numpy as np

import pandas as pd

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

def cosine\_similarity(x,y):

#Ensure length of x and y are the same

if len(x) != len(y):

return None

#Compute the dot product between x and y

dot\_product = np.dot(x, y)

#Compute the 1, 2 norms (magnitudes) of x and y

magnitude\_x = np.sqrt(np.sum(x \*\* 2))

magnitude\_y = np.sqrt(np.sum(y \*\* 2))

#Compute the cosine similarity

cosine\_similarity = dot\_product / (magnitude\_x \* magnitude\_y)

return cosine\_similarity

corpus = ['Data Science is one of the most important fields of science','This is one of the best data science courses',

'Data Scientists analyse data']

#Create a matrix to represent the corpus

X = CountVectorizer().fit\_transform(corpus).toarray()

print(X)

cos\_sim\_1\_2 = cosine\_similarity(X[0,:],X[1,:])

cos\_sim\_1\_3 = cosine\_similarity(X[0,:],X[2,:])

cos\_sim\_2\_3 = cosine\_similarity(X[1,:],X[2,:])

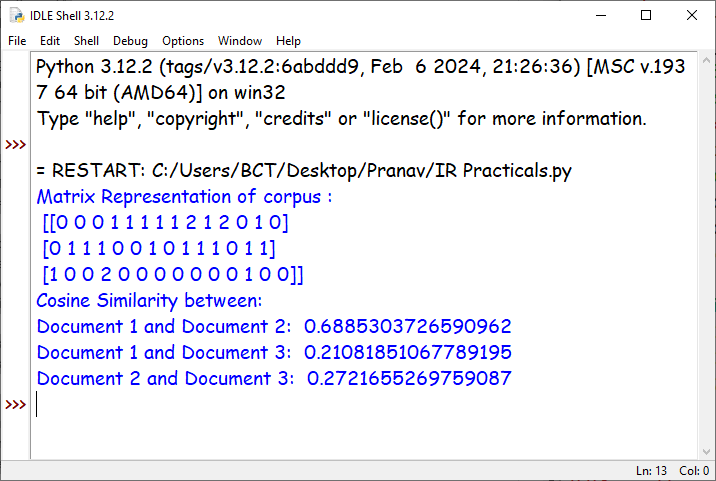
print('Cosine Similarity between: ')

print('Document 1 and Document 2: ', cos\_sim\_1\_2)

print('Document 1 and Document 3: ', cos\_sim\_1\_3)

print('Document 2 and Document 3: ', cos\_sim\_2\_3)

**Output :-**



**Practical No. 5**

**Aim :-** Write a program in python to implement text clustering using TFIDF and K Means

**Code :-**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score

# Sample corpus

documents = [

"The cat sat on the mat.",

"The dog barked at the cat.",

"I love programming in Python.",

"Python is a great programming language.",

"Dogs and cats are popular pets.",

"I enjoy solving coding challenges in Python."

]

print('Documents are : \n',documents,'\n')

# Step 1: Convert text to TF-IDF features

vectorizer = TfidfVectorizer(stop\_words="english")

tfidf\_matrix = vectorizer.fit\_transform(documents)

# Step 2: Apply K-Means clustering

num\_clusters = 3 # Define the number of clusters

kmeans = KMeans(n\_clusters=num\_clusters, random\_state=42)

kmeans.fit(tfidf\_matrix)

# Step 3: Assign documents to clusters

labels = kmeans.labels\_

# Display results

for i, doc in enumerate(documents):

print(f"Document {i + 1}: {doc}")

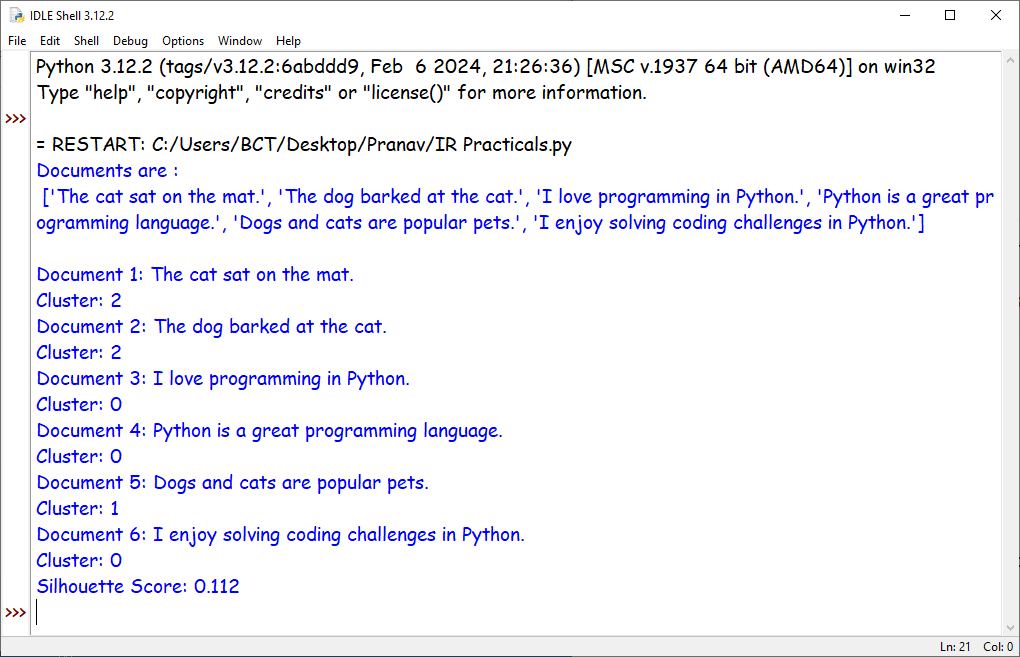
print(f"Cluster: {labels[i]}\n")

# Step 4: Evaluate clustering performance

silhouette\_avg = silhouette\_score(tfidf\_matrix, labels)

print(f"Silhouette Score: {silhouette\_avg:.3f}")

**Output :-**



**Practical No. 6**

**Aim :-** Write program for pre-processing of Text document: stop word removal

**Code :-**

import nltk

##nltk.download('stopwords')

##nltk.download('punkt')

##nltk.download('punkt\_tab')

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

example\_sentence = 'Quantum mechanics describes the behavior of particles at the atomic level.'

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

# Tokenize the sentence

word\_tokens = word\_tokenize(example\_sentence)

# Filter out stop words

filtered\_sentence = [w for w in word\_tokens if w.lower() not in stop\_words]

stopwords\_found = [w for w in word\_tokens if w.lower() in stop\_words]

# Print results

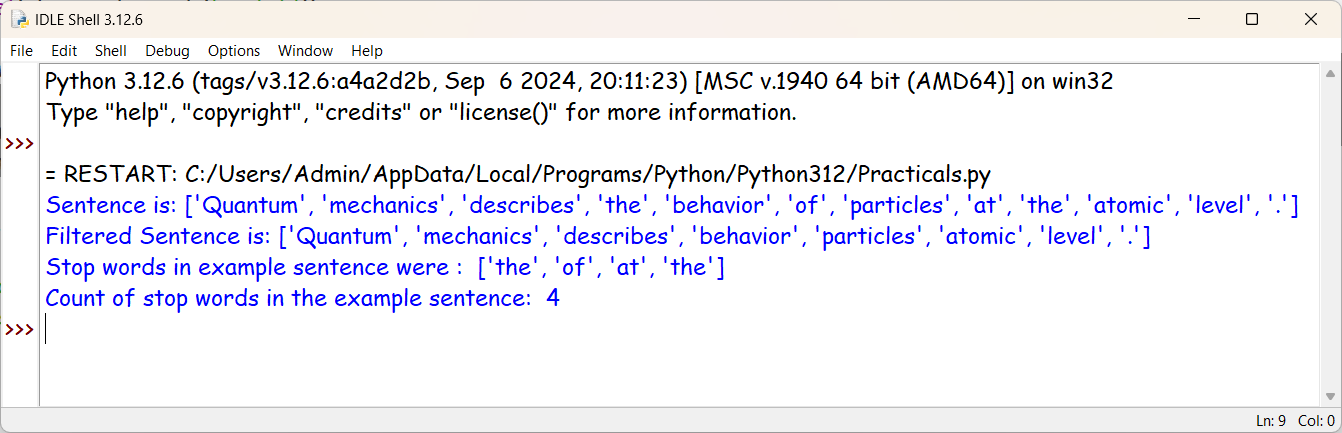
print(f'Sentence is: {word\_tokens}')

print(f'Filtered Sentence is: {filtered\_sentence}')

print('Stop words in example sentence were : ',stopwords\_found)

print('Count of stop words in the example sentence: ', len(stopwords\_found))

**Output :-**



**Practical No. 7**

**Aim:-** Write a python code to demonstrate the concept of summarization using hugging face transformers library

**Code:-**

from transformers import pipeline

summarizer = pipeline("summarization", model="Falconsai/text\_summarization")

##print(summarizer)

ARTICLE = """

Information retrieval is fundamentally concerned with the efficient process of locating the right data at the precise moment it is needed.

Imagine it as similar to searching for your favorite song on your smartphone; you simply type in the title or the artist's name and almost instantly the song appears ready for you to play and enjoy. In a comparable manner information retrieval systems are designed to assist us in navigating through vast amounts of information whether that be from extensive libraries of books or the endless resources available on the internet. This entire process often involves the use of specific keywords or phrases which are essential for generating the most relevant and useful results. By utilizing sophisticated algorithms and indexing techniques these systems streamline our search efforts making it more efficient and effective. Ultimately information retrieval significantly enhances our daily lives by allowing us to swiftly locate and access the precise information we seek whether for personal enjoyment educational purposes or professional development.

It acts as a vital tool in our increasingly data-driven world.

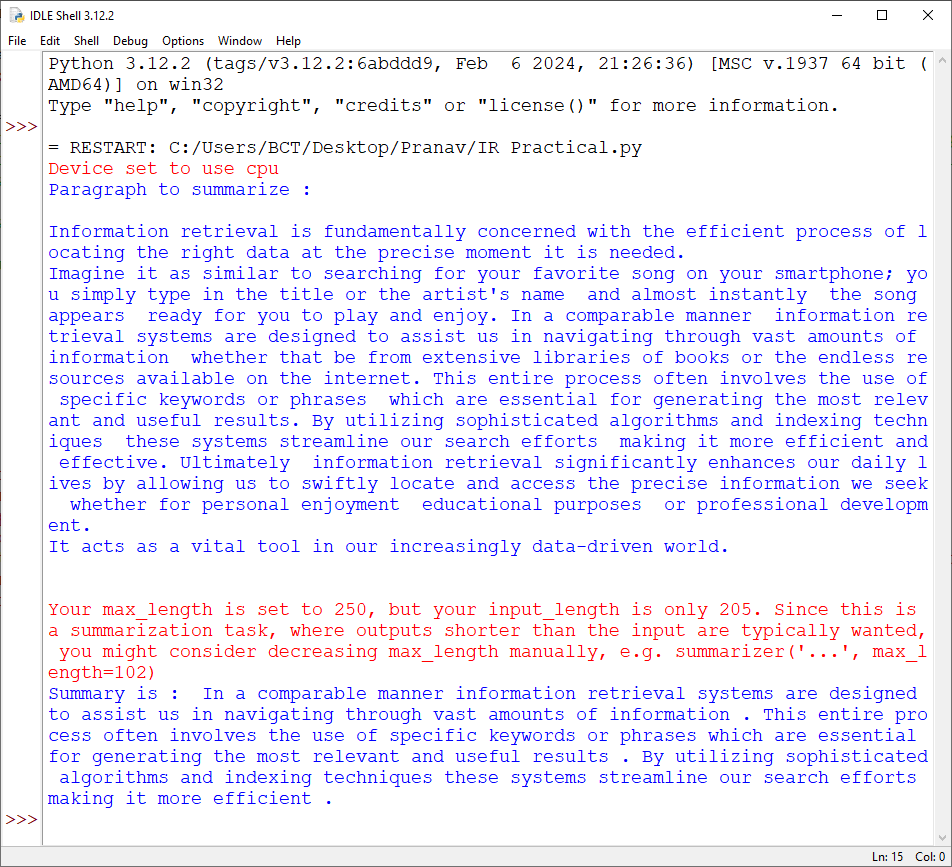
"""

print('Paragraph to summarize : \n',ARTICLE,'\n')

result = summarizer(ARTICLE, max\_length=250, min\_length=30, do\_sample=False)

print('Summary is : ',result[0]['summary\_text'])

**Output:-**



**Practical No. 8**

**Aim:-** Write a program in python to demonstrate a simple question answering system to demonstrate hugging face transformer model

**Code:-**

from transformers import pipeline

model\_name = "deepset/roberta-base-squad2"

nlp = pipeline('question-answering', model=model\_name, tokenizer=model\_name)

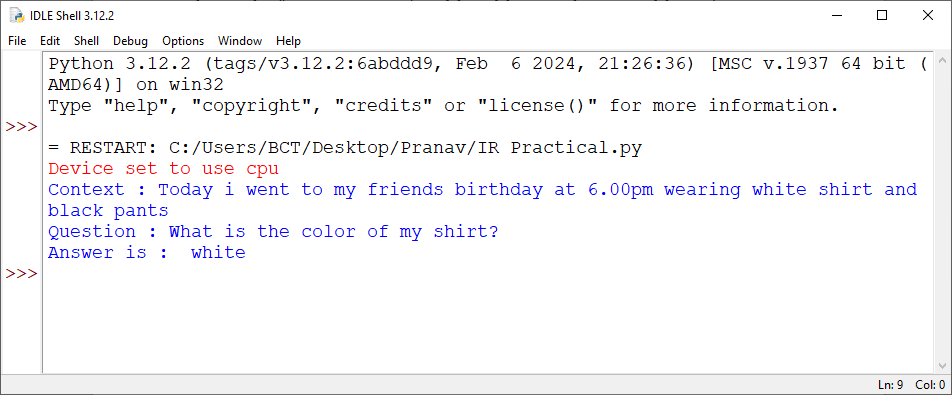
question = 'What is the color of my shirt?'

context = '''Today i went to my friends birthday at 6.00pm wearing white shirt and black pants '''

res = nlp(question = question,context= context)

print(f'Context : {context}\nQuestion : {question}\nAnswer is : ',res['answer'])

**Output :-**



**Practical No. 9**

**Aim: -** Implement Dynamic programming algorithm for computing the edit distance between strings s1 and s2.

**Code:**

def Levenshtein(s1, s2):

if s1 == "":

return len(s2)

elif s2 == "":

return len(s1)

elif s1[-1] == s2[-1]:

cost = 0

else:

cost = 1

res = min(

[

Levenshtein(s1[:-1], s2) + 1, # deletion

Levenshtein(s1, s2[:-1]) + 1, # insertion

Levenshtein(s1[:-1], s2[:-1]) + cost, # substitution

]

)

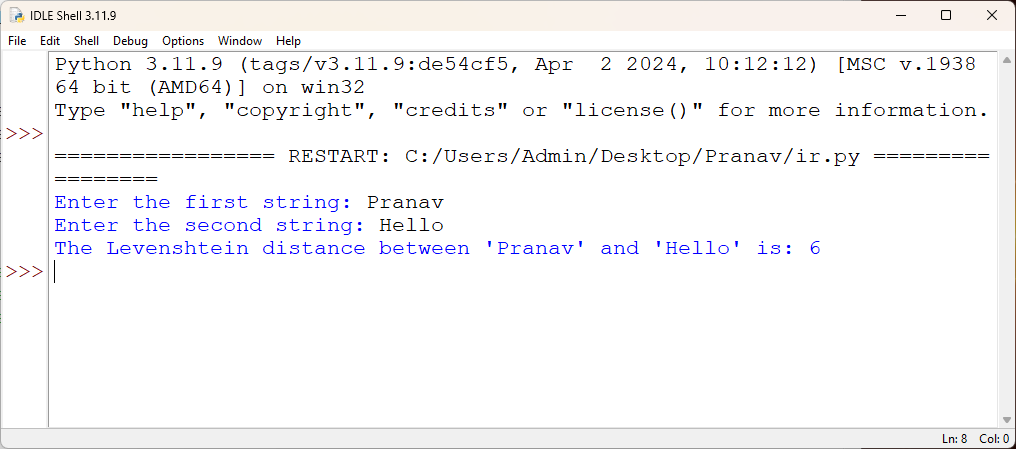
return res

s1 = input("Enter the first string: ")

s2 = input("Enter the second string: ")

print(f"The Levenshtein distance between '{s1}' and '{s2}' is: {Levenshtein(s1, s2)}")

**Output:**



**Practical No. 10**

**Aim: -** Write a program to implement simple crawler.

**Code:**

import requests

from bs4 import BeautifulSoup

from urllib.parse import urljoin

# URL of the website to crawl

base\_url = "https://www.google.com/"

# Set to store visited URLs

visited\_urls = set()

# List to store URLs to visit next

urls\_to\_visit = [base\_url]

# Function to crawl a page and extract links

def crawl\_page(url):

try:

response = requests.get(url)

response.raise\_for\_status() # Raise an exception for HTTP errors

soup = BeautifulSoup(response.content, "html.parser")

# Extract links and enqueue new URLs

links = []

for link in soup.find\_all("a", href=True):

next\_url = urljoin(url, link["href"])

links.append(next\_url)

return links

except requests.exceptions.RequestException as e:

print(f"Error crawling {url}: {e}")

return []

# Crawl the website

while urls\_to\_visit:

current\_url = urls\_to\_visit.pop(0) # Dequeue the first URL

if current\_url in visited\_urls:

continue

print(f"Crawling: {current\_url}")

new\_links = crawl\_page(current\_url)

visited\_urls.add(current\_url)

urls\_to\_visit.extend(new\_links)

print("Crawling finished.")

**Output:**



**Practical No. 11**

**Aim: -** Demonstrate a simple web scraping process using Python within the environment.

**Code:**

import requests

from bs4 import BeautifulSoup

# Specify the URL you want to scrape

url = "https://google.com"

# Send a GET request to the URL

response = requests.get(url)

# Check if the request was successful (status code 200)

if response.status\_code == 200:

# Parse the HTML content of the page

soup = BeautifulSoup(response.text, "html.parser")

# Find and print the text content (modify as needed based on the HTML structure)

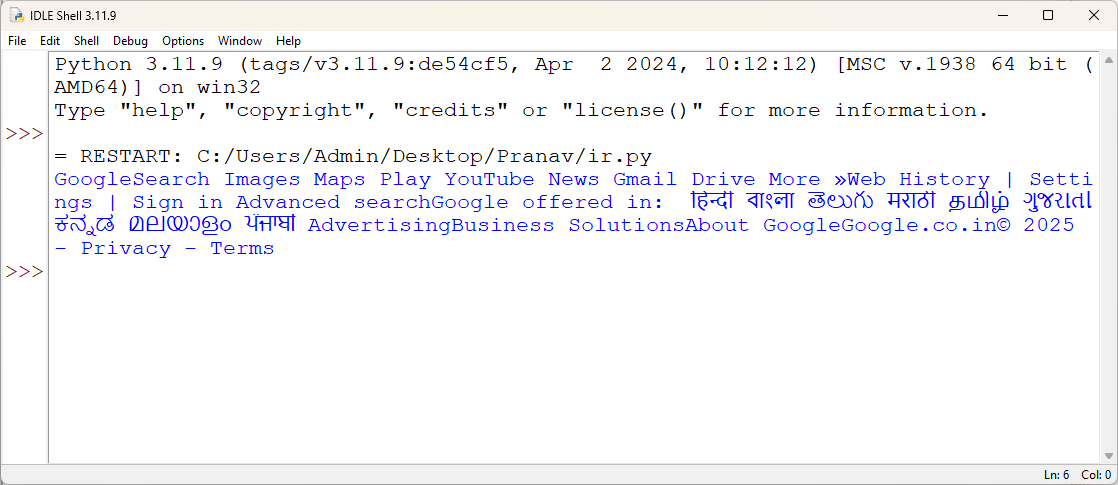
text\_content = soup.get\_text()

print(text\_content)

else:

print(f"Error: Unable to fetch content. Status code: {response.status\_code}")

**Output:**



**Practical No. 12**

**Aim: -** Write a program to parse XML text, generate Web graph and compute topic specific page rank.

**Code: -**

1. **movies.xml:**

<collection shelf="New Arrivals">

<movie title="One Piece: Stampede">

   <type>Anime, Action</type>

   <format>Blu-ray</format>

   <year>2019</year>

   <rating>PG-13</rating>

   <stars>9</stars>

   <description>The Pirates Festival</description>

</movie>

<movie title="Your Name">

   <type>Anime, Romance, Fantasy</type>

   <format>Blu-ray</format>

   <year>2016</year>

   <rating>PG</rating>

   <stars>9</stars>

   <description>A beautiful story of two strangers</description>

</movie>

<movie title="3 Idiots">

   <type>Comedy, Drama</type>

   <format>DVD</format>

   <year>2009</year>

   <rating>PG-13</rating>

   <stars>10</stars>

   <description>A story of three friends in an engineering college</description>

</movie>

<movie title="Dangal">

   <type>Action, Biography, Drama</type>

   <format>DVD</format>

   <year>2016</year>

   <rating>PG</rating>

   <stars>9</stars>

   <description>Based on the true story of wrestler Mahavir Singh Phogat</description>

</movie>

<movie title="Andhadhun">

   <type>Suspense, Thriller</type>

   <format>Blu-ray</format>

   <year>2018</year>

   <rating>R</rating>

   <stars>8</stars>

   <description>A blind pianist who witnesses a murder</description>

</movie>

</collection>

1. **Python code:**

import networkx as nx

import matplotlib.pyplot as plt

from xml.dom.minidom import parse

import xml.dom.minidom

# Open xml document using minidom parser

DOMTree = xml.dom.minidom.parse("movies.xml")

collection = DOMTree.documentElement

if collection.hasAttribute("shelf"):

print("Root element: %s" % collection.getAttribute("shelf"))

# get all the movies in the collection

movies = collection.getElementsByTagName("movie")

# print detail of each movie.

for movie in movies:

print("\*\*\*\*\*Movie\*\*\*\*\*")

if movie.hasAttribute("title"):

print("Title: %s" % movie.getAttribute("title"))

type = movie.getElementsByTagName('type')[0]

print("Type: %s" % type.childNodes[0].data)

format = movie.getElementsByTagName('format')[0]

print("Format: %s" % format.childNodes[0].data)

rating = movie.getElementsByTagName('rating')[0]

print("Rating: %s" % rating.childNodes[0].data)

description = movie.getElementsByTagName('description')[0]

print("Description: %s" % description.childNodes[0].data)

def GenerateGraph():

G = nx.Graph()

# adding just one node:

G.add\_node("a")

# adding a list of edges:

G.add\_edges\_from([("a", "b"), ("b", "c"), ("c", "d"), ("d", "a"), ("a", "c")])

nx.draw(G)

plt.savefig("simple\_path.png") # save as png

plt.show() # display

print("Nodes of graph: ")

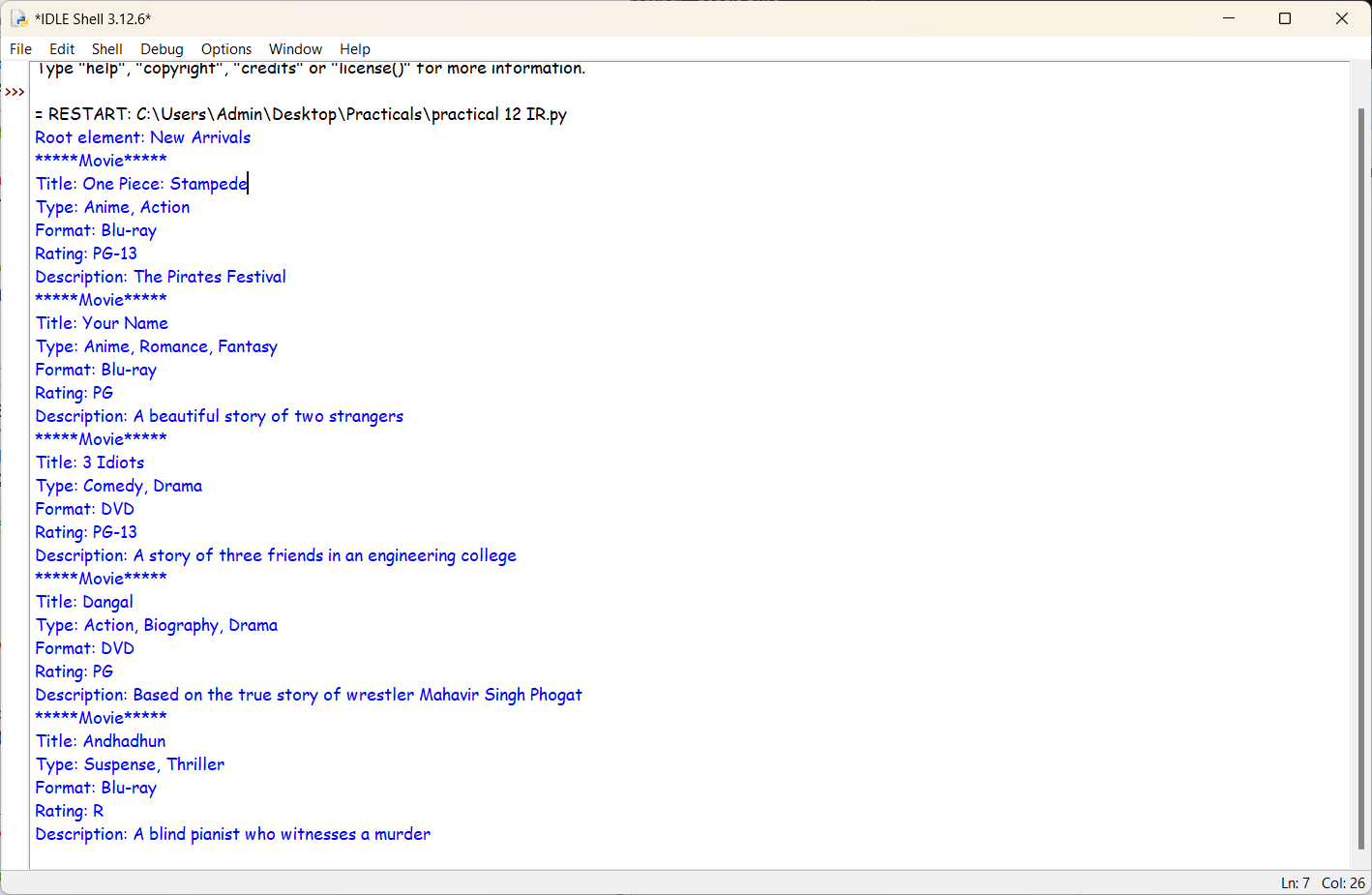
print(G.nodes())

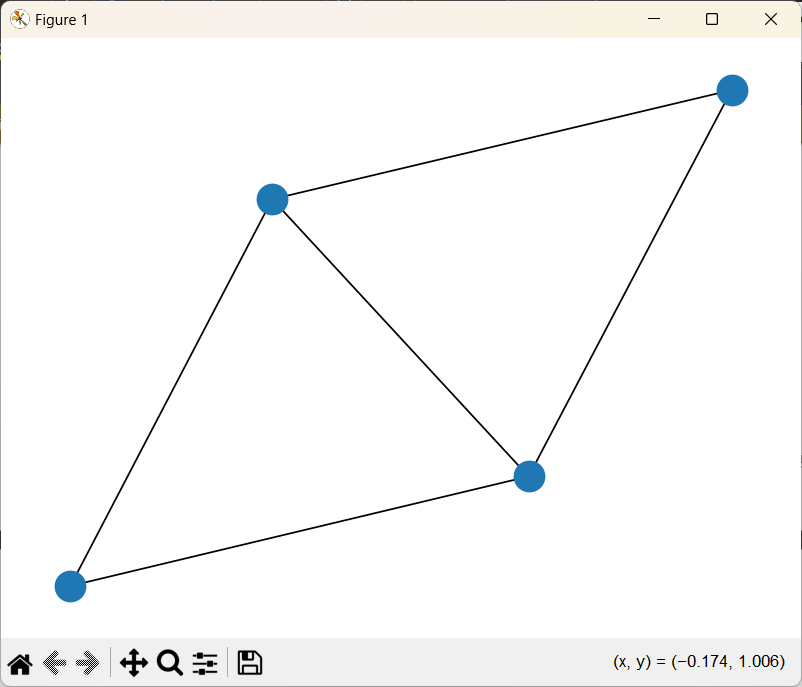
print("Edges of graph: ")

print(G.edges())

GenerateGraph()

1. **Output:**





**Practical No. 13**

**Aim: -** Calculate Page rank along with hubs and authorities.

**Code:**

import networkx as nx

import matplotlib.pyplot as plt

# Create a directed graph (replace this with your own graph)

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 1)])

# Calculate PageRank

pagerank\_scores = nx.pagerank(G)

# Calculate HITS (Hub and Authority) scores

hits\_scores = nx.hits(G)

# Print the results

print("PageRank Scores:", pagerank\_scores)

print("Hub Scores:", hits\_scores[0])

print("Authority Scores:", hits\_scores[1])

# Visualize the graph

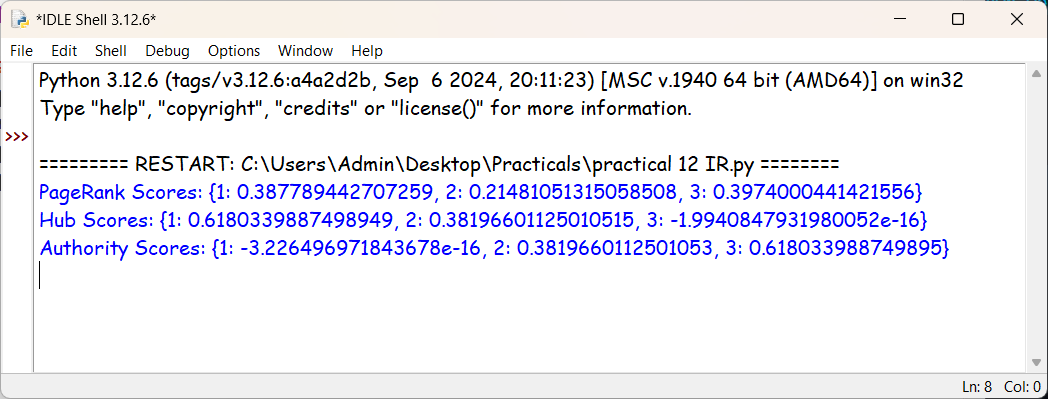
pos = nx.spring\_layout(G) # positions for all nodes

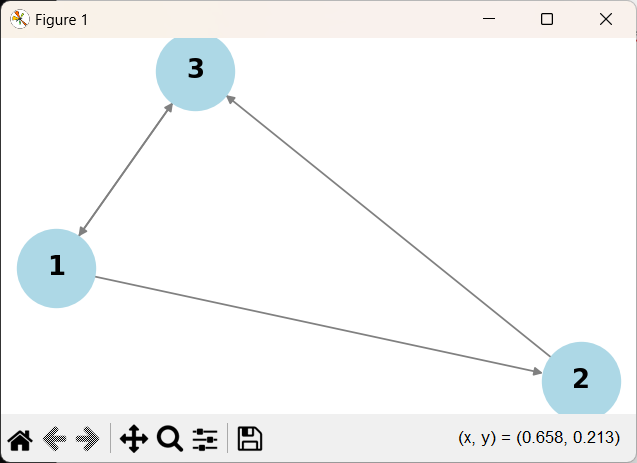
nx.draw(G, pos, with\_labels=True, node\_color='lightblue', edge\_color='gray', node\_size=2000, font\_size=15, font\_weight='bold')

plt.title("Directed Graph Visualization")

plt.show()

**Output:**





**Practical No. 14**

**Aim: -** Implement simple Word2Vec in python for information retrieval

**Code: -**

import nltk

import gensim

import numpy as np

from nltk.tokenize import word\_tokenize

from sklearn.metrics.pairwise import cosine\_similarity

nltk.download("punkt")#Download tokenizer

# Sample documents

documents = [

"Information retrieval is a key area in search engines.",

"Machine learning helps improve search relevance.",

"Deep learning and AI are advancing information retrieval.",

"Search engines use algorithms to rank results.",

"Artificial intelligence enhances text processing."

]

#Tokenize documents

tokenized\_docs = [word\_tokenize(doc.lower()) for doc in documents]

# Train Word2Vec model

word2vec\_model = gensim.models.Word2Vec(tokenized\_docs, vector\_size=100, window=5, min\_count=1, workers=4)

# Function to get sentence vector (average of word vectors)

def get\_sentence\_vector(sentence, model):

words = word\_tokenize(sentence.lower())

vectors = [model.wv[word] for word in words if word in model.wv]

if vectors:

return np.mean(vectors, axis=0) # Average word vectors

else:

return np.zeros(model.vector\_size) # Return zero vector if no valid words

# Function to search relevant documents

def search(query):

query\_vector = get\_sentence\_vector(query, word2vec\_model)

doc\_vectors = np.array([get\_sentence\_vector(doc, word2vec\_model) for doc in documents])

# Compute cosine similarity

scores = cosine\_similarity([query\_vector], doc\_vectors)[0]

# Rank results

ranked\_results = sorted(enumerate(scores), key=lambda x: x[1], reverse=True)

print("\nSearch Results for:", query)

for i, score in ranked\_results:

if score > 0:

print(f"Score: {score:.4f} | Document: {documents[i]}")

else:

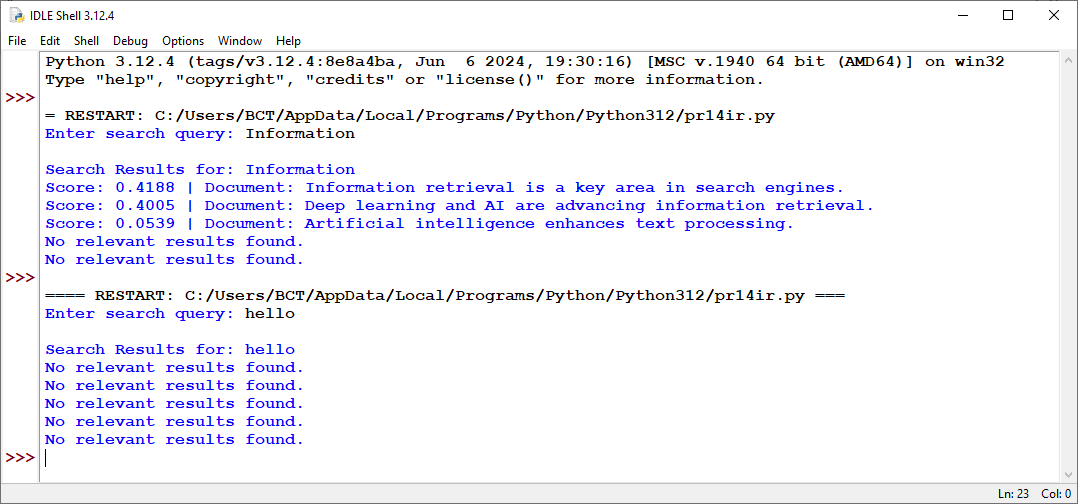
print("No relevant results found.")

# Example search query

query = input("Enter search query: ")

search(query)

**Output:**



**Practical No. 15**

**Aim:-** Write a program in python to Implement Page Rank Algorithm.

**Code:-**

import numpy as np

def pagerank(G, beta=0.85, tol=1.0e-6,max\_iter=100):

"""

Computes the PageRank for a given adjacency matrix G.

Parameters:

G : numpy array : Adjacency matrix representing the graph

beta : float : Damping factor (default is 0.85)

tol : float : Convergence tolerance (default is 1e-6)

max\_iter : int : Maximum number of iterations (default is 100)

Returns:

numpy array : PageRank scores for each page

"""

n= len(G)

#Convert adjacency matrix to a stochastic matrix

M= np.zeros((n,n))

for i in range(n):

row\_sum= np.sum(G[i])

if row\_sum==0: #Handling dangling nodes(pages with no outlinks)

M[i] = np.ones(n)/n

else:

M[i]=G[i]/row\_sum #Normalize row to make it stochastic

#Initialize rank vector with equal probability

R= np.ones(n)/n

#Teleportation matrix (used to handle rank sinks)

E= np.ones((n,n))/n

#Compute transition probability matrix with damping factor

A= beta \* M + (1-beta) \* E

#Iteratively compute PageRank

for \_ in range(max\_iter):

new\_R= A @ R #Matrix-vector multiplication

if np.linalg.norm(new\_R-R,ord=1) < tol: #Convergence Check

break

R= new\_R

return np.round(R,5) #Round results for better readability

#Example adjacency matrix(3 pages linking to each other)

G= np.array([

[0,1,1],

[1,0,1],

[0,1,0]

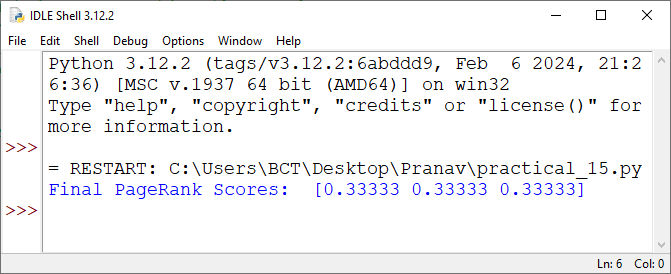
])

#Compute PageRank

page\_ranks = pagerank(G)

print("Final PageRank Scores: ",page\_ranks)

**Output:**

****