The **Bag of Words (BoW)** approach is a fundamental technique used in **Natural Language Processing (NLP)** for text representation. Here's a breakdown of the BoW model:

1. **Concept**:
   * The BoW model represents text as a collection (or "bag") of its words, disregarding grammar, word order, and syntax but maintaining frequency counts of words in the text.
2. **Steps**:
   * **Tokenization**: Split the text into individual words (tokens).
   * **Vocabulary creation**: Build a set of unique words (vocabulary) from the corpus.
   * **Vectorization**: For each document, create a vector where each element corresponds to a word in the vocabulary. The value of each element is the count of occurrences of that word in the document.
3. **Example**: Suppose you have two sentences:
   * "I love cats"
   * "I love dogs"

The vocabulary would be: ["I", "love", "cats", "dogs"]

Now, represent each sentence as a vector based on the word occurrences:

* + "I love cats" → [1, 1, 1, 0]
  + (1 occurrence of "I", 1 of "love", 1 of "cats", 0 of "dogs")
  + "I love dogs" → [1, 1, 0, 1]
  + (1 occurrence of "I", 1 of "love", 0 of "cats", 1 of "dogs")

1. **Limitations**:
   * **No semantics**: BoW doesn't capture the meaning or context of words.
   * **High dimensionality**: For large vocabularies, BoW can result in large sparse vectors.
   * **Word order ignored**: Sentences with the same words but different meanings (due to word order) are treated the same.
2. **Extensions**:
   * **TF-IDF (Term Frequency-Inverse Document Frequency)**: A more advanced variant that weighs words based on their importance in the document relative to the entire corpus, reducing the impact of common words like "the", "and", etc.

BoW is often used as a baseline for more complex text representation techniques like **word embeddings** (Word2Vec, GloVe) and **transformers** (BERT, GPT).

**CountVectorizer**: This is used to convert the text into a matrix of token counts (n-grams). **.**

from sklearn.feature\_extraction.text import CountVectorizer

* This imports the CountVectorizer class from scikit-learn, which is used to convert a collection of text documents into a matrix of token counts (bag-of-words model).
* **Purpose**: To transform textual data into numerical data (word counts) so that it can be used as input for machine learning models.

Parameter of CountVectorizer

* ngram\_range=(1, 1) for unigrams (single words).
* ngram\_range=(2, 2) for bigrams (two consecutive words).
* ngram\_range=(3, 3) for trigrams (three consecutive words).

 **Fitting and Transforming:** The fit\_transform method is used on the corpus/training data, and the transform method is used to convert the new document/test data into the same vector space.

 **Feature Names**: These are the actual unigrams, bigrams, or trigrams extracted.

**fit\_transform method**

The fit\_transform method performs two operations:

1. **Fit**: It analyzes the corpus, learns the vocabulary (all unique words, bigrams, trigrams, etc. based on the ngram\_range), and creates a mapping from words to feature indices.
2. **Transform**: It then converts the text data (corpus) into a Sparse numerical matrix or feature matrix where each row represents a document and each column represents a word/phrase (based on the n-gram chosen), and the values in the matrix represent the frequency of that word/phrase in each document.

This method is used for the **corpus**, which is the dataset that the model is initially trained on.

**Efficient Sparse Representation:**

* The matrix created by fit\_transform is automatically represented in a **sparse format**, meaning that only the positions and values of the **non-zero elements** are stored.
* This step optimizes the storage, avoiding memory waste by not saving the zero elements. For instance, a document that contains only a few words will have a lot of zeros in the original matrix, but the sparse representation will only store the words (features) that appear in that document.

**Transform method**

The transform method is used **only to transform new, unseen data** based on the vocabulary or features learned from the corpus (from the previous fit\_transform call).

It does not update the vocabulary but simply converts the new document into the same format (i.e., matrix of token counts) using the existing vocabulary. This method is used for the **new\_doc/test data** in this case.

Or

The transform method converts the new document new\_doc into the same unigram matrix using the previously learned vocabulary from CORPUS.

**a. Unigrams (1-gram) Extraction**

* vectorizer = CountVectorizer(ngram\_range=(1, 1)): Initializes the vectorizer for unigrams (single words).
* X\_corpus = vectorizer.fit\_transform(corpus):
  + **Fit**: The vectorizer learns the vocabulary from CORPUS (it will find all unique unigrams).
  + **Transform**: Converts CORPUS into a sparse matrix where each row is a document, and each column is a unigram, and the values are counts of how often each unigram appears in the document.
* X\_new\_doc = vectorizer.transform(new\_doc):
  + **Transform**: Converts the new document new\_doc into the same unigram matrix using the previously learned vocabulary from CORPUS.

**b. Bigrams (2-gram) Extraction**

* vectorizer = CountVectorizer(ngram\_range=(2, 2)): Initializes the vectorizer for bigrams (pairs of consecutive words).
* The same fit\_transform and transform methods are applied, but this time the vectorizer focuses on extracting bigrams.

**c. Trigrams (3-gram) Extraction**

* vectorizer = CountVectorizer(ngram\_range=(3, 3)): Initializes the vectorizer for trigrams (three consecutive words).
* The same process follows, but now trigrams are extracted.

**Mapping from words to feature indices:** This refers to how the CountVectorizer assigns each unique word (or n-gram) in the corpus to a specific position (or index) in the feature matrix. When fit is called on corpus, the following operations are performed:

**Here's what happens:**

1. **Vocabulary Learning**: When the fit method is called on a corpus of text data, CountVectorizer scans through all the text and identifies all unique words (or n-grams, depending on the ngram\_range setting). This collection of unique terms is known as the **vocabulary**.
2. **Mapping to Indices**: Once the vocabulary is created, CountVectorizer assigns a unique number (index) to each word or n-gram. This index is the "feature index." For example:
   * Let's say the unique words in a corpus are: ['apple', 'banana', 'orange', 'fruit'].
   * The CountVectorizer might assign indices like this(Mapping of n-grams to feature indices)
     + 'apple' → 0
     + 'banana' → 1
     + 'orange' → 2
     + 'fruit' → 3
3. **Feature Matrix Construction**: When you later transform text data into a numerical format (using transform or fit\_transform), each document is represented as a **vector** where the values correspond to the frequencies of the words, placed in the positions (indices) according to this mapping.
   * For example, for the sentence "apple banana banana fruit,"
   * the feature vector would be [1, 2, 0, 1] because:
     + 'apple' appears 1 time (at index 0),
     + 'banana' appears 2 times (at index 1),
     + 'orange' doesn't appear (0 at index 2),
     + 'fruit' appears 1 time (at index 3).

In this way, CountVectorizer turns each document into a vector of word counts, using the feature indices created during the **fit** step. This mapping is crucial for later steps, such as training machine learning models, which expect the input data in numerical (vectorized) form.

CORPUS = [ 'the sky is blue', '

sky is blue and sky is beautiful',

'the beautiful sky is so blue',

'i love blue cheese' ]

**1. Unigrams:**

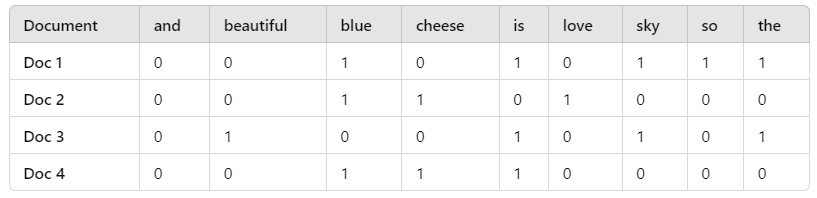
* **Feature Names**: ['and', 'beautiful', 'blue', 'cheese', 'is', 'love', 'sky', 'so', 'the']
* **Corpus Matrix** (4 rows representing the documents, 9 columns representing the unigrams.

Let's assume that we have a simple corpus of 4 documents:

1. **Doc 1**: "The sky is so blue"
2. **Doc 2**: "I love blue cheese"
3. **Doc 3**: "The sky is beautiful"
4. **Doc 4**: "Blue cheese is delicious"

The **Feature Names** represent the unigrams (words) that appear across the documents:  
['and', 'beautiful', 'blue', 'cheese', 'is', 'love', 'sky', 'so', 'the']

Now, the **Corpus Matrix** will represent the frequency of each word (unigram) from the feature names list in each document. The numbers represent the counts of each unigram in the respective document.



**2. Bigrams:**

* **Feature Names**: ['and sky', 'blue and', 'blue cheese', 'is beautiful', 'is blue', 'is so', 'sky is', 'the beautiful', 'the sky']
* **Corpus Matrix** (4 rows representing the documents, 9 columns representing the bigrams)

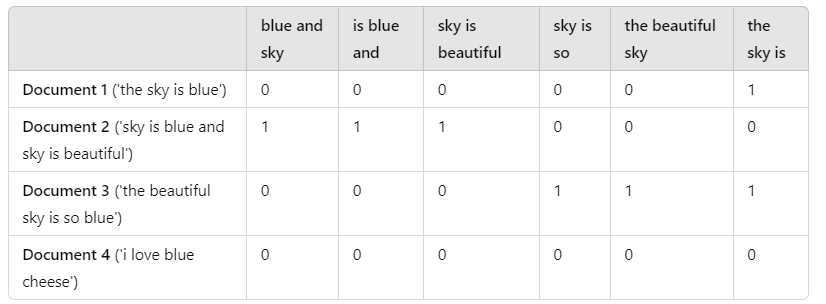


The bigram "i love" is not considered because CountVectorizer by default excludes words that appear only once in the entire corpus (based on the parameter min\_df=1). If the word "i" or "love" appears only once in the whole corpus, then the bigram "i love" may not be included due to the internal filtering that is taking place during vectorization.

This happens because:

1. **Minimum document frequency (min\_df)**: By default, CountVectorizer might be filtering out n-grams that appear too rarely, depending on the parameters used. If a term or a bigram appears only once across all documents, it may be ignored
2. **Stop words**: If you have stop-word removal enabled, common words like "i" or "love" may be filtered out.
3. **Trigrams:**

* **Feature Names**: ['blue and sky', 'is blue and', 'sky is beautiful', 'sky is so', 'the beautiful sky', 'the sky is']
* **Corpus Matrix** (4 rows representing the documents, 6 columns representing the trigrams)

****

Unigrams, bigrams, and trigrams are created from the corpus by breaking down sentences into contiguous sequences of words, as follows:

**1. Unigrams (1-grams):**

* Unigrams are single words extracted from each sentence.
* Each word in a sentence is treated as a separate token.

For example, for the sentence "the sky is blue":

* Unigrams: ["the", "sky", "is", "blue"]

**2. Bigrams (2-grams):**

* Bigrams are sequences of **two consecutive words** from the text.

For example, for the sentence "the sky is blue":

* Bigrams: ["the sky", "sky is", "is blue"]

**3. Trigrams (3-grams):**

* Trigrams are sequences of **three consecutive words** from the text.

For example, for the sentence "the sky is blue":

* Trigrams: ["the sky is", "sky is blue"]

Example of How Unigrams, Bigrams, and Trigrams Are Created from the Corpus

CORPUS = [

'the sky is blue',

'sky is blue and sky is beautiful',

'the beautiful sky is so blue',

'i love blue cheese'

]

 **Unigrams**: Each sentence is tokenized into individual words:

* Sentence 1: ['the', 'sky', 'is', 'blue']
* Sentence 2: ['sky', 'is', 'blue', 'and', 'sky', 'is', 'beautiful']
* Sentence 3: ['the', 'beautiful', 'sky', 'is', 'so', 'blue']
* Sentence 4: ['i', 'love', 'blue', 'cheese']

 **Bigrams**: Pairs of consecutive words are created:

* Sentence 1: ['the sky', 'sky is', 'is blue']
* Sentence 2: ['sky is', 'is blue', 'blue and', 'and sky', 'sky is', 'is beautiful']
* Sentence 3: ['the beautiful', 'beautiful sky', 'sky is', 'is so', 'so blue']
* Sentence 4: ['i love', 'love blue', 'blue cheese']

 **Trigrams**: Sequences of three consecutive words are created:

* Sentence 1: ['the sky is', 'sky is blue']
* Sentence 2: ['sky is blue', 'is blue and', 'blue and sky', 'and sky is', 'sky is beautiful']
* Sentence 3: ['the beautiful sky', 'beautiful sky is', 'sky is so', 'is so blue']
* Sentence 4: ['i love blue', 'love blue cheese']

**Get the feature names**

feature\_names = vectorizer.get\_feature\_names\_out()

* This retrieves the actual n-gram terms that were found in the corpus during the fitting process.
* **feature\_names**: A list of n-grams (unigrams, bigrams, or trigrams) that correspond to the columns in the sparse matrices X\_corpus and X\_new\_doc.

**Summing counts**:

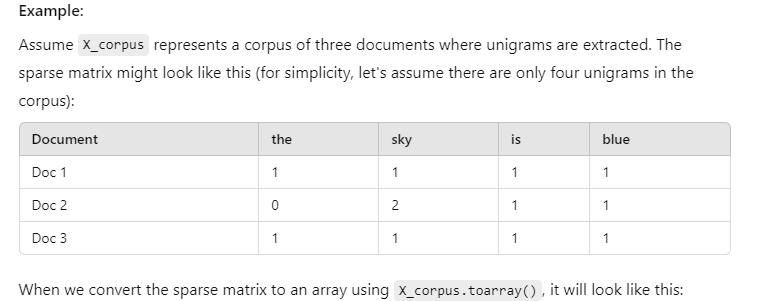
* X\_corpus.toarray().sum(axis=0): This sums the counts of each n-gram across all documents in the corpus to show the overall frequency.
* X\_new\_doc.toarray().flatten(): Converts the count matrix of the new document into a one-dimensional array.

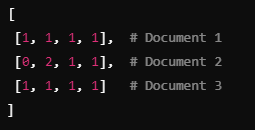
**Convert the Matrices to Arrays**:

corpus\_ngrams = X\_corpus.toarray().sum(axis=0)

new\_doc\_ngrams = X\_new\_doc.toarray().flatten()

* **X\_corpus.toarray()**:
  + Converts the sparse matrix for the corpus into a regular numpy array(dense array).
  + **.sum(axis=0)**: Sums the counts of each n-gram across all documents in the corpus to get the total count of each n-gram in the corpus.
* Converts the sparse matrix X\_corpus (which contains counts of n-grams for each document) into a dense numpy array.
* **X\_new\_doc.toarray().flatten()**:
  + Converts the sparse matrix for the new document into a numpy array.
  + **.flatten()**: Converts the 2D array into a 1D array to easily compare counts of n-grams in the new document.





Now, .sum(axis=0) will sum the counts for each unigram (column-wise) across all documents:



[2, 4, 3, 3]

# 'the' occurs 2 times, 'sky' 4 times, 'is' 3 times, 'blue' 3 times in the entire corpus

This array corpus\_ngrams holds the total count of each unigram across all documents in the corpus.

### ****Sparse Matrix:****

* A **sparse matrix** is a matrix where most of the elements are zero. This happens when you're dealing with data that has a lot of zero values (such as in text processing, where many words or n-grams don't appear in every document).
* In the case of **n-grams** in text, most documents only contain a small subset of all possible n-grams in the corpus. As a result, for a given document, many n-grams will have a count of zero.
* A sparse matrix is a memory-efficient way to store such data because it doesn't store all the zeros explicitly. Instead, it only stores the positions and values of the non-zero elements, which saves a lot of space when dealing with large corpora and vocabularies.

#### Example of Sparse Matrix:

Imagine a simple vocabulary of unigrams with 6 words: ["the", "sky", "is", "blue", "cheese", "love"], and a document "sky is blue".

In this document, only "sky", "is", and "blue" appear, while the rest of the words do not.

A sparse matrix would look something like this:

[0, 1, 1, 1, 0, 0] # Counts of [the, sky, is, blue, cheese, love]

In a sparse matrix format, this would only store the indices and values of the non-zero elements, i.e., something like:

* At position 1 (second column), the value is 1 (for 'sky').
* At position 2 (third column), the value is 1 (for 'is').
* At position 3 (fourth column), the value is 1 (for 'blue').

This sparse representation **saves memory** by not storing the zeros and only storing the relevant data (non-zero values and their positions).

# Non-zero counts at positions 1: (sky),

Position 2: (is),

Position 3: (blue)

[(1, 1), (2, 1), (3, 1)]

**Dense Array:**

* A **dense array** is a matrix where all elements are stored explicitly, including the zeros. This is the regular way to represent matrices or arrays in Python using libraries like NumPy.
* Even though most of the values might be zero, a dense array still stores those zeros, taking up more memory.

**Example of Dense Array:**

For the same document "sky is blue", a dense array representation would explicitly store every value, even the zeros:

[0, 1, 1, 1, 0, 0] # Counts of [the, sky, is, blue, cheese, love]

This stores each word's count, even if the word doesn’t appear in the document (which would result in a zero).

**Sparse Matrix Representation:**

This sparse matrix representation is for a document (or row of data) where only certain **words (or n-grams)** appear, and their counts are non-zero. Instead of storing all values (including zeros for the words that don't appear in the document), we only store the positions and the corresponding counts of the words that appear.

Each tuple (position, count) in the sparse matrix means:

* **Position**: The index in the vector (which corresponds to a word or n-gram).
* **Count**: How many times that word or n-gram appears in the document.

So, in the sparse matrix [(1, 1), (2, 1), (3, 1)]:

* **(1, 1)**: At index 1 (which corresponds to the word "sky"), there is a count of 1. This means "sky" appears once in this document.
* **(2, 1)**: At index 2 (which corresponds to the word "is"), there is a count of 1. "Is" appears once in this document.
* **(3, 1)**: At index 3 (which corresponds to the word "blue"), there is a count of 1. "Blue" appears once in this document.

**Key Difference:**

* A **sparse matrix** only stores the non-zero elements and their positions, saving memory, especially when the data contains many zero values.
* A **dense array** stores every element explicitly, including the zeros, which can take up more memory.

In the code I provided

* **X\_corpus** is initially stored as a **sparse matrix** because most n-grams in the vocabulary will not appear in every document, leading to a lot of zeros.
* **toarray()** converts the sparse matrix into a **dense array**, which includes all the zero counts. This is useful for operations that require direct access to all elements, like summing or counting across documents.

**Key Reasons for Converting a Sparse Matrix into a Dense Array:**

**Ease of Manipulation and Computation:**

* **Dense arrays** are much easier to manipulate for simple operations like summing values or flattening data. Libraries like **NumPy** are optimized for these types of operations when using dense arrays.

In this case:

corpus\_ngrams = X\_corpus.toarray().sum(axis=0)

The summing operation across the corpus works efficiently on a dense array. It treats all elements, including zeros, consistently and allows the result to be a simple 1D array of n-gram counts. Doing this directly on a sparse matrix is more complex and less intuitive in certain cases.

**Compatibility with Other Python Libraries:**

* Many libraries, especially for data manipulation and visualization (e.g., **pandas**), expect dense arrays or standard data structures like lists or NumPy arrays as inputs.
* For example, when creating a **pandas DataFrame**, you need the data in a form that pandas can work with:

Using a dense array ensures smooth integration with pandas.

 **Simplicity for Small Data:**

* For small corpora or datasets (like the one in the example), the memory overhead of converting the sparse matrix to a dense array is negligible. This makes dense arrays a good choice for simplicity in implementation.
* The example corpus only has a few n-grams, so the cost of storing zeros is very low. However, with a large corpus, keeping data in sparse format might be more appropriate.

 **Summing Across Documents:**

* The code uses .sum(axis=0) to get the total counts of each n-gram across all documents. With a dense array, this operation is very straightforward. It sums each column (representing an n-gram) across all rows (representing documents). This is easy to understand and implement in code.
* In a **sparse matrix**, summing operations may not be as intuitive and would require different methods (like using sum() or .A.sum() for sparse matrices in SciPy or directly working with the non-zero elements).

 **Flattening the Array for New Documents:**

* When transforming the new document's n-grams, the .flatten() operation works on the dense array to convert a 2D array into a 1D array, making it easier to access counts. In this case:

### When to Avoid Converting to Dense Arrays:

In practice, converting sparse matrices to dense arrays should be avoided for **large datasets** or **high-dimensional data**, because storing all zeros explicitly can consume a lot of memory. For example, if you have a corpus with millions of documents and tens of thousands of unique n-grams, the number of zero entries would grow significantly, leading to inefficient memory usage.

However, for smaller datasets like the one in the example, the simplicity and ease of working with dense arrays can outweigh the minor memory overhead.

The sparse matrix representation [(1, 1), (2, 1), (3, 1)] shows a more compact way of representing data, where only **non-zero** elements are stored.

**What It Represents:**

This sparse matrix representation is for a document (or row of data) where only certain **words (or n-grams)** appear, and their counts are non-zero. Instead of storing all values (including zeros for the words that don't appear in the document), we only store the positions and the corresponding counts of the words that appear.

Each tuple (position, count) in the sparse matrix means:

* **Position**: The index in the vector (which corresponds to a word or n-gram).
* **Count**: How many times that word or n-gram appears in the document.

So, in the sparse matrix [(1, 1), (2, 1), (3, 1)]:

* **(1, 1)**: At index 1 (which corresponds to the word "sky"), there is a count of 1. This means "sky" appears once in this document.
* **(2, 1)**: At index 2 (which corresponds to the word "is"), there is a count of 1. "Is" appears once in this document.
* **(3, 1)**: At index 3 (which corresponds to the word "blue"), there is a count of 1. "Blue" appears once in this document.

### Example Context:

Let's assume we have a simple **vocabulary** of 6 words: ["the", "sky", "is", "blue", "cheese", "love"]. When you create a bag-of-words model, each document is represented as a vector of counts corresponding to these words.

For the document "sky is blue", this is how the vector representation would look in a **dense array** (count of each word in the vocabulary):

**[0, 1, 1, 1, 0, 0] # Counts of ["the", "sky", "is", "blue", "cheese", "love"]**

* The word "the" appears 0 times.
* The word "sky" appears 1 time.
* The word "is" appears 1 time.
* The word "blue" appears 1 time.
* The words "cheese" and "love" appear 0 times.

In a **sparse matrix** format, only the **non-zero values** (i.e., the words that actually appear in the document) are stored, along with their positions:

* "sky" is at index 1 and appears once.
* "is" is at index 2 and appears once.
* "blue" is at index 3 and appears once.

**This gives: [(1, 1), (2, 1), (3, 1)]**

**Why Sparse Representation?**

* **Memory Efficiency**: Instead of storing all the zeros (which could be many if the vocabulary is large), sparse representation only stores the non-zero values. This is more memory-efficient, especially when working with large corpora where most words (or n-grams) will not appear in most documents.
* **Position-Based**: The sparse matrix stores the index (position) where a word or n-gram occurs in the vectorized representation, along with how many times it occurs (the count).

**Comparison with Dense Representation:**

In contrast, a **dense array** representation of the document "sky is blue" would look like this:

**[0, 1, 1, 1, 0, 0] # Include all words, even with zero counts**

But the sparse matrix only stores:

**[(1, 1), (2, 1), (3, 1)] # Only store non-zero counts**

This sparse representation helps save memory and is often used for text data because most documents will have a lot of zeros in their vectorized form, especially with a large vocabulary.

**1. print(df.isnull().sum())**

* This line prints the number of missing values (**NaN** or **None**) in each column of the DataFrame df.
* df.isnull() returns a DataFrame of the same shape as df where each element is True if the corresponding element in df is missing (null), and False otherwise.
* .sum() adds up the True values for each column, effectively counting the number of null values in each column.
* **Purpose**: To check if there are any missing values in the dataset and in which columns they occur.

**2. print(df['Label'].value\_counts())**

* This line prints the frequency (count) of each unique value in the 'Label' column of the DataFrame df.
* df['Label'] accesses the 'Label' column, and value\_counts() counts how many times each unique label appears.
* **Purpose**: To show the distribution of classes (or categories) in the 'Label' column, which is helpful for understanding if the data is balanced or imbalanced.

**3. import re**

* This line imports Python's **regular expressions (re)** library, which is used for text pattern matching and string manipulation.
* **Purpose**: Allows you to perform text cleaning, such as removing unwanted characters or patterns from strings.

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

* This imports the train\_test\_split function from the scikit-learn library.
* **Purpose**: This function is used to split the dataset into training and testing sets, which is important for machine learning tasks to evaluate model performance.

**5****. from sklearn.feature\_extraction.text import CountVectorizer**

* This imports the CountVectorizer class from scikit-learn, which is used to convert a collection of text documents into a matrix of token counts (bag-of-words model).
* **Purpose**: To transform textual data into numerical data (word counts) so that it can be used as input for machine learning models.

**6. def preprocess\_text(text):**

This function defines the steps for preprocessing the text.

**Steps Inside preprocess\_text:**

* **Lowercasing**: text = text.lower()
  + Converts all characters in the text to lowercase to ensure that "Apple" and "apple" are treated as the same word.
  + **Purpose**: Helps standardize the text and avoid case sensitivity.
* **Remove Punctuation and Non-Alphanumeric Characters**: text =
* re.sub(r'[^a-zA-Z0-9\s]', '', text)
  + Uses the re.sub() function from the re library to substitute any character that is not a letter (a-zA-Z), number (0-9), or whitespace (\s) with an empty string ('').
  + This effectively removes punctuation, special symbols, and other non-alphanumeric characters from the text.
  + **Purpose**: To clean up the text by removing unnecessary characters, leaving only letters, numbers, and spaces.

**7. df['Cleaned\_Message'] = df['Message'].apply(preprocess\_text)**

* This line applies the preprocess\_text function to the 'Message' column in the DataFrame df and stores the cleaned text in a new column called 'Cleaned\_Message'.
* .apply(preprocess\_text) applies the preprocess\_text function to each value (message) in the 'Message' column.
* **Purpose**: Preprocess and clean the text data before using it for machine learning, making it more suitable for analysis and model training.

In the pattern r'[^a-zA-Z0-9\s]', the components r and \s have specific meanings in the context of regular expressions in Python:

**1. r (Raw String Literal):**

* The r before the string indicates a **raw string**.
* In a normal string, backslashes (\) are treated as escape characters (e.g., \n for newline, \t for tab). However, in a raw string (denoted by r'...'), backslashes are treated as literal characters, and Python does not interpret them as escape sequences.
* **Purpose**: Using r ensures that the backslashes in the regular expression are treated literally, which makes it easier to write regex patterns without worrying about Python’s escape rules.

For example:

* '\\s' (without r) would need to escape the backslash, but r'\s' (with r) is easier to read and write.

**2. \s (Whitespace Character)**

* \s is a **special character** in regular expressions that matches any **whitespace character**. This includes:
  + Spaces
  + Tabs (\t)
  + Newlines (\n)
  + Carriage returns (\r)
* **Purpose**: In the pattern r'[^a-zA-Z0-9\s]', the \s part tells the regex engine to consider whitespace (spaces, tabs, etc.) as valid characters, which means they won’t be removed when applying this pattern.

**Breakdown of the Full Pattern r'[^a-zA-Z0-9\s]':**

* ^ inside the square brackets [] means **"not"**, so it negates the pattern that follows.
* a-zA-Z matches any **uppercase** (A-Z) or **lowercase** (a-z) letter.
* 0-9 matches any **digit** (0-9).
* \s matches any **whitespace character** (spaces, tabs, etc.).

Thus, [^a-zA-Z0-9\s] matches any character that is **not** a letter (uppercase or lowercase), a number, or a whitespace character. This includes punctuation, special symbols, and other non-alphanumeric characters. The re.sub(r'[^a-zA-Z0-9\s]', '', text) removes these unwanted characters from the text.

**1. X = df['Cleaned\_Text'] and y = df['category']**

* X and y represent the **features** and **target labels** respectively.
  + X is the input text data (features), which in this case is the Cleaned\_Text column from the DataFrame df. This column contains preprocessed text (cleaned versions of messages or documents).
  + y is the target variable, which contains the labels for each corresponding text document. The category column likely represents some classification labels (e.g., spam/ham, positive/negative, etc.).

**2. Splitting Data into Training and Testing Sets**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)**

**train\_test\_split()**: This function from sklearn.model\_selection splits the dataset into two subsets: a **training set** and a **testing set**.

* **X and y** are split into:
  + X\_train: 80% of the X data used for training.
  + X\_test: 20% of the X data used for testing.
  + y\_train: 80% of the y labels corresponding to the training data.
  + y\_test: 20% of the y labels corresponding to the testing data.
* **test\_size=0.2** means that 20% of the data is allocated to the testing set, and 80% is for training.
* **random\_state=42** ensures that the split is reproducible. Using the same random state ensures the same split every time the code is run.

1. **Vectorizing the Text Data**

vectorizer = CountVectorizer()

X\_train\_vectorized = vectorizer.fit\_transform(X\_train)

X\_test\_vectorized = vectorizer.transform(X\_test)

The value 42 for random\_state is not special or necessary—it's simply a commonly used value for reproducibility in machine learning examples. You can use any integer as the random\_state. The key is that specifying **any fixed integer** ensures that the split of the data is **reproducible** across different runs of the code.

1. **Why Use random\_state?**

* **Reproducibility**: When you specify a random\_state, it ensures that the data is split in the same way every time the code is run, which is useful for consistency in experiments and debugging.
* **Without random\_state**: If you do not specify random\_state, the data will be split randomly every time the code is executed, which can lead to different training and test sets on different runs, potentially causing varying results.

Other Valuescan use any integer for random\_state. For example:

* random\_state=1
* random\_state=123
* random\_state=7

All of these will create different, but still consistent and reproducible, splits. The choice of the integer itself doesn’t affect the quality of the model or the split; it just controls the randomness.

If you want non-reproducible random splits, you can leave random\_state unset, and it will split randomly each time.

**Initialize the Multinomial Naive Bayes Classifier**

classifier = MultinomialNB()

 This line creates an instance of the **Multinomial Naive Bayes** classifier from the sklearn.naive\_bayes library.

 Multinomial **Naive Bayes** is commonly used for text classification problems where features (word counts or frequencies) are represented as discrete data (hence the use of CountVectorizer for vectorization).

**Fit the Model on Training Data**

**classifier.fit(X\_train\_vectorized, y\_train)**

* **fit()**: This method trains the model using the training data.
* X\_train\_vectorized: This is the feature matrix that has been transformed using CountVectorizer. It contains the word count representation of the training documents.
* y\_train: This contains the actual labels (categories) of the training data.

During this step, the model learns the relationship between the word counts (features) and the labels (categories), which it will use to make predictions.

**Make Predictions on the Test Data**

y\_pred = classifier.predict(X\_test\_vectorized)

**predict()**: This method uses the trained classifier to make predictions on new (unseen) data.

X\_test\_vectorized: This is the test feature matrix, where each test document is represented by word counts.

The output y\_pred contains the predicted labels for the test data.

**Evaluate the Model**

Print (classification\_report(y\_test, y\_pred))

* **classification\_report()**: This function generates a comprehensive evaluation report for the classification model.
* y\_test: These are the true labels for the test set.
* y\_pred: These are the predicted labels made by the model.

The classification report includes:

* **Precision**: The ratio of true positives to the sum of true and false positives.
* **Recall**: The ratio of true positives to the sum of true positives and false negatives.
* **F1-Score**: The harmonic means of precision and recall.
* **Support**: The number of actual occurrences of each class.

The printed report will show these metrics for each class, as well as averages (macro, micro, and weighted).

. **Store the Classification Report as a Dictionary**

report = classification\_report(y\_test, y\_pred, output\_dict=True)

 The **output\_dict=True** parameter converts the classification report into a dictionary for further analysis or processing.

The report variable now holds a dictionary that contains the precision, recall, F1 score, and support for each class, as well as overall averages.