**Machine Learning Projects**

**End To End Movie Recommender System**

### **Overview:**

* Diving into the world of machine learning, I created my first machine learning project.
* This project helped me explore practical concepts like:
  + **Vectorization techniques**
  + **Cosine distance**
  + **Similarity measurements**
* This journey has been **very rewarding**, and I **enjoyed it a lot** while working on the project.
* Below, I provide a **detailed explanation** of the project.
* I also discuss the **challenges** I faced and how I **overcame** them.

**Project Explanation Step By Step:**

**Importing Libraries:**

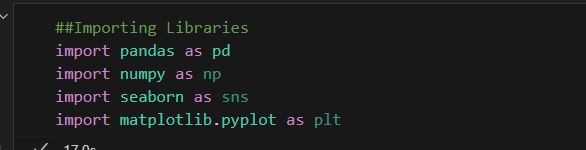
In this project i used few python libraries which are listed below

.Pandas(For understanding and analyzing the data)

.Numpy(It deals with numerical data)

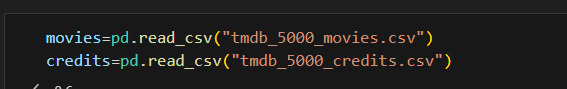
.Matplotlib(It is used for graphical representation,visualization)

.Seborn(it is alsoused for visualize the data)



### **Explanation and Loading the Dataset:**

* I fetched the dataset from **Kaggle**.
* The dataset contains **5,000 movies**.
* Our goal is to find **similar movies** based on their **context and features**.



### **Exploratory Data Analysis (EDA)**

* Earlier, we imported the necessary **libraries** and **loaded the dataset**.
* Now, we focus on **cleaning the data**, which involves:
  + **Removing irrelevant information**
  + **Handling missing values**
  + **Making data well-structured and presentable**
* The purpose of **data cleaning** is to prepare the dataset for **machine learning algorithms**, ensuring **better results** and **valuable insights**.

#### **Importance of Data Cleaning in Data Science**

* As a **Data Scientist**, a significant portion of time is spent on **data cleaning and preprocessing**.
* **80% of the time** is spent on **EDA and data cleaning**, while only **20% is spent on applying machine learning algorithms**.

### **Step-by-Step EDA Analysis**

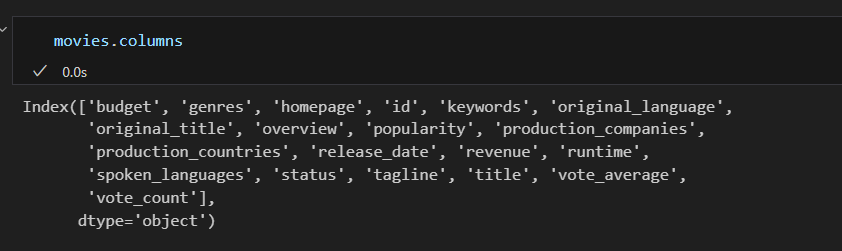
#### **Fundamental Steps for EDA:**

1. **Store data in a variable** → df = pd.read\_csv('file.csv')
2. **Check the shape of the data** → df.shape
3. **Check the information of the data** → df.info()
4. **Describe the dataset** → df.describe()
5. **Check for missing values** → df.isnull().sum()
6. **Find duplicate values** → df.duplicated().sum()

### **Dataset Overview:**

* Our dataset consists of **two CSV files**:
  + **movies.csv**
  + **credits.csv**

**.Find the columns in moies.csv**

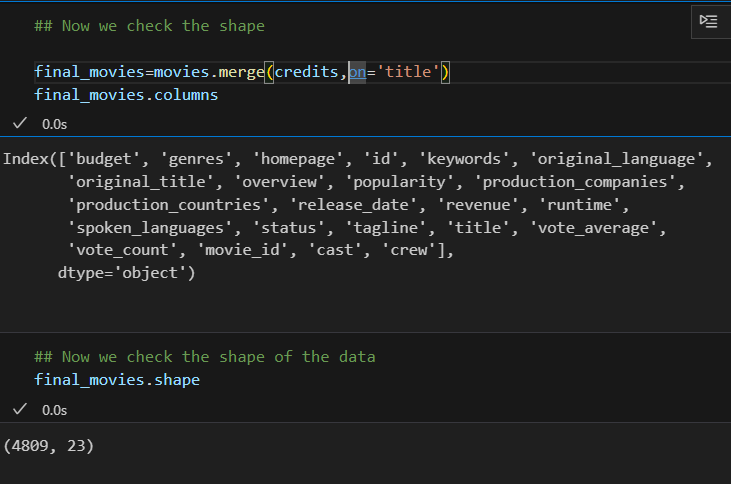
**.Find the columns in credits.csv**

### **Merging Both DataFrames**

* To simplify our dataset, we **merge both datasets** into a **single DataFrame**.
* We merge **credits.csv** with **movies.csv** based on the **title** column.
* Once merged, we can remove the **second DataFrame (credits)** as it is no longer needed.

### **. Final DataFrame**

* After merging both **DataFrames**, we store the result in **final\_df**, which contains all columns from both datasets.
* We then check the **shape** of the merged DataFrame to confirm the number of rows and columns.



### **Basic EDA Analysis**

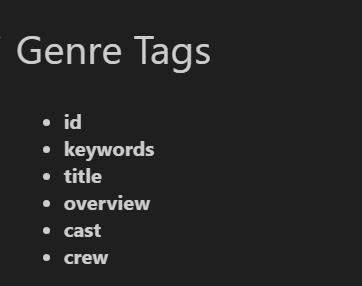
* In this step, we **merged both DataFrames** to create a structured dataset.
* The main goal of this project is to build a **Movie Recommender System**.

### **Key Requirements for the Recommender System**

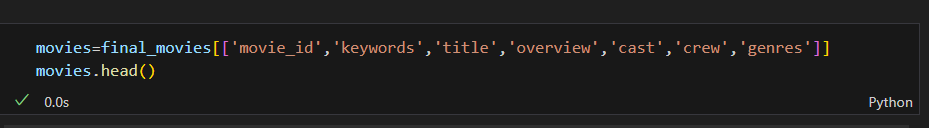
* To build the system, we need to identify the **most important columns** that define a movie.
* These columns will be used to determine **similarity** between movies.
* This process is known as **Genre Tagging**, where we extract and utilize key attributes like:
  + **Genres**
  + **Keywords**
  + **Cast**
  + **Director**

**Genre Tags:**

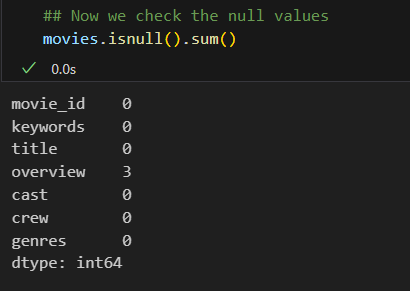
Basicaly these are the columns which are funadmental for any movies



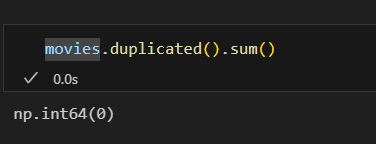
**Creating a new DataFrame in which we stored theses columns of genre tags :**

We stored these columns in movies variable.And this variable is based on these columns.

**Check the Null values in movies variable:**



**Check the duplicate values:**

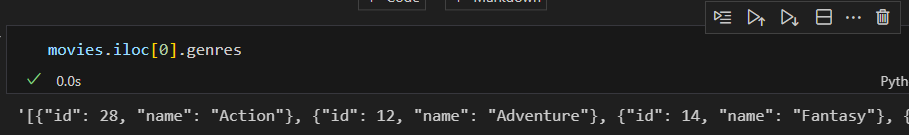


### **. Using iloc Function**

* The iloc function in **Pandas** is used for **index-based selection** of rows and columns.
* It helps in extracting specific parts of a DataFrame using numerical indices.

### **Why Use iloc?**

* It allows us to access data based on **integer positions** rather than labels.
* Useful when working with **structured datasets** where column names may not be needed.



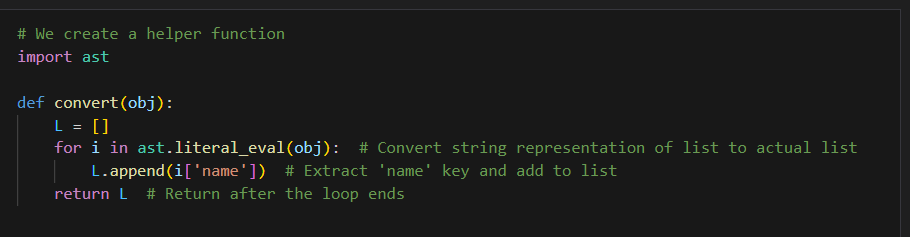
Now we convert above string representation into list like this

#we convert this [{"id": 28, "name": "Action"}, {"id": 12, "name": "Adventure"}, {"id": 14, "name": "Fantasy"}, {"id": 878, "name": "Science Fiction"}]

# into ['Action',;Adventure,'Fantasy','ScienceFiction']

**Process to Convert String Representation into Actual List of Genre**

1. We created a helper function this function or code snippet helps usto to convert string representation into actual list



Now we discuss this code snippet in detail

### **Step-by-Step Explanation:**

#### **Step 1: Initialize an empty list**

L = []

* Creates an empty list L to store the extracted values from obj.

#### **Step 2: Convert the string representation of a list to an actual list**

for i in ast.literal\_eval(obj):

* The function takes obj as input.
* obj is expected to be a **string representation of a list** (like '[{"id": 28, "name": "Action"}, {"id": 12, "name": "Adventure"}]').
* ast.literal\_eval(obj) converts this string into an **actual Python list of dictionaries**.
* Then, we iterate (for i in ...) over each dictionary in the list.

#### **Step 3: Extract the 'name' value from each dictionary**

L.append(i['name'])

* i represents each dictionary inside the list.
* i['name'] extracts the value of the "name" key from the dictionary.
* This value is then **appended** to the list L.

📌 **Example Input & Iteration:**

obj = '[{"id": 28, "name": "Action"}, {"id": 12, "name": "Adventure"}]'

Converted list:

[{"id": 28, "name": "Action"}, {"id": 12, "name": "Adventure"}]

**Iteration 1:**

* i = {"id": 28, "name": "Action"}
* i['name'] = "Action" → Added to L

**Iteration 2:**

* i = {"id": 12, "name": "Adventure"}
* i['name'] = "Adventure" → Added to L

#### **Step 4: Return the final list**

return L

* Once all "name" values are extracted and added to L, the function returns L.

### **Example Usage:**

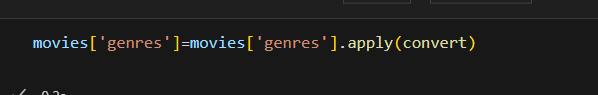
obj = '[{"id": 28, "name": "Action"}, {"id": 12, "name": "Adventure"}]'  
print(convert(obj))

### **Output:**

['Action', 'Adventure']

### **This is just a explanation of this code.**

After this code execution we write this line of the code

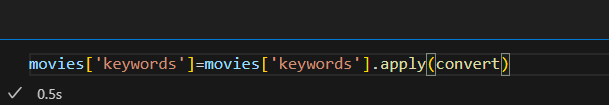


By executing this line of code snippet we are able to and also our genre tage column is completely converted into list

. This is all about our genre column .

**Keyword Column:**

Similalry also we convert this keyword column by running this line of code



This keywords column is converted into list

**Now we deal with Cast Column:**

### **Understanding print(movies['cast'][0])**

This code retrieves and prints the first value from the **"cast"** column in the movies DataFrame. Let's break it down step by step.

### **Step 1: Understanding movies['cast']**

movies['cast']

* This selects the **"cast"** column from the movies DataFrame.
* The result is a **pandas Series**, where each row contains a value corresponding to the "cast" of that movie.

Example:

makefile

0 '[{"id": 101, "name": "Leonardo DiCaprio"}, {"id": 102, "name": "Brad Pitt"}]'  
1 '[{"id": 201, "name": "Tom Hanks"}, {"id": 202, "name": "Morgan Freeman"}]'  
2 '[{"id": 301, "name": "Robert Downey Jr."}, {"id": 302, "name": "Chris Evans"}]'  
Name: cast, dtype: object

* Notice that the values in the "cast" column are stored as **string representations of lists of dictionaries**.

### **Step 2: Accessing the First Row ([0])**

movies['cast'][0]

* movies['cast'] is a Series, and [0] selects the first value (first row).
* The first value is likely a **string** that represents a list of dictionaries.

Example Output:

'[{"id": 101, "name": "Leonardo DiCaprio"}, {"id": 102, "name": "Brad Pitt"}]'

* This is a **string**, not a real Python list yet.

### **Step 3: If You Want to Convert It to a List**

Since the output is a string representation of a list, you need to **convert it to a real list** using ast.literal\_eval() or json.loads():

#### **Using ast.literal\_eval()**

import ast  
cast\_list = ast.literal\_eval(movies['cast'][0])  
print(cast\_list)

**Output:**

[{"id": 101, "name": "Leonardo DiCaprio"}, {"id": 102, "name": "Brad Pitt"}]

Now cast\_list is a **real list of dictionaries**.

#### **Extracting Names from the Cast**

actor\_names = [actor['name'] for actor in cast\_list]  
print(actor\_names)

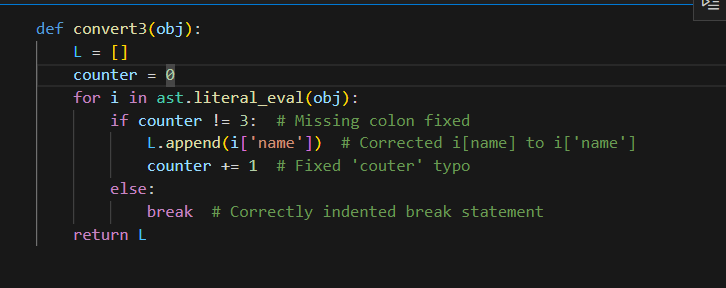
**Output:**

['Leonardo DiCaprio', 'Brad Pitt']

### **Final Summary**

|  |  |
| --- | --- |
| **Code** | **Explanation** |
| movies['cast'] | Selects the "cast" column (a Series). |
| movies['cast'][0] | Selects the first row from the "cast" column (a string). |
| ast.literal\_eval(movies['cast'][0]) | Converts the string to an actual Python list. |
| [actor['name'] for actor in cast\_list] | Extracts actor names from the list of dictionaries. |

## **Step-by-Step Explanation:**



### **Step 1: Initialize an empty list**

L = []

* Creates an empty list L to store extracted names.

### **Step 2: Initialize a counter**

counter = 0

* This variable keeps track of how many names have been added to the list.
* The function will stop once **3 names** have been added.

### **Step 3: Convert string to list & iterate over it**

for i in ast.literal\_eval(obj):

* obj is expected to be a **string representation of a list** (like '[{"id": 28, "name": "Action"}, {"id": 12, "name": "Adventure"}]').
* ast.literal\_eval(obj) **converts** this string into an actual **Python list of dictionaries**.
* The for loop iterates over each dictionary in the list.

### **Step 4: Check if counter is less than 3**

if counter != 3:

* Ensures that **only the first 3 names** are added to the list.
* If 3 names have already been added, the loop stops.

### **Step 5: Extract 'name' and append to the list**

L.append(i['name'])

* i is a dictionary from the list.
* i['name'] extracts the **value of the "name" key**.
* This extracted name is added to L.

### **Step 6: Increment the counter**

counter += 1

* After adding a name, the counter increases by **1**.

### **Step 7: Stop the loop after adding 3 elements**

else:  
 break

* If counter == 3, the loop **breaks** immediately, stopping further iterations.

### **Step 8: Return the final list**

return L

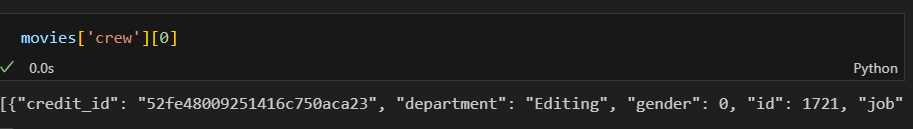
* Once the loop stops, the function returns the final list L containing **up to 3 names**.

After this we run this line of code

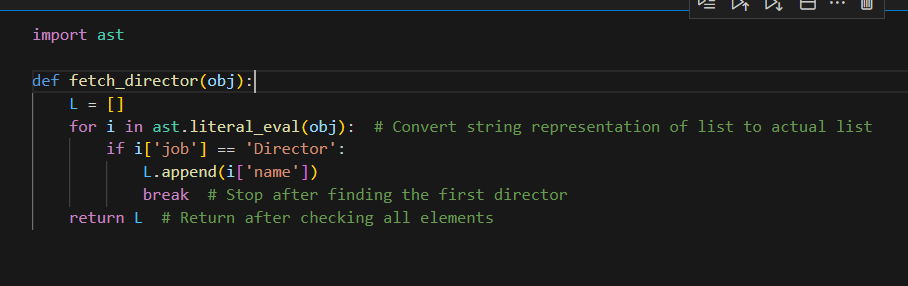


By running this line of command our cast column is easily ready .

**Now we deal or make a better understandable of CREW Column**



Above crew line of code is very messy data now we solve this code .



### **Code Breakdown:**

import ast # Step 1: Import the 'ast' module

* The **ast** (Abstract Syntax Tree) module is used to safely **convert a string representation** of a list (like '[{"job": "Director", "name": "Nolan"}]') into an **actual Python list**.
* **Why use ast.literal\_eval()?**
  + It safely evaluates a string containing a Python literal (like a list or dictionary).
  + It avoids the security risks of using **eval()**.

def fetch\_director(obj): # Step 2: Define a function named 'fetch\_director' that takes 'obj' as input

* This function is designed to extract the **first director's name** from obj, which is expected to be a **string representation of a list of dictionaries**.

t

L = [] # Step 3: Initialize an empty list to store the director's name

* Creates an **empty list L** to store the **director's name**.
* If no director is found, the function will return an **empty list**.

python

CopyEdit

for i in ast.literal\_eval(obj): # Step 4: Convert 'obj' (string) to a list and iterate over it

* **ast.literal\_eval(obj)** converts the **string** into an **actual Python list of dictionaries**.
* The for loop iterates through **each dictionary** in this list.

if i['job'] == 'Director': # Step 5: Check if the 'job' field is 'Director'

* **Each i** represents a dictionary inside the list.
* This condition checks if **the dictionary has 'job': 'Director'**.

L.append(i['name']) # Step 6: If 'job' is 'Director', add the 'name' value to the list 'L'

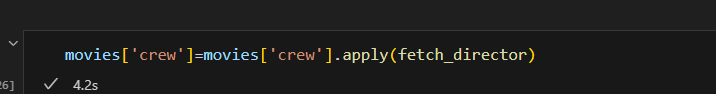
* **Extracts the name** from the dictionary (i['name']) and **adds it to the list L**.

break # Step 7: Stop the loop after finding the first director

* **Stops the loop immediately** after adding the **first** director's name.
* Prevents the function from adding multiple director names.

return L # Step 8: Return the final list containing the director's name

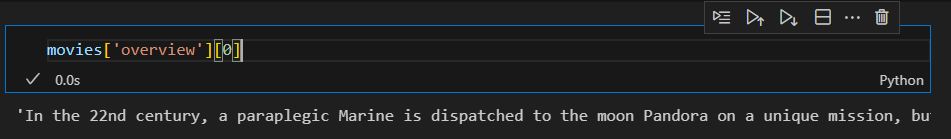
* Returns **list L**, which contains the **first director's name** (or an empty list if no director is found).

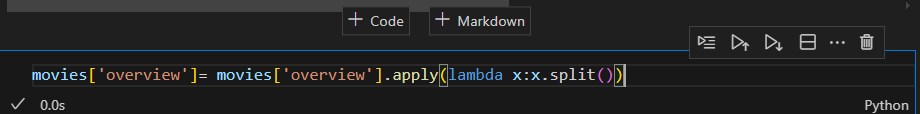
 By runing this code snippet we are able to see and make understandable crew column.

Mostly our columns are clean and gives understandable meanings.

**Now We Check Or clean the Overview Column:**

Basicaly the overview column is textual data so in order to solve this we use some different approaches . First we simply check overview column



 **Step by Step Explanation of this code :**

**1)movies[‘overview’]**

This peice of the code help us to find the overview column in movies dataframe and also this column is based on textual data.

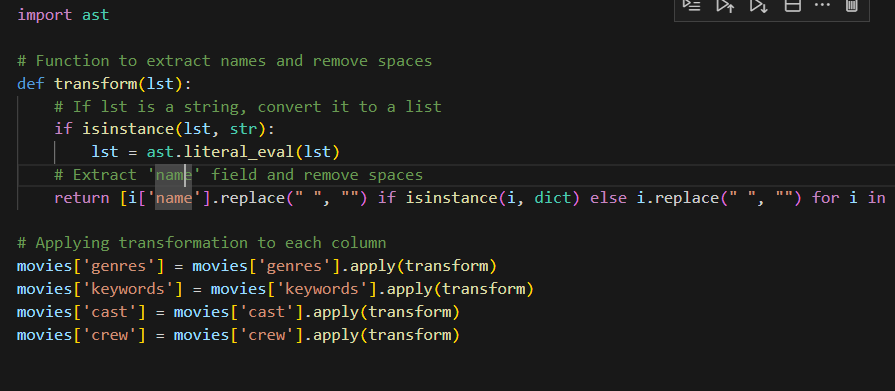
**2).apply(lambda x:x.split())**

.apply() is a pandas function that applies a given function to each value

lambda x: x.split() is an **anonymous function** (a function without a name) that:

* Takes **x** (which is each row value from the **"overview"** column).
* Uses .split() to **split** the string into a list of word

**Converting String into List:**

Explanation of this code snippet in detail:

## **. Importing the ast Module**

python

CopyEdit

import ast

* ast stands for **Abstract Syntax Tree**.
* ast.literal\_eval() is used to safely evaluate a **string representation of a Python data structure** (like a list, dictionary, or tuple) and convert it back into a real Python object.
* This is useful when your dataset contains list-like strings (e.g., "[{'name': 'Action'}, {'name': 'Adventure'}]"), and you need to convert them into actual lists.

## **2. Defining the transform Function**

python

CopyEdit

def transform(lst):  
 # If lst is a string, convert it to a list  
 if isinstance(lst, str):  
 lst = ast.literal\_eval(lst)  
 # Extract 'name' field and remove spaces  
 return [i['name'].replace(" ", "") if isinstance(i, dict) else i.replace(" ", "") for i in lst]

### **Step-by-Step Explanation:**

1. **Checking if lst is a string:**

python

CopyEdit

if isinstance(lst, str):  
 lst = ast.literal\_eval(lst)

* 1. If lst is a **string representation of a list**, it is converted into an **actual list** using ast.literal\_eval().
  2. This is important because data from CSV or JSON files often store lists as strings.

1. **Extracting the 'name' field and removing spaces:**

python

CopyEdit

return [i['name'].replace(" ", "") if isinstance(i, dict) else i.replace(" ", "") for i in lst]

* 1. This is a **list comprehension** that iterates through the list (lst).
  2. It checks if each item (i) is a **dictionary** (which happens in JSON-like data).
     1. If **i is a dictionary**, it extracts the 'name' value (i['name']) and removes spaces using .replace(" ", "").
     2. Otherwise, it removes spaces from the string directly (i.replace(" ", "")).

## **3. Applying the Function to Movie Data**

python

CopyEdit

movies['genres'] = movies['genres'].apply(transform)  
movies['keywords'] = movies['keywords'].apply(transform)  
movies['cast'] = movies['cast'].apply(transform)  
movies['crew'] = movies['crew'].apply(transform)

* The **apply()** function applies the transform function to each row of the respective columns (genres, keywords, cast, crew).
* These columns originally contain **list-like string values**, and this transformation:
  + Converts them into **real lists**.
  + Extracts the 'name' field from dictionaries.
  + Removes spaces from all words.

## **Example: Before and After Transformation**

### **Before Transformation**

|  |  |
| --- | --- |
| **genres (as string)** | **keywords (as string)** |
| "[{'name': 'Action'}, {'name': 'Adventure'}]" | "[{'name': 'hero'}, {'name': 'fight'}]" |

### **After Transformation**

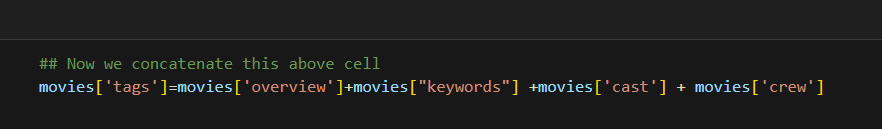
|  |  |
| --- | --- |
| **genres (as list)** | **keywords (as list)** |
| ['Action', 'Adventure'] | ['hero', 'fight'] |

## **Why This is Useful?**

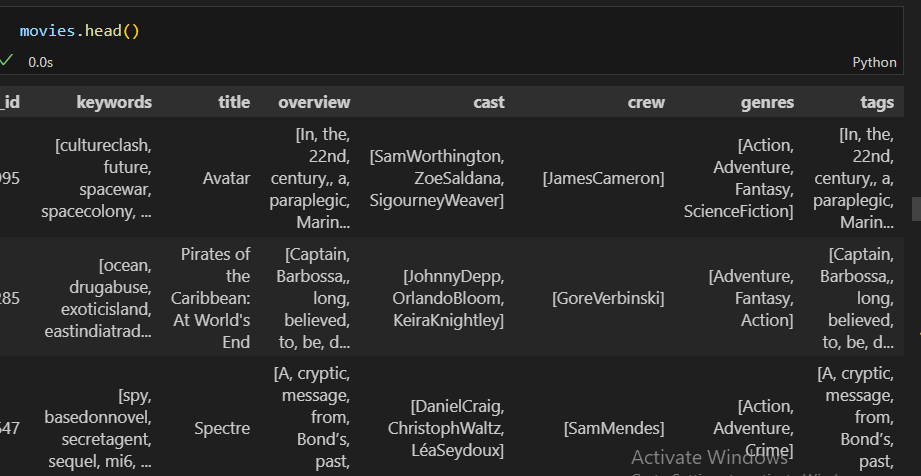
* **Data Cleaning:** Converts messy **stringified lists** into real Python lists.
* **Feature Engineering for NLP & ML:** Helps in text-based ML tasks like:
  + **Movie Recommendation Systems** (e.g., finding similar movies based on genres, keywords, or cast).
  + **Text Vectorization** (e.g., converting words into numerical vectors for ML models).
  + **Search & Filtering** (e.g., searching for movies by a genre or cast member).

After these transformations Now we are able to cocatenate all column

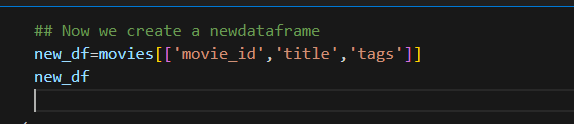
**Cocatination of the string:**

In above code snippet we create a new variable which is movies[’tags’] which is consist of 4 columns. The purpose of creating this new variable is to reduce unnecessary information .

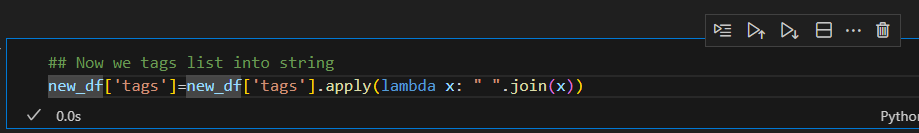
After creating this we have an other column in main variable which is movies dataframe so so when we merge it then it is easy for us to remove irrelevant information.

 After creating this we create a new dataframe which is easy for us and also remove those separate columns because thses columns we already stored in tags dataframe.

**New DataFrame:**



**Converted Tags List into String:**

**Step By Step explanation of this code:**

#### **1. Accessing the 'tags' Column**

python

CopyEdit

new\_df['tags']

* This selects the **'tags'** column from the new\_df DataFrame.
* Before this transformation, each row in the 'tags' column is a **list of words**.

#### **2. Applying the Transformation Using apply()**

python

CopyEdit

new\_df['tags'] = new\_df['tags'].apply(lambda x: " ".join(x))

* The **apply()** function is used to apply a function to each row in the column.
* The function inside apply() is a **lambda function**:

python

CopyEdit

lambda x: " ".join(x)

* + **x** represents each row (which is a **list of words**).
  + " ".join(x) **joins the list elements into a single string** with spaces between them.

### **Example to Understand the Transformation**

#### **Before Applying the Code**

|  |  |
| --- | --- |
| **Movie Title** | **tags (as a list)** |
| Inception | ['Action', 'SciFi', 'Thriller'] |
| Titanic | ['Romance', 'Drama', 'Ship'] |

#### **After Applying the Code**

|  |  |
| --- | --- |
| **Movie Title** | **tags (as a string)** |
| Inception | "Action SciFi Thriller" |
| Titanic | "Romance Drama Ship" |

### Our New DataFrame:**Step 1: Accessing the Column**

python

CopyEdit

new\_df['tags']

* This selects the **'tags'** column from the new\_df DataFrame.
* Each row in this column **currently contains a list of words** (e.g., ['Action', 'SciFi', 'Thriller']).

### **Step 2: Applying the Transformation**

python

CopyEdit

new\_df['tags'].apply(lambda x: " ".join(x))

* **.apply()** applies a function to **each row** in the column.
* The function used inside apply() is a **lambda function**:

python

CopyEdit

lambda x: " ".join(x)

* + Here, x represents each row (which is a **list of words**).
  + " ".join(x) converts the list into a **single string** by **joining words with spaces**.

### **Step 3: Updating the Column**

python

CopyEdit

new\_df['tags'] = new\_df['tags'].apply(lambda x: " ".join(x))

* The transformed data (now a string) replaces the existing 'tags' column.

## **Example: Before and After Transformation**

### **Before Applying the Code**

|  |  |
| --- | --- |
| **Movie Title** | **tags (as a list)** |
| Inception | ['Action', 'SciFi', 'Thriller'] |
| Titanic | ['Romance', 'Drama', 'Ship'] |

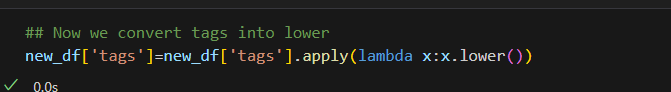
### **After Applying the Code**

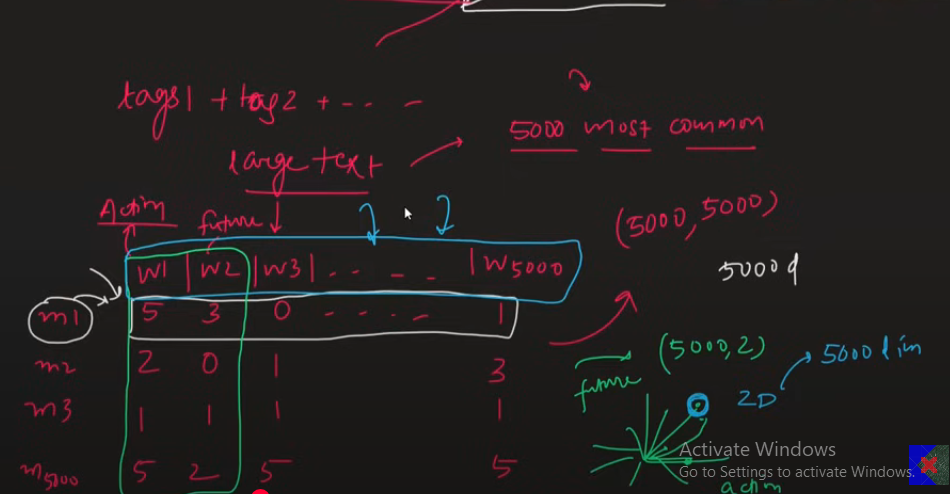
|  |  |
| --- | --- |
| **Movie Title** | **tags (as a string)** |
| Inception | "Action SciFi Thriller" |
| Titanic | "Romance Drama Ship" |

## **Why is This Useful?**

1. **Prepares Data for Natural Language Processing (NLP) & Machine Learning**
   1. Many ML models and NLP techniques (like **TF-IDF, CountVectorizer**) require text **as a single string**, not a list.
2. **Makes Text Similarity Calculations Easier**
   1. Useful for **content-based movie recommendations** where we compute similarity between movies based on their **tags**.
3. **Better Data Representation for Vectorization**
   1. This transformation helps in converting the text into numerical vectors, which can then be used in ML models.

**Converted Into Lower alphabets:**





### **Why We Use Vectorization in This Project?**

* The main goal of this project is to find movies that are **similar** based on their content.
* We achieve this by analyzing **tags**, which consist of important textual data for each movie.
* To measure similarity, we need to convert textual data into a numerical format—this is where **vectorization** comes in.

### **What is Vectorization?**

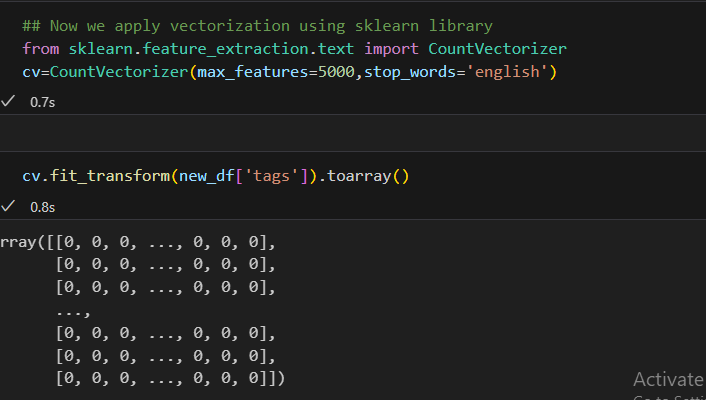
* **Vectorization** is a technique used to convert textual data into numerical vectors.
* In this project, since our **tags contain textual data**, we apply vectorization to compute the similarity between movies.
* Various vectorization techniques exist, but we use one of the most popular methods: **Bag of Words (BoW)**.

### **Bag of Words (BoW) Method**

* **Bag of Words (BoW)** is a widely used vectorization technique in NLP.
* It converts each movie's textual data (tags) into a numerical vector.
* Given that we have **5,000 movies**, each movie is represented as a **vector** in a high-dimensional space.
* By converting movies into vectors, we can calculate the **similarity** between them.
* After concatenating all tags, we obtain a large set of unique words.
* Instead of just counting the most frequent words, **BoW calculates the distance between movie vectors** to determine which movies are **most similar**.

**Not Using Stop Words:**

When performing vectorization, we exclude stop words. Stop words are common words (such as "and," "the," "is") that do not contribute significant meaning to the overall content of a sentence. In vectorization, these words are ignored to focus on the more meaningful words. We don’t need to manually remove them because libraries like scikit-learn already have built-in functionality to handle stop words automatically.



**Step By Step Explanation of this Code:**

### **Step 1: Importing Required Library**

python

CopyEdit

from sklearn.feature\_extraction.text import CountVectorizer

* This imports the CountVectorizer class from sklearn.feature\_extraction.text, which is used to convert text data into a matrix of token counts.

### **Step 2: Creating an Instance of CountVectorizer**

cv = CountVectorizer(max\_features=5000, stop\_words='english')

* CountVectorizer() initializes an instance for text vectorization.
* max\_features=5000 limits the number of unique words considered to the top 5000 most frequent words in the dataset.
* stop\_words='english' removes common English stop words (like "the", "is", "and") that do not add much meaning to the text.

### **Step 3: Fitting and Transforming the Text Data**

python

CopyEdit

cv.fit\_transform(new\_df['tags']).toarray()

* .fit\_transform(new\_df['tags']):
  + fit\_transform() learns the vocabulary from the text data (new\_df['tags'] column) and converts it into a sparse matrix where each row represents a document (movie in this case) and each column represents a word from the vocabulary.
* .toarray():
  + Converts the sparse matrix into a dense NumPy array for better visualization.

### **Step 4: Understanding the Output**

* The output is a 2D NumPy array where:
  + Each row represents a movie (or document).
  + Each column represents a word from the vocabulary.
  + A value of 1 or higher in a column means the corresponding word appears in that movie's tags column.
  + 0 means the word is absent.

**What is Sparse Matrix?**

A **sparse matrix** is a matrix in which most of the elements are **zero**. It is the opposite of a **dense matrix**, where most elements have nonzero values. Sparse matrices are commonly used in machine learning, data science, and natural language processing (NLP) to efficiently store and process data.

### **Why Use a Sparse Matrix?**

When dealing with text data (like in your movie recommender system), we convert words into numerical representations using vectorization techniques like **CountVectorizer** or **TF-IDF**. However, not every document (or movie tag) will contain every word from the vocabulary, leading to many zeros in the representation. A sparse matrix helps:

* **Save memory** by storing only nonzero values.
* **Improve computational efficiency** by avoiding unnecessary calculations on zero values.

### **Example of a Sparse Matrix**

Consider the following text corpus:

Movie 1: "action hero adventure"  
Movie 2: "romantic love story"  
Movie 3: "sci-fi adventure space"

After applying **CountVectorizer**, the document-term matrix might look like this:

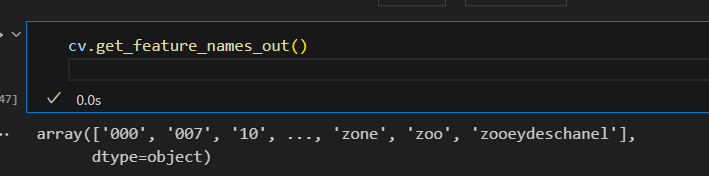
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **action** | **adventure** | **hero** | **love** | **romantic** | **sci-fi** | **space** | **story** |
| Movie 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| Movie 2 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| Movie 3 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |

Since most of the entries are 0, storing the entire matrix as a dense array wastes memory. Instead, a **sparse matrix** stores only the nonzero values along with their positions.

### **Types of Sparse Matrix Representations**

1. **COO (Coordinate List) Format:** Stores (row, column, value) for nonzero elements.
2. **CSR (Compressed Sparse Row) Format:** Efficient for row-based operations.
3. **CSC (Compressed Sparse Column) Format:** Efficient for column-based operations.

Scikit-learn’s CountVectorizer by default returns a sparse matrix, which you converted to a **dense array** using .toarray().

The code in the image is using the CountVectorizer object (cv) to retrieve the list of unique words (features) that were extracted from the text data. Here's a step-by-step breakdown:

### **Step 1: Calling get\_feature\_names\_out()**

python

CopyEdit

cv.get\_feature\_names\_out()

* This function returns an **array of feature names** (i.e., words) that the CountVectorizer has learned from the dataset.
* These are the words that appear in the transformed document-term matrix.
* It reflects the vocabulary used for vectorization.

### **Step 2: Understanding the Output**

The output is a NumPy array that contains all the unique words that CountVectorizer has kept after preprocessing (removing stopwords, limiting vocabulary size to max\_features=5000, etc.).

Example output:

python

CopyEdit

array(['000', '007', '10', ..., 'zone', 'zoo', 'zooeydeschanel'], dtype=object)

* '000', '007', and '10' → These are words (or numbers) extracted from the text.
* 'zone', 'zoo', 'zooeydeschanel' → These are some other extracted words.
* The ... in the output represents that there are many more words in the array.

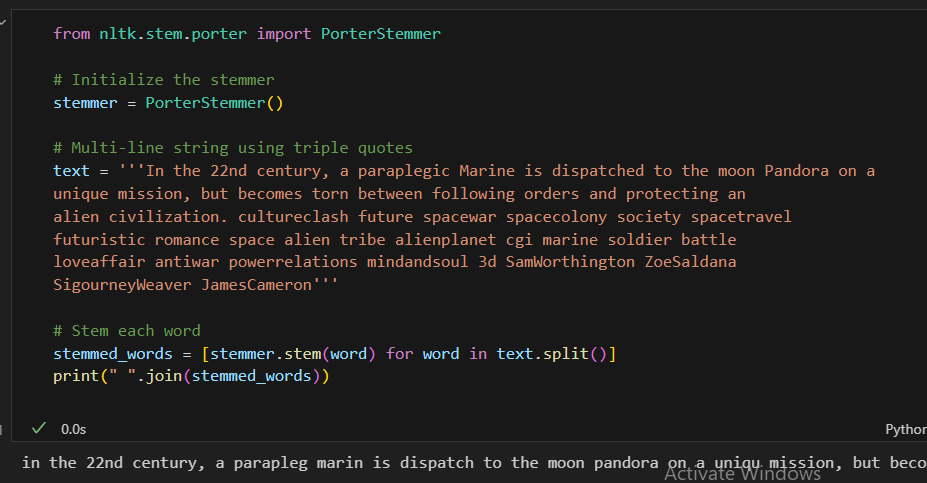
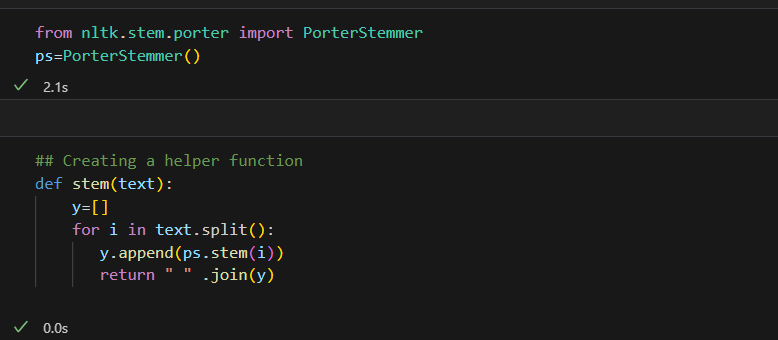
### **Step 3: How It Works**

1. CountVectorizer reads the text data (e.g., movie descriptions, tags, etc.).
2. It tokenizes the text into words.
3. It removes stopwords (if specified).
4. It keeps only the max\_features=5000 most frequent words.
5. The final list of words becomes the "features" used in the vectorized dataset.
6. get\_feature\_names\_out() displays these words.

### **Use Cases**

* Helps **understand the vocabulary** extracted from the dataset.
* Useful for **debugging** to check which words are included or removed.
* Can be used to **interpret the vectorized data**.

**Applying Stemming:**

**Step 1: Importing the PorterStemmer**

from nltk.stem.porter import PorterStemmer

* The PorterStemmer is a stemming algorithm that reduces words to their root form (e.g., "running" → "run", "flies" → "fli").
* It is part of the **Natural Language Toolkit (NLTK)**.

### **Step 2: Creating an Instance of PorterStemmer**

python

CopyEdit

ps = PorterStemmer()

* This initializes an object ps of the PorterStemmer class.
* ps.stem(word) will return the stemmed form of any given word.

### **Step 3: Defining a Helper Function**

python

CopyEdit

def stem(text):  
 y = [] # Initialize an empty list to store stemmed words  
 for i in text.split(): # Split the input text into words  
 y.append(ps.stem(i)) # Apply stemming to each word and add to the list  
 return " ".join(y) # Join the stemmed words into a sentence and return

* **Splitting the text**: text.split() breaks the input sentence into words.
* **Applying stemming**: ps.stem(i) converts each word to its root form.
* **Joining the words**: " ".join(y) combines the stemmed words back into a sentence.

### **Example Usage**

#### **Input:**

python

CopyEdit

text = "playing played plays"  
print(stem(text))

#### **Processing:**

1. "playing" → "play"
2. "played" → "play"
3. "plays" → "play"

#### **Output:**

arduino

CopyEdit

"play play play"

### **Potential Issue in Code**

There's an indentation mistake in the function:

python

CopyEdit

return " ".join(y)

* This line should be **outside** the for loop; otherwise, it returns only the first stemmed word.
* **Corrected Code:**

python

CopyEdit

def stem(text):  
 y = []  
 for i in text.split():  
 y.append(ps.stem(i))  
 return " ".join(y) # Correct indentation

### **Why Use Stemming?**

* Reduces words to their base form, making text processing more efficient.
* Helps in **search engines**, **chatbots**, and **natural language processing (NLP)** tasks.

### **Stemming vs. Lemmatization**

Both **stemming** and **lemmatization** are text normalization techniques used in **Natural Language Processing (NLP)** to reduce words to their root forms. However, they have key differences.

## **1️⃣ Stemming**

**Definition:**  
Stemming is a rule-based process of **chopping off word endings** to reduce words to their root form. It does **not consider the meaning of words**, which can sometimes lead to incorrect root words.

**Example using PorterStemmer:**

python

CopyEdit

from nltk.stem import PorterStemmer  
  
ps = PorterStemmer()  
words = ["running", "flies", "better", "studies", "caring"]  
stemmed\_words = [ps.stem(word) for word in words]  
print(stemmed\_words)

#### **Output:**

['run', 'fli', 'better', 'studi', 'care']

🔹 **Issues with stemming:**

* "flies" → "fli" (not a real word)
* "studies" → "studi" (incorrect root)
* "caring" → "care" (correct)

**Pros:** ✔ Faster and simpler  
✔ Works well for applications like search engines

**Cons:** ❌ Can produce non-existent words  
❌ Doesn’t always give meaningful roots

## **2️⃣ Lemmatization**

**Definition:**  
Lemmatization reduces words to their **dictionary base form (lemma)** by considering the meaning and part of speech (POS). It produces **real words**.

**Example using WordNetLemmatizer:**

python

CopyEdit

from nltk.stem import WordNetLemmatizer  
  
lemmatizer = WordNetLemmatizer()  
words = ["running", "flies", "better", "studies", "caring"]  
lemmatized\_words = [lemmatizer.lemmatize(word, pos="v") for word in words]  
print(lemmatized\_words)

#### **Output:**

css

CopyEdit

['run', 'fly', 'be', 'study', 'care']

🔹 **Why is lemmatization better?**

* "flies" → "fly" (correct word)
* "studies" → "study" (correct word)
* "better" → "be" (since it recognizes it as an adjective and changes meaning based on POS)

**Pros:** ✔ Produces actual dictionary words  
✔ More accurate compared to stemming

**Cons:** ❌ Slower since it uses a dictionary  
❌ Requires specifying **POS tags** for better accuracy

## **Key Differences:**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Stemming** | **Lemmatization** |
| **Method** | Chops word endings based on rules | Uses a dictionary and linguistic analysis |
| **Speed** | Fast 🚀 | Slower ⏳ |
| **Accuracy** | Lower ❌ | Higher ✅ |
| **Real Words?** | No ❌ | Yes ✅ |
| **Example (flies)** | "fli" ❌ | "fly" ✅ |

## **Which One to Use?**

✅ **Use Stemming** when:

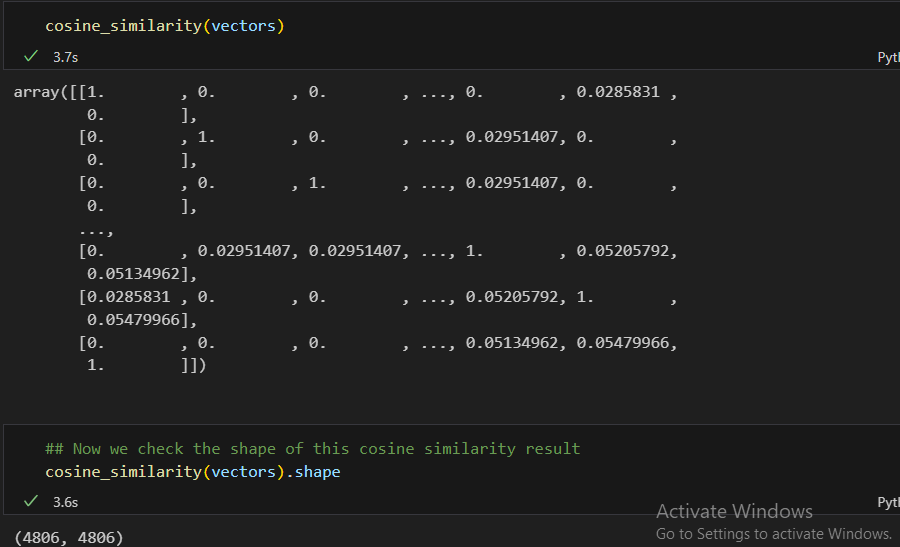
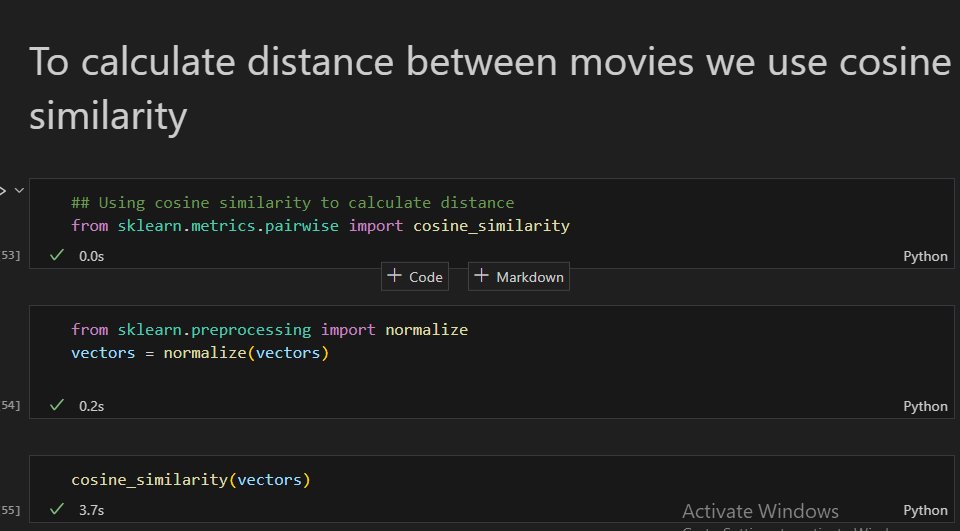
* You need **fast processing** (e.g., search engines, quick NLP tasks).
* Accuracy is **not critical**.

✅ **Use Lemmatization** when:

* You need **accurate meaning and real words**.
* You are working with **chatbots, machine learning models, or text summarization**.

**Calculating Distances Between Movies:**

* We have 5,000 movies, and each movie is represented as a vector in a high-dimensional space.
* There are several ways to calculate distances between vectors, but we use **cosine distance** because it is effective for measuring similarity between high-dimensional data.
* **Euclidean distance** is not suitable for large-dimensional spaces because it can become less meaningful as the number of dimensions increases.
* Movies that have a smaller distance (or higher cosine similarity) are considered more similar to each other, indicating they share common characteristics.
* Cosine distance measures the angle between two vectors, ensuring that even if the vectors have different magnitudes, the similarity based on direction is captured.



**Why we use enumerate Function:**

The enumerate() function in Python is used to add a counter to an iterable (like a list, tuple, or string) and return it as an enumerate object. It allows you to loop over the iterable while also keeping track of the index (or position) of the current item. This can be particularly useful when you need both the item and its index during a loop.

### **Key reasons for using enumerate():**

1. **Provides Index and Value**: It helps in iterating over the iterable while keeping track of the index automatically, which avoids the need for manually incrementing a counter.
2. **Cleaner Code**: Using enumerate() eliminates the need for extra code (like a separate counter variable) to keep track of the index.
3. **Improved Readability**: The code becomes easier to read and understand because you directly get both the index and the value in the same loop.

### **Example:**

fruits = ['apple', 'banana', 'cherry']  
  
# Using enumerate to get both index and value  
for index, fruit in enumerate(fruits):  
 print(f"Index: {index}, Fruit: {fruit}")

**Output:**

Index: 0, Fruit: apple  
Index: 1, Fruit: banana  
Index: 2, Fruit: cherry

In this example, enumerate() helps you access both the index and the value (fruit) in a clean and readable way.