AI VIET NAM
@aivietnam.edu.vn

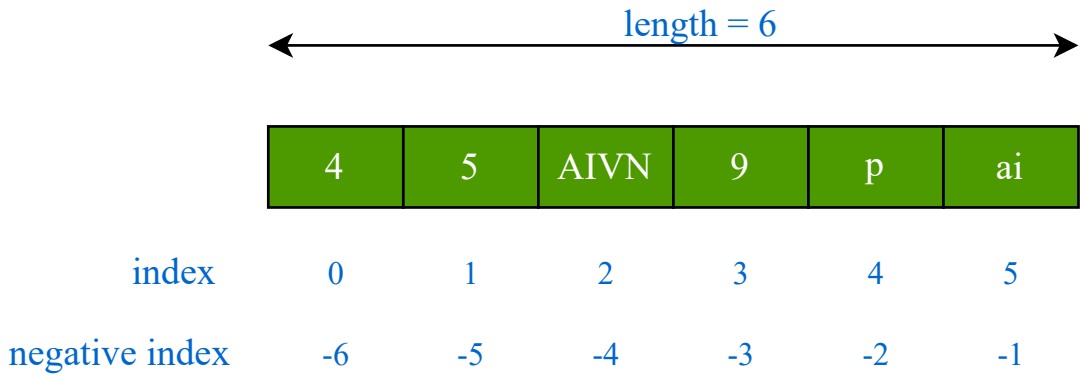
AI VIET NAM – AI COURSE 2025

Tutorial: Ứng dụng của List trong các bài toán thực tế

Duc-Huy Tran, Phuc-Thinh Nguyen, và Quang-Vinh Dinh
(Viết dựa trên tài liệu tổng hợp từ bạn Đoàn Chiến Thắng - AIO2025)

I. Giới thiệu

List (Danh sách) là một trong những cấu trúc dữ liệu cốt lõi và linh hoạt nhất trong Python. Về bản chất, list là một tập hợp các phần tử có thứ tự (ordered) và có thể thay đổi (mutable). Nhờ khả năng lưu trữ các phần tử thuộc nhiều kiểu dữ liệu khác nhau (số, chuỗi, thậm chí là một list khác) trong cùng một danh sách, nó đã trở thành công cụ không thể thiếu cho lập trình viên. Trong lĩnh vực trí tuệ nhân tạo, list chính là nền tảng cho vô số thuật toán và ứng dụng, từ việc lưu trữ dữ liệu đầu vào, quản lý các tham số của mô hình cho đến việc xử lý kết quả đầu ra.



Hình 1: Minh họa về List trong Python.

Tuy nhiên, việc học và vận dụng `list` trong bối cảnh thực tế không chỉ đơn thuần là hiểu cú pháp hoặc ghi nhớ phương thức cơ bản như `append()` hay `pop()`. Trong các dự án ML/DL, chúng ta thường phải xử lý khối lượng lớn dữ liệu thô từ: ảnh, chuỗi văn bản, đến các nhãn phân loại. Ở mỗi bước – từ tiền xử lý dữ liệu, chia tập huấn luyện, mã hóa nhãn cho đến tính toán

độ chính xác – cấu trúc `list` thường xuyên được sử dụng để thao tác, tổ chức và xử lý thông tin. Việc nắm vững cách sử dụng `list` hiệu quả chính là bước đệm quan trọng để xử lý dữ liệu linh hoạt, tối ưu hóa pipeline học máy, và chuẩn bị tốt hơn cho việc làm việc với các thư viện chuyên sâu như `NumPy`, `Pandas` hoặc `PyTorch`.

Tài liệu này được thiết kế gồm một chuỗi 10 bài tập thực hành, mỗi bài đặt trong một ngữ cảnh cụ thể thường gặp trong các dự án Machine Learning. Mỗi tình huống đều đi kèm phần mô tả bối cảnh thực tế, yêu cầu bài toán rõ ràng và các *test case* để kiểm tra kết quả. Toàn bộ nội dung tập trung vào việc sử dụng `list` để giải quyết vấn đề, qua đó giúp người học hiểu rõ bản chất và ứng dụng của `list` trong môi trường lập trình chuyên nghiệp.

Mục lục

II. Bài tập thực hành

Bài 1: Chuẩn hóa độ dài chuỗi đầu vào bằng kỹ thuật Padding

1. Bối cảnh sử dụng

Trong nhiều bài toán học sâu, đặc biệt là trong lĩnh vực xử lý ngôn ngữ tự nhiên (NLP) hoặc phân tích chuỗi thời gian (Time Series), dữ liệu đầu vào thường có chiều dài không đồng nhất. Chẳng hạn, một tập văn bản có thể bao gồm các câu với số lượng từ khác nhau — có câu chỉ 5 từ, trong khi câu khác có thể dài tới 15 từ. Tuy nhiên, để có thể xử lý dữ liệu một cách hiệu quả trong quá trình huấn luyện mô hình, đặc biệt là khi sử dụng GPU, ta cần đảm bảo rằng tất cả các phần tử trong một batch đều có kích thước bằng nhau để thuận tiện trong việc biểu diễn dưới dạng tensor.

Đây là lúc kỹ thuật **padding** (đệm) trở nên cần thiết. Padding là thao tác thêm các giá trị cố định (thường là số 0) vào cuối các chuỗi ngắn hơn, sao cho tất cả các chuỗi trong cùng một batch đều có độ dài bằng chuỗi dài nhất. Kỹ thuật này không chỉ giúp chuẩn hóa đầu vào mà còn tối ưu hóa khả năng xử lý song song trong các mô hình học sâu.

2. Bài tập thực tiễn

Mô tả yêu cầu bài toán

Cho một danh sách gồm nhiều chuỗi số nguyên có độ dài khác nhau (list các list), hãy viết một hàm thực hiện việc:

- Tìm độ dài lớn nhất trong tất cả các chuỗi.
- Đệm 0 vào các chuỗi ngắn hơn sao cho tất cả chuỗi đều có cùng độ dài như chuỗi dài nhất.

Input: Một list chứa nhiều list con, mỗi list con là một chuỗi số nguyên.
Output: Một list mới với các chuỗi đã được đệm 0 để có cùng độ dài.

Ví dụ:

```
Input: [[1, 2], [3, 4, 5], [6]]
Output: [[1, 2, 0], [3, 4, 5], [6, 0, 0]]
```

Hàm padding

```python
def pad_sequences(sequences):
    # Bước 1: Tìm độ dài lớn nhất của các chuỗi
    # Bước 2: Khởi tạo danh sách lưu kết quả

```

```
    # Bước 3: Với mỗi chuỗi trong danh sách:
        # Nếu chuỗi ngắn, thêm các phần tử đệm (ví dụ: 0) vào cuối
        # Thêm chuỗi đã được đệm vào danh sách kết quả

    # Bước 4: Trả về danh sách các chuỗi sau khi padding
    return ...
```

Test case

```
# Test 1: Các chuỗi có độ dài khác nhau
assert pad_sequences([[1, 2], [3, 4, 5], [6]]) == [[1, 2, 0], [3, 4, 5], [6, 0, 0]],
                                                  "Test 1 Failed"

# Test 2: Tất cả các chuỗi đã có cùng độ dài, không cần padding
assert pad_sequences([[1, 1], [2, 2], [3, 3]]) == [[1, 1], [2, 2], [3, 3]],
                                                   "Test 2 Failed"

# Test 3: Một chuỗi rỗng và một chuỗi dài, chuỗi rỗng được đệm
assert pad_sequences([[1, 2, 3], []]) == [[1, 2, 3], [0, 0, 0]], "Test 3 Failed"

# Test 4: Chỉ có một chuỗi, giữ nguyên
assert pad_sequences([[7, 8]]) == [[7, 8]], "Test 4 Failed"

# Test 5: Danh sách đầu vào rỗng, trả về danh sách rỗng
assert pad_sequences([]) == [], "Test 5 Failed"
```

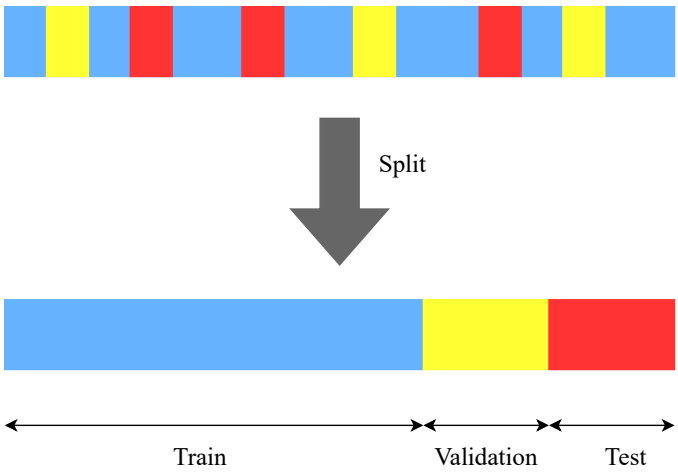
Bài 2: Phân tách dữ liệu: Train - Validation - Test

1. Bối cảnh sử dụng

Trong bất kỳ quy trình huấn luyện mô hình Machine Learning nào, việc đánh giá hiệu suất mô hình một cách khách quan là điều tối quan trọng. Để đạt được điều đó, dữ liệu ban đầu cần được phân chia hợp lý thành ba tập chính:

- **Tập huấn luyện (Training set)**: dùng để mô hình học các đặc trưng từ dữ liệu.
- **Tập kiểm định (Validation set)**: dùng để tinh chỉnh các siêu tham số (*hyperparameters*) và chọn mô hình tốt nhất.
- **Tập kiểm tra (Test set)**: dùng để đánh giá hiệu suất cuối cùng của mô hình trên dữ liệu mà nó *chưa từng được nhìn thấy*.

Cách phân chia này giúp đảm bảo rằng quy trình huấn luyện và đánh giá không bị lệ thuộc vào cùng một tập dữ liệu, tránh hiện tượng học vẹt (overfitting) và mang lại kết quả phản ánh đúng năng lực khái quát hóa của mô hình.



Hình 2: Minh họa chia dữ liệu train-val-test.

2. Bài tập thực tiễn

Mô tả yêu cầu bài toán

Cho một danh sách dữ liệu và hai tỉ lệ số thực biểu diễn phần trăm cho tập huấn luyện và kiểm định, hãy viết một hàm thực hiện việc:

- Chia danh sách dữ liệu thành ba phần: `train`, `validation`, và `test`.
- Tập `test` được tính là phần còn lại sau khi trừ hai tỉ lệ trên.

Input:

- Một list dữ liệu bất kỳ.
- Hai số thực `train_ratio` và `val_ratio` ($0 \leq$ tỉ lệ ≤ 1).

Output: Ba list con tương ứng với tập `train`, `validation`, và `test`.

Ví dụ:

```
Input: data = list(range(10)), train_ratio = 0.6, val_ratio = 0.2
Output: ([0, 1, 2, 3, 4, 5], [6, 7], [8, 9])
```

Hàm split data

```python
def split_dataset(data, train_ratio, val_ratio):
    # Bước 1: Tính tổng số phần tử trong tập dữ liệu

    # Bước 2: Tính số phần tử cho mỗi tập dựa theo tỉ lệ
```

```
        # train_end = số phần tử của tập huấn luyện
        # val_end = vị trí kết thúc của tập kiểm định

    # Bước 3: Dùng slicing để tạo 3 tập con:
        # train_set = từ đầu đến train_end
        # val_set = từ train_end đến val_end
        # test_set = từ val_end đến hết

    # Bước 4: Trả về ba tập train, validation, test
    return ...
```

Test case

```
# Test 1: Chia tỷ lệ chuẩn
dataset = list(range(100))
train, val, test = split_dataset(dataset, 0.7, 0.15)
assert (len(train), len(val), len(test)) == (70, 15, 15), "Test 1 Failed"

# Test 2: Dữ liệu nhỏ, chia đều, kiểm tra làm tròn
dataset = ['a', 'b', 'c']
train, val, test = split_dataset(dataset, 0.5, 0.5)
assert (len(train), len(val), len(test)) == (1, 1, 1), "Test 2 Failed"

# Test 3: Không có tập validation
dataset = list(range(5))
train, val, test = split_dataset(dataset, 0.8, 0.0)
assert (train, val, test) == (list(range(4)), [], [4]), "Test 3 Failed"

# Test 4: Dữ liệu rỗng
dataset = []
train, val, test = split_dataset(dataset, 0.6, 0.2)
assert (train, val, test) == ([], [], []), "Test 4 Failed"

# Test 5: Tổng tỉ lệ nhỏ hơn 1, phần còn lại là test
dataset = list(range(10))
train, val, test = split_dataset(dataset, 0.2, 0.3)  # 2, 3, 5
assert (train, val, test) == \
(list(range(2)), list(range(2, 5)), list(range(5, 10))), "Test 5 Failed"
```

Bài 3: Làm Phẳng Token Để Tạo Vocabulary NLP

1. Bối cảnh sử dụng

Trong xử lý ngôn ngữ tự nhiên (NLP), dữ liệu văn bản thường được biểu diễn dưới dạng danh sách các câu, trong đó mỗi câu là một danh sách các từ đã được tách (token hóa). Tuy nhiên, để xây dựng các mô hình học máy hoặc học sâu trên văn bản, ta cần mã hóa toàn bộ dữ liệu này thành các dạng số. Một bước quan trọng đầu tiên là xây dựng từ điển (vocabulary) — tập hợp tất cả các từ duy nhất xuất hiện trong dữ liệu.

Để làm điều này, ta cần "làm phẳng" (flatten) cấu trúc dữ liệu lồng nhau — từ danh sách các câu (mỗi câu là một danh sách từ) thành một danh sách đơn chứa tất cả từ trong toàn bộ tập văn bản. Việc làm phẳng này tạo nền tảng cho quá trình rút trích từ điển, gán chỉ số cho từng từ, và phục vụ các bước mã hóa tiếp theo.

2. Bài tập thực tiễn

Mô tả yêu cầu bài toán

Cho một kho văn bản được biểu diễn dưới dạng danh sách các câu, trong đó mỗi câu là một danh sách các từ (token), hãy viết một hàm thực hiện việc:

- Làm phẳng toàn bộ danh sách lồng nhau thành một danh sách chứa tất cả các từ có trong kho văn bản, theo đúng thứ tự ban đầu.

Input: Một list các list, mỗi list con là một danh sách các từ.
Output: Một list chứa tất cả các từ, theo thứ tự xuất hiện ban đầu.

Ví dụ:

```
Input: [["I", "love", "AI"], ["NLP", "is", "fun"]]
Output: ["I", "love", "AI", "NLP", "is", "fun"]
```

Hàm flatten_tokens

```python
def flatten_tokens(corpus):
    # Bước 1: Khởi tạo danh sách rỗng để lưu kết quả

    # Bước 2: Duyệt qua từng câu trong kho văn bản (corpus)

        # Bước 2.1: Duyệt qua từng từ trong câu

            # Bước 2.2: Thêm từ đó vào danh sách kết quả

    # Bước 3: Trả về danh sách kết quả đã được làm phẳng
    return ...
```

Test case

```python
# Test 1: Hai câu, mỗi câu có nhiều từ, làm phẳng toàn bộ
assert flatten_tokens([["hello", "world"], ["this", "is", "a", "test"]]) == \
["hello", "world", "this", "is", "a", "test"], "Test 1 Failed"

# Test 2: Một câu ngắn và một câu 1 từ, kiểm tra xử lý danh sách không đều
assert flatten_tokens([["a", "b"], ["c"]]) == ["a", "b", "c"], "Test 2 Failed"

# Test 3: Kho văn bản rỗng, đầu ra là list rỗng
assert flatten_tokens([]) == [], "Test 3 Failed"

```

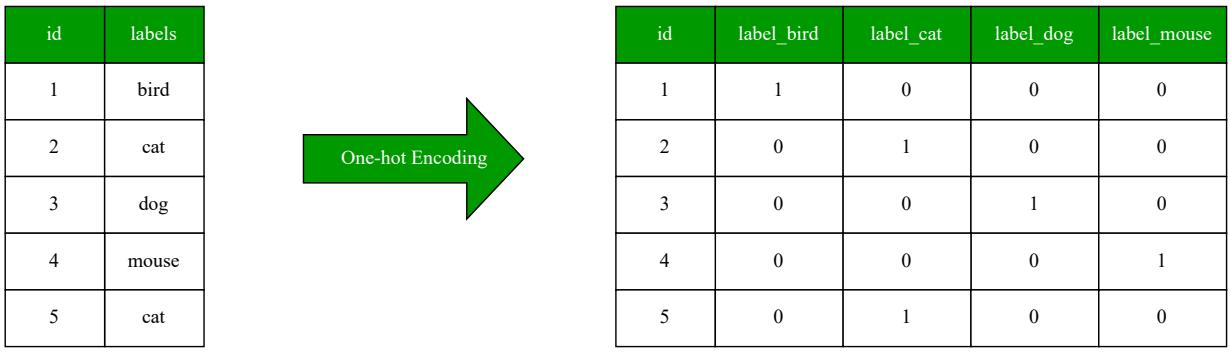
```
# Test 4: Chỉ có một câu, kiểm tra hoạt động đơn lẻ
assert flatten_tokens([["single", "sentence"]]) == ["single", "sentence"], \
"Test 4 Failed"

# Test 5: Nhiều câu có độ dài khác nhau, kiểm tra tính ổn định của kết quả
assert flatten_tokens([["deep", "learning"], ["rocks"], ["NLP", "is", "fun"]]) == \
["deep", "learning", "rocks", "NLP", "is", "fun"], "Test 5 Failed"
```

Bài 4: Xử Lý Nhãn Văn Bản Bằng Kỹ Thuật One-Hot

1. Bối cảnh sử dụng

Trong các mô hình học máy, đặc biệt là các mô hình học có giám sát, dữ liệu đầu ra thường là các nhãn thuộc loại rời rạc (categorical labels), ví dụ như: "cat", "dog", hoặc "bird". Tuy nhiên, hầu hết các mô hình học máy không thể xử lý trực tiếp chuỗi ký tự mà yêu cầu dữ liệu đầu vào và đầu ra phải ở dạng số.

Để giải quyết điều này, *One-Hot Encoding* được sử dụng như một kỹ thuật tiền xử lý giúp chuyển đổi các nhãn phân loại thành dạng vector nhị phân. Cụ thể, với n lớp khác nhau, mỗi nhãn được mã hóa thành một vector độ dài n, trong đó chỉ có một phần tử có giá trị 1 (tương ứng với lớp của nhãn), các phần tử còn lại là 0.



Hình 3: Minh họa với input và output của One-hot Encoding.

2. Bài tập thực tiễn

Mô tả yêu cầu bài toán

Cho một danh sách các nhãn văn bản và danh sách tất cả các lớp có thể có (đã được sắp xếp), hãy viết một hàm thực hiện:

- Chuyển đổi từng nhãn thành một vector one-hot có độ dài bằng số lớp.
- Trong mỗi vector, chỉ có một phần tử là 1 ở vị trí tương ứng với lớp của nhãn đó, các phần tử còn lại là 0.

Input:

- `labels`: list các nhãn văn bản (ví dụ: `["cat", "dog"]`)
- `classes`: list tất cả các lớp có thể có, đã được sắp xếp thứ tự (ví dụ: `["bird", "cat", "dog"]`)

Output: List các vector one-hot tương ứng với từng nhãn đầu vào.

Ví dụ:

```
Input:
labels = ["cat", "dog"]
classes = ["bird", "cat", "dog"]

Output:
[[0, 1, 0],
 [0, 0, 1]]
```

Hàm one_hot_encode

```python
def one_hot_encode(labels, classes):
    # Bước 1: Xác định số lượng lớp cần encode (số chiều của vector one-hot)

    # Bước 2: Khởi tạo danh sách kết quả rỗng

    # Bước 3: Duyệt qua từng nhãn trong danh sách labels:
        # Tìm chỉ số của nhãn trong danh sách classes
        # Tạo vector one-hot với tất cả phần tử là 0, chỉ có vị trí tương ứng là 1
        # Thêm vector này vào danh sách kết quả

    # Bước 4: Trả về danh sách các vector one-hot
    return ...
```

Test case

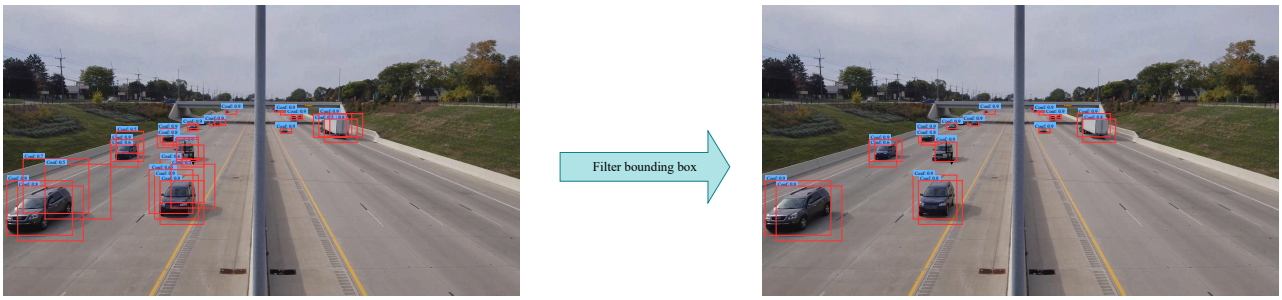
```python
# Test 1: Danh sách nhãn đầy đủ, kiểm tra thứ tự và ánh xạ chính xác
assert one_hot_encode(["dog", "cat", "bird", "dog"], ["cat", "dog", "bird"]) == [[0, 1,
                                                0], [1, 0, 0], [0, 0, 1], [0, 1, 0]],
                                                "Test 1 Failed"

# Test 2: Danh sách nhãn rỗng, kết quả cũng phải là danh sách rỗng
assert one_hot_encode([], ["cat", "dog", "bird"]) == [], "Test 2 Failed"

# Test 3: Trường hợp chỉ có hai lớp, lặp lại nhãn nhiều lần
assert one_hot_encode(["A", "A", "B"], ["A", "B"]) == [[1, 0], [1, 0], [0, 1]],
                                                    "Test 3 Failed"

```

```python
# Test 4: Chỉ có một nhãn duy nhất, kiểm tra đầu ra đơn
assert one_hot_encode(["cat"], ["cat", "dog", "bird"]) == [[1, 0, 0]], "Test 4 Failed"

# Test 5: Lớp có nhãn dài, kiểm tra ánh xạ không bị ảnh hưởng bởi độ dài chuỗi
assert one_hot_encode(["sad", "happy"], ["happy", "sad", "neutral"]) == [[0, 1, 0], [1,
                                                0, 0]], "Test 5 Failed"
```

Bài 5: Lọc Các Bounding Box Dựa Trên Confidence Score

1. Bối cảnh sử dụng

Trong các hệ thống nhận dạng vật thể (Object Detection), mô hình thường trả về rất nhiều bounding boxes, kể cả những boxes không đáng tin cậy với điểm tự tin (confidence score) thấp. Việc lọc các hộp này giúp giảm nhiễu và tăng độ chính xác của kết quả đầu ra. Đây là một bước hậu xử lý quan trọng trong pipeline của các mô hình như YOLO, SSD, Faster R-CNN.



Hình 4: Minh họa ứng dụng của việc filter bounding boxes trong thực tế.

2. Bài tập thực tiễn

Mô tả yêu cầu bài toán

Cho một danh sách các dự đoán từ mô hình nhận dạng vật thể, trong đó mỗi dự đoán là một danh sách gồm 6 phần tử theo định dạng:

$$[class\_id, confidence, x, y, w, h]$$

Hãy viết một hàm để:

- Lọc ra và giữ lại các dự đoán có độ tự tin (`confidence`) lớn hơn hoặc bằng một ngưỡng cho trước (`threshold`).

Input:

- `predictions`: một list các list, mỗi list con đại diện cho một dự đoán với 6 phần tử như trên.

- `threshold`: một số thực trong đoạn $[0, 1]$, là ngưỡng lọc độ tự tin.

Output: Một list mới chứa các dự đoán có confidence $\geq$ threshold.

Ví dụ:

```
Input: [[0, 0.9, 10, 10, 100, 100], [1, 0.3, 15, 15, 80, 80]]
Threshold: 0.5
Output: [[0, 0.9, 10, 10, 100, 100]]
```

Hàm filter_low_confidence_boxes

```python
def filter_low_confidence_boxes(predictions, threshold):
    # Bước 1: Khởi tạo danh sách rỗng để lưu các bounding boxes được giữ lại

    # Bước 2: Duyệt qua từng bounding boxes dự đoán trong danh sách đầu vào

        # Bước 2.1: Kiểm tra nếu điểm tự tin của bounding boxes >= ngưỡng threshold

            # Bước 2.2: Thêm bounding boxes đó vào danh sách kết quả

    # Bước 3: Trả về danh sách các bounding boxes có điểm tự tin đủ cao
    return ...
```

Test case

```python
# Test 1: Có 2 boxes đủ điểm tự tin (>= 0.8), 1 box bị loại
predictions1 = [[0, 0.95, 10, 10, 50, 50], [1, 0.4, 20, 20, 30, 30], [0, 0.88, 15, 15,
                                            40, 40]]
assert filter_low_confidence_boxes(predictions1, 0.8) == \
[[0, 0.95, 10, 10, 50, 50], [0, 0.88, 15, 15, 40, 40]], "Test 1 Failed"

# Test 2: Threshold cao hơn mọi boxes -> kết quả rỗng
assert filter_low_confidence_boxes(predictions1, 0.99) == [], "Test 2 Failed"

# Test 3: Dữ liệu đầu vào rỗng -> đầu ra cũng rỗng
assert filter_low_confidence_boxes([], 0.5) == [], "Test 3 Failed"

# Test 4: Một box có confidence đúng bằng threshold -> vẫn được giữ lại
predictions2 = [[0, 0.5, 5, 5, 10, 10]]
assert filter_low_confidence_boxes(predictions2, 0.5) == [[0, 0.5, 5, 5, 10, 10]], \
"Test 4 Failed"

# Test 5: Tất cả các boxes đều đủ điểm -> không loại boxes nào
predictions3 = [[0, 0.85, 1, 2, 3, 4], [1, 0.95, 4, 5, 6, 7]]
assert filter_low_confidence_boxes(predictions3, 0.5) == predictions3, \
"Test 5 Failed"
```

Bài 6: Chuẩn Hóa Đặc Trưng (Feature Scaling)

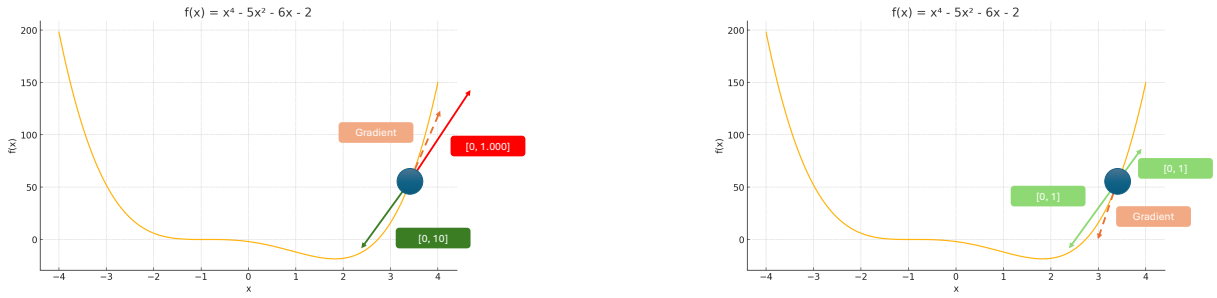
1. Bối cảnh sử dụng

Trong các mô hình học máy, đặc biệt là những thuật toán nhạy cảm với khoảng cách (như K-Nearest Neighbors) hoặc dựa trên gradient (như Linear Regression, Logistic Regression, Neural Networks), các đặc trưng có thang đo khác nhau có thể gây ra vấn đề. Ví dụ, một đặc trưng có giá trị từ 0 đến 1000 sẽ lấn át một đặc trưng khác có giá trị từ 0 đến 1 trong quá trình học.

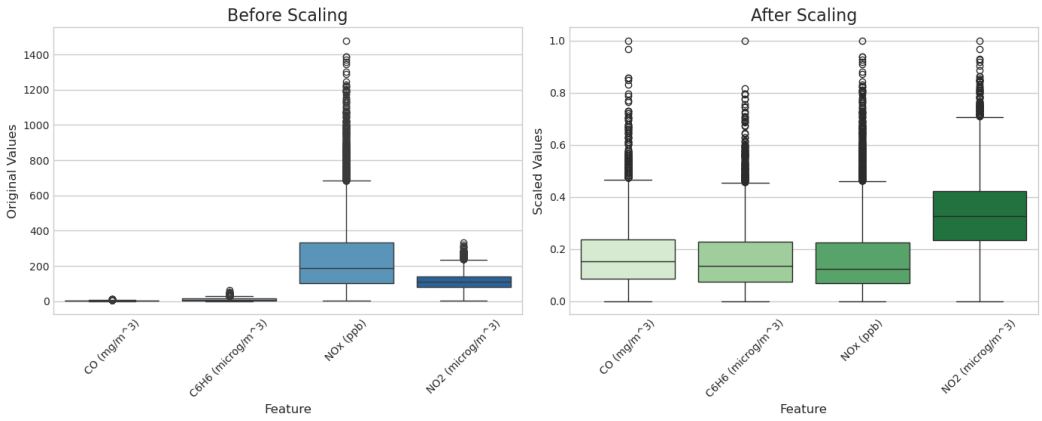
Kỹ thuật **Feature Scaling** được sử dụng để biến đổi thang đo của các đặc trưng, đưa chúng về một phạm vi chung. Những phương pháp phổ biến nhất là **Min-Max Scaling** (một hàm Normalization), StandardScaler, và RobustScaler (các hàm Standardization). Trong đó Min-Max Scaling giúp biến đổi giá trị của mỗi đặc trưng về đoạn $[0, 1]$ với công thức là:

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Việc này đảm bảo rằng tất cả các đặc trưng đều có tầm quan trọng như nhau khi bắt đầu quá trình huấn luyện, giúp mô hình có cơ hội được hội tụ nhanh hơn và hoạt động hiệu quả hơn.



Hình 5: So sánh sự thay đổi của hàm $f(x)$ trước (trái) và sau (phải) khi sử dụng Min-Max Scaling để chuẩn hóa đặc trưng. Ta thấy rằng đối với hình bên trái, tuy rằng hàm sắp đạt tới cực tiểu, nhưng vì hướng đạo hàm của các biến trong khoảng $[0, 1.000]$ quá lớn, dẫn đến lấn áp hướng đi của biến trong khoảng $[0, 10]$. Ở hình sau, sau khi ta chuẩn hóa giá trị các biến về khoảng $[0, 1]$, lúc này hướng đi của hàm sẽ ổn định hơn và tiến về cực tiểu nhanh hơn.



Hình 6: Phân phối của các giá trị trong tập dataset air-quality trước và sau khi Min-Max Scaling. Ta thấy rằng Min–Max Scaling đưa mọi feature về chung thang đo, bảo toàn hình dạng phân phối (bao gồm outliers). Nguồn: Link.

2. Bài tập thực tiễn

Mô tả yêu cầu bài toán

Cho một danh sách các số thực đại diện cho một đặc trưng, hãy viết một hàm thực hiện:

- Áp dụng công thức Min-Max Scaling để chuẩn hóa tất cả các giá trị trong danh sách về đoạn $[0, 1]$.
- Nếu tất cả các giá trị trong danh sách bằng nhau ($X_{max} = X_{min}$), hãy trả về một danh sách chứa các giá trị 0.

Input: Một list các số (số nguyên hoặc số thực).
Output: Một list mới với các giá trị đã được chuẩn hóa theo Min-Max Scaling.

Ví dụ:

```
Input: [10, 20, 50]
Output: [0.0, 0.25, 1.0]
```

Hàm min_max_scale

```python
def min_max_scale(data):
    # Bước 1: Kiểm tra các trường hợp đặc biệt (ví dụ: danh sách rỗng)

    # Bước 2: Tìm giá trị nhỏ nhất (min_val) và lớn nhất (max_val) trong danh sách

    # Bước 3: Kiểm tra nếu min_val bằng max_val để tránh chia cho 0

    # Bước 4: Khởi tạo danh sách kết quả
    # Duyệt qua từng giá trị trong danh sách đầu vào:
```

```
    # Áp dụng công thức Min-Max Scaling và thêm vào danh sách kết quả

    # Bước 5: Trả về danh sách đã được chuẩn hóa
    return ...
```

Test case

```
# Test 1: Chuẩn hóa danh sách số nguyên dương
assert min_max_scale([10, 20, 50, 30]) == [0.0, 0.25, 1.0, 0.5], "Test 1 Failed"

# Test 2: Danh sách chứa số âm và số 0
assert min_max_scale([-10, 0, 10]) == [0.0, 0.5, 1.0], "Test 2 Failed"

# Test 3: Tất cả các phần tử giống nhau, tránh chia cho 0
assert min_max_scale([5, 5, 5, 5]) == [0.0, 0.0, 0.0, 0.0], "Test 3 Failed"

# Test 4: Danh sách đầu vào rỗng
assert min_max_scale([]) == [], "Test 4 Failed"

# Test 5: Dữ liệu đã ở trong khoảng [0, 1]
# Lưu ý: Do sai số floating point, ta cần so sánh với một sai số nhỏ
scaled_data = min_max_scale([0.1, 0.5, 0.9])
expected_data = [0.0, 0.5, 1.0]
assert all(abs(a - b) < 1e-9 for a, b in \
zip(scaled_data, expected_data)), "Test 5 Failed"
```

Bài 7: Tính Toán Độ Chính Xác (Accuracy)

1. Bối cảnh sử dụng

Trong các bài toán phân loại (classification), **độ chính xác (Accuracy)** là một trong những độ đo cơ bản và phổ biến nhất để đánh giá hiệu suất của mô hình. Nó đo lường tỷ lệ giữa số lượng dự đoán đúng trên tổng số dự đoán được thực hiện. Công thức tính độ chính xác rất đơn giản:

$$Accuracy = \frac{\text{Số lượng dự đoán đúng}}{\text{Tổng số dự đoán}}$$

2. Bài tập thực tiễn

Mô tả yêu cầu bài toán

Cho hai danh sách có cùng độ dài: một danh sách chứa các nhãn thực tế (‘y_true‘) và một danh sách chứa các nhãn dự đoán từ mô hình (‘y_pred‘). Hãy viết một hàm để:

- So sánh từng cặp phần tử tương ứng trong hai danh sách.
- Tính toán và trả về độ chính xác (accuracy) dưới dạng một số thực trong đoạn $[0.0, 1.0]$.

Input:

- 'y_true': list các nhãn thực tế.
- 'y_pred': list các nhãn dự đoán.

Output: Một số thực là giá trị độ chính xác.

Ví dụ:

```
Input:
y_true = ["cat", "dog", "cat", "bird"]
y_pred = ["cat", "cat", "cat", "bird"]

Output: 0.75
```

Hàm calculate_accuracy

```python
def calculate_accuracy(y_true, y_pred):
    # Bước 1: Kiểm tra nếu danh sách đầu vào rỗng, trả về 0.0

    # Bước 2: Khởi tạo biến đếm số dự đoán đúng (correct_predictions)

    # Bước 3: Duyệt qua các cặp nhãn tương ứng trong y_true và y_pred
    # Nếu nhãn dự đoán khớp với nhãn thực tế, tăng biến đếm

    # Bước 4: Tính toán độ chính xác bằng cách chia số dự đoán đúng cho tổng số dự đoán

    # Bước 5: Trả về kết quả
    return ...
```

Test case

```python
# Test 1: Các nhãn là chuỗi, độ chính xác 80%
y_true1 = ["cat", "dog", "cat", "bird", "dog"]
y_pred1 = ["cat", "dog", "cat", "dog", "dog"]
assert calculate_accuracy(y_true1, y_pred1) == 0.8, "Test 1 Failed"

# Test 2: Các nhãn là số, độ chính xác 100%
y_true2 = [1, 0, 1, 1, 0]
y_pred2 = [1, 0, 1, 1, 0]
assert calculate_accuracy(y_true2, y_pred2) == 1.0, "Test 2 Failed"

# Test 3: Độ chính xác 0%
y_true3 = [0, 0, 0]
y_pred3 = [1, 1, 1]
assert calculate_accuracy(y_true3, y_pred3) == 0.0, "Test 3 Failed"

# Test 4: Danh sách rỗng
```

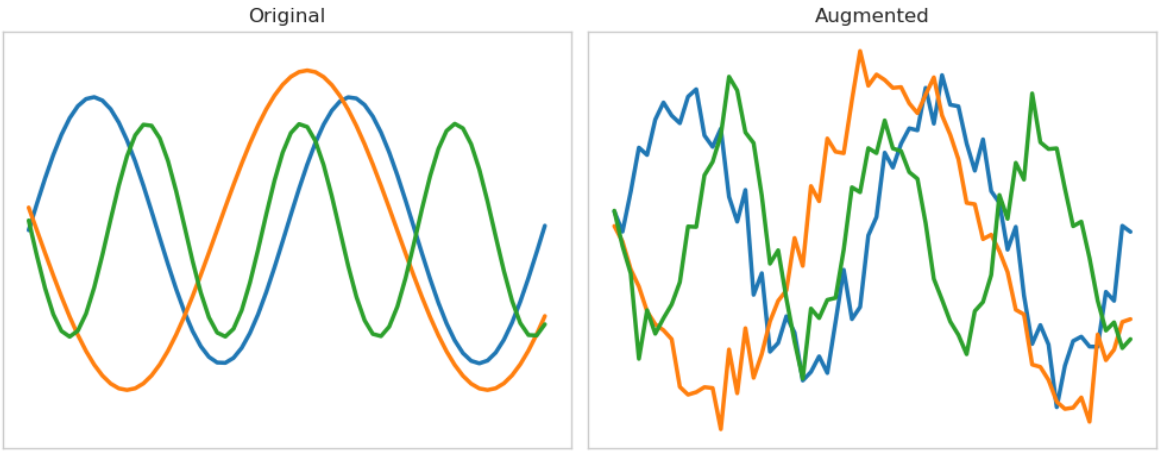
```python
assert calculate_accuracy([], []) == 0.0, "Test 4 Failed"

# Test 5: Độ chính xác 50%
y_true5 = ["A", "B"]
y_pred5 = ["A", "C"]
assert calculate_accuracy(y_true5, y_pred5) == 0.5, "Test 5 Failed"
```

Bài 8: Tăng Cường Dữ Liệu Chuỗi Thời Gian

1. Bối cảnh sử dụng

Huấn luyện các mô hình học sâu trên dữ liệu chuỗi thời gian thường đòi hỏi một lượng lớn dữ liệu để có thể khái quát hóa tốt và tránh học vẹt. Tuy nhiên, việc thu thập dữ liệu chuỗi thời gian trong thực tế có thể tốn kém và mất nhiều thời gian.

Time-Series Augmentation là một kỹ thuật giúp tạo ra các mẫu dữ liệu huấn luyện mới và hợp lý từ tập dữ liệu gốc. Bằng cách áp dụng các phép biến đổi nhỏ như thêm nhiễu (adding noise), co giãn (scaling), hoặc dịch chuyển (shifting), chúng ta có thể làm tăng đáng kể kích thước và sự đa dạng của tập huấn luyện. Điều này giúp mô hình ít nhạy cảm hơn với các biến thể nhỏ trong dữ liệu thực tế.



Hình 7: Ba chuỗi thời gian mẫu ban đầu (bên trái) và sau khi tăng cường bằng nhiễu Gaussian (bên phải) với độ lệch chuẩn $\sigma = 1.5$, minh hoạ quá trình làm biến đổi tập dữ liệu chuỗi thời gian.

2. Bài tập thực tiễn

Mô tả yêu cầu bài toán

Cho một chuỗi thời gian (được biểu diễn dưới dạng một list các số) và một độ lệch chuẩn của nhiễu, hãy viết một hàm thực hiện:

- Tạo ra một chuỗi thời gian mới có cùng độ dài.
- Mỗi điểm dữ liệu trong chuỗi mới bằng điểm dữ liệu tương ứng trong chuỗi gốc cộng với một giá trị nhiễu ngẫu nhiên.
- Giá trị nhiễu này được lấy mẫu từ một phân phối chuẩn (Gaussian distribution) với giá trị trung bình là 0 và độ lệch chuẩn (‘noise_level‘) cho trước.

Input:

- ‘time_series‘: một list các số.
- ‘noise_level‘: một số thực không âm, đại diện cho độ lệch chuẩn của nhiễu.

Output: Một list mới là chuỗi thời gian đã được tăng cường bằng nhiễu.
Gợi ý: Sử dụng thư viện ‘random‘ của Python, cụ thể là hàm ‘random.gauss(mu, sigma)‘.

Hàm add_noise_augmentation

```python
import random

def add_noise_augmentation(time_series, noise_level):
    # Bước 1: Khởi tạo danh sách kết quả rỗng

    # Bước 2: Duyệt qua từng điểm dữ liệu trong chuỗi thời gian gốc
    # Tạo một giá trị nhiễu ngẫu nhiên từ phân phối Gaussian (mu=0, sigma=noise_level)
    # Tính giá trị mới bằng cách cộng điểm dữ liệu gốc với nhiễu
    # Thêm giá trị mới vào danh sách kết quả

    # Bước 3: Trả về chuỗi thời gian đã được thêm nhiễu
    return ...
```

Test case

```python
import random

# Để đảm bảo kết quả có thể tái lập, ta cố định seed cho random
random.seed(0)

# Test 1: Thêm nhiễu vào một chuỗi thời gian
ts1 = [10, 11, 12, 11, 10]
augmented_ts1 = add_noise_augmentation(ts1, 0.1)
assert len(augmented_ts1) == len(ts1), "Test 1 Failed: Length mismatch"
```

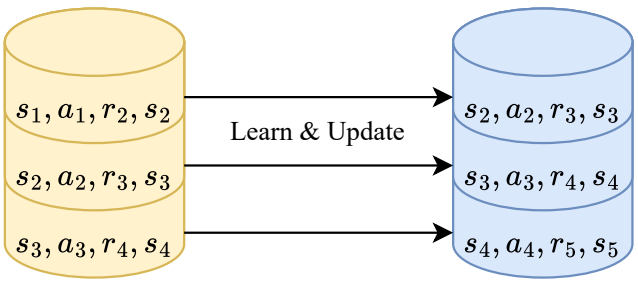
```
assert augmented_ts1 != ts1, \
"Test 1 Failed: Series should be different after adding noise"

# Test 2: noise_level = 0, chuỗi không thay đổi
ts2 = [100, 200, 150]
augmented_ts2 = add_noise_augmentation(ts2, 0.0)
assert augmented_ts2 == ts2, \
"Test 2 Failed: Series should be identical with zero noise"

# Test 3: Chuỗi rỗng
assert add_noise_augmentation([], 0.5) == [], \
"Test 3 Failed: Empty list should return empty list"

# Test 4: Kiểm tra xem giá trị có thay đổi không
# Vì kết quả là ngẫu nhiên, ta chỉ kiểm tra xem nó có khác bản gốc không
ts4 = [5]
augmented_ts4 = add_noise_augmentation(ts4, 1.0)
assert augmented_ts4 != ts4, "Test 4 Failed: Single element should change"
```

Bài 9: Experience Replay Buffer

1. Bối cảnh sử dụng

Trong học tăng cường (Reinforcement Learning), đặc biệt là các thuật toán off-policy như Deep Q-Networks (DQN), tác nhân (agent) học hỏi từ các kinh nghiệm trong quá khứ. Tuy nhiên, nếu tác nhân chỉ học từ kinh nghiệm ngay sau khi nó xảy ra, các mẫu dữ liệu sẽ có tương quan cao với nhau theo thời gian, làm cho quá trình học không ổn định.

Experience Replay Buffer là một cấu trúc dữ liệu (thường là một hàng đợi có kích thước cố định) dùng để lưu trữ các kinh nghiệm của tác nhân. Một kinh nghiệm thường bao gồm bộ $(state, action, reward, next\_state)$. Thay vì học ngay lập tức, tác nhân lưu kinh nghiệm vào bộ đệm. Trong quá trình huấn luyện, tác nhân sẽ lấy ra một lô (mini-batch) kinh nghiệm ngẫu nhiên từ bộ đệm này để cập nhật mạng nơ-ron. Việc này giúp phá vỡ sự tương quan về mặt thời gian của dữ liệu, giúp quá trình học ổn định và hiệu quả hơn.



Hình 7: Minh họa cơ chế *Experience Replay Buffer* – các transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ được lưu trữ, sau đó được lấy ra để Learn & Update và bộ đệm được bổ sung thêm các trải nghiệm mới $(s_{t+1}, a_{t+1}, r_{t+2}, s_{t+2})$ nhằm phá vỡ tương quan theo thời gian và ổn định quá trình huấn luyện.

2. Bài tập thực tiễn

Mô tả yêu cầu bài toán

Hãy triển khai một lớp 'ExperienceReplayBuffer' với các chức năng sau:

- Khởi tạo với một 'capacity' (sức chứa) tối đa.
- Phương thức 'add(experience)': thêm một kinh nghiệm vào bộ đệm. Nếu bộ đệm đầy, kinh nghiệm cũ nhất sẽ bị loại bỏ (cơ chế FIFO - First-In, First-Out).
- Phương thức 'sample(batch_size)': lấy ngẫu nhiên một lô gồm 'batch_size' kinh nghiệm từ bộ đệm. Nếu số kinh nghiệm trong bộ đệm ít hơn 'batch_size', trả về tất cả kinh nghiệm hiện có.

Input: Các kinh nghiệm (có thể là tuple, list,...) và kích thước lô.
Output: Lớp 'ExperienceReplayBuffer' hoạt động đúng chức năng.

Gợi ý: Sử dụng 'collections.deque(maxlen=capacity)' để dễ dàng triển khai hàng đợi có kích thước cố định.

Lớp ExperienceReplayBuffer

```python
import collections
import random

class ExperienceReplayBuffer:
    def __init__(self, capacity):
        # Bước 1: Khởi tạo một deque với sức chứa tối đa là capacity
        self.buffer = collections.deque(maxlen=capacity)

    def add(self, experience):
        # Bước 2: Thêm kinh nghiệm vào buffer
        # Lưu ý: deque sẽ tự động xử lý việc loại bỏ phần tử cũ khi đầy
        self.buffer.append(experience)

    def sample(self, batch_size):
        # Bước 3: Lấy ngẫu nhiên một mẫu từ buffer
        # Sử dụng random.sample()
        # Kích thước mẫu là min(batch_size, số lượng kinh nghiệm hiện có)
        actual_batch_size = min(batch_size, len(self.buffer))
        return random.sample(self.buffer, actual_batch_size)

    def __len__(self):
        # Bước 4: Trả về số lượng kinh nghiệm hiện có trong buffer
        return len(self.buffer)
```

Test case

```python
# Test 1: Thêm và lấy mẫu
buffer = ExperienceReplayBuffer(capacity=100)
buffer.add(("s1", "a1", 1, "s2"))
buffer.add(("s2", "a2", 0, "s3"))
buffer.add(("s3", "a3", -1, "s4"))
assert len(buffer) == 3, "Test 1 Failed: Incorrect buffer size"
sample = buffer.sample(2)
assert len(sample) == 2, "Test 1 Failed: Incorrect sample size"

# Test 2: Vượt quá sức chứa
buffer = ExperienceReplayBuffer(capacity=3)
for i in range(5):
    buffer.add(i) # Thêm 0, 1, 2, 3, 4
assert len(buffer) == 3, "Test 2 Failed: Capacity exceeded but size is wrong"
# Vì cơ chế FIFO, các phần tử còn lại phải là 2, 3, 4
assert list(buffer.buffer) == [2, 3, 4], \
"Test 2 Failed: Old experiences not discarded correctly"

# Test 3: Lấy mẫu nhiều hơn số lượng hiện có
buffer = ExperienceReplayBuffer(capacity=10)
buffer.add(1)
buffer.add(2)
sample = buffer.sample(5)
assert len(sample) == 2, \
"Test 3 Failed: Should return all elements if batch_size is larger"

# Test 4: Lấy mẫu từ buffer rỗng
buffer = ExperienceReplayBuffer(capacity=10)
sample = buffer.sample(5)
assert sample == [], \
"Test 4 Failed: Sampling from empty buffer should return empty list"
```

Bài 10: Biểu Diễn Văn Bản Bằng Túi Từ (Bag-of-Words)

1. Bối cảnh sử dụng

Để các mô hình học máy có thể xử lý văn bản, chúng ta cần chuyển đổi văn bản từ dạng chuỗi ký tự sang dạng vector số. **Bag-of-Words (BoW)** là một trong những kỹ thuật đơn giản và cổ điển nhất để làm việc này. Mô hình BoW biểu diễn một đoạn văn bản bằng một vector, trong đó mỗi phần tử của vector tương ứng với tần suất xuất hiện của một từ trong từ điển (vocabulary) đã xây dựng trước.

Cách tiếp cận này được gọi là "túi từ" vì nó bỏ qua hoàn toàn ngữ pháp và thứ tự của các từ, chỉ quan tâm đến việc từ nào xuất hiện và xuất hiện bao nhiêu lần — giống như bỏ tất cả các từ vào một chiếc túi và đếm chúng. Mặc dù đơn giản, BoW vẫn rất hiệu quả trong nhiều bài toán NLP như phân loại văn bản.

2. Bài tập thực tiễn

Mô tả yêu cầu bài toán

Cho một kho văn bản (corpus) được biểu diễn dưới dạng danh sách các tài liệu (mỗi tài liệu là một danh sách các từ đã được token hóa) và một từ điển (vocabulary) đã được xây dựng sẵn. Hãy viết một hàm để:

- Chuyển đổi mỗi tài liệu trong kho văn bản thành một vector Bag-of-Words.
- Vector BoW có độ dài bằng kích thước của từ điển.
- Mỗi phần tử thứ i trong vector biểu diễn số lần từ thứ i trong từ điển xuất hiện trong tài liệu đó. Các từ không có trong từ điển sẽ bị bỏ qua.

Input:

- 'corpus': list các list, mỗi list con là một tài liệu chứa các từ.
- 'vocabulary': list các từ duy nhất, đã được sắp xếp, dùng làm từ điển.

Output: Một list các vector BoW.

Ví dụ:

```
Input:
corpus = [
    ["ai", "is", "great"],
    ["ai", "is", "fun", "and", "ai", "is", "cool"]
    ]
vocabulary = ["ai", "cool", "fun", "great", "is"]

Output:
[[1, 0, 0, 1, 1],  # "ai":1, "is":1, "great":1
 [2, 1, 1, 0, 2]]  # "ai":2, "is":2, "fun":1, "cool":1
```

Hàm create_bow_vectors

```python
def create_bow_vectors(corpus, vocabulary):
    # Bước 1: Tạo một map (dictionary) từ từ sang chỉ số để tra cứu nhanh
    vocab_map = {word: i for i, word in enumerate(vocabulary)}

    # Bước 2: Khởi tạo danh sách rỗng để chứa các vector BoW
    bow_vectors = []

    # Bước 3: Duyệt qua từng tài liệu (document) trong kho văn bản (corpus)
    #   Khởi tạo một vector 0 có độ dài bằng kích thước từ điển cho tài liệu hiện tại
    #   Duyệt qua từng từ (token) trong tài liệu
    #   Nếu từ đó có trong vocab_map:
```

```
    #   Lấy chỉ số của từ và tăng giá trị tại chỉ số đó trong vector lên 1
    #   Thêm vector đã hoàn thành vào danh sách bow_vectors

    # Bước 4: Trả về danh sách các vector BoW
    return ...
```

Test case

```
vocab = ["apple", "banana", "fruit", "orange"]

# Test 1: Hai tài liệu với các từ trong từ điển
corpus1 = [["apple", "banana", "apple"], ["fruit", "orange"]]
expected1 = [[2, 1, 0, 0], [0, 0, 1, 1]]
assert create_bow_vectors(corpus1, vocab) == expected1, "Test 1 Failed"

# Test 2: Tài liệu chứa từ không có trong từ điển
corpus2 = [["apple", "grape", "banana"]] # "grape" không có trong vocab
expected2 = [[1, 1, 0, 0]]
assert create_bow_vectors(corpus2, vocab) == expected2, "Test 2 Failed"

# Test 3: Một tài liệu rỗng
corpus3 = [[]]
expected3 = [[0, 0, 0, 0]]
assert create_bow_vectors(corpus3, vocab) == expected3, "Test 3 Failed"

# Test 4: Kho văn bản rỗng
assert create_bow_vectors([], vocab) == [], "Test 4 Failed"

# Test 5: Từ điển lớn hơn, tài liệu chỉ chứa một vài từ
vocab5 = ["a", "b", "c", "d", "e"]
corpus5 = [["a", "c", "e", "a"]]
expected5 = [[2, 0, 1, 0, 1]]
assert create_bow_vectors(corpus5, vocab5) == expected5, "Test 5 Failed"
```

III. Tài liệu tham khảo

[1] Chien-Thang Doan, *Notebook file*, Colab, 2025. [Online]. Available: https://tinyurl.com/yx6ev3xd.

[2] (Solution), *Notebook file*, Colab, 2025. [Online]. Available: https://drive.google.com/file/d/1gwzcQBYzd8k4u_WcyEgYaAX2FUOvIsYn/view?usp=sharing.