Practical No:1

import pandas as pd

from sklearn.feature\_extraction.text import CountVectorizer

corpus = [

    'this is the first document',

    'this document is second document',

    'and this is a third one',

    'is this first document?',

]

*# Create CountVectorizer object*

vectorizer = CountVectorizer()

*# Fit and transform the corpus into a word count matrix*

X = vectorizer.fit\_transform(corpus)

*# Convert the result to a DataFrame*

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

*# Output the DataFrame*

print("Generated DataFrame:")

print(df)

*# Find indices where both 'this' and 'first' are present*

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

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

*# Find indices where either 'this' or 'first' is present*

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

print("Indices where either 'this' or 'first' terms are present:", ordata.index.tolist())

*# Find indices where 'and' term is present*

notdata = df[df['and'] == 1]

print("Indices where 'and' term is present:", notdata.index.tolist())

**pandas**: This library is used for data manipulation and analysis

**CountVectorizer**: This is a class from **sklearn.feature\_extraction.text** that converts a collection of text documents into a matrix of token counts. It tokenizes the text and counts the occurrences of each word in the corpus.

**corpus**: This is a list of text documents (strings) that will be processed. In this case, there are four sentences in the corpus.

**fit\_transform(corpus)**: This method first learns the vocabulary (terms) in the corpus (i.e., the set of unique words) and then transforms the text into a matrix of word counts (also known as a document-term matrix or DTM).

 **Matrix** is more mathematical and is limited to numerical operations.

 **DataFrame** is more flexible, used in data analysis, and supports multiple data types.

A **sparse matrix** is a matrix where most of the elements are zero, stored efficiently by only keeping track of non-zero values and their positions.

**vectorizer.get\_feature\_names\_out()**: Retrieves the feature names (terms or words) that were identified during vectorization. These are used as the column names in the DataFrame.

 **df['this'] == 1**: This checks where the term **'this'** appears in the document (a value of 1 indicates presence).

 **df['first'] == 1**: Similarly, this checks where **'first'** appears.

 The **&** operator is used to find the intersection of both conditions, meaning we are looking for documents where both **'this'** and **'first'** are present. The result is stored in **alldata**.

 **alldata.index.tolist()**: This returns the indices (document numbers) where both terms are present.

Imagine you are building a **library catalog** system. You need to find books based on keywords like **"this"** or **"first"**. If a user searches for books containing both of these words, you can quickly identify those books using the code. Similarly, if the user wants books with either word, the code helps find those too, speeding up the search process.

Practical No:2

# NLTK (Natural Language Toolkit)

import nltk

nltk.download('punkt\_tab')

import nltk

from nltk import word\_tokenize

from nltk.util import ngrams

text="This is a text for unigrams,bigrams,and trigrams extraction using NLTK"

tokens=word\_tokenize(text.lower())

unigrams=list(ngrams(tokens,1))

bigrams=list(ngrams(tokens,2))

trigram=list(ngrams(tokens,3))

print("Original Text:",text)

print("\nUnigrams:",unigrams)

print("\Bigrams:",bigrams)

print("\Trigram:",trigram)

 **word\_tokenize**: This function is used to split the input text into individual words. It handles punctuation, contractions, and other complexities of English text.

 **ngrams**: This function is used to generate n-grams (sequences of 'n' contiguous words) from the tokenized text. It can generate unigrams (1 word), bigrams (2 words), trigrams (3 words), etc.

**Purpose of the Code**

The purpose of this code is to demonstrate how to extract **unigrams**, **bigrams**, and **trigrams** from a given text using NLTK in Python. These n-grams are essential for various natural language processing (NLP) tasks like:

* **Text analysis**
* **Machine learning models** for text classification
* **Language modeling** (predicting the next word in a sequence)
* **Information retrieval systems** (e.g., search engines)
* **Recommendation systems** (based on text input)

**Real-life Scenario**

Let's consider a real-life scenario where this code could be useful:

**Scenario: Sentiment Analysis for Reviews**

Suppose you have a collection of product reviews, and you want to analyze the sentiment of these reviews. Using **n-grams** can help you better capture the context of words in a sentence rather than just analyzing individual words.

For instance, in the sentence:

* **"The product is amazing and works perfectly."**
* The **unigrams** might include: ['the', 'product', 'is', 'amazing', 'and', 'works', 'perfectly']
* The **bigrams** could be: [('the', 'product'), ('product', 'is'), ('is', 'amazing'), ('amazing', 'and'), ('and', 'works'), ('works', 'perfectly')]
* The **trigrams** could be: [('the', 'product', 'is'), ('product', 'is', 'amazing'), ('is', 'amazing', 'and'), ('amazing', 'and', 'works'), ('and', 'works', 'perfectly')]

In sentiment analysis, these n-grams (especially bigrams and trigrams) can help capture important context, like "amazing product" or "works perfectly," which can be significant in determining that the sentiment of the review is positive.

**Scenario: Language Modeling**

In language modeling, n-grams are used to predict the next word in a sentence. For example:

* In the phrase "I love to play \_\_\_", bigrams and trigrams can help predict that the next word could be "football" or "guitar," based on common patterns in the text data.

By using **unigrams**, **bigrams**, and **trigrams**, you can improve the accuracy of models that require context understanding, such as:

* Chatbots
* Predictive text systems (like in mobile phones)
* Automated content generation systems

Practical No:3

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

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

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

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

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

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

print("Precison:",precision)

print("Recall:",recall)

print("F1 score",f1)

These metrics give you insights into how well your model is doing in terms of correctly identifying positive and negative cases.

**Ground Truth and Predicted Values:**

* ground\_truth = [1, 0, 1, 0, 1, 1, 0, 0, 1, 1]
  + These are the actual labels for the data. 1 represents a positive class, and 0 represents a negative class.

### Detailed Breakdown:

* **True Positives (TP):** These are the instances where both ground truth and prediction are 1.
  + Here, they are the values at indices 0, 2, 5, 8.
  + TP = 4.
* **False Positives (FP):** These are the instances where ground truth is 0, but the model predicts 1.
  + Here, they are the values at indices 1, 7.
  + FP = 2.
* **False Negatives (FN):** These are the instances where ground truth is 1, but the model predicts 0.
  + Here, they are the values at indices 3, 4.
  + FN = 2.
* **True Negatives (TN):** These are the instances where both ground truth and prediction are 0.
  + Here, they are the values at indices 6, 9.
  + TN = 2.

Using the formulas above:

* **Precision = 4 / (4 + 2) = 0.6**
* **Recall = 4 / (4 + 2) = 0.6**
* **F1 Score = 2 \* (0.6 \* 0.6) / (0.6 + 0.6) = 0.6**