In [5]:

```
Text = "I am learning NLP"
```

In [7]:

```
import pandas as pd
pd.get_dummies(Text.split())
```

Out[7]:

	I	NLP	am	learning
0	True	False	False	False
1	False	False	True	False
2	False	False	False	True
3	False	True	False	False

In [9]:

```
text = ["i love NLP and i will learn NLP in 2month"]
```

In [13]:

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
vectorizer.fit(text)
vector = vectorizer.transform(text)
```

In [14]:

```
print(vectorizer.vocabulary_)
print(vector.toarray())
```

```
{'love': 4, 'nlp': 5, 'and': 1, 'will': 6, 'learn': 3, 'in': 2, '2month': 0}
[[1 1 1 1 1 2 1]]
```

In [17]:

```
print(vector)
```

```
  (0, 0)	1
  (0, 1)	1
  (0, 2)	1
  (0, 3)	1
  (0, 4)	1
  (0, 5)	2
  (0, 6)	1
```

In [19]:

```
CountVectorizer?
```

```
Init signature:
CountVectorizer(
    *,
    input='content',
    encoding='utf-8',
    decode_error='strict',
    strip_accents=None,
    lowercase=True,
    preprocessor=None,
    tokenizer=None,
    stop_words=None,
    token_pattern='(?u)\\b\\w\\w+\\b',
    ngram_range=(1, 1),
    analyzer='word',
    max_df=1.0,
    min_df=1,
    max_features=None,
    vocabulary=None,
    binary=False,
    dtype=<class 'numpy.int64'>,
)
Docstring:
Convert a collection of text documents to a matrix of token counts.

This implementation produces a sparse representation of the counts using
scipy.sparse.csr_matrix.

If you do not provide an a-priori dictionary and you do not use an analyzer
that does some kind of feature selection then the number of features will
be equal to the vocabulary size found by analyzing the data.

For an efficiency comparison of the different feature extractors, see
:ref:`sphx_glr_auto_examples_text_plot_hashing_vs_dict_vectorizer.py`.

Read more in the :ref:`User Guide <text_feature_extraction>`.

Parameters
----------
input : {'filename', 'file', 'content'}, default='content'
    - If `'filename'`, the sequence passed as an argument to fit is
      expected to be a list of filenames that need reading to fetch
      the raw content to analyze.

    - If `'file'`, the sequence items must have a 'read' method (file-like
      object) that is called to fetch the bytes in memory.

    - If `'content'`, the input is expected to be a sequence of items that
      can be of type string or byte.

encoding : str, default='utf-8'
    If bytes or files are given to analyze, this encoding is used to
    decode.

decode_error : {'strict', 'ignore', 'replace'}, default='strict'
    Instruction on what to do if a byte sequence is given to analyze that
    contains characters not of the given `encoding`. By default, it is
    'strict', meaning that a UnicodeDecodeError will be raised. Other
    values are 'ignore' and 'replace'.

strip_accents : {'ascii', 'unicode'} or callable, default=None
    Remove accents and perform other character normalization
    during the preprocessing step.
    'ascii' is a fast method that only works on characters that have
    a direct ASCII mapping.
    'unicode' is a slightly slower method that works on any characters.
    None (default) means no character normalization is performed.

    Both 'ascii' and 'unicode' use NFKD normalization from
    :func:`unicodedata.normalize`.

lowercase : bool, default=True
    Convert all characters to lowercase before tokenizing.

preprocessor : callable, default=None
    Override the preprocessing (strip_accents and lowercase) stage while
    preserving the tokenizing and n-grams generation steps.
    Only applies if ``analyzer`` is not callable.

tokenizer : callable, default=None
    Override the string tokenization step while preserving the
    preprocessing and n-grams generation steps.
    Only applies if ``analyzer == 'word'``.

stop_words : {'english'}, list, default=None
    If 'english', a built-in stop word list for English is used.
    There are several known issues with 'english' and you should
    consider an alternative (see :ref:`stop_words`).

    If a list, that list is assumed to contain stop words, all of which
    will be removed from the resulting tokens.
    Only applies if ``analyzer == 'word'``.

    If None, no stop words will be used. In this case, setting `max_df`
    to a higher value, such as in the range (0.7, 1.0), can automatically detect
    and filter stop words based on intra corpus document frequency of terms.

token_pattern : str or None, default=r"(?u)\\b\\w\\w+\\b"
    Regular expression denoting what constitutes a "token", only used
    if ``analyzer == 'word'``. The default regexp select tokens of 2
    or more alphanumeric characters (punctuation is completely ignored
    and always treated as a token separator).

    If there is a capturing group in token_pattern then the
    captured group content, not the entire match, becomes the token.
    At most one capturing group is permitted.

ngram_range : tuple (min_n, max_n), default=(1, 1)
    The lower and upper boundary of the range of n-values for different
    word n-grams or char n-grams to be extracted. All values of n such
    such that min_n <= n <= max_n will be used. For example an
    ``ngram_range`` of ``(1, 1)`` means only unigrams, ``(1, 2)`` means
    unigrams and bigrams, and ``(2, 2)`` means only bigrams.
    Only applies if ``analyzer`` is not callable.

analyzer : {'word', 'char', 'char_wb'} or callable, default='word'
    Whether the feature should be made of word n-gram or character
    n-grams.
    Option 'char_wb' creates character n-grams only from text inside
    word boundaries; n-grams at the edges of words are padded with space.

    If a callable is passed it is used to extract the sequence of features
    out of the raw, unprocessed input.

    .. versionchanged:: 0.21

    Since v0.21, if ``input`` is ``filename`` or ``file``, the data is
    first read from the file and then passed to the given callable
    analyzer.

max_df : float in range [0.0, 1.0] or int, default=1.0
    When building the vocabulary ignore terms that have a document
    frequency strictly higher than the given threshold (corpus-specific
    stop words).
    If float, the parameter represents a proportion of documents, integer
    absolute counts.
    This parameter is ignored if vocabulary is not None.

min_df : float in range [0.0, 1.0] or int, default=1
    When building the vocabulary ignore terms that have a document
    frequency strictly lower than the given threshold. This value is also
    called cut-off in the literature.
    If float, the parameter represents a proportion of documents, integer
    absolute counts.
    This parameter is ignored if vocabulary is not None.

max_features : int, default=None
    If not None, build a vocabulary that only consider the top
    `max_features` ordered by term frequency across the corpus.
    Otherwise, all features are used.

    This parameter is ignored if vocabulary is not None.

vocabulary : Mapping or iterable, default=None
    Either a Mapping (e.g., a dict) where keys are terms and values are
    indices in the feature matrix, or an iterable over terms. If not
    given, a vocabulary is determined from the input documents. Indices
    in the mapping should not be repeated and should not have any gap
    between 0 and the largest index.

binary : bool, default=False
    If True, all non zero counts are set to 1. This is useful for discrete
    probabilistic models that model binary events rather than integer
    counts.

dtype : dtype, default=np.int64
    Type of the matrix returned by fit_transform() or transform().

Attributes
```

```
----------
vocabulary_ : dict
    A mapping of terms to feature indices.

fixed_vocabulary_ : bool
    True if a fixed vocabulary of term to indices mapping
    is provided by the user.

stop_words_ : set
    Terms that were ignored because they either:

      - occurred in too many documents (`max_df`)
      - occurred in too few documents (`min_df`)
      - were cut off by feature selection (`max_features`).

    This is only available if no vocabulary was given.

See Also
--------
HashingVectorizer : Convert a collection of text documents to a
    matrix of token counts.

TfidfVectorizer : Convert a collection of raw documents to a matrix
    of TF-IDF features.

Notes
-----
The ``stop_words_`` attribute can get large and increase the model size
when pickling. This attribute is provided only for introspection and can
be safely removed using delattr or set to None before pickling.

Examples
--------
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
...     'This is the first document.',
...     'This document is the second document.',
...     'And this is the third one.',
...     'Is this the first document?',
... ]
>>> vectorizer = CountVectorizer()
>>> X = vectorizer.fit_transform(corpus)
>>> vectorizer.get_feature_names_out()
array(['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third',
       'this'], ...)
>>> print(X.toarray())
[[0 1 1 1 0 0 1 0 1]
 [0 2 0 1 0 1 1 0 1]
 [1 0 0 1 1 0 1 1 1]
 [0 1 1 1 0 0 1 0 1]]
>>> vectorizer2 = CountVectorizer(analyzer='word', ngram_range=(2, 2))
>>> X2 = vectorizer2.fit_transform(corpus)
>>> vectorizer2.get_feature_names_out()
array(['and this', 'document is', 'first document', 'is the', 'is this',
       'second document', 'the first', 'the second', 'the third', 'third one',
       'this document', 'this is', 'this the'], ...)
```

```
>>> print(X2.toarray())
[[0 0 1 1 0 0 1 0 0 0 0 1 0]
[0 1 0 1 0 1 0 1 0 0 1 0 0]
[1 0 0 1 0 0 0 0 1 1 0 1 0]
[0 0 1 0 1 0 1 0 0 0 0 0 1]]
File:           c:\users\vaish\anaconda3\lib\site-packages\sklearn\feature_extractio
n\text.py
Type:           type
Subclasses:     TfidfVectorizer
```

In [21]:

```
df = pd.DataFrame(data=vector.toarray(), columns=vectorizer.get_feature_names_out()
df
```

Out[21]:

	2month	and	in	learn	love	nlp	will
0	1	1	1	1	1	2	1

In [23]:

```
#text = ["i love NLP and i will learn NLP in 2month"]
# single characters are ignored by count vectorizer
```

In [27]:

```
text = "I am learning NLP"
```

In [36]:

```
from textblob import TextBlob
TextBlob(text).ngrams(1)
```

Out[36]:

```
[WordList(['I']), WordList(['am']), WordList(['learning']), WordList(['NLP'])]
```

In [32]:

```
TextBlob(text).ngrams(2)
```

Out[32]:

```
[WordList(['I', 'am']),
 WordList(['am', 'learning']),
 WordList(['learning', 'NLP'])]
```

In [34]:

```
TextBlob(text).ngrams(3)
```

Out[34]:

```
[WordList(['I', 'am', 'learning']), WordList(['am', 'learning', 'NLP'])]
```

In [38]:

```
TextBlob(text).ngrams(4)
```

Out[38]:

```
[WordList(['I', 'am', 'learning', 'NLP'])]
```

CONVERTING TEXT TO FEATURES USING TFIDF

In [47]:

```
Text = ["The quick brown fox jumps over a lazy dog.","The fox","The dog"]
```

In [51]:

```
#Import TfidfVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
#Create the transform
vectorizer = TfidfVectorizer()
```

```
#Tokenize and build vocab
vectorizer.fit(Text)
```

Out[51]:

```
▾ TfidfVectorizer ⓘ ?

TfidfVectorizer()
```

In [53]:

```
print(vectorizer.vocabulary_)
print(vectorizer.idf_)
```

```
{'the': 7, 'quick': 6, 'brown': 0, 'fox': 2, 'jumps': 3, 'over': 5, 'lazy': 4, 'do
g': 1}
[1.69314718 1.28768207 1.28768207 1.69314718 1.69314718 1.69314718
 1.69314718 1.        ]
```

In [ ]:

```

```

In [ ]:

```

```