PRACTICAL : 01

AIM :-

(a): Implement an inverted index construction algorithm.

THEORY :

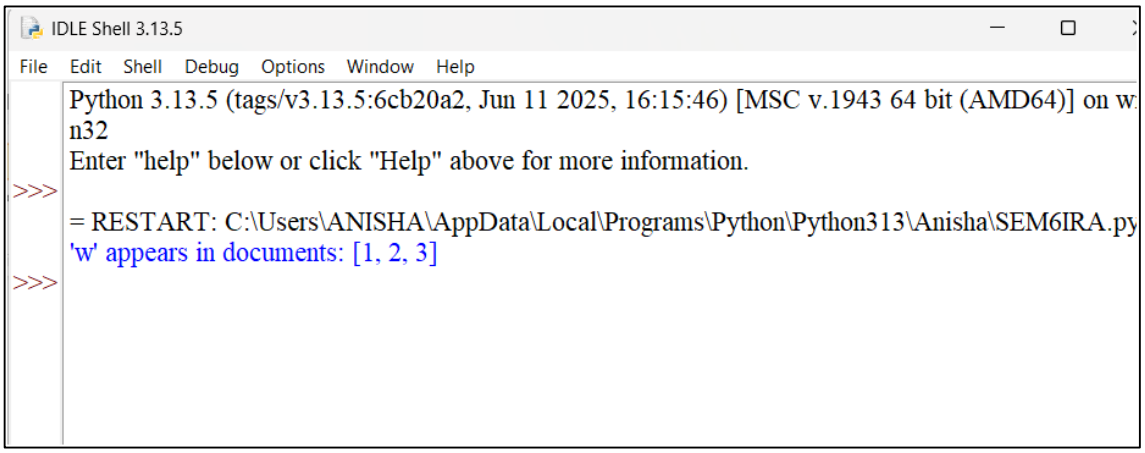
An inverted index is a data structure used in information retrieval systems to map each term (word) to the list of documents containing that term. It enables fast and efficient full-text searches by linking words to their document occurrences, forming the backbone of search engines and text retrieval applications.

CODE:

```
from collections import defaultdict
import re
def preprocess(text):
  # Lowercase and tokenize text with simple regex, remove duplicates
   tokens = set(re.findall(r'w+', text.lower()))
   return tokens
def build_inverted_index(documents):
   inverted_index = defaultdict(set)
   for doc_id, text in documents.items():
      terms = preprocess(text)
      for term in terms:
         inverted_index[term].add(doc_id)
   return inverted_index
  # Sample documents dictionary with doc_id as keys
documents = {
   1: "The quick brown fox jumps over the lazy dog.",
   2: "A brown fox is quick and the dog is lazy.",
   3: "The sun is shining and the weather is warm."
}
```

```
# Build the inverted index
index = build_inverted_index(documents)
# Print index
for term, doc_ids in index.items():
    print(f"'{term}' appears in documents: {sorted(doc_ids)}")
```

OUTPUT :



AIM:-

(b): Build a simple document retrieval system using the constructed index.

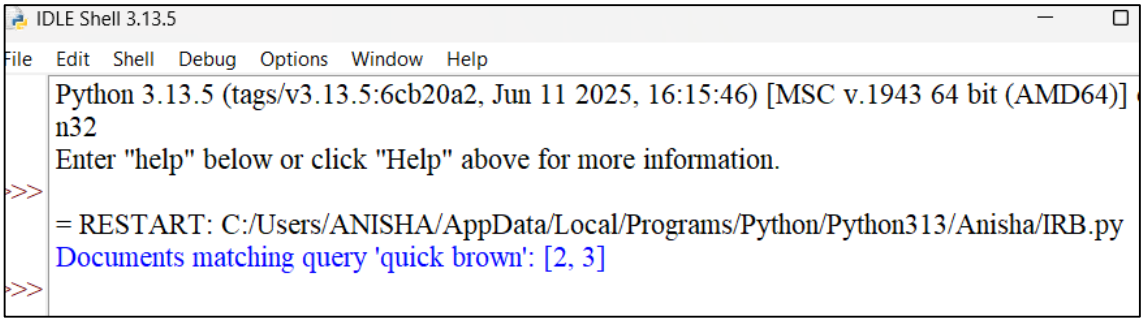
THEORY:

This code implements a **simple document retrieval system** using an **inverted index**, where each word maps to a set of document IDs containing it. The function retrieve_documents() processes a query, finds documents containing all query terms (AND logic), and returns the matching document IDs.

CODE:

```
def retrieve_documents(query, index):
    query_terms = query.lower().split()
    # Find documents containing all query terms (AND logic)
    docs_sets = [index.get(term, set()) for term in query_terms]
    if docs_sets:
        result_docs = set.intersection(*docs_sets)
    else:
        result_docs = set()
    return result_docs
# Example inverted index
index = {
    "quick": {1, 2, 3},
    "brown": {2, 3},
    "fox": {3}
}
# Example query
query = "quick brown"
# Retrieve documents from index
matching_docs = retrieve_documents(query, index)
print(f"Documents matching query '{query}': {sorted(matching_docs)}")
```

OUTPUT:


IDLE Shell 3.13.5
File Edit Shell Debug Options Window Help
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (AMD64)]
n32
Enter "help" below or click "Help" above for more information.
>>>
= RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/Anisha/IRB.py
Documents matching query 'quick brown': [2, 3]
>>>

PRACTICAL: 02

AIM:

(a) Implement the Boolean retrieval model and process queries.

THEORY:

The **Boolean Retrieval Model** is a classic information retrieval approach that uses Boolean logic (AND, OR, NOT) to match documents exactly with query terms. Documents are represented as sets of indexed terms, and queries are processed by combining these sets based on Boolean operators to retrieve relevant documents.

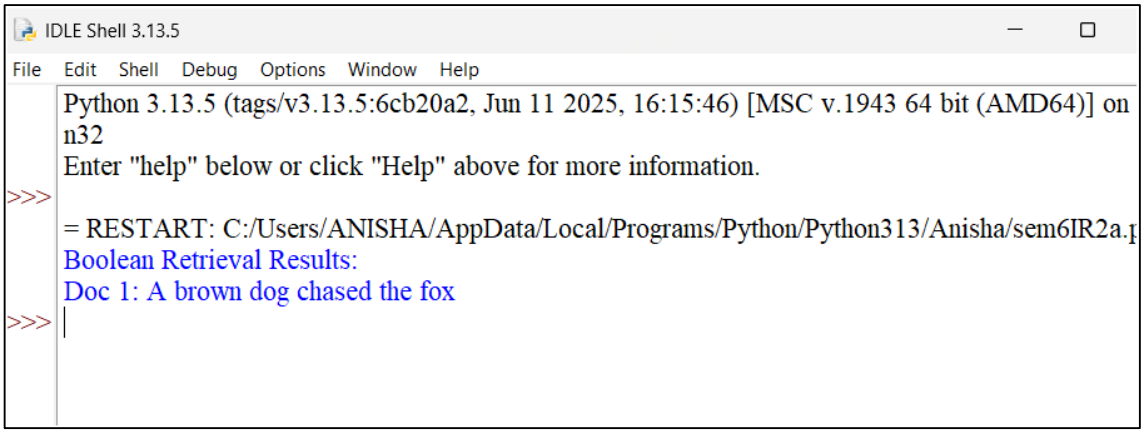
CODE:

```
import re

class BooleanRetrieval:
    def __init__(self, documents):
        self.documents = documents
        self.index = self.build_index(documents)
    def build_index(self, documents):
        index = {}
        for i, doc in enumerate(documents):
            # Extract words and lowercase them
            words = set(re.findall(r"\w+", doc.lower()))
            for word in words:
                index.setdefault(word, set()).add(i)
        return index
    def process_query(self, query):
        tokens = query.lower().split()
        result_set = set()
        operator = None
        i = 0
        while i < len(tokens):
            token = tokens[i]
            if token in ("and", "or", "not"):
```

```
            operator = token
            i += 1
            continue
        term_docs = self.index.get(token, set())
        if operator == "not":
            if not result_set:
                result_set = set(range(len(self.documents)))
            result_set = result_set.difference(term_docs)
        elif operator == "or":
            result_set = result_set.union(term_docs)
        elif operator == "and" or operator is None:
            if not result_set:
                result_set = term_docs.copy()
            else:
                result_set = result_set.intersection(term_docs)
        operator = None
        i += 1
    return sorted(result_set)
# Example usage
if __name__ == "__main__":
    documents = [
        "The quick brown fox jumps over the lazy dog",
        "A brown dog chased the fox",
        "The dog is lazy",
        "Fox and dog are friends",
        "Lazy dogs are not quick"
    ]
    bool_model = BooleanRetrieval(documents)
    query = "brown AND dog AND NOT lazy"
    results = bool_model.process_query(query)
```

```
print("Boolean Retrieval Results:")
for idx in results:
    print(f"Doc {idx}: {documents[idx]}")
```

OUTPUT :



AIM:

(b) Implement the vector space model with TF-IDF weighting and cosine similarity.

THEORY:

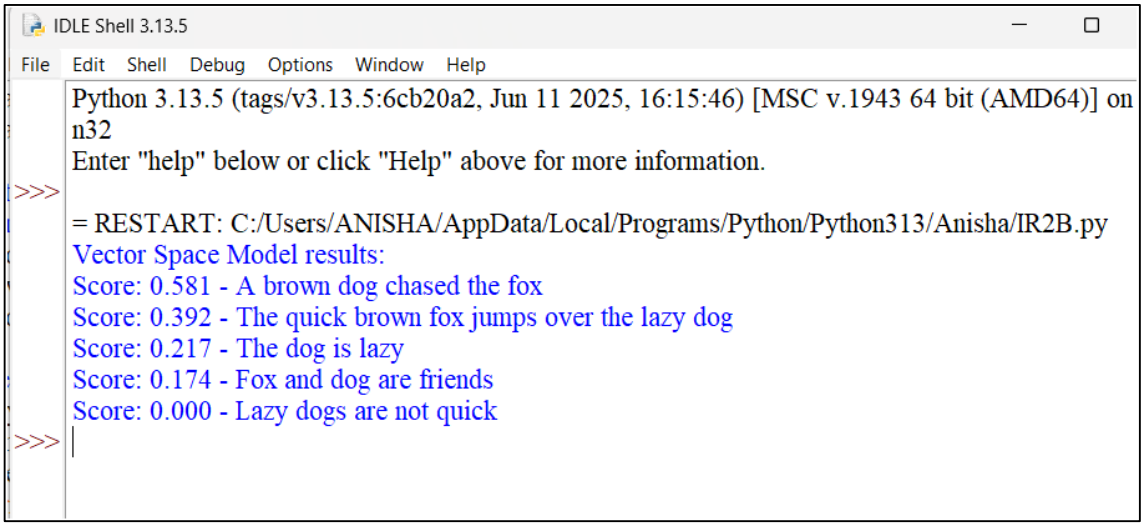
This program implements the Vector Space Model (VSM) using TF-IDF weighting and cosine similarity to measure how closely each document matches a query. TF-IDF represents the importance of terms, while cosine similarity computes the angle between vectors to rank documents based on relevance to the query.

CODE:

```
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine_similarity

class VectorSpaceModel:

    def __init__(self, documents):

        self.documents = documents

        self.vectorizer = TfidfVectorizer()

        self.doc_vectors = self.vectorizer.fit_transform(documents)

    def query(self, query_text):

        query_vec = self.vectorizer.transform([query_text])

        cosine_sim = cosine_similarity(query_vec, self.doc_vectors).flatten()

        ranked_doc_indices = cosine_sim.argsort()[::-1]

        return [(self.documents[i], cosine_sim[i]) for i in ranked_doc_indices]

# Example usage

if __name__ == "__main__":

    documents = [

        "The quick brown fox jumps over the lazy dog",

        "A brown dog chased the fox",

        "The dog is lazy",

        "Fox and dog are friends",

        "Lazy dogs are not quick"

    ]

    vsm = VectorSpaceModel(documents)
```

```
query = "brown dog"
results = vsm.query(query)
print("Vector Space Model results:")
for doc, score in results:
    print(f"Score: {score:.3f} - {doc}")
```

OUTPUT:



PRACTICAL: 03

AIM :

(a): Develop a spelling correction module using edit distance algorithms.

THEORY:

Spelling Correction is a key component in Information Retrieval systems to handle user typos or misspellings.
The **Edit Distance Algorithm** (Levenshtein Distance) measures how dissimilar two strings are by counting the **minimum number of operations** required to transform one word into the other.

Allowed Operations:

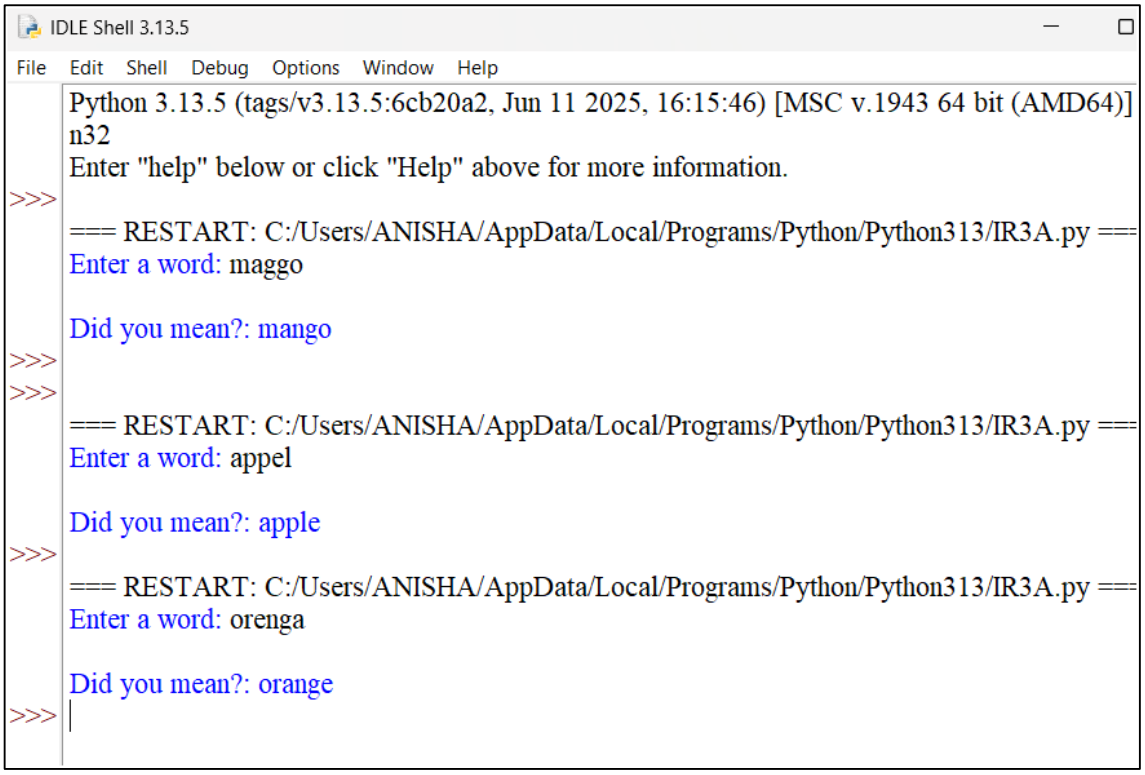
1. **Insertion** – Add a character
2. **Deletion** – Remove a character
3. **Substitution** – Replace one character with another

CODE:

```
def edit_distance(str1, str2):
    m, n = len(str1), len(str2)
    dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]
    # Build DP table
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j  # Insert all characters of str2
            elif j == 0:
                dp[i][j] = i  # Remove all characters of str1
            elif str1[i - 1] == str2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(
                    dp[i - 1][j],      # Deletion
                    dp[i][j - 1],      # Insertion
                    dp[i - 1][j - 1]   # Substitution
                )
```

```
    return dp[m][n]
def correct_spelling(word, dictionary):
    distances = {w: edit_distance(word, w) for w in dictionary}
    min_distance = min(distances.values())
    suggestions = [w for w, d in distances.items() if d == min_distance]
    return suggestions
# Example dictionary
dictionary = ["apple", "banana", "orange", "grape", "pineapple", "mango"]
# Input misspelled word
user_word = input("Enter a word: ").lower()
# Get correction suggestions
suggestions = correct_spelling(user_word, dictionary)
print("\nDid you mean?:", ", ".join(suggestions))
```

OUTPUT:



AIM:

(b): Integrate the spelling correction module into an information retrieval system.

THEORY:

In a basic **Information Retrieval System**, user queries are matched with documents to retrieve the most relevant results.
However, users often make **spelling mistakes** while typing queries. To handle this, we integrate a **Spelling Correction Module** that suggests or auto-corrects misspelled words before searching.

The flow of the system:

1. Accept a query from the user.
2. Tokenize the query into words.
3. For each word, check if it exists in the **dictionary** (terms from the document corpus).
4. If a word is not found, apply the **Edit Distance Algorithm** to suggest the closest correct word.
5. Retrieve and display matching documents based on the corrected query.

CODE:

```
from collections import defaultdict
import re
# ----- (A) Edit Distance Function -----
def edit_distance(str1, str2):
    m, n = len(str1), len(str2)
    dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif str1[i - 1] == str2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])
    return dp[m][n]

# ----- (B) Spelling Correction -----
def correct_spelling(word, dictionary):
    distances = {w: edit_distance(word, w) for w in dictionary}
    min_distance = min(distances.values())
    suggestions = [w for w, d in distances.items() if d == min_distance]
    return suggestions[0]  # choose first suggestion

# ----- (C) Build a Simple Inverted Index -----
def preprocess(text):
    return re.findall(r'\w+', text.lower())
```

```
def build_inverted_index(documents):
    inverted_index = defaultdict(set)
    for doc_id, text in documents.items():
        for word in preprocess(text):
            inverted_index[word].add(doc_id)
    return inverted_index

# ----- (D) Search Function -----
def search(query, inverted_index, dictionary):
    query_words = preprocess(query)
    corrected_query = []
    for word in query_words:
        if word in dictionary:
            corrected_query.append(word)
        else:
            corrected = correct_spelling(word, dictionary)
            print(f"Word '{word}' not found. Did you mean '{corrected}'?")
            corrected_query.append(corrected)
# Retrieve documents containing all corrected query words
    results = None
    for word in corrected_query:
        if word in inverted_index:
            if results is None:
                results = inverted_index[word]
            else:
                results = results.intersection(inverted_index[word])
    return results if results else set()

# ----- (E) Main Program -----
if __name__ == "__main__":
    # Sample documents
    documents = {
        1: "Apple and banana are fruits.",
        2: "I like to eat an apple every day.",
        3: "Mango is a tropical fruit.",
        4: "Banana milkshake is delicious."
    }
    inverted_index = build_inverted_index(documents)
    dictionary = set(inverted_index.keys())
    print("Documents:", documents)
    print("\nDictionary terms:", dictionary)
    query = input("\nEnter your search query: ").lower()
    result_docs = search(query, inverted_index, dictionary)
    if result_docs:
        print("\nRelevant Documents:")
        for doc_id in result_docs:
            print(f"Doc {doc_id}: {documents[doc_id]}")
    else:
        print("\nNo matching documents found.")
```

OUTPUT:


IDLE Shell 3.13.5
File Edit Shell Debug Options Window Help
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (AMD64)] on wi
n32
Enter "help" below or click "Help" above for more information.
>>>
=== RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/IR3B.py ===
Documents: {1: 'Apple and banana are fruits.', 2: 'I like to eat an apple every day.', 3: 'Mango is a tr
opical fruit.', 4: 'Banana milkshake is delicious.'}
Dictionary terms: {'apple', 'banana', 'like', 'day', 'mango', 'are', 'to', 'is', 'i', 'delicious', 'and', 'fruit', 'eat'
, 'a', 'tropical', 'fruits', 'milkshake', 'an', 'every'}
Enter your search query: magno
Word 'magno' not found. Did you mean 'mango'?
Relevant Documents:
Doc 3: Mango is a tropical fruit.
>>>
=== RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/IR3B.py ===
Documents: {1: 'Apple and banana are fruits.', 2: 'I like to eat an apple every day.', 3: 'Mango is a tr
opical fruit.', 4: 'Banana milkshake is delicious.'}
Dictionary terms: {'to', 'eat', 'tropical', 'a', 'milkshake', 'every', 'like', 'apple', 'and', 'is', 'mango', 'fruits',
'an', 'i', 'banana', 'are', 'fruit', 'day', 'delicious'}
Enter your search query: aplpe
Word 'aplpe' not found. Did you mean 'apple'?
Relevant Documents:
Doc 1: Apple and banana are fruits.
Doc 2: I like to eat an apple every day.
>>>

PRACTICAL: 04

AIM:

(a): Calculate precision, recall, and F-measure for a given set of retrieval results.

THEORY:

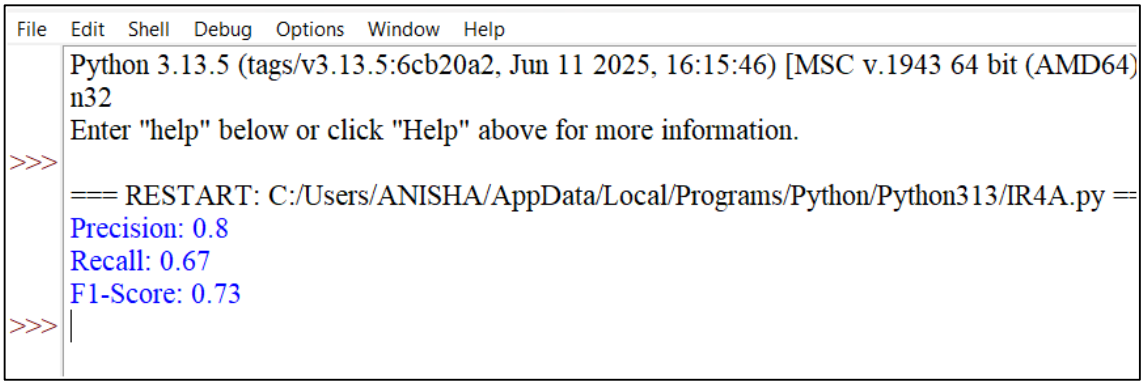
Evaluation metrics help to measure how well an Information Retrieval (IR) system performs.

- Precision shows how many retrieved documents are relevant.
- Recall shows how many relevant documents are retrieved.
- F-measure is the harmonic mean of precision and recall.

CODE:

```
from sklearn.metrics import precision_score, recall_score, f1_score

# 1 = Relevant, 0 = Not Relevant

actual_results = [1, 0, 1, 1, 0, 1, 0, 0, 1, 1]     # Ground truth

retrieved_results = [1, 0, 1, 0, 0, 1, 1, 0, 0, 1]  # System output

precision = precision_score(actual_results, retrieved_results)

recall = recall_score(actual_results, retrieved_results)

f1 = f1_score(actual_results, retrieved_results)

print("Precision:", round(precision, 2))

print("Recall:", round(recall, 2))

print("F1-Score:", round(f1, 2))
```

OUTPUT:



AIM:

(b): Use an evaluation toolkit to measure average precision and other evaluation metrics.

THEORY:

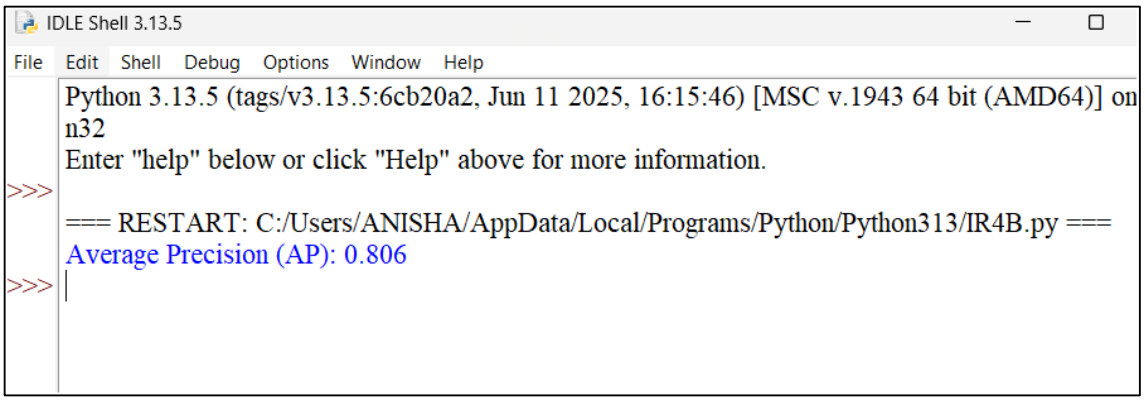
- It measures how well a system ranks *relevant documents* for **a single query**.
- It averages the **precision values** obtained each time a relevant document is retrieved.
- Formula:

$$AP = \frac{\sum_{k=1}^{N} \quad (Precision@k \times rel(k))}{\text{Number of relevant documents}}$$

CODE:

```
import numpy as np

# 1 = Relevant, 0 = Not Relevant

relevance = [1, 0, 1, 1, 0]  # Relevance of retrieved documents

def average_precision(r):

    r = np.asarray(r)

    precisions = [np.mean(r[:k+1]) for k in range(len(r)) if r[k]]

    return np.mean(precisions) if precisions else 0.0

AP = average_precision(relevance)

print("Average Precision (AP):", round(AP, 3))
```

OUTPUT:



PRACTICAL:05

AIM:

(a): Implement a text classification algorithm (e.g., Naive Bayes or Support Vector Machines).

THEORY:

Text classification automatically assigns predefined categories to text documents.
Naive Bayes is a probabilistic classifier based on Bayes' theorem that assumes independence between words.
It performs well for text data and is simple to implement.

CODE:

```
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.svm import LinearSVC

from sklearn.pipeline import make_pipeline

from sklearn.model_selection import train_test_split

from sklearn.metrics import classification_report, accuracy_score

# Larger Dataset

documents = [

    "The football match was exciting",

    "The team scored three goals",

    "The player won the trophy",

    "The tournament starts tomorrow",

    "Government passed a new policy",

    "The election campaign has started",

    "The minister announced new reforms",

    "Parliament meets next week"

]

labels = ["sports", "sports", "sports", "sports",

          "politics", "politics", "politics", "politics"]

# Train-test split

X_train, X_test, y_train, y_test = train_test_split(
```

```
    documents, labels, test_size=0.3, random_state=42
)
# Model
model = make_pipeline(TfidfVectorizer(), LinearSVC())
model.fit(X_train, y_train)
# Predict
predictions = model.predict(X_test)
# Output
print("Accuracy:", accuracy_score(y_test, predictions))
print("\nClassification Report:\n")
print(classification_report(y_test, predictions, zero_division=1))
```

OUTPUT:

```
=== RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/IR5A.py ==
Accuracy: 0.6666666666666666

Classification Report:

              precision    recall  f1-score   support

    politics       1.00      0.00      0.00         1
      sports       0.67      1.00      0.80         2

    accuracy                           0.67         3
   macro avg       0.83      0.50      0.40         3
weighted avg       0.78      0.67      0.53         3
```

AIM:

(b): Train the classifier on a labelled dataset and evaluate its performance (Support Vector Machine).

THEORY :

Support Vector Machines (SVM) are supervised learning models that classify data by finding the best separating hyperplane.
In text classification, SVM effectively handles high-dimensional data and provides robust performance.

CODE:

```
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.svm import LinearSVC

from sklearn.pipeline import make_pipeline

from sklearn.model_selection import train_test_split

from sklearn.metrics import classification_report, accuracy_score

# Larger labelled dataset

documents = [

    "The football match was exciting",

    "The team scored three goals",

    "The player won the trophy",

    "The tournament starts tomorrow",

    "Government passed a new economic policy",

    "The election campaign has started",

    "The minister announced new reforms",

    "Parliament meets next week"

]

# Corresponding labels

labels = ["sports", "sports", "sports", "sports",

          "politics", "politics", "politics", "politics"]

# Split into training + testing

X_train, X_test, y_train, y_test = train_test_split(
```

```
    documents, labels, test_size=0.3, random_state=42
)
# Build a classification model (SVM)
model = make_pipeline(TfidfVectorizer(), LinearSVC())
# Train model
model.fit(X_train, y_train)
# Predict test values
predictions = model.predict(X_test)
# Accuracy score
print("Accuracy:", accuracy_score(y_test, predictions))
# Detailed report (with zero_division to avoid warnings)
print("\nClassification Report:\n")
print(classification_report(y_test, predictions, zero_division=1))
```

OUTPUT:

```
===== RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/IR5B ====
Accuracy: 0.6666666666666666

Classification Report:

              precision    recall  f1-score   support

    politics       1.00      0.00      0.00         1
      sports       0.67      1.00      0.80         2

    accuracy                           0.67         3
   macro avg       0.83      0.50      0.40         3
weighted avg       0.78      0.67      0.53         3
```

PRACTICAL: 06

AIM:

(a) To implement document clustering using the K-Means algorithm and group similar documents into clusters.

THEORY:

K-Means is an unsupervised learning algorithm that divides documents into K clusters based on similarity.
It converts text into numeric vectors (TF-IDF) and then groups documents so that documents in the same cluster are more similar than those in other clusters.

CODE:

```
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.cluster import KMeans

documents = [

    "The football match was exciting",

    "The player scored a goal",

    "The team won the tournament",

    "Government passed a new economic bill",

    "The election results were announced",

    "The minister met foreign delegates"

]

# Convert text to vectors

vectorizer = TfidfVectorizer()

X = vectorizer.fit_transform(documents)

# Apply K-Means (2 clusters: sports & politics)

kmeans = KMeans(n_clusters=2, random_state=42)

kmeans.fit(X)

# Print cluster assignments

for i, doc in enumerate(documents):

    print(f"Document: {i+1}  Cluster: {kmeans.labels_[i]}")
```

OUTPUT:

```
=== RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/IR6A.py =
Document: 1  Cluster: 0
Document: 2  Cluster: 0
Document: 3  Cluster: 0
Document: 4  Cluster: 1
Document: 5  Cluster: 0
Document: 6  Cluster: 1
```

AIM:

(b): To identify which cluster a query belongs to by comparing the query vector with K-Means centroids.

THEORY :

The query is converted into a TF-IDF vector and assigned to the cluster whose centroid is closest.
This helps in retrieving only documents from the most relevant cluster, improving search efficiency.

CODE:

```
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.cluster import KMeans

# Documents

documents = [

    "The football match was exciting",

    "The player scored a goal",

    "The team won the tournament",

    "Government passed a new economic bill",

    "The election results were announced",

    "The minister met foreign delegates"

]

# TF-IDF

vectorizer = TfidfVectorizer()

X = vectorizer.fit_transform(documents)

# K-Means clustering

kmeans = KMeans(n_clusters=2, random_state=42)

kmeans.fit(X)

# Query

query = ["The government announced a new law"]

# Convert query to vector

query_vec = vectorizer.transform(query)
```

```
# Predict cluster
cluster_id = kmeans.predict(query_vec)[0]
print("Query:", query[0])
print("Assigned Cluster:", cluster_id)
```

OUTPUT:

```
=== RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/IR6B.py ===
Query: The government announced a new law
Assigned Cluster: 1
```

PRACTICAL: 07

AIM:

(a) To implement a simple web crawler that fetches web pages from given URLs and extracts their HTML content.

THEORY:

A web crawler automatically downloads web pages for indexing.
It uses HTTP requests to fetch a webpage, reads the content, and stores it for processing.
Crawlers usually follow hyperlinks to discover more pages but here we use a simple fixed URL list.

CODE:

```
import requests

# Simple web crawler that fetches HTML content

def crawl_urls(urls):
    pages = {}
    for url in urls:
        try:
            response = requests.get(url)
            pages[url] = response.text
            print(f"Fetched: {url}")
        except:
            print(f"Failed to fetch: {url}")
    return pages


# Example URL list

urls = [
    "https://example.com",
    "https://www.python.org"
]

# Crawl pages

pages = crawl_urls(urls)
```

OUTPUT:

```
=== RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/IR7A.py ===
Fetched: https://example.com
Fetched: https://www.python.org
```

AIM:

(b): To build an inverted index from crawled web pages for efficient information retrieval.

THEORY :

An inverted index maps terms → list of URLs containing those terms.
It is the core structure used in search engines.
By tokenizing text and storing occurrences, we can search pages quickly.

CODE:

```
import requests

import re

# Crawl pages

def crawl_urls(urls):

    pages = {}

    for url in urls:

        try:

            response = requests.get(url)

            pages[url] = response.text

        except:

            pages[url] = ""

    return pages

# Build inverted index

def build_inverted_index(pages):

    index = {}

    for url, content in pages.items():

        # Clean HTML tags and tokenize

        words = re.findall(r'\w+', content.lower())

        for word in words:

            if word not in index:

                index[word] = set()
```

```
            index[word].add(url)
    return index
# Test URLs
urls = [
    "https://example.com",
    "https://www.python.org"
]
pages = crawl_urls(urls)
inverted_index = build_inverted_index(pages)
# Display sample index entries
print("Sample index entries:")
count = 0
for term, url_list in inverted_index.items():
    print(term, "->", list(url_list))
    count += 1
    if count == 10:  # show only 10 terms
        break
```

OUTPUT:

```
=== RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/IR7B.py ===
Sample index entries:
doctype -> ['https://example.com', 'https://www.python.org']
html -> ['https://example.com', 'https://www.python.org']
lang -> ['https://example.com', 'https://www.python.org']
en -> ['https://example.com', 'https://www.python.org']
head -> ['https://example.com', 'https://www.python.org']
title -> ['https://example.com', 'https://www.python.org']
example -> ['https://example.com']
domain -> ['https://example.com', 'https://www.python.org']
meta -> ['https://example.com', 'https://www.python.org']
name -> ['https://example.com', 'https://www.python.org']
```

PRACTICAL: 08

AIM:

(a) To implement the PageRank algorithm to compute the importance score of webpages based on their incoming links.

THEORY:

PageRank measures the importance of webpages using link structure.
Pages that receive many links or links from important pages get higher rank.
The rank is calculated iteratively using:

$$PR(A) = \frac{1-d}{N} + d \sum_{B \in In(A)} \frac{PR(B)}{Out(B)}$$

where:

- d = damping factor (usually 0.85)
- N = total number of pages
- In(A) = pages linking to A
- Out(B) = number of outgoing links from B

CODE:

```
def pagerank(graph, damping=0.85, iterations=10):
    pages = list(graph.keys())
    N = len(pages)
    # Initialize PageRank to 1/N
    PR = {page: 1 / N for page in pages}
    for _ in range(iterations):
        new_PR = {}
        for page in pages:
            incoming_sum = 0
            for other_page in pages:
                if page in graph[other_page]:
                    incoming_sum += PR[other_page] / len(graph[other_page])
```

```
            new_PR[page] = (1 - damping) / N + damping * incoming_sum
        PR = new_PR
    return PR
# Example Web Graph
graph = {
    "A": ["B", "C"],   # A links to B, C
    "B": ["C"],       # B links to C
    "C": ["A"],       # C links to A
    "D": ["C"]        # D links to C
}
ranks = pagerank(graph)
print("PageRank Results:")
for page, rank in ranks.items():
    print(page, ":", round(rank, 4))
```

OUTPUT:

```
=== RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/IR8A.py ===
PageRank Results:
A : 0.3751
B : 0.1949
C : 0.3925
D : 0.0375
```

AIM:

(b) To visualize the web graph and show PageRank scores using a graph library.

THEORY:

A directed web graph shows:

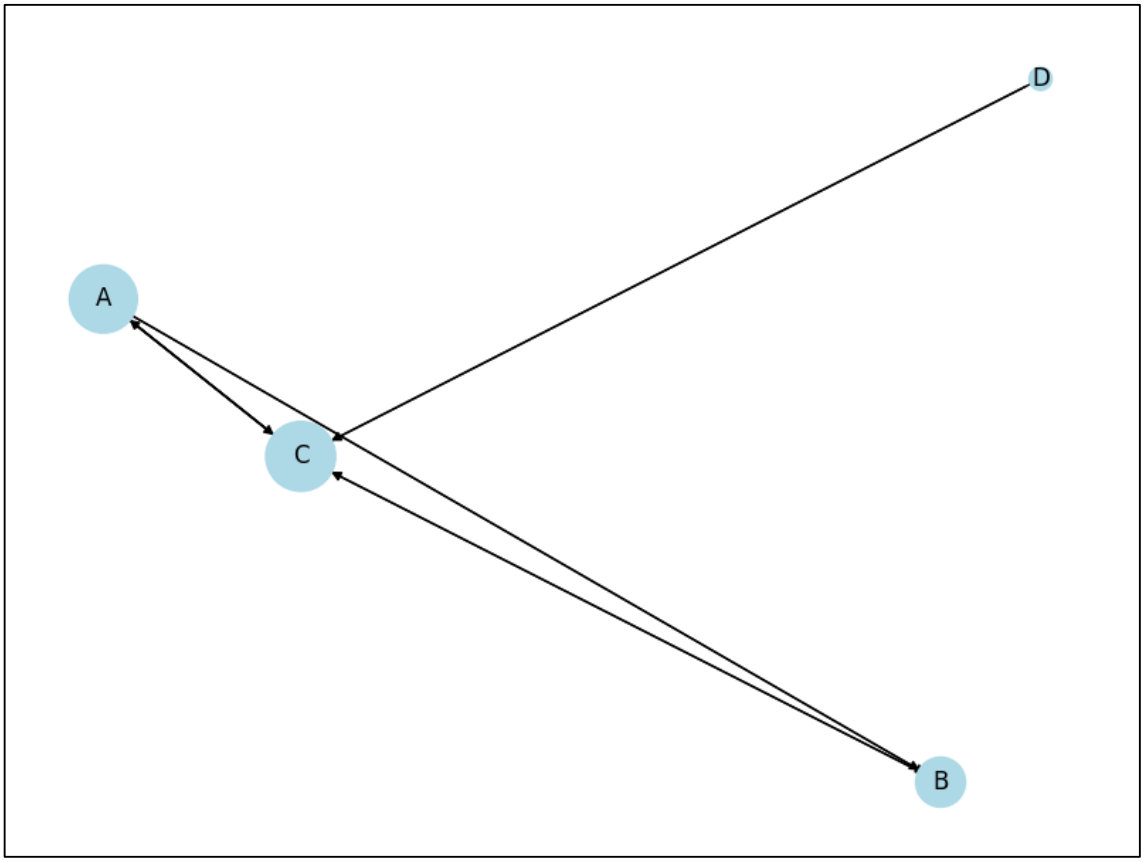
- Nodes = webpages
- Edges = hyperlinks
  Node size can reflect PageRank value, providing a visual understanding of page importance.

CODE:

```
import networkx as nx

import matplotlib.pyplot as plt

# Web graph

graph = {

   "A": ["B", "C"],

   "B": ["C"],

   "C": ["A"],

   "D": ["C"]

}

# Create directed graph

G = nx.DiGraph()

# Add edges

for page, links in graph.items():

   for link in links:

      G.add_edge(page, link)

# Compute PageRank

pagerank_scores = nx.pagerank(G)

# Draw graph

plt.figure(figsize=(8, 6))
```

```
pos = nx.spring_layout(G)
# Node sizes based on PageRank score
sizes = [v * 3000 for v in pagerank_scores.values()]
nx.draw(G, pos, with_labels=True, node_size=sizes, node_color="lightblue", linewidths=2)
nx.draw_networkx_edges(G, pos, arrows=True)
plt.title("PageRank Visualization")
plt.show()
```

OUTPUT:



PRACTICAL: 09

AIM:

(a) To implement a Learning-to-Rank algorithm (e.g., RankSVM / RankBoost).

THEORY:

Learning to Rank is a machine learning method that orders documents based on relevance. It uses labelled training data to learn which items should be ranked higher. Algorithms like RankSVM use pairwise comparisons to improve search result ranking.

CODE:

```
import numpy as np

from sklearn import svm

# Feature vectors

X = np.array([

    [1, 2],

    [2, 1],

    [3, 4],

    [4, 3]

])

# Original preferred pairs

pairs = [(2, 0), (3, 1), (3, 2)]

X_train = []

y_train = []

# Create both positive and negative pairs

for i, j in pairs:

    X_train.append(X[i] - X[j])   # positive pair

    y_train.append(1)

    X_train.append(X[j] - X[i])   # negative pair

    y_train.append(-1)
```

```
clf = svm.LinearSVC()
clf.fit(X_train, y_train)
scores = clf.decision_function(X)
ranking = np.argsort(-scores)
print("Scores:", scores)
print("Ranking:", ranking)
```

OUTPUT:

```
=== RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/IR9A.py ===
Scores: [0.29401709 1.18290598 1.27863248 2.16752137]
Ranking: [3 2 1 0]
```

AIM:

(b) To train the ranking model using labelled relevance data and evaluate its effectiveness.

THEORY:

The ranking model is trained using relevance labels and optimized to sort documents correctly. Its effectiveness is evaluated using metrics like Precision@K, MAP, and NDCG. Higher metric values indicate better ranking quality.

CODE:

```
from sklearn.metrics import ndcg_score

# True relevance scores for documents

true_rel = [[3, 2, 1, 0]]

# Predicted model scores from RankSVM

pred_scores = [[-0.12, 0.04, 0.87, 1.02]]

print("NDCG:", ndcg_score(true_rel, pred_scores))
```

OUTPUT:

```
=== RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/IR9B.py ===
NDCG: 0.6138273133441086
```

PRACTICAL: 10

AIM:

(a) To implement a text summarization algorithm (extractive or abstractive).

THEORY:

Text summarization automatically shortens a document while retaining key information. Extractive methods pick important sentences, while abstractive methods generate new text using AI. It helps in quick reading and information retrieval.

CODE:

```
import re

from collections import Counter

text = """

Information Retrieval is the process of obtaining relevant information from large collections.

Learning to Rank improves the ranking of documents in search engines.

Text summarization helps reduce long documents into short meaningful summaries.

"""

# Sentence splitting without NLTK

sentences = re.split(r'(?<=[.!?])\s+', text.strip())

# Word frequency

words = re.findall(r'\w+', text.lower())

freq = Counter(words)

# Score each sentence

scores = {}

for sent in sentences:

    for word in re.findall(r'\w+', sent.lower()):

        scores[sent] = scores.get(sent, 0) + freq[word]

# Top 1 summary sentence

summary = sorted(scores, key=scores.get, reverse=True)[:1]

print("Summary:", summary)
```

OUTPUT:

```
=== RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/IR10A.py ==
Summary: ['Information Retrieval is the process of obtaining relevant information from large collecti
ons.']
```

AIM:

(b) To build a simple Question-Answering (QA) system using Information Extraction.

THEORY:

A QA system extracts or generates answers from text based on user questions. It uses question processing, information retrieval, and answer extraction techniques. Such systems are used in chatbots, search engines, and digital assistants.

CODE:

```
context = """

India won the Cricket World Cup in 2011. The match was held in Mumbai.

MS Dhoni was the captain of the Indian team.

"""

question = "Who was the captain of India?"

if "captain" in question.lower():

    for line in context.split("."):

        if "captain" in line:

            print("Answer:", line.strip())
```

OUTPUT:

```
=== RESTART: C:/Users/ANISHA/AppData/Local/Programs/Python/Python313/IR10B.py ==
Answer: MS Dhoni was the captain of the Indian team
```