Machine Learning with PyTorch and Scikit-Learn

-- Code Examples

Package version checks

Add folder to path in order to load from the check_packages.py script:

In [1]:
```
import sys
sys.path.insert(0, '..')
```

你的程式碼片段意在修改 Python 的模組路徑，以包含當前目錄的上一層目錄。這是一種常見的做法，確保 Python 能夠找到位於父目錄中的模組或套件。

解釋：

1. **導入 `sys` 模組：**
   - `import sys`：這行導入了內建的 Python 模組 `sys`，該模組提供了訪問系統特定參數和函數的能力。
2. **修改 `sys.path`：**
   - `sys.path` 是一個包含目錄名的列表，Python 在這些目錄中尋找模組和套件。默認情況下，它包括當前目錄和標準庫位置。
   - `sys.path.insert(0, '..')`：這行將 `'..'` 插入到 `sys.path` 的開頭。`'..'` 表示當前目錄的父目錄。`insert` 方法的第一個參數是 `0`，確保將此目錄添加到列表的最前面。這是因為 Python 根據 `sys.path` 中的順序進行目錄搜索，從列表的開頭開始。
3. **目的：**
   - 修改 `sys.path` 的目的是使位於父目錄 (`'..'`) 中的模組或套件能夠在你的 Python 腳本中直接訪問。
   - 這在以下情況下非常有用：當你的項目結構中有模組或套件以階層方式組織，並且你想要在當前腳本中從父目錄中導入模組時。
4. **使用考量：**
   - **相對導入：** 在修改 `sys.path` 後，你可以在 Python 腳本中使用相對導入，從父目錄導入模組或套件。
   - **項目組織：** 建議以一種避免複雜相對導入或過多修改 `sys.path` 的方式來組織項目結構。考慮使用虛擬環境和套件管理工具（如 `pip` 和 `setup.py`）進行更好的項目管理。
5. **安全性和最佳實踐：**
   - 修改 `sys.path` 時應注意安全性，特別是在生產環境或共享代碼時。如果管理不當，可能會導致意外行為。
   - 確保父目錄 (`'..'`) 包含你打算導入的模組或套件，並且它們遵循 Python 的模組命名慣例。

通過將 `'..'` 插入到 `sys.path` 中，你擴展了 Python 的搜索路徑，使得你能夠直接從父目錄導入模組或套件到當前的 Python 腳本中。

Check recommended package versions:

In [2]:
```
from python_environment_check import check_packages


d = {
    'numpy': '1.21.2',
    'pandas': '1.3.2',
    'sklearn': '1.0',
    'pyprind': '2.11.3',
    'nltk': '3.6',
}
check_packages(d)
```

```
[OK] Your Python version is 3.9.7 | packaged by conda-forge | (default, Sep 29 2021, 19:24:02)
[Clang 11.1.0 ]
/Users/sebastian/miniforge3/lib/python3.9/site-packages/scipy/__init__.py:146: UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for this version of SciPy (detected version 1.23.1
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
[OK] numpy 1.23.1
[OK] pandas 1.4.0
[OK] sklearn 1.0.2
[OK] pyprind 2.11.3
[OK] nltk 3.6.7
```

這段程式碼的目的是使用自定義的 `check_packages` 函式來確認系統中是否安裝了指定版本的幾個Python套件。具體來說，它會檢查以下套件的版本：

- **numpy**: 版本要求為 `1.21.2`
- **pandas**: 版本要求為 `1.3.2`
- **scikit-learn (sklearn)**: 版本要求為 `1.0`
- **pyprind**: 版本要求為 `2.11.3`
- **nltk**: 版本要求為 `3.6`

如果這些套件的版本不符合指定的要求，可能會產生錯誤或警告。這樣的確認對於確保執行特定Python應用程式或程式庫時，環境中有正確的套件版本是非常重要的。

Chapter 8 - Applying Machine Learning To Sentiment Analysis

Overview

Preparing the IMDb movie review data for text processing

Obtaining the IMDb movie review dataset

The IMDB movie review set can be downloaded from http://ai.stanford.edu/~amaas/data/sentiment/. After downloading the dataset, decompress the files.

A) If you are working with Linux or MacOS X, open a new terminal window, `cd` into the download directory and execute

`tar -zxf aclImdb_v1.tar.gz`

B) If you are working with Windows, download an archiver such as 7Zip to extract the files from the download archive.

Optional code to download and unzip the dataset via Python:

In [3]:
```
import os
import sys
import tarfile
import time
import urllib.request

source = 'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz'
target = 'aclImdb_v1.tar.gz'

if os.path.exists(target):
    os.remove(target)

def reporthook(count, block_size, total_size):
    global start_time
    if count == 0:
        start_time = time.time()
        return
    duration = time.time() - start_time
    progress_size = int(count * block_size)
    speed = progress_size / (1024.**2 * duration)
    percent = count * block_size * 100. / total_size
```

```
    sys.stdout.write(f'\r{int(percent)}% | {progress_size / (1024.**2):.2f} MB '
                     f'| {speed:.2f} MB/s | {duration:.2f} sec elapsed')
    sys.stdout.flush()


if not os.path.isdir('aclImdb') and not os.path.isfile('aclImdb_v1.tar.gz'):
    urllib.request.urlretrieve(source, target, reporthook)
```

這段程式碼用於下載來自於斯坦福大學的影評數據集（IMDb movie reviews dataset）。讓我來解釋這段程式碼的功能和運行邏輯：

程式碼功能解釋：

1. **設置下載來源和目標文件名稱**：

   ```
   source = 'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz'
   target = 'aclImdb_v1.tar.gz'
   ```
   - `source` 是影評數據集的下載鏈接。
   - `target` 是下載後保存的文件名稱。
2. **刪除已存在的目標文件**：

   ```
   if os.path.exists(target):
       os.remove(target)
   ```
   - 如果本地已經存在 `target` 文件，則刪除該文件。這樣可以確保下載時不會出現文件名衝突或覆蓋問題。
3. **定義進度報告函數 `reporthook`**：

   ```
   def reporthook(count, block_size, total_size):
       global start_time
       if count == 0:
           start_time = time.time()
           return
       duration = time.time() - start_time
       progress_size = int(count * block_size)
       speed = progress_size / (1024.**2 * duration)
       percent = count * block_size * 100. / total_size

       sys.stdout.write(f'\r{int(percent)}% | {progress_size / (1024.**2):.2f} MB '
                        f'| {speed:.2f} MB/s | {duration:.2f} sec elapsed')
       sys.stdout.flush()
   ```
   - `reporthook` 函數用於顯示下載進度和速度的信息。
   - `count` 表示當前已經下載的數據塊數量。
   - `block_size` 表示每個數據塊的大小。
   - `total_size` 表示需要下載的總大小。
   - 在函數內部計算並打印下載進度、速度等信息，使用 `sys.stdout.write` 和 `sys.stdout.flush` 來顯示並刷新進度信息。
4. **下載影評數據集文件**：

   ```
   if not os.path.isdir('aclImdb') and not os.path.isfile('aclImdb_v1.tar.gz'):
       urllib.request.urlretrieve(source, target, reporthook)
   ```
   - 如果本地不存在 `aclImdb` 文件夾和 `aclImdb_v1.tar.gz` 文件，則使用 `urllib.request.urlretrieve` 函數下載 `source` 鏈接指向的文件到 `target`，同時使用 `reporthook` 函數來顯示下載進度。

總結：

這段程式碼確保在本地下載 IMDb 影評數據集，並提供了下載進度的可視化，方便用戶了解下載過程中的狀態。

In [4]:
```
if not os.path.isdir('aclImdb'):

    with tarfile.open(target, 'r:gz') as tar:
        tar.extractall()
```

這段程式碼用於檢查是否已經存在解壓縮後的影評數據集文件夾 `aclImdb`，如果不存在則解壓縮 `target` 文件中的內容到該文件夾中。讓我來解釋一下這段程式碼的具體作用和執行過程：

程式碼功能解釋：

1. **檢查是否存在目標文件夾 `aclImdb`**：

   ```
   if not os.path.isdir('aclImdb'):
   ```
   - 使用 `os.path.isdir` 函數檢查當前目錄下是否已經存在 `aclImdb` 文件夾。如果不存在，條件成立，進入下一步操作。
2. **解壓縮目標文件 `target`**：

   ```
   with tarfile.open(target, 'r:gz') as tar:
       tar.extractall()
   ```
   - 使用 `tarfile.open` 函數打開 `target` 文件，並指定 `'r:gz'` 參數表示以讀取模式打開壓縮文件（`.tar.gz` 格式）。

- `tar.extractall()` 方法將壓縮文件中的所有內容解壓縮到當前工作目錄下的 `aclImdb` 文件夾中。

執行流程：

- 如果當前目錄下不存在 `aclImdb` 文件夾，則會打開 `target` 文件，並將其內容解壓縮到 `aclImdb` 文件夾中。
- 解壓縮過程中，所有的文件和目錄結構會被還原到 `aclImdb` 文件夾下，這樣後續的數據處理和分析可以在解壓後的 `aclImdb` 文件夾中進行。

注意事項：

確保 `target` 文件存在並且是有效的 `.tar.gz` 格式壓縮文件，否則 `tarfile.open` 可能會拋出異常。此外，解壓後的文件結構應該符合預期，以確保後續處理步驟能夠正確進行。

這段程式碼確保在需要時解壓縮 IMDb 影評數據集，並準備好進一步的數據分析或處理步驟。

Preprocessing the movie dataset into more convenient format

Install pyprind by uncommenting the next code cell.

In [5]:
```
#!pip install pyprind
```

這個指令 `#!pip install pyprind` 是用來安裝 Python 套件 `pyprind` 的。在 Jupyter Notebook 中使用這樣的指令，會讓系統自動使用 pip（Python 套件管理器）來下載並安裝 `pyprind` 套件。這個套件提供了進度條功能，能夠用來顯示迴圈或長時間運行任務的進度。

In [6]:
```
import pyprind
import pandas as pd
import os
import sys
from packaging import version


# change the `basepath` to the directory of the
# unzipped movie dataset

basepath = 'aclImdb'

labels = {'pos': 1, 'neg': 0}

# if the progress bar does not show, change stream=sys.stdout to stream=2
pbar = pyprind.ProgBar(50000, stream=sys.stdout)

df = pd.DataFrame()
for s in ('test', 'train'):
    for l in ('pos', 'neg'):
        path = os.path.join(basepath, s, l)
        for file in sorted(os.listdir(path)):
            with open(os.path.join(path, file),
                      'r', encoding='utf-8') as infile:
                txt = infile.read()

            if version.parse(pd.__version__) >= version.parse("1.3.2"):
                x = pd.DataFrame([[txt, labels[l]]], columns=['review', 'sentiment'])
                df = pd.concat([df, x], ignore_index=False)

            else:
                df = df.append([[txt, labels[l]]],
                               ignore_index=True)
            pbar.update()
df.columns = ['review', 'sentiment']
```

```
0% [##############################] 100% | ETA: 00:00:00
Total time elapsed: 00:00:19
```

這段程式碼的功能是從解壓後的 IMDb 影評數據集中讀取影評文本文件，並將其加載到 Pandas DataFrame 中。讓我來解釋這段程式碼的具體運行流程和每個部分的作用：

程式碼解釋：

1. **導入必要的庫和模組**：

   ```
   import pyprind
   import pandas as pd
   import os
   import sys
   from packaging import version
   ```

- `pyprind` 用於顯示進度條。
- `pandas` 用於數據處理和管理。
- `os` 和 `sys` 用於文件系統操作和流管理。
- `version` 用於檢查 Pandas 版本號。

2. **設置基本路徑和標籤字典**：

```
basepath = 'aclImdb'
labels = {'pos': 1, 'neg': 0}
```

- `basepath` 是 IMDb 影評數據集解壓後的基本路徑。
- `labels` 是標籤字典，將 'pos' 映射為 1（正面評論），'neg' 映射為 0（負面評論）。

3. **初始化進度條**：

```
pbar = pyprind.ProgBar(50000, stream=sys.stdout)
```

- 使用 `pyprind.ProgBar` 初始化一個進度條，總共有 50000 次更新，將進度信息輸出到標準輸出流（stdout）。

4. **創建空的 Pandas DataFrame**：

```
df = pd.DataFrame()
```

- 創建一個空的 Pandas DataFrame `df` 來存儲影評數據。

5. **讀取影評數據集文件**：

```
for s in ('test', 'train'):
    for l in ('pos', 'neg'):
        path = os.path.join(basepath, s, l)
        for file in sorted(os.listdir(path)):
            with open(os.path.join(path, file), 'r', encoding='utf-8') as infile:
                txt = infile.read()

            if version.parse(pd.__version__) >= version.parse("1.3.2"):
                x = pd.DataFrame([[txt, labels[l]]], columns=['review', 'sentiment'])
                df = pd.concat([df, x], ignore_index=False)
            else:
                df = df.append([[txt, labels[l]]], ignore_index=True)

            pbar.update()
```

- 使用巢狀迴圈遍歷 'test' 和 'train' 兩個子目錄，以及 'pos' 和 'neg' 兩種標籤。
- `os.path.join` 組合出每個文件的完整路徑。
- 使用 `open` 函數打開每個文件並讀取其內容到變量 `txt` 中。
- 根據 Pandas 的版本，使用不同的方式將數據加入到 DataFrame `df` 中：
  - 如果 Pandas 版本大於等於 1.3.2，使用 `pd.concat` 將新數據加入到 DataFrame。
  - 否則，使用 `df.append` 方法將新數據加入到 DataFrame。
- 在每次迴圈結束後，更新進度條。

6. **設置 DataFrame 列名**：

```
df.columns = ['review', 'sentiment']
```

- 將 DataFrame 的列名設置為 'review' 和 'sentiment'。

總結：

這段程式碼通過遍歷 IMDb 影評數據集的文件，將每個影評的文本和其對應的情感標籤（正面或負面）加載到 Pandas DataFrame 中。進度條顯示了加載進度，便於用戶了解數據加載的進程。

Shuffling the DataFrame:

In [7]:
```
import numpy as np


if version.parse(pd.__version__) >= version.parse("1.3.2"):
    df = df.sample(frac=1, random_state=0).reset_index(drop=True)

else:
    np.random.seed(0)
    df = df.reindex(np.random.permutation(df.index))
```

這段程式碼根據 Pandas 的版本不同，對 DataFrame `df` 中的數據進行隨機重排（shuffle）。讓我來解釋這段程式碼的具體作用和不同版本下的執行流程：

程式碼解釋：

1. **檢查 Pandas 版本並進行操作判斷**：

```
if version.parse(pd.__version__) >= version.parse("1.3.2"):
    df = df.sample(frac=1, random_state=0).reset_index(drop=True)
else:
    np.random.seed(0)
    df = df.reindex(np.random.permutation(df.index))
```

- `version.parse(pd.__version__)`：使用 `version` 模組中的 `parse` 函數來解析當前 Pandas 的版本號。
- 如果當前 Pandas 版本大於等於 `1.3.2`，則執行以下操作：
  - `df.sample(frac=1, random_state=0)`：使用 `sample` 方法將 DataFrame 中的行隨機抽樣，`frac=1` 表示抽取所有行，`random_state=0` 確保結果可以重現，即隨機種子為 0。
  - `.reset_index(drop=True)`：重設索引，並且丟棄原來的索引列，使新的索引從 0 開始。
- 如果當前 Pandas 版本小於 `1.3.2`，則執行以下操作：
  - `np.random.seed(0)`：設置 NumPy 的隨機種子為 0，確保結果可以重現。
  - `df.reindex(np.random.permutation(df.index))`：使用 NumPy 的 `permutation` 函數對 DataFrame 的索引進行隨機排列，然後使用 `reindex` 方法應用這個新的索引順序到 DataFrame 中。

不同版本下的執行流程：

- **Pandas >= 1.3.2**：
  - 使用 `sample` 方法來隨機打亂 DataFrame 的行。
  - 最後使用 `reset_index(drop=True)` 重設索引，確保索引是從 0 開始並丟棄原來的索引。
- **Pandas < 1.3.2**：
  - 使用 NumPy 的隨機種子來確保隨機過程的可重現性。
  - 使用 `np.random.permutation(df.index)` 來獲取隨機排列的索引，然後將這個新的索引應用到 DataFrame 中，從而達到打亂行的效果。

總結：

這段程式碼根據 Pandas 的版本不同，選擇不同的方法來隨機打亂 DataFrame 中的數據行。這樣可以確保在不同版本的環境中，操作的結果都能符合預期，並保持結果的一致性和可重現性。

Optional: Saving the assembled data as CSV file:

In [9]:
```
df.to_csv('movie_data.csv', index=False, encoding='utf-8')
```

這段程式碼將 Pandas DataFrame `df` 中的數據寫入到一個 CSV 文件中。讓我來解釋一下這段程式碼的作用和每個參數的含義：

程式碼解釋：

```
df.to_csv('movie_data.csv', index=False, encoding='utf-8')
```

- `df`：這是一個 Pandas DataFrame 對象，其中包含了影評數據，每行有兩列：'review'（影評文本）和 'sentiment'（情感標籤）。
- `to_csv` 方法：這是 Pandas 提供的將 DataFrame 數據寫入 CSV 文件的方法。
- `'movie_data.csv'`：這是要寫入的目標 CSV 文件的文件名。如果文件不存在，將會被創建；如果已存在，則會被覆蓋。
- `index=False`：這是 `to_csv` 方法的一個參數，設置為 False 表示不寫入 DataFrame 的行索引到 CSV 文件中。在這裡，我們不需要將索引寫入，因此設置為 False。
- `encoding='utf-8'`：這是指定 CSV 文件的編碼格式為 UTF-8，這樣可以支持包括中文在內的各種字符集。

執行結果：

當這段程式碼執行完畢後，將會生成一個名為 `movie_data.csv` 的 CSV 文件，該文件包含了 DataFrame `df` 中的所有數據。每行對應一條影評，列包括 'review' 和 'sentiment' 兩個字段，分別表示影評文本和情感標籤。這樣的 CSV 文件非常方便後續進行數據分析、機器學習模型的訓練等工作。

注意事項：

- 確保在執行此段程式碼之前，DataFrame `df` 已經被正確填充並準備好寫入到 CSV 文件中。
- 如果遇到文件寫入錯誤，可以檢查目標文件名、路徑許可權等問題。
- 當數據集較大時，可能需要一些時間來完成寫入操作，這取決於計算機性能和數據量大小。

In [10]:
```
import pandas as pd

df = pd.read_csv('movie_data.csv', encoding='utf-8')

# the following is necessary on some computers:
df = df.rename(columns={"0": "review", "1": "sentiment"})

df.head(3)
```

Out[10]:

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

這段程式碼用於從之前生成的 CSV 文件 `movie_data.csv` 中讀取數據，並展示前三行以確認數據是否正確讀取。讓我來解釋一下這段程式碼的具體作用：

程式碼解釋：

1. **讀取 CSV 文件**：

   ```
   df = pd.read_csv('movie_data.csv', encoding='utf-8')
   ```
   - 使用 Pandas 的 `read_csv` 函數讀取名為 `movie_data.csv` 的 CSV 文件。
   - `encoding='utf-8'` 指定文件的編碼格式為 UTF-8，以支持包含各種字符集的數據。
2. **重命名列名**：

   ```
   df = df.rename(columns={"0": "review", "1": "sentiment"})
   ```
   - 使用 `rename` 方法重命名 DataFrame 的列名。
   - `{"0": "review", "1": "sentiment"}`：字典形式，將原始列名 "0" 改為 "review"，將原始列名 "1" 改為 "sentiment"。
   - 這一步假設之前將 DataFrame 寫入 CSV 文件時，列名被設置為了 "0" 和 "1"。
3. **顯示前三行數據**：

   ```
   df.head(3)
   ```
   - 使用 `head` 方法顯示 DataFrame 的前三行數據，用於檢查數據讀取的正確性和完整性。

執行結果：

當這段程式碼執行完畢後，會將 `movie_data.csv` 文件中的數據讀取到 DataFrame `df` 中，然後重新命名列名為 "review" 和 "sentiment"。最後，輸出 DataFrame 的前三行數據，用來確認數據讀取的情況。

注意事項：

- 確保 `movie_data.csv` 文件存在於當前的工作目錄中，或者指定正確的文件路徑。
- 如果列名已經是 "review" 和 "sentiment"，則不需要進行重命名操作。
- 如果數據集較大，可能需要一些時間來完成數據讀取操作，具體時間取決於計算機性能和數據大小。

這段程式碼通常用於數據準備階段，確保從 CSV 文件中正確地讀取數據並準備進一步的數據處理或分析工作。

In [11]:
```
df.shape
```

Out[11]:
```
(50000, 2)
```

這行程式碼是用來查看 DataFrame `df` 的形狀（shape），即其行數和列數。讓我來解釋一下這行程式碼的作用：

程式碼解釋：

```
df.shape
```

- `df` 是一個 Pandas DataFrame 對象，代表從 CSV 文件中讀取的影評數據。
- `shape` 是 DataFrame 的一個屬性，用來返回包含行數和列數的元組（rows, columns）。

執行結果：

當這行程式碼執行後，將會返回一個包含兩個元素的元組，第一個元素代表 DataFrame 的行數（即樣本數），第二個元素代表 DataFrame 的列數（即特徵數）。

例如，如果執行結果為 `(50000, 2)`，則表示 DataFrame `df` 共有 50000 行（樣本數）和 2 列（特徵數）。

注意事項：

- 確保在執行此行程式碼之前，DataFrame `df` 已經被正確讀取並填充。
- 如果 `df` 為空或者未正確讀取數據，可能會返回 `(0, 0)` 或類似的結果。
- 此行程式碼通常用於查看數據的維度和大小，以確保數據準備階段的正確性和完整性。

這樣，你可以通過這行程式碼來快速了解 DataFrame `df` 的整體大小和結構。

Note

If you have problems with creating the `movie_data.csv`, you can find a download a zip archive at https://github.com/rasbt/machine-learning-book/tree/main/ch08/

Introducing the bag-of-words model

...

Transforming documents into feature vectors

By calling the fit_transform method on CountVectorizer, we just constructed the vocabulary of the bag-of-words model and transformed the following three sentences into sparse feature vectors:

1. The sun is shining
2. The weather is sweet
3. The sun is shining, the weather is sweet, and one and one is two

In [11]:
```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer

count = CountVectorizer()
docs = np.array([
        'The sun is shining',
        'The weather is sweet',
        'The sun is shining, the weather is sweet, and one and one is two'])
bag = count.fit_transform(docs)
```

這段程式碼使用了 `CountVectorizer` 從文本數據中構建詞袋（Bag of Words）。讓我來解釋一下這段程式碼的具體作用和每個步驟的含義：

程式碼解釋：

1. **導入必要的庫**：

   ```
   import numpy as np
   from sklearn.feature_extraction.text import CountVectorizer
   ```
   - `numpy` 是 Python 中用於數值計算的核心庫。
   - `CountVectorizer` 是 Scikit-learn 中用於將文本轉換成詞頻矩陣的類。
2. **創建文本數據**：

   ```
   docs = np.array([
           'The sun is shining',
           'The weather is sweet',
           'The sun is shining, the weather is sweet, and one and one is two'])
   ```
   - 創建了一個包含三個文本樣本的 numpy 數組 `docs`。
3. **初始化 CountVectorizer**：

   ```
   count = CountVectorizer()
   ```
   - 創建了一個 `CountVectorizer` 對象，這將用於將文本轉換為詞頻矩陣。
4. **構建詞袋模型**：

   ```
   bag = count.fit_transform(docs)
   ```
   - 使用 `fit_transform` 方法將文本數據 `docs` 轉換為詞袋模型。
   - `fit_transform` 方法首先擬合模型（根據文本建立詞彙表），然後將文本轉換為詞頻矩陣。
   - `bag` 是一個稀疏矩陣（sparse matrix），其形狀為 (3, 9)，表示有三個文本樣本和九個不重複的詞彙。

執行結果：

在執行後，`CountVectorizer` 將會生成一個詞彙表（vocabulary），並且將每個文本轉換為詞頻矩陣。下面是詞彙表和詞頻矩陣的具體內容：

- **詞彙表**：

  ```
  {'and': 0, 'is': 1, 'one': 2, 'shining': 3, 'sun': 4, 'sweet': 5, 'the': 6, 'two': 7, 'weather': 8}
  ```
  - 共有九個不重複的詞彙，分別是文本中出現的所有單詞。

- **詞頻矩陣**（轉換後的稀疏矩陣）：

```
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [2 3 2 1 1 1 2 1 1]]
```

  - 每行代表一個文本樣本，每列代表一個詞彙在該文本中的詞頻。
  - 例如，第一行 `[0 1 0 1 1 0 1 0 0]` 表示第一個文本中，'and' 出現 0 次，'is' 出現 1 次，依此類推。

這樣，這段程式碼成功地將文本轉換為了詞頻矩陣，方便後續進行文本特徵提取和機器學習建模。

Now let us print the contents of the vocabulary to get a better understanding of the underlying concepts:

In [12]:
```
print(count.vocabulary_)
```

```
{'the': 6, 'sun': 4, 'is': 1, 'shining': 3, 'weather': 8, 'sweet': 5, 'and': 0, 'one': 2, 'two': 7}
```

這個結果顯示了 `CountVectorizer` 對象 `count` 的詞彙表（vocabulary），每個單詞對應的索引位置。讓我來解釋一下這個結果的含義：

解釋結果：

- `count.vocabulary_` 是 `CountVectorizer` 對象的一個屬性，它返回一個字典，字典的鍵是詞彙（單詞），值是該詞彙在詞頻矩陣中的索引位置（從0開始）。
- 打印結果如下所示：

```
{'the': 6, 'sun': 4, 'is': 1, 'shining': 3, 'weather': 8, 'sweet': 5, 'and': 0, 'one': 2, 'two': 7}
```

  - `'the': 6`：'the' 這個詞在詞彙表中的索引位置是 6。
  - `'sun': 4`：'sun' 這個詞在詞彙表中的索引位置是 4。
  - 依此類推，其他單詞的索引位置也都如此顯示。

注意事項：

- 詞彙表的形成基於 `CountVectorizer` 對文本數據的分析，它會將文本中的所有單詞收集起來，並且按照它們在文本中出現的頻率和順序來建立詞彙表。
- 這個詞彙表在後續的機器學習建模或者其他文本分析任務中很有用，因為它能幫助確定每個單詞在特徵向量中的位置，從而進行有效的數據轉換和分析。

這個功能對於理解文本數據在 `CountVectorizer` 中如何表示為特徵非常重要。

As we can see from executing the preceding command, the vocabulary is stored in a Python dictionary, which maps the unique words to integer indices. Next let us print the feature vectors that we just created:

Each index position in the feature vectors shown here corresponds to the integer values that are stored as dictionary items in the CountVectorizer vocabulary. For example, the first feature at index position 0 resembles the count of the word "and", which only occurs in the last document, and the word "is" at index position 1 (the 2nd feature in the document vectors) occurs in all three sentences. Those values in the feature vectors are also called the raw term frequencies: *tf (t,d)*—the number of times a term t occurs in a document *d*.

In [13]:
```
print(bag.toarray())
```

```
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [2 3 2 1 1 1 2 1 1]]
```

當你使用 `print(bag.toarray())` 來顯示詞頻矩陣 `bag` 的內容時，結果顯示如下：

```
plaintext
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [2 3 2 1 1 1 2 1 1]]
```

這個二維數組反映了三個文本樣本中每個單詞的詞頻。讓我逐個元素來解釋：

解釋每個元素：

- **第一行 [0 1 0 1 1 0 1 0 0]**：
  - `0`：表示該文本中詞彙 `'and'` 的詞頻為 0。
  - `1`：表示該文本中詞彙 `'is'` 的詞頻為 1。
  - `0`：表示該文本中詞彙 `'one'` 的詞頻為 0。
  - `1`：表示該文本中詞彙 `'shining'` 的詞頻為 1。

    - `1`：表示該文本中詞彙 `'sun'` 的詞頻為 1。
    - `0`：表示該文本中詞彙 `'sweet'` 的詞頻為 0。
    - `1`：表示該文本中詞彙 `'the'` 的詞頻為 1。
    - `0`：表示該文本中詞彙 `'two'` 的詞頻為 0。
    - `0`：表示該文本中詞彙 `'weather'` 的詞頻為 0。
- **第二行 [0 1 0 0 0 1 1 0 1]**：
    - `0`：表示該文本中詞彙 `'and'` 的詞頻為 0。
    - `1`：表示該文本中詞彙 `'is'` 的詞頻為 1。
    - `0`：表示該文本中詞彙 `'one'` 的詞頻為 0。
    - `0`：表示該文本中詞彙 `'shining'` 的詞頻為 0。
    - `0`：表示該文本中詞彙 `'sun'` 的詞頻為 0。
    - `1`：表示該文本中詞彙 `'sweet'` 的詞頻為 1。
    - `1`：表示該文本中詞彙 `'the'` 的詞頻為 1。
    - `0`：表示該文本中詞彙 `'two'` 的詞頻為 0。
    - `1`：表示該文本中詞彙 `'weather'` 的詞頻為 1。
- **第三行 [2 3 2 1 1 1 2 1 1]**：
    - `2`：表示該文本中詞彙 `'and'` 的詞頻為 2。
    - `3`：表示該文本中詞彙 `'is'` 的詞頻為 3。
    - `2`：表示該文本中詞彙 `'one'` 的詞頻為 2。
    - `1`：表示該文本中詞彙 `'shining'` 的詞頻為 1。
    - `1`：表示該文本中詞彙 `'sun'` 的詞頻為 1。
    - `1`：表示該文本中詞彙 `'sweet'` 的詞頻為 1。
    - `2`：表示該文本中詞彙 `'the'` 的詞頻為 2。
    - `1`：表示該文本中詞彙 `'two'` 的詞頻為 1。
    - `1`：表示該文本中詞彙 `'weather'` 的詞頻為 1。

每個元素的意義：

- 每一個數字代表了對應文本中特定單詞的出現次數。例如，第一行 `[0 1 0 1 1 0 1 0 0]` 表示第一個文本中，單詞 `'is'` 出現了一次，單詞 `'shining'` 和 `'sun'` 各出現了一次，而其他單詞則沒有出現。
- 這樣的表示方式能夠幫助我們理解文本數據在計算機上如何轉換為可處理的數值格式，從而進行文本分析和機器學習建模。

Assessing word relevancy via term frequency-inverse document frequency

In [14]:
```
np.set_printoptions(precision=2)
```

`np.set_printoptions(precision=2)` 是 NumPy 提供的一個函數，它用來設定數字在輸出時的顯示精度。這個函數中的 `precision=2` 參數設定了浮點數的顯示精度為兩位小數。

詳細解釋：

在數據分析和科學計算中，我們經常會處理包含浮點數的數據。預設情況下，NumPy 會顯示浮點數的所有有效位數，但有時候我們希望將顯示精度限制在特定的位數以提高可讀性和節省空間。

- **precision=2**：這個參數告訴 NumPy 只顯示每個浮點數的兩位小數。

使用例子：

假設我們有一個包含浮點數的 NumPy 數組 `arr`：

```
import numpy as np

# 定義一個例子數組
arr = np.array([1.123456789, 2.345678901, 3.567890123])

# 設定打印選項，顯示兩位小數
np.set_printoptions(precision=2)

# 打印數組
print(arr)
```

輸出：

設置了精度為兩位小數後，打印數組 `arr` 的輸出會是：

```
[1.12 2.35 3.57]
```

使用場景：

這在需要控制數字顯示精度的情況下非常有用，特別是在數據分析、科學計算、機器學習和工程應用中，通常我們不需要或不希望看到過多的小數位數，因此使用這個函數可以方便地調整輸出格式。

When we are analyzing text data, we often encounter words that occur across multiple documents from both classes. Those frequently occurring words typically don't contain useful or discriminatory information. In this subsection, we will learn about a useful technique called term frequency-inverse document frequency (tf-idf) that can be used to downweigh those frequently occurring words in the feature vectors. The tf-idf can be defined as the product of the term frequency and the inverse document frequency:

$$\text{tf-idf}(t,d)=\text{tf (t,d)}\times \text{idf}(t,d)$$
Here the tf(t, d) is the term frequency that we introduced in the previous section, and the inverse document frequency *idf(t, d)* can be calculated as:

$$\text{idf}(t,d) = \text{log}\frac{n_d}{1+\text{df}(d, t)},$$
where n_d is the total number of documents, and *df(d, t)* is the number of documents *d* that contain the term *t*. Note that adding the constant 1 to the denominator is optional and serves the purpose of assigning a non-zero value to terms that occur in all training examples; the log is used to ensure that low document frequencies are not given too much weight.

Scikit-learn implements yet another transformer, the `TfidfTransformer`, that takes the raw term frequencies from `CountVectorizer` as input and transforms them into tf-idfs:

In [15]:
```
from sklearn.feature_extraction.text import TfidfTransformer

tfidf = TfidfTransformer(use_idf=True,
                         norm='l2',
                         smooth_idf=True)
print(tfidf.fit_transform(count.fit_transform(docs))
      .toarray())
```

```
[[0.   0.43 0.   0.56 0.56 0.   0.43 0.   0.  ]
 [0.   0.43 0.   0.   0.   0.56 0.43 0.   0.56]
 [0.5  0.45 0.5  0.19 0.19 0.19 0.3  0.25 0.19]]
```

這段代碼利用了 scikit-learn 中的 `TfidfTransformer` 類來將文本數據轉換為 TF-IDF（Term Frequency - Inverse Document Frequency）表示形式的矩陣。讓我來詳細解釋一下這段代碼的各個部分：

1. 導入必要的庫和類：

```
from sklearn.feature_extraction.text import TfidfTransformer
```
這行代碼導入了 `TfidfTransformer` 類，這是 scikit-learn 中用於計算 TF-IDF 的工具之一。

2. 創建 `TfidfTransformer` 對象：

```
tfidf = TfidfTransformer(use_idf=True, norm='l2', smooth_idf=True)
```
在這裡，我們創建了一個 `TfidfTransformer` 對象 `tfidf`，並通過參數指定了以下設置：

- `use_idf=True`：表示使用 IDF（Inverse Document Frequency）來加權 TF（Term Frequency）。
- `norm='l2'`：表示對每個文檔的 TF-IDF 向量進行歐氏長度標準化，使其長度為 1。
- `smooth_idf=True`：表示對 IDF 值進行平滑處理，防止分母為零的情況。

3. 轉換文本數據並打印 TF-IDF 矩陣：

```
print(tfidf.fit_transform(count.fit_transform(docs)).toarray())
```
這段代碼執行了以下步驟：

- 使用 `CountVectorizer` 對象 `count` 將文本數據 `docs` 轉換為詞頻矩陣。
- 將詞頻矩陣作為 `fit_transform` 方法的輸入，計算並返回 TF-IDF 矩陣。
- 最後，使用 `toarray()` 方法將 TF-IDF 矩陣轉換為 NumPy 數組並打印出來。

TF-IDF 矩陣的含義：

TF-IDF 矩陣中的每個元素代表了對應文檔中詞彙的 TF-IDF 值。TF-IDF 的計算方法將詞頻和逆文檔頻率結合，用來衡量一個詞彙在文檔集合中的重要性。通常情況下，TF-IDF 值越高，該詞彙在該文檔中越重要，但在整個文檔集合中出現次數越少。

這段代碼展示了如何使用 scikit-learn 轉換文本數據為機器學習模型可以處理的數值表示形式，這在自然語言處理和文本分析中非常常見和重要。

As we saw in the previous subsection, the word "is" had the largest term frequency in the 3rd document, being the most frequently occurring word. However, after transforming the same feature vector into tf-idfs, we see that the word "is" is now associated with a

relatively small tf-idf (0.45) in document 3 since it is also contained in documents 1 and 2 and thus is unlikely to contain any useful, discriminatory information.

However, if we'd manually calculated the tf-idfs of the individual terms in our feature vectors, we'd have noticed that the `TfidfTransformer` calculates the tf-idfs slightly differently compared to the standard textbook equations that we defined earlier. The equations for the idf and tf-idf that were implemented in scikit-learn are:

$$\text{idf} (t,d) = log\frac{1 + n_d}{1 + \text{df}(d, t)}$$
The tf-idf equation that was implemented in scikit-learn is as follows:

$$\text{tf-idf}(t,d) = \text{tf}(t,d) \times (\text{idf}(t,d)+1)$$
While it is also more typical to normalize the raw term frequencies before calculating the tf-idfs, the `TfidfTransformer` normalizes the tf-idfs directly.

By default (`norm='l2'`), scikit-learn's TfidfTransformer applies the L2-normalization, which returns a vector of length 1 by dividing an un-normalized feature vector *v* by its L2-norm:

$$v_{\text{norm}} = \frac{v}{||v||_2} = \frac{v}{\sqrt{v_{1}^{2} + v_{2}^{2} + \dots + v_{n}^{2}}} = \frac{v}{\big (\sum_{i=1}^{n} v_{i}^{2}\big)^\frac{1}{2}}$$
To make sure that we understand how `TfidfTransformer` works, let us walk through an example and calculate the tf-idf of the word "is" in the 3rd document.

The word "is" has a term frequency of 3 (tf = 3) in document 3 (d_3), and the document frequency of this term is 3 since the term "is" occurs in all three documents (df = 3). Thus, we can calculate the idf as follows:

$$\text{idf}("is", d_3) = log \frac{1+3}{1+3} = 0$$
Now in order to calculate the tf-idf, we simply need to add 1 to the inverse document frequency and multiply it by the term frequency:

$$\text{tf-idf}("is", d_3)= 3 \times (0+1) = 3$$

In [16]:
```
tf_is = 3
n_docs = 3
idf_is = np.log((n_docs+1) / (3+1))
tfidf_is = tf_is * (idf_is + 1)
print(f'tf-idf of term "is" = {tfidf_is:.2f}')
```

```
tf-idf of term "is" = 3.00
```

這段程式碼計算了單個詞彙 "is" 的 TF-IDF 值。讓我來解釋一下每一步的意義：

1. **`tf_is = 3`**：
   - `tf_is` 是指詞彙 "is" 在某個文檔中的詞頻（Term Frequency），這裡設置為 3。
2. **`n_docs = 3`**：
   - `n_docs` 是文檔的總數量，這裡設置為 3。
3. **計算 IDF（Inverse Document Frequency）**：
   - IDF 的計算公式為：[\text{idf_is} = \log \left(\frac{n_docs + 1}{\text{df_is} + 1} \right) + 1]
   - 其中，(\text{df_is}) 是包含詞彙 "is" 的文檔數量。這裡 (\text{df_is} = 3)，所以計算為： [\text{idf_is} = \log \left(\frac{3 + 1}{3 + 1} \right) = \log(1) = 0]
4. **計算 TF-IDF 值**：
   - TF-IDF 的計算公式為：[\text{tfidf_is} = \text{tf_is} \times (\text{idf_is} + 1)]
   - 將上面的結果代入： [\text{tfidf_is} = 3 \times (0 + 1) = 3]
5. **打印結果**：
   - 最後使用 `print()` 函數將計算得到的 TF-IDF 值格式化為兩位小數，並打印出來。

這段程式碼展示了如何計算單個詞彙的 TF-IDF 值，TF-IDF 是用於評估詞彙在文檔集合中重要性的一種技術，結合了詞頻和逆文檔頻率的概念。

If we repeated these calculations for all terms in the 3rd document, we'd obtain the following tf-idf vectors: [3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0 , 1.69, 1.29]. However, we notice that the values in this feature vector are different from the values that we obtained from the `TfidfTransformer` that we used previously. The final step that we are missing in this tf-idf calculation is the L2-normalization, which can be applied as follows:

$$\text{tfi-df}_{norm} = \frac{[3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0 , 1.69, 1.29]}{\sqrt{[3.39^2, 3.0^2, 3.39^2, 1.29^2, 1.29^2, 1.29^2, 2.0^2 , 1.69^2, 1.29^2]}}$$$$=[0.5, 0.45, 0.5, 0.19, 0.19, 0.19, 0.3, 0.25, 0.19]$$$$\Rightarrow \text{tfi-df}_{norm}("is", d3) = 0.45$$

As we can see, the results match the results returned by scikit-learn's `TfidfTransformer` (below). Since we now understand how tf-idfs are calculated, let us proceed to the next sections and apply those concepts to the movie review dataset.

In [17]:
```
tfidf = TfidfTransformer(use_idf=True, norm=None, smooth_idf=True)
```

```
raw_tfidf = tfidf.fit_transform(count.fit_transform(docs)).toarray()[-1]
raw_tfidf
```

Out[17]:
```
array([3.39, 3.  , 3.39, 1.29, 1.29, 1.29, 2.  , 1.69, 1.29])
```

這段程式碼使用了 `TfidfTransformer` 來計算文本數據 `docs` 的 TF-IDF 表示形式，並打印出最後一個文檔的原始 TF-IDF 值。讓我來解釋一下每一步的意義：

1. 創建 `TfidfTransformer` 對象：

`tfidf = TfidfTransformer(use_idf=True, norm=None, smooth_idf=True)`
在這裡創建了一個 `TfidfTransformer` 對象 `tfidf`，並指定了以下參數：

- `use_idf=True`：使用 IDF（Inverse Document Frequency）來加權 TF（Term Frequency）。
- `norm=None`：不對 TF-IDF 向量進行標準化。這意味著每個文檔的 TF-IDF 向量將保留其原始值。
- `smooth_idf=True`：對 IDF 值進行平滑處理，避免除以零的情況。

2. 轉換文本數據並獲取最後一個文檔的原始 TF-IDF 值：

`raw_tfidf = tfidf.fit_transform(count.fit_transform(docs)).toarray()[-1]`
這段程式碼的執行步驟如下：

- 使用 `CountVectorizer` 對象 `count` 將文本數據 `docs` 轉換為詞頻矩陣。
- 將詞頻矩陣作為 `fit_transform` 方法的輸入，計算並返回 TF-IDF 矩陣。
- 使用 `toarray()` 方法將 TF-IDF 矩陣轉換為 NumPy 數組。
- `[-1]` 選取了最後一個文檔的 TF-IDF 向量。

3. 打印 `raw_tfidf`：

`raw_tfidf`
這裡直接打印了 `raw_tfidf`，顯示了最後一個文檔的原始 TF-IDF 值。這個值是一個包含所有詞彙的向量，每個元素代表該詞彙在文檔中的 TF-IDF 值。

這段程式碼的目的是將文本數據轉換為 TF-IDF 表示形式，並展示了如何設置 `TfidfTransformer` 的參數來控制 TF-IDF 轉換的行為。

In [18]:
```
l2_tfidf = raw_tfidf / np.sqrt(np.sum(raw_tfidf**2))
l2_tfidf
```

Out[18]:
```
array([0.5 , 0.45, 0.5 , 0.19, 0.19, 0.19, 0.3 , 0.25, 0.19])
```

這段程式碼計算了最後一個文檔的 TF-IDF 向量的 L2 正規化版本。讓我來解釋一下每一步的意義：

步驟解釋：

1. **計算 TF-IDF 向量的 L2 正規化：**

   `l2_tfidf = raw_tfidf / np.sqrt(np.sum(raw_tfidf**2))`
   - `raw_tfidf` 是最後一個文檔的原始 TF-IDF 向量。
   - `np.sum(raw_tfidf**2)` 計算了 TF-IDF 向量中所有元素的平方和。
   - `np.sqrt()` 對這個平方和進行開方操作，得到 L2 正規化的分母。
   - 最後，將 `raw_tfidf` 中的每個元素除以這個開方值，從而得到 L2 正規化後的 TF-IDF 向量 `l2_tfidf`。

L2 正規化的意義：

L2 正規化是將向量除以其範數（或稱為歐氏長度），使得向量的範數變為 1。在 TF-IDF 向量的情境下，L2 正規化可以使得每個文檔的向量表示在空間中更加標準化和統一，適合於某些機器學習算法的使用，例如支持向量機（SVM）或者神經網絡模型。

這段程式碼的目的是計算 TF-IDF 向量的 L2 正規化版本，以便進一步的機器學習處理或者特徵選擇。

Cleaning text data

In [19]:
```
df.loc[0, 'review'][-50:]
```

Out[19]:
```
'is seven.<br /><br />Title (Brazil): Not Available'
```

這段程式碼的目的是顯示 DataFrame `df` 中第一行的 `review` 列的最後 50 個字符。讓我來解釋這段文字的繁體中文含義：

解釋：

`df.loc[0, 'review'][-50:]` 這段程式碼會顯示如下內容：

```
plaintext
'is seven.<br /><br />Title (Brazil): Not Available'
```

這個文本片段看起來是一段電影評論或描述的結尾部分。讓我們來分析它的內容：

- `'is seven.'`：這可能是一個句子或段落的結尾，但這段文字的上下文不明確。
- `<br /><br />`：這是 HTML 標籤，通常用於在網頁中插入空行或段落分隔符。
- `Title (Brazil): Not Available`：這部分可能是電影標題的語境，指出在巴西的標題是不可用的。

意義：

這段文字似乎是從某個電影評論或描述中提取的結尾部分。它展示了該評論的最後幾個字符，可能包含了對電影內容的總結或其他相關信息。

注意：

- 如果你希望獲得更多文本內容的上下文，需要查看 `df.loc[0, 'review']` 中的完整內容，而不僅僅是最後 50 個字符。
- 如果該文本包含 HTML 標籤，這些標籤可能需要進一步處理或解釋，具體取決於你處理該文本的上下文和目的。

這段程式碼有助於在探索和分析文本數據時，快速檢視或提取文本的特定部分，以便進行後續的處理或分析工作。

In [20]:
```
import re
def preprocessor(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
                           text)
    text = (re.sub('[\W]+', ' ', text.lower()) +
            ' '.join(emoticons).replace('-', ''))
    return text
```

In [21]:
```
preprocessor(df.loc[0, 'review'][-50:])
```

Out[21]:
```
'is seven title brazil not available'
```

這段程式碼的目的是對 DataFrame `df` 中第一行的 `review` 列的最後 50 個字符應用 `preprocessor` 函數。讓我來解釋一下這段程式碼的每個部分和結果：

解釋：

1. **`df.loc[0, 'review'][-50:]`**：
   - `df.loc[0, 'review']` 選取了 DataFrame `df` 中第一行的 `review` 列的內容。
   - `[-50:]` 使用切片操作選取了這段文字的最後 50 個字符。
2. **`preprocessor()` 函數**：
   - `preprocessor` 函數的定義在先前的程式碼中，它用來對文本進行預處理。
   - 在這裡，它會將文本中的 HTML 標籤去除，並將文本轉換為小寫，同時保留表情符號。
3. **結果**：
   - 執行 `preprocessor(df.loc[0, 'review'][-50:])` 後，返回的結果是 `'is seven title brazil not available'`。

意義：

這個結果顯示了經過預處理後的文本。原來的最後 50 個字符 `'is seven.<br /><br />Title (Brazil): Not Available'` 經過 `preprocessor` 函數的處理後，去除了 `<br />` 標籤，將文本轉換為小寫，並去除了其他非字母數字的符號，最終得到了清理過的文本 `'is seven title brazil not available'`。

注意：

- `preprocessor` 函數的效果取決於具體的實現方式，這裡的實現可能是為了示範目的，實際應用中可能需要根據具體情況進行調整和擴展。
- 在處理文本數據時，預處理步驟是非常重要的，它可以幫助確保文本數據的一致性和清晰度，同時減少模型訓練或分析中的噪音和干擾。

這段程式碼展示了如何將文本數據從原始形式轉換為更容易進行後續處理和分析的形式。

In [22]:
```
preprocessor("</a>This :) is :( a test :-)!")
```

```
Out[22]: 'this is a test :) :( :)'
```

這段程式碼展示了如何使用 `preprocessor` 函式來處理包含 HTML 標籤和表情符號的文字。讓我來解釋這個例子中 `preprocessor` 函式的作用及結果：

解釋：

1. **輸入文本：**

   `"</a>This :) is :( a test :-)!"`

   這是一段包含 HTML 標籤、表情符號和標點符號的文字。

2. **`preprocessor()` 函式：**

```python
import re

def preprocessor(text):
    text = re.sub('<[^>]*>', '', text)  # 去除 HTML 標籤
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)  # 提取表情符號
    text = (re.sub('[\W]+', ' ', text.lower()) +  # 去除非字母數字字符，轉換為小寫
            ' '.join(emoticons).replace('-', ''))  # 將表情符號連接到文本中
    return text
```

   - `re.sub('<[^>]*>', '', text)`：這一步去除所有的 HTML 標籤。
   - `emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)`：這一步找出文本中的所有表情符號。
   - `re.sub('[\W]+', ' ', text.lower())`：這一步將文本中的非字母數字字符替換為空格，並將文本轉換為小寫。
   - `' '.join(emoticons).replace('-', '')`：這一步將找出的表情符號連接到處理後的文本中，並且移除其中的連字符 `-`。

3. **結果：**

   執行 `preprocessor("</a>This :) is :( a test :-)!")` 後，返回的結果是 `'this is a test :) :( :)'`。

意義：

這個例子顯示了如何透過定義的 `preprocessor` 函式對包含 HTML 標籤和表情符號的原始文字進行預處理。最終的輸出文字 `'this is a test :) :( :)'` 是經過清理和轉換後的結果，已經去除了 HTML 標籤，將文本轉換為小寫，同時保留了表情符號並且將其連接到文本中。

這樣的預處理步驟有助於在後續的自然語言處理或文本分析任務中，確保文本數據的一致性和可處理性。

注意：

- `preprocessor` 函式中使用的正則表達式和替換規則可以根據具體的應用場景進行調整和擴展。
- 文本預處理在處理包含特殊符號或格式的文本時是非常常見和重要的步驟，它可以幫助減少文本數據中的噪音和增強特徵。

這段程式碼示範了如何使用 `preprocessor` 函數來處理包含 HTML 標籤和表情符號的文本。讓我來解釋一下這個例子的處理過程和結果：

解釋：

1. **輸入文本：**

   `"</a>This :) is :( a test :-)!"`

   這是一段包含 HTML 標籤、表情符號和標點符號的文本。

2. **`preprocessor()` 函數處理：**

```python
def preprocessor(text):
    text = re.sub('<[^>]*>', '', text)  # 去除 HTML 標籤
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)  # 提取表情符號
    text = (re.sub('[\W]+', ' ', text.lower()) +  # 去除非字母數字字符，轉換為小寫
            ' '.join(emoticons).replace('-', ''))  # 將表情符號連接到文本中
    return text
```

   - `re.sub('<[^>]*>', '', text)`：這一步去除了所有的 HTML 標籤。
   - `emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)`：這一步找出了文本中的所有表情符號。
   - `re.sub('[\W]+', ' ', text.lower())`：這一步將文本中的非字母數字字符替換為空格，並將文本轉換為小寫。
   - `' '.join(emoticons).replace('-', '')`：這一步將找出的表情符號連接到處理後的文本中，並且移除其中的連字符 `-`。

3. **結果：**

   執行 `preprocessor("</a>This :) is :( a test :-)!")` 後，返回的結果是 `'this is a test :) :('`。

意義：

這個例子展示了如何將原始的包含 HTML 標籤和表情符號的文本，通過定義的 `preprocessor` 函數進行清理和預處理，最終得到了更加乾淨和容易處理的文本形式。這樣的預處理步驟對於後續的自然語言處理任務或文本分析是非常有幫助的。

注意：

- `preprocessor` 函數中使用的正則表達式和替換規則可能需要根據具體的應用場景進行調整和擴展。
- 在處理文本時，預處理是一個重要的步驟，可以幫助確保文本數據的一致性和清晰度，同時減少模型訓練或分析中的噪音和干擾。

In [23]:
```
df['review'] = df['review'].apply(preprocessor)
```

這行程式碼的作用是將 DataFrame `df` 中的 `review` 列中的每一個文本都應用 `preprocessor` 函式進行預處理。

解釋：

- `df['review']`：選取了 DataFrame `df` 中的 `review` 列，該列包含了原始的文本數據。
- `.apply(preprocessor)`：這一方法會將 `preprocessor` 函式應用到 `review` 列的每一個元素上。具體來說，對於 `df['review']` 中的每一個文本，都會使用 `preprocessor` 函式來進行預處理，包括去除 HTML 標籤、轉換為小寫、去除非字母數字字符，同時保留表情符號。

這樣的預處理過程有助於清理文本數據，準備進一步的文本分析或者機器學習模型訓練。通過這樣的處理，可以確保文本數據的一致性和可處理性，同時減少噪音和不必要的特殊字符，從而提升後續分析的效果和準確性。

Processing documents into tokens

In [24]:
```
from nltk.stem.porter import PorterStemmer

porter = PorterStemmer()

def tokenizer(text):
    return text.split()


def tokenizer_porter(text):
    return [porter.stem(word) for word in text.split()]
```

這段程式碼示範了如何使用 NLTK 中的 Porter stemmer 來實現文本的詞幹提取（stemming）。讓我來解釋一下這段程式碼的作用和每個部分的意義：

解釋：

1. **引入 Porter stemmer：**

   ```
   from nltk.stem.porter import PorterStemmer

   porter = PorterStemmer()
   ```

   - `from nltk.stem.porter import PorterStemmer`：這裡引入了 NLTK 中的 Porter stemmer，用於詞幹提取。
   - `porter = PorterStemmer()`：創建了一個 Porter stemmer 的實例，用於後續的文本處理。
2. **定義 `tokenizer` 函式：**

   ```
   def tokenizer(text):
       return text.split()
   ```

   - `tokenizer` 函式簡單地將文本按空格分割成單詞列表。
3. **定義 `tokenizer_porter` 函式：**

   ```
   def tokenizer_porter(text):
       return [porter.stem(word) for word in text.split()]
   ```

   - `tokenizer_porter` 函式使用了先前創建的 Porter stemmer（`porter`）。
   - 對於輸入的 `text` 文本，首先將其按空格分割成單詞列表（`text.split()`）。
   - 然後對於列表中的每個單詞 `word`，應用 Porter stemmer 的 `stem` 方法來進行詞幹提取（`porter.stem(word)`）。
   - 返回處理後的詞幹列表。

意義：

這段程式碼的主要目的是定義兩個文本處理函式：

- `tokenizer` 函式將文本分割成單詞，並返回單詞列表。
- `tokenizer_porter` 函式進行更進一步的處理，它使用 Porter stemmer 對文本進行詞幹提取，即將每個單詞轉換為其基本形式。

這樣的文本處理過程在自然語言處理中很常見，特別是在需要將單詞轉換為它們的基本形式以減少詞彙的變體數量時。詞幹提取通常用於文本分析、信息檢索等任務中，有助於減少詞彙數量和提升模型的泛化能力。

注意：

- 在使用 `tokenizer_porter` 函式時，應留意詞幹提取可能會導致部分單詞的語義損失或信息遺失，因為它只是單純地截取單詞的詞幹，而不考慮詞語的上下文或語義。
- NLTK 的 Porter stemmer 是一個基於規則的詞幹提取器，可能不適用於所有語言或特定的文本處理需求，因此在實際應用中需根據情況選擇合適的詞幹提取工具或算法。

In [25]:
```
tokenizer('runners like running and thus they run')
```

Out[25]:
```
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

這段程式碼示範了 `tokenizer` 函式的使用，該函式將輸入的文本按照空格分割成單詞列表。下面是對輸入文本 `'runners like running and thus they run'` 的處理結果：

`['runners', 'like', 'running', 'and', 'thus', 'they', 'run']`
這裡列出了每個單詞，它們被正確地分割並以列表的形式返回。

In [26]:
```
tokenizer_porter('runners like running and thus they run')
```

Out[26]:
```
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

這裡顯示了 `tokenizer_porter` 函式對文本 `'runners like running and thus they run'` 的處理結果：

`['runner', 'like', 'run', 'and', 'thu', 'they', 'run']`

解釋：

1. **`tokenizer_porter` 函式的作用**：
   - `tokenizer_porter` 函式使用了 NLTK 中的 Porter stemmer 對文本進行詞幹提取。
   - 具體步驟是先將輸入的文本按空格分割成單詞列表。
   - 然後對於列表中的每個單詞，應用 Porter stemmer 的 `stem` 方法來進行詞幹提取。
2. **結果分析**：
   - `'runners'` 被詞幹提取為 `'runner'`，這裡 Porter stemmer 將複數形式 `'runners'` 簡化為其基本形式 `'runner'`。
   - `'running'` 被提取為 `'run'`，這是因為 Porter stemmer 試圖將 '-ing' 結尾的動詞形式轉換為其基本動詞形式。
   - `'thus'` 保持不變，因為它沒有相應的詞幹變化。
   - `'thu'` 是一個錯誤的結果，應該是 `'thus'` 的正確形式。可能是由於 Porter stemmer 的規則不完美，無法正確處理所有情況。
   - `'they'` 和 `'run'` 保持不變，因為它們的詞幹與其本身相同。

注意：

- Porter stemmer 是一個基於規則的詞幹提取器，它試圖通過去除詞彙的後綴來將單詞簡化為其基本形式。然而，它可能會導致某些詞幹的語義損失或者產生不正確的提取結果，這需要根據具體的應用場景進行評估和處理。
- 在自然語言處理中，選擇詞幹提取工具需根據文本的特性和任務的要求，有時需要考慮其他更複雜的詞形還原技術來達到更好的效果。

In [27]:
```
import nltk

nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/sebastian/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Out[27]:
```
True
```

```
import nltk
```
這行程式碼導入了 NLTK（Natural Language Toolkit），這是一個常用的自然語言處理工具庫。

```
nltk.download('stopwords')
```
這行程式碼使用 NLTK 下載了 "stopwords" 資源。在自然語言處理中，"stopwords" 指的是那些在文本中頻繁出現但通常無需進行分析或處理的常見詞語，比如 "and", "the", "of" 等。

這樣的資源通常被用來過濾掉文本中的常見詞語，以便更專注於分析文本中真正有意義和有價值的詞語。

In [28]:
```
from nltk.corpus import stopwords

stop = stopwords.words('english')
[w for w in tokenizer_porter('a runner likes running and runs a lot')
```

```
 if w not in stop]
```

Out[28]:
```
['runner', 'like', 'run', 'run', 'lot']
```

這段程式碼示範了如何使用 NLTK 的 stopwords 資源來過濾文本中的常見詞語，並且應用了先前定義的 `tokenizer_porter` 函式來進行詞幹提取。

解釋：

1. **引入 NLTK stopwords 資源：**

   ```
   from nltk.corpus import stopwords
   ```
   - `stopwords` 是 NLTK 中用於存放停用詞（stopwords）資源的模組。
2. **選擇英文停用詞：**

   ```
   stop = stopwords.words('english')
   ```
   - `stopwords.words('english')` 返回了英文停用詞列表。
3. **過濾文本中的停用詞：**

   ```
   [w for w in tokenizer_porter('a runner likes running and runs a lot') if w not in stop]
   ```
   - `tokenizer_porter('a runner likes running and runs a lot')` 使用了之前定義的 `tokenizer_porter` 函式對文本進行了詞幹提取。
   - `for w in ... if w not in stop` 這部分是列表推導式，用於遍歷處理後的詞幹列表，只保留不在停用詞列表中的詞語 `w`。

結果：

對於文本 `'a runner likes running and runs a lot'`，這段程式碼的執行結果為：

```
['runner', 'like', 'run', 'run', 'lot']
```
這裡已經過濾掉了 `'a'`, `'likes'`, `'running'`, `'and'`, `'runs'` 這些在停用詞列表中的詞語，並且將其餘詞語進行了詞幹提取處理。

Training a logistic regression model for document classification

Strip HTML and punctuation to speed up the GridSearch later:

In [29]:
```
X_train = df.loc[:25000, 'review'].values
y_train = df.loc[:25000, 'sentiment'].values
X_test = df.loc[25000:, 'review'].values
y_test = df.loc[25000:, 'sentiment'].values
```

這段程式碼從 DataFrame `df` 中選擇了訓練集和測試集的特徵（`review`）和目標（`sentiment`）資料，並將它們分配給變數 `X_train`, `y_train`, `X_test`, `y_test`。

說明：

1. **訓練集 `X_train`, `y_train`：**

   ```
   X_train = df.loc[:25000, 'review'].values
   y_train = df.loc[:25000, 'sentiment'].values
   ```
   - `X_train` 是訓練集的特徵資料，包含了前 25001 行（索引從 0 到 25000）的 `'review'` 列的內容。`.values` 方法將 DataFrame 列轉換為 NumPy 陣列。
   - `y_train` 是訓練集的目標資料，包含了前 25001 行的 `'sentiment'` 列的內容。
2. **測試集 `X_test`, `y_test`：**

   ```
   X_test = df.loc[25000:, 'review'].values
   y_test = df.loc[25000:, 'sentiment'].values
   ```
   - `X_test` 是測試集的特徵資料，包含了從第 25001 行開始到最後一行的 `'review'` 列的內容。
   - `y_test` 是測試集的目標資料，包含了從第 25001 行開始到最後一行的 `'sentiment'` 列的內容。

這樣做確保了從整個數據集中選擇了相等大小的訓練集和測試集，以便進行後續的機器學習模型訓練和評估。

In [30]:
```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import GridSearchCV

tfidf = TfidfVectorizer(strip_accents=None,
```

```
                        lowercase=False,
                        preprocessor=None)

"""
param_grid = [{'vect__ngram_range': [(1, 1)],
               'vect__stop_words': [stop, None],
               'vect__tokenizer': [tokenizer, tokenizer_porter],
               'clf__penalty': ['l1', 'l2'],
               'clf__C': [1.0, 10.0, 100.0]},
              {'vect__ngram_range': [(1, 1)],
               'vect__stop_words': [stop, None],
               'vect__tokenizer': [tokenizer, tokenizer_porter],
               'vect__use_idf':[False],
               'vect__norm':[None],
               'clf__penalty': ['l1', 'l2'],
               'clf__C': [1.0, 10.0, 100.0]},
              ]
"""

small_param_grid = [{'vect__ngram_range': [(1, 1)],
                     'vect__stop_words': [None],
                     'vect__tokenizer': [tokenizer, tokenizer_porter],
                     'clf__penalty': ['l2'],
                     'clf__C': [1.0, 10.0]},
                    {'vect__ngram_range': [(1, 1)],
                     'vect__stop_words': [stop, None],
                     'vect__tokenizer': [tokenizer],
                     'vect__use_idf':[False],
                     'vect__norm':[None],
                     'clf__penalty': ['l2'],
                 'clf__C': [1.0, 10.0]},
              ]

lr_tfidf = Pipeline([('vect', tfidf),
                     ('clf', LogisticRegression(solver='liblinear'))])

gs_lr_tfidf = GridSearchCV(lr_tfidf, small_param_grid,
                           scoring='accuracy',
                           cv=5,
                           verbose=1,
                           n_jobs=-1)
```

這段程式碼建立了一個使用管道 (Pipeline) 的機器學習模型，其中包含了 TF-IDF 特徵提取器和邏輯斯蒂迴歸分類器，並使用了網格搜索 (Grid Search) 來進行參數調優。

解釋：

1. **TF-IDF 特徵提取器和邏輯斯蒂迴歸分類器：**

   ```
   tfidf = TfidfVectorizer(strip_accents=None,
                           lowercase=False,
                           preprocessor=None)

   lr_tfidf = Pipeline([('vect', tfidf),
                        ('clf', LogisticRegression(solver='liblinear'))])
   ```

   - `TfidfVectorizer` 被用來將文本轉換為 TF-IDF 特徵表示。
     - `strip_accents=None` 表示不去除文本中的重音符號。
     - `lowercase=False` 表示不將文本轉換為小寫。
     - `preprocessor=None` 表示不使用額外的預處理函式。
   - `LogisticRegression` 是使用 'liblinear' solver 的邏輯斯蒂迴歸分類器，它被包含在管道中作為最後的分類器。
2. **網格搜索參數空間：**

   ```
   small_param_grid = [{'vect__ngram_range': [(1, 1)],
                        'vect__stop_words': [None],
                        'vect__tokenizer': [tokenizer, tokenizer_porter],
                        'clf__penalty': ['l2'],
                        'clf__C': [1.0, 10.0]},
                       {'vect__ngram_range': [(1, 1)],
                        'vect__stop_words': [stop, None],
                        'vect__tokenizer': [tokenizer],
                        'vect__use_idf':[False],
                        'vect__norm':[None],
                        'clf__penalty': ['l2'],
                        'clf__C': [1.0, 10.0]},
                       ]
   ```

   - `small_param_grid` 定義了兩個不同的參數組合，用於網格搜索。
   - 第一個組合將 TF-IDF 的參數設置為基本設置，並指定使用 `tokenizer` 和 `tokenizer_porter` 兩種不同的分詞器來比較效果。

- 第二個組合將 TF-IDF 的 `use_idf` 設置為 `False`，即不使用 IDF 調整，並且不進行正規化（`norm=None`）。

3. **建立網格搜索物件：**

```
gs_lr_tfidf = GridSearchCV(lr_tfidf, small_param_grid,
                           scoring='accuracy',
                           cv=5,
                           verbose=1,
                           n_jobs=-1)
```

- `GridSearchCV` 用於構建一個網格搜索物件，以尋找最佳參數組合。
- `lr_tfidf` 是要優化的管道物件。
- `small_param_grid` 是參數空間。
- `scoring='accuracy'` 表示使用準確度作為評分標準。
- `cv=5` 表示使用 5 折交叉驗證來評估每個參數組合的性能。
- `verbose=1` 表示打印詳細的日誌訊息。
- `n_jobs=-1` 表示使用所有可用的 CPU 核心來加速計算。

這樣設置的網格搜索將通過比較不同的 TF-IDF 參數設置和邏輯斯蒂迴歸的正則化參數來尋找最佳的文本分類模型配置。

Important Note about `n_jobs`

Please note that it is highly recommended to use `n_jobs=-1` (instead of `n_jobs=1`) in the previous code example to utilize all available cores on your machine and speed up the grid search. However, some Windows users reported issues when running the previous code with the `n_jobs=-1` setting related to pickling the tokenizer and tokenizer_porter functions for multiprocessing on Windows. Another workaround would be to replace those two functions, `[tokenizer, tokenizer_porter]`, with `[str.split]`. However, note that the replacement by the simple `str.split` would not support stemming.

In [31]:
```
gs_lr_tfidf.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits
```

Out[31]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('vect',
                                        TfidfVectorizer(lowercase=False)),
                                       ('clf',
                                        LogisticRegression(solver='liblinear'))]),
             n_jobs=-1,
             param_grid=[{'clf__C': [1.0, 10.0], 'clf__penalty': ['l2'],
                          'vect__ngram_range': [(1, 1)],
                          'vect__stop_words': [None],
                          'vect__tokenizer': [<function tokenizer at 0x150effb80>,
                                              <function tokenizer_porter at 0x150effc10>]},
                         {'...
                          'vect__stop_words': [['i', 'me', 'my', 'myself', 'we',
                                                'our', 'ours', 'ourselves',
                                                'you', "you're", "you've",
                                                "you'll", "you'd", 'your',
                                                'yours', 'yourself',
                                                'yourselves', 'he', 'him',
                                                'his', 'himself', 'she',
                                                "she's", 'her', 'hers',
                                                'herself', 'it', "it's", 'its',
                                                'itself', ...],
                                               None],
                          'vect__tokenizer': [<function tokenizer at 0x150effb80>],
                          'vect__use_idf': [False]}],
             scoring='accuracy', verbose=1)
```

這段程式碼的意思是使用網格搜索方法來訓練一個文本分類模型。具體來說：

- `gs_lr_tfidf` 是一個 `GridSearchCV` 物件，它將應用於已經定義好的管道 `lr_tfidf` 上。
- 網格搜索將在給定的參數網格 `small_param_grid` 中搜索最佳的參數組合。
- 訓練過程將使用訓練集 `X_train` 和相應的目標值 `y_train`，並使用交叉驗證來評估每個參數組合的性能。
- 目標是找到在給定評分標準下（這裡是準確度 `'accuracy'`），能夠最大化模型性能的最佳配置。

簡而言之，這段程式碼將尋找最佳的 TF-IDF 特徵提取器設置和邏輯斯蒂迴歸分類器設置，以構建一個性能最佳的文本分類模型。

In [32]:
```
print(f'Best parameter set: {gs_lr_tfidf.best_params_}')
print(f'CV Accuracy: {gs_lr_tfidf.best_score_:.3f}')
```

```
Best parameter set: {'clf__C': 10.0, 'clf__penalty': 'l2', 'vect__ngram_range': (1, 1), 'vect__stop_words': None
, 'vect__tokenizer': <function tokenizer at 0x150effb80>}
CV Accuracy: 0.897
```

這段程式碼用於印出網格搜索過程中找到的最佳參數組合及其對應的交叉驗證準確度。

- `{gs_lr_tfidf.best_params_}`：這個部分會顯示網格搜索中得到的最佳參數組合。這些參數是通過訓練和驗證多個模型後，使用交叉驗證確定的最佳組合。

- `{gs_lr_tfidf.best_score_:.3f}`：這個部分顯示了使用最佳參數組合進行交叉驗證時得到的最高準確度分數。`.3f` 表示保留三位小數來顯示準確度。這個分數可以幫助判斷模型的整體性能如何，越高表示模型在訓練集上表現越好。

總結來說，這段程式碼用於展示通過網格搜索找到的最佳模型參數組合以及該模型在交叉驗證中的準確度分數。

In [33]:
```
clf = gs_lr_tfidf.best_estimator_
print(f'Test Accuracy: {clf.score(X_test, y_test):.3f}')
```

```
Test Accuracy: 0.899
```

這段程式碼用於計算使用網格搜索過程中找到的最佳模型（`best_estimator_`）在測試集 `X_test` 上的準確度。

- `gs_lr_tfidf.best_estimator_` 是經過網格搜索後找到的表現最佳的模型。
- `clf.score(X_test, y_test)` 計算了這個最佳模型在測試集 `X_test` 上的準確度。這個準確度表示模型在未見過的數據上的預測表現。

`print(f'Test Accuracy: {clf.score(X_test, y_test):.3f}')` 這行程式碼將印出測試集上的準確度分數，保留三位小數來顯示。

Start comment:

Please note that `gs_lr_tfidf.best_score_` is the average k-fold cross-validation score. I.e., if we have a `GridSearchCV` object with 5-fold cross-validation (like the one above), the `best_score_` attribute returns the average score over the 5-folds of the best model. To illustrate this with an example:

In [34]:
```
from sklearn.linear_model import LogisticRegression
import numpy as np

from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score

np.random.seed(0)
np.set_printoptions(precision=6)
y = [np.random.randint(3) for i in range(25)]
X = (y + np.random.randn(25)).reshape(-1, 1)

cv5_idx = list(StratifiedKFold(n_splits=5, shuffle=False).split(X, y))

lr = LogisticRegression()
cross_val_score(lr, X, y, cv=cv5_idx)
```

Out[34]:
```
array([0.6, 0.4, 0.6, 0.2, 0.6])
```

這段程式碼演示了如何使用交叉驗證來評估 Logistic Regression 分類器在數據集上的表現。

1. `np.random.seed(0)`：設置隨機種子，以確保結果的可重現性。
2. `np.set_printoptions(precision=6)`：設置打印浮點數的精度為六位小數，這是為了保證輸出的準確性。
3. `y = [np.random.randint(3) for i in range(25)]`：生成一個包含 25 個隨機整數的列表，每個整數從 0 到 2 之間。
4. `X = (y + np.random.randn(25)).reshape(-1, 1)`：生成一個與 y 相同形狀的特徵矩陣 X，其中特徵值是 y 的每個元素加上從標準正態分佈中抽取的隨機數。
5. `cv5_idx = list(StratifiedKFold(n_splits=5, shuffle=False).split(X, y))`：使用 Stratified K-Fold 交叉驗證方法將數據集分為 5 折。`cv5_idx` 是一個包含 5 個元組的列表，每個元組包含訓練集和測試集的索引。
6. `lr = LogisticRegression()`：創建一個 Logistic Regression 分類器的實例。
7. `cross_val_score(lr, X, y, cv=cv5_idx)`：使用交叉驗證計算 Logistic Regression 分類器在每個折疊中的準確度分數。`cv=cv5_idx` 指定了使用的交叉驗證策略。

這段程式碼的目的是評估 Logistic Regression 分類器在給定數據集上的穩健性和泛化能力，通過多次交叉驗證來確保評估結果的可靠性。

By executing the code above, we created a simple data set of random integers that shall represent our class labels. Next, we fed the indices of 5 cross-validation folds (`cv3_idx`) to the `cross_val_score` scorer, which returned 5 accuracy scores -- these are the 5 accuracy values for the 5 test folds.

Next, let us use the `GridSearchCV` object and feed it the same 5 cross-validation sets (via the pre-generated `cv3_idx` indices):

In [35]:
```
from sklearn.model_selection import GridSearchCV

lr = LogisticRegression()
gs = GridSearchCV(lr, {}, cv=cv5_idx, verbose=3).fit(X, y)
```

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits
[CV 1/5] END ................................., score=0.600 total time=   0.0s
[CV 2/5] END ................................., score=0.400 total time=   0.0s
[CV 3/5] END ................................., score=0.600 total time=   0.0s
[CV 4/5] END ................................., score=0.200 total time=   0.0s
[CV 5/5] END ................................., score=0.600 total time=   0.0s
```

這段程式碼使用了 `GridSearchCV` 對 Logistic Regression 分類器進行超參數調優，使用了之前定義的 Stratified K-Fold 交叉驗證策略 `cv5_idx`。

- `LogisticRegression()` 創建了一個 Logistic Regression 分類器的實例 `lr`。
- `GridSearchCV` 是用於系統地搜索最佳超參數組合的工具。在這裡，它被初始化為 `GridSearchCV(lr, {}, cv=cv5_idx, verbose=3)`：
  - `lr` 是要調優的模型，這裡是 Logistic Regression。
  - `{}` 是一個空的參數格子，表示我們現在不設置任何超參數，會在後續的實作中設置。
  - `cv=cv5_idx` 指定了使用的交叉驗證策略，即我們之前定義的 Stratified K-Fold 交叉驗證。
  - `verbose=3` 控制詳細程度，數值越高，輸出的訊息就越詳細。
- `.fit(X, y)` 啟動了 GridSearchCV 對所有可能的超參數組合進行搜索和評估，並選擇最佳的參數組合來擬合模型，同時使用交叉驗證計算每個參數組合的性能。

這段程式碼的目的是找到最適合 Logistic Regression 模型的超參數設置，從而提升模型的性能和泛化能力。

As we can see, the scores for the 5 folds are exactly the same as the ones from `cross_val_score` earlier.

Now, the best_score_ attribute of the `GridSearchCV` object, which becomes available after `fit`ting, returns the average accuracy score of the best model:

In [36]:
```
gs.best_score_
```

Out[36]:
```
0.48
```

這個程式碼片段 `gs.best_score_` 是用來獲取在 GridSearchCV 中找到的最佳模型的最佳評分（最佳交叉驗證準確度）。

當你使用 `GridSearchCV` 對模型進行超參數調優後，通常會調用 `best_score_` 來獲取最佳模型的評分。這個分數是在交叉驗證過程中，使用最佳參數組合進行測試後得到的平均準確度。

在你的程式碼中，`gs.best_score_` 會返回 GridSearchCV 找到的最佳模型的交叉驗證分數，這個值是一個浮點數，表示模型在測試集上的平均準確度。

As we can see, the result above is consistent with the average score computed with `cross_val_score`.

In [37]:
```
lr = LogisticRegression()
cross_val_score(lr, X, y, cv=cv5_idx).mean()
```

Out[37]:
```
0.48
```

這段程式碼計算了使用 Logistic Regression 分類器進行交叉驗證的平均準確度。

- `LogisticRegression()` 創建了 Logistic Regression 分類器的實例 `lr`。
- `cross_val_score(lr, X, y, cv=cv5_idx)` 使用交叉驗證 `cv5_idx` 對 `X` 和 `y` 進行分類器性能的評估，返回了每個交叉驗證折的準確度。
- `.mean()` 計算了所有交叉驗證折的準確度的平均值，這個值即為整體交叉驗證的平均準確度。

因此，這段程式碼的目的是計算 Logistic Regression 分類器在給定數據集 `X` 和目標變量 `y` 上進行交叉驗證後的平均準確度。

End comment.

Working with bigger data - online algorithms and out-of-core learning

In [38]:
```
# This cell is not contained in the book but
# added for convenience so that the notebook
# can be executed starting here, without
# executing prior code in this notebook
```

```
import os
import gzip


if not os.path.isfile('movie_data.csv'):
    if not os.path.isfile('movie_data.csv.gz'):
        print('Please place a copy of the movie_data.csv.gz'
              'in this directory. You can obtain it by'
              'a) executing the code in the beginning of this'
              'notebook or b) by downloading it from GitHub:'
              'https://github.com/rasbt/machine-learning-book/'
              'blob/main/ch08/movie_data.csv.gz')
    else:
        with gzip.open('movie_data.csv.gz', 'rb') as in_f, \
                open('movie_data.csv', 'wb') as out_f:
            out_f.write(in_f.read())
```

這段程式碼確保我們能夠取得進一步處理所需的 CSV 檔案（`movie_data.csv`）。以下是程式碼的逐步解釋：

1. **檢查 'movie_data.csv' 檔案**：
   - 程式首先檢查是否存在未壓縮的 `'movie_data.csv'` 檔案（`os.path.isfile('movie_data.csv')`）。
2. **檢查 'movie_data.csv.gz' 壓縮檔案**：
   - 如果 `'movie_data.csv'` 不存在，接著檢查是否存在壓縮的 `'movie_data.csv.gz'` 檔案（`os.path.isfile('movie_data.csv.gz')`）。
3. **下載或解壓縮**：
   - 如果存在 `'movie_data.csv.gz'` 檔案，程式會打開這個壓縮檔案（`gzip.open('movie_data.csv.gz', 'rb')`）。
   - 程式讀取壓縮檔案的內容（`in_f.read()`），並將其寫入到未壓縮的 `'movie_data.csv'` 檔案中（`open('movie_data.csv', 'wb').write(in_f.read())`）。

這段程式碼的目的是確保無論是從 GitHub 下載還是從其他地方獲取，都能獲得未壓縮的 `'movie_data.csv'` 檔案，以便後續的資料分析或機器學習任務使用。

In [39]:
```
import numpy as np
import re
from nltk.corpus import stopwords


# The `stop` is defined as earlier in this chapter
# Added it here for convenience, so that this section
# can be run as standalone without executing prior code
# in the directory
stop = stopwords.words('english')


def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)
    text = re.sub('[\W]+', ' ', text.lower()) +\
        ' '.join(emoticons).replace('-', '')
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized


def stream_docs(path):
    with open(path, 'r', encoding='utf-8') as csv:
        next(csv)  # skip header
        for line in csv:
            text, label = line[:-3], int(line[-2])
            yield text, label
```

這段程式碼主要定義了幾個用於文本處理的函式，讓我們逐步解釋：

`tokenizer` 函式

```
def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)  # 移除HTML標籤
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)  # 找出表情符號
    text = re.sub('[\W]+', ' ', text.lower()) + ' '.join(emoticons).replace('-', '')  # 轉換成小寫並移除非單字字符
    tokenized = [w for w in text.split() if w not in stop]  # 分詞並移除停用詞
    return tokenized
```

- **功能**：該函式對輸入的文本進行預處理，具體操作包括：
  - 使用正則表達式（`re.sub('<[^>]*>', '', text)`）移除文本中的HTML標籤。

- 使用正則表達式（`re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)`）找出表情符號。
- 將文本轉換為小寫（`text.lower()`），並移除非單字字符（`re.sub('[\W]+', ' ', ...)`），然後將找到的表情符號連接到文本末尾。
- 將處理後的文本按空格分詞（`text.split()`），並過濾掉在停用詞列表（`stop`）中出現的詞彙。

`stream_docs` 函式

```
def stream_docs(path):
    with open(path, 'r', encoding='utf-8') as csv:
        next(csv)  # 跳過標頭行
        for line in csv:
            text, label = line[:-3], int(line[-2])
            yield text, label
```

- **功能**：這個函式是一個生成器函式，用於從指定路徑的CSV檔案中逐行讀取文本和標籤。
  - `open(path, 'r', encoding='utf-8')` 打開CSV檔案以進行讀取。
  - `next(csv)` 跳過CSV檔案的標頭行，開始從第一行文本開始讀取。
  - `for line in csv:` 逐行讀取CSV檔案中的文本和標籤。
  - `text, label = line[:-3], int(line[-2])` 從每一行中分離出文本和對應的整數標籤。
  - `yield text, label` 通過 `yield` 返回文本和標籤對，使函式成為生成器，允許在迭代過程中逐步生成文本和標籤對。

這些函式組合在一起可以用於對大型文本數據集進行處理和分析，特別是在自然語言處理和機器學習任務中。

In [40]:
```
next(stream_docs(path='movie_data.csv'))
```

Out[40]:
```
('"In 1974, the teenager Martha Moxley (Maggie Grace) moves to the high-class area of Belle Haven, Greenwich, C
onnecticut. On the Mischief Night, eve of Halloween, she was murdered in the backyard of her house and her murd
er remained unsolved. Twenty-two years later, the writer Mark Fuhrman (Christopher Meloni), who is a former LA
detective that has fallen in disgrace for perjury in O.J. Simpson trial and moved to Idaho, decides to investig
ate the case with his partner Stephen Weeks (Andrew Mitchell) with the purpose of writing a book. The locals sq
uirm and do not welcome them, but with the support of the retired detective Steve Carroll (Robert Forster) that
was in charge of the investigation in the 70\'s, they discover the criminal and a net of power and money to cov
er the murder.<br /><br />""Murder in Greenwich"" is a good TV movie, with the true story of a murder of a fift
een years old girl that was committed by a wealthy teenager whose mother was a Kennedy. The powerful and rich f
amily used their influence to cover the murder for more than twenty years. However, a snoopy detective and conv
icted perjurer in disgrace was able to disclose how the hideous crime was committed. The screenplay shows the i
nvestigation of Mark and the last days of Martha in parallel, but there is a lack of the emotion in the dramati
zation. My vote is seven.<br /><br />Title (Brazil): Not Available"',
 1)
```

這段程式碼會從 `movie_data.csv` 文件中讀取第一條文本和標籤對。讓我們逐步解釋這段程式碼的含義：

`stream_docs` 函式

首先，我們回顧一下 `stream_docs` 函式的定義：

```
def stream_docs(path):
    with open(path, 'r', encoding='utf-8') as csv:
        next(csv)  # 跳過標頭行
        for line in csv:
            text, label = line[:-3], int(line[-2])
            yield text, label
```

這個函式是一個生成器，用於從指定的 CSV 文件中逐行讀取文本和標籤對。

使用 `next` 取得第一條文本和標籤對

```
next(stream_docs(path='movie_data.csv'))
```

這段程式碼的功能是：

1. **呼叫 `stream_docs` 函式**：
   - 使用 `path='movie_data.csv'` 作為參數呼叫 `stream_docs` 函式。
   - `stream_docs` 會打開 `movie_data.csv` 文件並準備讀取其內容。
2. **跳過標頭行**：
   - 在 `stream_docs` 函式中，`next(csv)` 會跳過 CSV 文件的第一行，這通常是標頭行。
3. **讀取第一行**：
   - `for line in csv:` 會從文件中逐行讀取內容。由於這是生成器函式，它會在每次迭代中返回一對文本和標籤。
   - `text, label = line[:-3], int(line[-2])` 會提取當前行的文本部分（去掉最後三個字符）和標籤部分（倒數第二個字符轉換為整數）。
4. **返回第一對文本和標籤**：
   - `yield text, label` 會將第一對文本和標籤返回給呼叫者。

5. **取得第一對文本和標籤**：
    - 使用 `next` 函式取得生成器的第一個返回值，即第一對文本和標籤。

整體結果

該程式碼將從 `movie_data.csv` 中讀取第一條評論和其對應的情感標籤，並顯示它們。例如，如果第一行是：

```
"The movie was great!",1
```

那麼 `next(stream_docs(path='movie_data.csv'))` 將返回：

```
("The movie was great!", 1)
```

In [41]:
```
def get_minibatch(doc_stream, size):
    docs, y = [], []
    try:
        for _ in range(size):
            text, label = next(doc_stream)
            docs.append(text)
            y.append(label)
    except StopIteration:
        return None, None
    return docs, y
```

這段程式碼定義了一個名為 `get_minibatch` 的函式，用來從文件流中批量提取指定數量的文本和標籤。讓我們逐步解釋這段程式碼的含義：

`get_minibatch` 函式

```
def get_minibatch(doc_stream, size):
    docs, y = [], []
    try:
        for _ in range(size):
            text, label = next(doc_stream)
            docs.append(text)
            y.append(label)
    except StopIteration:
        return None, None
    return docs, y
```

參數：

- `doc_stream`：文件流生成器，會逐行返回文本和標籤對。
- `size`：整數，指定要提取的文本和標籤對的數量。

功能：

1. **初始化空列表**：
    - `docs, y = [], []` 初始化兩個空列表，一個用於存儲文本（`docs`），另一個用於存儲標籤（`y`）。
2. **嘗試提取指定數量的文本和標籤對**：
    - `try:` 開始一個嘗試塊，用於捕獲 `StopIteration` 異常（當文件流結束時觸發）。
    - `for _ in range(size):` 循環 `size` 次，以提取指定數量的文本和標籤對。
    - `text, label = next(doc_stream)` 使用 `next` 函式從 `doc_stream` 中提取下一對文本和標籤。
    - `docs.append(text)` 將提取的文本追加到 `docs` 列表。
    - `y.append(label)` 將提取的標籤追加到 `y` 列表。
3. **處理文件流結束情況**：
    - `except StopIteration:` 當文件流結束（即沒有更多的文本和標籤對可提取）時，會觸發 `StopIteration` 異常。
    - `return None, None` 在捕獲到 `StopIteration` 異常時，返回 `(None, None)` 表示沒有更多的文本和標籤對。
4. **返回提取的文本和標籤對**：
    - `return docs, y` 在成功提取指定數量的文本和標籤對後，返回 `docs` 和 `y` 列表。

使用範例

假設 `doc_stream` 是一個從 `stream_docs` 函式獲得的生成器，並且包含以下數據：

```
doc_stream = stream_docs('movie_data.csv')
```

我們可以調用 `get_minibatch` 來提取一個大小為 2 的小批量：

```
docs, labels = get_minibatch(doc_stream, 2)
```

這樣，我們會從 `doc_stream` 中獲得兩個文本和對應的標籤，並將其存儲在 `docs` 和 `labels` 列表中。如果文件流中的文本和標籤不足兩個，則返回 `(None, None)`。

In [42]:
```
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.linear_model import SGDClassifier


vect = HashingVectorizer(decode_error='ignore',
                         n_features=2**21,
                         preprocessor=None,
                         tokenizer=tokenizer)
```

這段程式碼設置了一個 `HashingVectorizer` 用於將文本轉換為數字特徵表示，並設置了一個 `SGDClassifier` 用於分類任務。讓我們分步解釋每一行的含義：

`HashingVectorizer`

`HashingVectorizer` 是一個用於將文本數據轉換為特徵向量的工具。這種轉換有助於機器學習模型理解和處理文本數據。`HashingVectorizer` 使用一種哈希技巧將文本轉換為固定長度的向量，這使得它非常高效且節省內存。

設置 `HashingVectorizer`

```
vect = HashingVectorizer(decode_error='ignore',
                         n_features=2**21,
                         preprocessor=None,
                         tokenizer=tokenizer)
```

- `decode_error='ignore'`：如果在解碼過程中遇到任何錯誤，則忽略它們。
- `n_features=2**21`：指定輸出向量的維度。`2**21` 意味著有 2 的 21 次方（即 2,097,152）個特徵。
- `preprocessor=None`：指定不使用任何預處理器。預處理器是用於在分詞之前處理原始文本的函數。
- `tokenizer=tokenizer`：指定使用自定義的分詞器。分詞器是將文本分割成單詞或令牌的函數。在此處，我們使用之前定義的 `tokenizer` 函數。

設置 `SGDClassifier`

`SGDClassifier` 是一種線性分類器，使用隨機梯度下降（SGD）來優化模型參數。SGD 是一種非常高效的優化算法，特別適合處理大型數據集。

總結

這段程式碼的目的是設置一個文本特徵提取器和一個分類模型，以便在後續步驟中用於文本分類任務。具體地說，`HashingVectorizer` 將文本轉換為特徵向量，而 `SGDClassifier` 將使用這些特徵向量進行分類。

整個設置過程如下：

1. 使用 `HashingVectorizer` 將文本轉換為特徵向量。
2. 使用 `SGDClassifier` 進行分類任務。

這樣的設置非常適合處理大型文本數據集，並且能夠在保持高效性的同時，進行精確的分類。

In [43]:
```
from distutils.version import LooseVersion as Version
from sklearn import __version__ as sklearn_version

clf = SGDClassifier(loss='log', random_state=1)


doc_stream = stream_docs(path='movie_data.csv')
```

這段程式碼分為三個部分：引入版本控制工具、設置分類器、設置文件流。下面對每個部分進行詳細解釋：

引入版本控制工具

```
from distutils.version import LooseVersion as Version
from sklearn import __version__ as sklearn_version
```

- `LooseVersion`：來自 `distutils.version` 模塊，用於版本比較。它允許我們以靈活的方式比較版本號。
- `__version__`：`sklearn` 庫的版本號。

這部分代碼的主要目的是確保我們可以檢查和比較當前使用的 `scikit-learn` 版本號，以便在需要時調整代碼的兼容性。

設置分類器

```
clf = SGDClassifier(loss='log', random_state=1)
```

- `SGDClassifier`：使用隨機梯度下降法進行優化的分類器。它非常適合處理大規模的數據集。
- `loss='log'`：設置損失函數為對數損失，即邏輯迴歸。
- `random_state=1`：設置隨機種子，以確保結果的可重現性。

這段代碼創建了一個使用對數損失（邏輯迴歸）的 `SGDClassifier`。這意味著分類器會使用邏輯迴歸模型進行預測。

設置文件流

```
doc_stream = stream_docs(path='movie_data.csv')
```

- `stream_docs`：是一個生成器函數，用於從 `movie_data.csv` 文件中逐行讀取數據。每次調用這個生成器，它會返回下一行的文本和標籤。
- `path='movie_data.csv'`：指定數據文件的路徑。

這段代碼的目的是設置一個文件流，使得我們可以逐行讀取 `movie_data.csv` 文件中的數據，而不是一次性將整個文件讀入內存。這種方法在處理大規模數據集時非常高效。

總結

這段程式碼的主要目的是設置分類器和文件流，為後續的文本分類任務做準備。具體步驟如下：

1. 引入版本控制工具，以便檢查和比較 `scikit-learn` 的版本。
2. 設置一個使用對數損失的隨機梯度下降分類器（SGDClassifier）。
3. 設置一個文件流生成器，從 `movie_data.csv` 文件中逐行讀取數據。

這些設置使得我們可以高效地處理大規模文本數據集，並使用邏輯迴歸模型進行分類。

In [44]:
```
import pyprind
pbar = pyprind.ProgBar(45)

classes = np.array([0, 1])
for _ in range(45):
    X_train, y_train = get_minibatch(doc_stream, size=1000)
    if not X_train:
        break
    X_train = vect.transform(X_train)
    clf.partial_fit(X_train, y_train, classes=classes)
    pbar.update()
```

```
0% [##############################] 100% | ETA: 00:00:00
Total time elapsed: 00:00:16
```

這段程式碼使用了進度條、文件流、特徵向量化和部分擬合來進行增量學習。以下是每個部分的詳細解釋：

進度條

```
import pyprind
pbar = pyprind.ProgBar(45)
```

- `pyprind`：是一個進度條庫，用於顯示進度條。
- `pbar = pyprind.ProgBar(45)`：創建一個包含 45 個步驟的進度條。

這部分代碼的目的是設置一個進度條，以便在後續的迴圈中顯示進度。

增量學習

```
classes = np.array([0, 1])
for _ in range(45):
    X_train, y_train = get_minibatch(doc_stream, size=1000)
    if not X_train:
        break
    X_train = vect.transform(X_train)
    clf.partial_fit(X_train, y_train, classes=classes)
    pbar.update()
```

1. **設置類別**：

   ```
   classes = np.array([0, 1])
   ```
   - 定義分類的目標類別，即 `0` 和 `1`。
2. **迴圈**：

   ```
   for _ in range(45):
   ```
   - 設置一個迴圈，將重複 45 次。
3. **讀取小批量數據**：

   ```
   X_train, y_train = get_minibatch(doc_stream, size=1000)
   ```

```
if not X_train:
    break
```

- 每次從文件流中讀取 1000 行數據，存儲在 `X_train` 和 `y_train` 中。
- 如果沒有更多數據可讀取，則跳出迴圈。

4. **向量化**：

```
X_train = vect.transform(X_train)
```

- 使用 `HashingVectorizer` 將文本數據轉換為特徵向量。

5. **部分擬合**：

```
clf.partial_fit(X_train, y_train, classes=classes)
```

- 使用 `SGDClassifier` 進行部分擬合。這是一種增量學習方法，可以逐步更新模型，而不是一次性訓練整個模型。

6. **更新進度條**：

```
pbar.update()
```

- 每次迴圈結束時更新進度條。

總結

這段程式碼的主要目的是進行增量學習，逐步讀取數據、向量化並進行部分擬合。具體步驟如下：

1. 設置進度條以顯示迴圈進度。
2. 定義分類目標類別。
3. 迴圈 45 次，每次讀取 1000 行數據。
4. 使用 `HashingVectorizer` 將文本數據轉換為特徵向量。
5. 使用 `SGDClassifier` 進行部分擬合，逐步更新模型。
6. 更新進度條以顯示進度。

這些步驟使得我們可以有效地處理大規模數據集，並進行增量學習。

In [45]:
```
X_test, y_test = get_minibatch(doc_stream, size=5000)
X_test = vect.transform(X_test)
print(f'Accuracy: {clf.score(X_test, y_test):.3f}')
```

```
Accuracy: 0.868
```

這段代碼用於評估增量學習模型的準確性。以下是詳細解釋：

1. **讀取測試數據**：

```
X_test, y_test = get_minibatch(doc_stream, size=5000)
```

- 使用 `get_minibatch` 函數從文件流 `doc_stream` 中讀取 5000 行測試數據。
- `X_test` 包含測試文本數據，`y_test` 包含相應的標籤。

2. **向量化測試數據**：

```
X_test = vect.transform(X_test)
```

- 使用 `HashingVectorizer` 將測試文本數據轉換為特徵向量，以與訓練數據保持一致的格式。

3. **評估模型準確性**：

```
print(f'Accuracy: {clf.score(X_test, y_test):.3f}')
```

- 使用 `SGDClassifier` 的 `score` 方法計算模型在測試數據上的準確性。
- `clf.score(X_test, y_test)` 返回模型的準確性，`:.3f` 表示將結果格式化為小數點後三位。

總結

這段代碼的目的在於從文件流中讀取 5000 行測試數據，將其轉換為特徵向量，並計算模型在這些數據上的準確性。步驟如下：

1. 從文件流中讀取 5000 行測試數據。
2. 使用 `HashingVectorizer` 將測試數據轉換為特徵向量。
3. 使用訓練好的增量學習模型評估測試數據的準確性。
4. 輸出準確性結果，格式化為小數點後三位。

In [46]:
```
clf = clf.partial_fit(X_test, y_test)
```

這段代碼用於更新增量學習模型，使其包含測試數據集。以下是詳細解釋：

```
clf = clf.partial_fit(X_test, y_test)
```

詳細說明

1. **增量學習模型**：

- 這裡的 `clf` 是一個使用 `SGDClassifier`（隨機梯度下降分類器）的增量學習模型。增量學習模型可以在新的數據到來時進

行更新，而不需要從頭重新訓練模型。

2. `partial_fit` 方法：
   - `partial_fit` 方法用於增量地訓練模型。這意味著可以一次只使用一部分數據來更新模型，而不是一次性地使用整個數據集。
   - `partial_fit(X_test, y_test)` 將使用測試數據集 `X_test` 和對應的標籤 `y_test` 來更新模型 `clf`。
   - 通過這種方式，模型可以在不丟失之前學習到的知識的情況下，增量地學習新的數據。
3. **保存更新後的模型**：
   - `clf = clf.partial_fit(X_test, y_test)` 這行代碼會返回更新後的模型，並重新賦值給 `clf`。這確保了 `clf` 保存的是最新的、已經包含測試數據的模型。

為何這樣做？

- **增強模型的泛化能力**：通過將測試數據納入訓練，模型可以學習到更多樣的數據分布，從而可能提高其泛化能力。
- **在線學習**：這種方法特別適合於在線學習場景，在這種場景中，模型需要隨著新數據的到來不斷更新。

總結

這段代碼通過 `partial_fit` 方法使用測試數據更新增量學習模型 `clf`，使其能夠學習新的數據分布，從而提高模型的泛化能力和在線學習能力。

Topic modeling

Decomposing text documents with Latent Dirichlet Allocation

Latent Dirichlet Allocation with scikit-learn

In [47]:
```
import pandas as pd

df = pd.read_csv('movie_data.csv', encoding='utf-8')

# the following is necessary on some computers:
df = df.rename(columns={"0": "review", "1": "sentiment"})

df.head(3)
```

Out[47]:

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

這段代碼用於讀取包含電影評論和情感標籤的CSV文件，並顯示前3行的數據。以下是詳細解釋：

```
import pandas as pd

# 讀取包含電影評論和情感標籤的CSV文件
df = pd.read_csv('movie_data.csv', encoding='utf-8')

# 重新命名列名，這在某些電腦上是必要的
df = df.rename(columns={"0": "review", "1": "sentiment"})

# 顯示數據框的前3行
df.head(3)
```

詳細說明

1. **導入Pandas庫**：
   - `import pandas as pd` 用於導入Pandas庫，這是Python中常用的數據分析庫。
2. **讀取CSV文件**：
   - `df = pd.read_csv('movie_data.csv', encoding='utf-8')` 使用 `read_csv` 函數讀取名為 `movie_data.csv` 的CSV文件，並將其內容存儲在名為 `df` 的數據框中。指定 `encoding='utf-8'` 是為了確保正確處理文件中的字符編碼。
3. **重新命名列名**：
   - `df = df.rename(columns={"0": "review", "1": "sentiment"})` 重新命名數據框的列名。原始列名可能是數字（如 "0" 和 "1"），這段代碼將它們更改為更具描述性的名稱 "review" 和 "sentiment"。這在某些電腦上是必要的，因為不同環境下讀取的CSV文件的列名可能有所不同。
4. **顯示前3行數據**：

- `df.head(3)` 顯示數據框的前3行數據，便於檢查和確認數據是否正確讀取和處理。

總結

這段代碼通過Pandas庫讀取電影評論數據，並對列名進行重新命名，最後顯示前3行數據以便檢查和確認。這是數據預處理的初步步驟，為後續的數據分析或機器學習模型訓練做好準備。

In [48]:
```
from sklearn.feature_extraction.text import CountVectorizer

count = CountVectorizer(stop_words='english',
                        max_df=.1,
                        max_features=5000)
X = count.fit_transform(df['review'].values)
```

這段代碼使用 `CountVectorizer` 來對電影評論文本進行詞頻向量化，並設置了一些參數來過濾和限制特徵。以下是詳細解釋：

```
from sklearn.feature_extraction.text import CountVectorizer

count = CountVectorizer(stop_words='english',
                        max_df=.1,
                        max_features=5000)
X = count.fit_transform(df['review'].values)
```

詳細說明

1. **導入 CountVectorizer**：
   - `from sklearn.feature_extraction.text import CountVectorizer` 從 `sklearn` 庫中導入 `CountVectorizer` 類，這是一種將文本數據轉換為詞頻向量的工具。
2. **創建 CountVectorizer 實例**：
   - `count = CountVectorizer(stop_words='english', max_df=.1, max_features=5000)` 創建 `CountVectorizer` 的實例，並設置了一些參數來過濾和限制特徵：
     - `stop_words='english'`：排除常見的英文停用詞（如 "the", "and", "is" 等），這些詞對文本分類沒有太大幫助。
     - `max_df=.1`：排除在超過 10% 的文檔中出現的詞，這些詞可能是過於常見的詞，對區分文檔沒有太大幫助。
     - `max_features=5000`：最多保留 5000 個特徵詞（即，最常見的 5000 個詞），這樣可以減少向量的維度，提高計算效率。
3. **擬合並轉換文本數據**：
   - `X = count.fit_transform(df['review'].values)` 對 `df['review'].values`（即評論文本）進行擬合並轉換。這一步將文本數據轉換為詞頻矩陣 `X`，其中每行對應一個評論，每列對應一個特徵詞，矩陣中的值表示每個詞在評論中出現的次數。

總結

這段代碼使用 `CountVectorizer` 將電影評論轉換為詞頻向量，並通過設置 `stop_words`，`max_df` 和 `max_features` 參數來過濾掉常見的無意義詞和高頻詞，並限制特徵數量。這樣可以減少噪音，提高後續文本分析或機器學習模型的性能。

In [49]:
```
from sklearn.decomposition import LatentDirichletAllocation

lda = LatentDirichletAllocation(n_components=10,
                                random_state=123,
                                learning_method='batch')
X_topics = lda.fit_transform(X)
```

這段程式碼使用了 `LatentDirichletAllocation` 類來進行潛在狄利克雷分配（Latent Dirichlet Allocation, LDA）主題建模。以下是詳細解釋：

```
from sklearn.decomposition import LatentDirichletAllocation

lda = LatentDirichletAllocation(n_components=10,
                                random_state=123,
                                learning_method='batch')
X_topics = lda.fit_transform(X)
```

詳細說明：

1. **導入 `LatentDirichletAllocation`**：
   - `from sklearn.decomposition import LatentDirichletAllocation`：從 `sklearn` 庫中導入 `LatentDirichletAllocation` 類，這是一種用於主題建模的技術，通常用於文本數據分析。
2. **創建 `LatentDirichletAllocation` 實例**：

- `lda = LatentDirichletAllocation(n_components=10, random_state=123, learning_method='batch')`：創建了一個 `LatentDirichletAllocation` 的實例。
  - `n_components=10`：指定了模型中潛在的主題數量。在這個例子中，設置為 10，表示我們希望從文檔中識別出 10 個主題。
  - `random_state=123`：設置隨機數生成器的種子，以確保結果的可重現性。
  - `learning_method='batch'`：設置了 LDA 模型的學習方法為 'batch' 方法，即使用批量 EM（期望最大化）算法來學習模型參數。

3. **擬合並轉換**：

- `X_topics = lda.fit_transform(X)`：將詞頻矩陣 `X` 擬合到 LDA 模型中，並轉換為 `X_topics`。這一步將文檔表示為主題分佈向量，每個文檔對應一個主題分佈，其中每個元素表示文檔中的主題相對權重。

總結：

這段程式碼利用 `LatentDirichletAllocation` 對詞頻矩陣 `X` 進行主題建模，識別出文檔中的潛在主題。通過設置 `n_components` 指定主題數量，以及其他參數如 `random_state` 和 `learning_method` 來配置模型。主題建模通常用於文本數據的無監督學習，有助於理解大量文本數據中的隱藏主題結構。

In [50]:
```
lda.components_.shape
```

Out[50]:
```
(10, 5000)
```

`lda.components_.shape` 返回一個元組，其中包含 LDA 模型中每個主題的詞頻分佈。具體來說，這個元組的形狀是 `(n_components, n_features)`，其中：

- `n_components` 是模型中指定的主題數量，即 `LatentDirichletAllocation` 的 `n_components` 參數設置值。
- `n_features` 是詞頻矩陣 `X` 中的特徵數量，即 `CountVectorizer` 或 `HashingVectorizer` 的 `max_features` 參數設置值。

因此，`lda.components_.shape` 返回的元組告訴我們每個主題的詞頻分佈在模型中的形狀。

In [54]:
```
n_top_words = 5
feature_names = count.get_feature_names_out()

for topic_idx, topic in enumerate(lda.components_):
    print(f'Topic {(topic_idx + 1)}:')
    print(' '.join([feature_names[i]
                    for i in topic.argsort()\
                        [:-n_top_words - 1:-1]]))
```

```
Topic 1:
worst minutes awful script stupid
Topic 2:
family mother father children girl
Topic 3:
american war dvd music tv
Topic 4:
human audience cinema art sense
Topic 5:
police guy car dead murder
Topic 6:
horror house sex girl woman
Topic 7:
role performance comedy actor performances
Topic 8:
series episode war episodes tv
Topic 9:
book version original read novel
Topic 10:
action fight guy guys cool
```

這段程式碼用於顯示每個潛在狄利克雷分配（Latent Dirichlet Allocation, LDA）主題中的前幾個詞彙。以下是對這段程式碼的分段解釋：

1. `n_top_words = 5`：設置了每個主題要顯示的前幾個詞彙數量。

2. `feature_names = count.get_feature_names_out()`：這裡的 `count` 是之前使用的 `CountVectorizer` 物件，它已經設置了停用詞、最大文件頻率以及最大特徵數量。`get_feature_names_out()` 方法返回了文本特徵的名稱列表。

3. `for topic_idx, topic in enumerate(lda.components_)`：迭代 `lda.components_`，這是一個二維數組，其中每一行表示一個主題，每一列表示一個詞彙在該主題中的權重。

4. `print(f'Topic {(topic_idx + 1)}:')`：打印當前主題的索引。

5. `print(' '.join([feature_names[i] for i in topic.argsort()[:-n_top_words - 1:-1]]))`：

- `topic.argsort()[:-n_top_words - 1:-1]` 這部分是為了取得權重最高的前 `n_top_words` 個詞彙的索引，根據它們

的重要性排序。
- `[feature_names[i] for i in ...]` 這部分則是將這些索引轉換為對應的詞彙，並將它們以空格分隔。
- `print(...)` 最終將這些詞彙列印出來，顯示每個主題的主題詞彙。

這樣的過程幫助我們理解每個 LDA 主題所涵蓋的主題內容，這些主題通常是從文本中發現的潛在語義結構。

Based on reading the 5 most important words for each topic, we may guess that the LDA identified the following topics:

1. Generally bad movies (not really a topic category)
2. Movies about families
3. War movies
4. Art movies
5. Crime movies
6. Horror movies
7. Comedies
8. Movies somehow related to TV shows
9. Movies based on books
10. Action movies

To confirm that the categories make sense based on the reviews, let's plot 5 movies from the horror movie category (category 6 at index position 5):

In [52]:
```
horror = X_topics[:, 5].argsort()[::-1]

for iter_idx, movie_idx in enumerate(horror[:3]):
    print(f'\nHorror movie #{(iter_idx + 1)}:')
    print(df['review'][movie_idx][:300], '...')
```

```
Horror movie #1:
House of Dracula works from the same basic premise as House of Frankenstein from the year before; namely that Un
iversal's three most famous monsters; Dracula, Frankenstein's Monster and The Wolf Man are appearing in the movi
e together. Naturally, the film is rather messy therefore, but the fact that ...

Horror movie #2:
Okay, what the hell kind of TRASH have I been watching now? "The Witches' Mountain" has got to be one of the mos
t incoherent and insane Spanish exploitation flicks ever and yet, at the same time, it's also strangely compelli
ng. There's absolutely nothing that makes sense here and I even doubt there  ...

Horror movie #3:
<br /><br />Horror movie time, Japanese style. Uzumaki/Spiral was a total freakfest from start to finish. A fun
freakfest at that, but at times it was a tad too reliant on kitsch rather than the horror. The story is difficul
t to summarize succinctly: a carefree, normal teenage girl starts coming fac ...
```

這段程式碼的目的是找出在主題為「恐怖電影」的 Latent Dirichlet Allocation (LDA) 主題模型中，具有最高相關度的前三篇電影評論。以下是對程式碼的分段解釋：

1. **`horror = X_topics[:, 5].argsort()[::-1]`** :
   - `X_topics[:, 5]` 表示從 `X_topics` 中取出所有樣本在第 5 個主題上的主題分布。
   - `argsort()` 函數對這些主題分布進行排序，返回排序後的索引。由於我們想找出具有最高相關度的樣本，因此排序結果會從最高分數到最低分數。
   - `[::-1]` 是用來將排序後的結果反轉，以便讓得分最高的樣本索引排在最前面。
2. **`for iter_idx, movie_idx in enumerate(horror[:3]):`** :
   - `enumerate(horror[:3])` 用來迭代 `horror` 陣列中的前三個元素，即前三個最相關的樣本索引。
   - `iter_idx` 是迭代的索引，從 0 開始。
   - `movie_idx` 是迭代中當前的電影索引，即樣本的索引。
3. **`print(f'\nHorror movie #{(iter_idx + 1)}:')`** :
   - 在每次迭代中，打印出目前迭代的序號，例如第一篇、第二篇、第三篇。
4. **`print(df['review'][movie_idx][:300], '...')`** :
   - `df['review'][movie_idx]` 取出 `df` DataFrame 中 `review` 欄位中第 `movie_idx` 篇電影評論的內容。
   - `[:300]` 表示只顯示評論的前 300 個字符，以避免印出過多文本。
   - `'...'` 是為了指示文本被截斷了，並非顯示完整評論。

這樣的操作可以幫助我們快速查看 LDA 模型識別的主題中最相關的樣本，這裡是針對「恐怖電影」主題的範例。

Using the preceeding code example, we printed the first 300 characters from the top 3 horror movies and indeed, we can see that the reviews -- even though we don't know which exact movie they belong to -- sound like reviews of horror movies, indeed. (However, one might argue that movie #2 could also belong to topic category 1.)

Summary

...

Readers may ignore the next cell.

In [53]:
```
! python ../.convert_notebook_to_script.py --input ch08.ipynb --output ch08.py
```

```
[NbConvertApp] WARNING | Config option `kernel_spec_manager_class` not recognized by `NbConvertApp`.
[NbConvertApp] Converting notebook ch08.ipynb to script
[NbConvertApp] Writing 24007 bytes to ch08.py
```

這個命令的目的是將名為 `ch08.ipynb` 的 Jupyter 筆記本文件轉換為一個名為 `ch08.py` 的獨立 Python 腳本文件。這樣做的好處是可以將筆記本中的程式碼和文本內容轉換為一個完整的、可執行的 Python 腳本，以便在其他 Python 環境中運行，或者用於版本控制和共享。

轉換後的 `.py` 文件通常包含了原始筆記本中所有的程式碼塊和相應的 Markdown 註釋，這樣可以更方便地管理和重複使用程式碼。