AI VIET NAM
@aivietnam.edu.vn

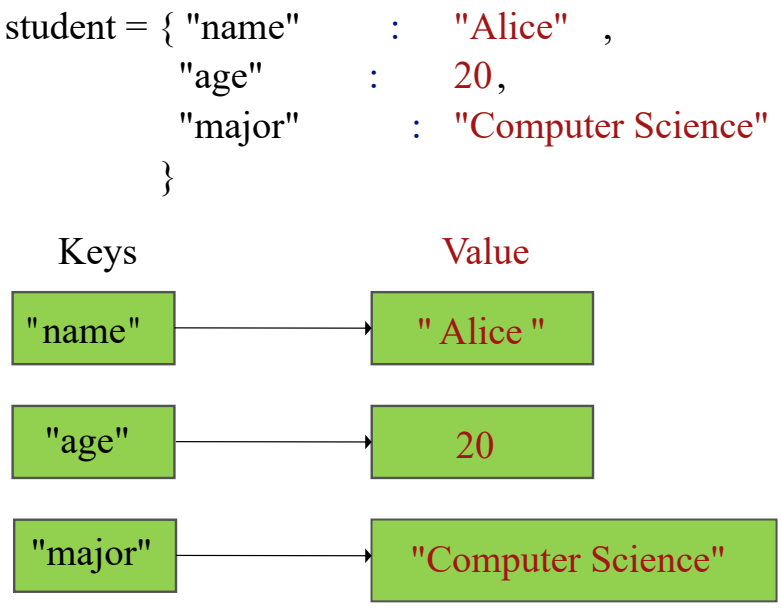
AI VIET NAM – All-in-One Course 2025

Tutorial: Ứng dụng của Python Dictionary trong các bài toán thực tế

Nguyễn Thế Hào, Nguyễn Phúc Thịnh và Dinh Quang Vinh

I. Giới thiệu

Trong Python, **dict (dictionary)** là một kiểu dữ liệu được xây dựng sẵn để lưu trữ các cặp "khóa – giá trị" (key-value). Mỗi khóa là duy nhất và dùng để truy xuất tới giá trị tương ứng. Đây là cấu trúc ánh xạ (mapping), khác với các cấu trúc tuần tự như danh sách (list) hoặc bộ (tuple).



Hình 1: Minh họa về Dict trong Python.

Trong ví dụ trên, "name", "age" và "major" là các khóa, còn "Alice", 20 và "Computer Science" là các giá trị tương ứng. Việc truy cập giá trị thực hiện nhanh chóng qua cú pháp `student["name"]` trả về "Alice". Python hỗ trợ nhiều thao tác tiện ích trên dict như thêm, xoá, kiểm tra sự tồn tại của khóa, duyệt qua các phần tử, cập nhật dữ liệu,... khiến dict trở thành một cấu trúc dữ liệu linh hoạt và mạnh mẽ.

Đối với lĩnh vực xử lý ngôn ngữ tự nhiên (Natural Language Processing – NLP), dict được xem là một trong những cấu trúc dữ liệu nền tảng, đóng vai trò trung tâm trong hầu hết các bước tiền xử lý, biểu diễn dữ liệu và phân tích. Điều này xuất phát từ chính đặc điểm của văn bản: thông tin trong ngôn ngữ tự nhiên thường không có cấu trúc rõ ràng, nhưng lại chứa nhiều mối quan hệ có thể ánh xạ một cách hiệu quả bằng key–value.

Trong bài viết này, chúng ta sẽ khám phá cách sử dụng cấu trúc dữ liệu dict trong Python để giải quyết các bài toán thực tế trong xử lý ngôn ngữ tự nhiên (NLP) và AI. Thông qua các ví dụ cụ thể, bạn sẽ thấy rõ dict không chỉ là một kiểu dữ liệu cơ bản mà còn là công cụ cực kỳ mạnh mẽ trong việc tổ chức, thống kê và ánh xạ dữ liệu phi cấu trúc như văn bản. Cụ thể bài viết này sẽ lần lượt trình bày ba ví dụ nền tảng:

- **Character Counting** – Đếm số lần xuất hiện của từng ký tự trong một chuỗi văn bản.
- **Word Counting** - Thống kê tần suất các từ trong văn bản.
- **N-grams** - Phân chia văn bản thành các chuỗi con gồm n từ hoặc n ký tự liên tiếp để phân tích hoặc dự đoán các phần tiếp theo dựa trên ngữ cảnh trước đó.

Sau khi xây dựng nền tảng qua các ví dụ trên, chúng ta sẽ bước vào một mini-project có tính ứng dụng thực tế cao: **Phân tích người viết văn bản (Author Profiling)**

Mục lục

II. Các ứng dụng của dict trong nlp

II.1. Character Counting

II.1.1. Mục đích sử dụng

Bài toán này phản ánh một nhu cầu phổ biến trong xử lý văn bản - đó là thống kê tần suất xuất hiện của các ký tự để phục vụ cho các bước phân tích tiếp theo. Thống kê tần suất ký tự giúp chúng ta phát hiện các mẫu xuất hiện phổ biến, kiểm tra định dạng dữ liệu, hoặc phục vụ các tác vụ như mã hóa văn bản, phân tích lỗi gõ phím, và phát hiện ngôn ngữ dựa trên tần suất chữ cái.

Với dict, việc cập nhật và truy xuất tần suất ký tự trở nên đơn giản và tối ưu, đặc biệt hữu ích trong các tập văn bản lớn. Ngoài ra, nhờ khả năng mở rộng và hỗ trợ nhiều phương pháp tích hợp sẵn như get() hoặc collections.Counter, dict trở thành công cụ lý tưởng để thực hiện các thao tác đếm tần suất trong NLP nói chung và bài toán character counting nói riêng.

II.1.2. Bài toán thực tế

Mô tả yêu cầu bài toán

Viết thuật toán trả về một dictionary đếm số lượng chữ xuất hiện trong một từ, với key là chữ cái và value là số lần xuất hiện.
Input: Một từ
Output: Dictionary đếm số lần các chữ xuất hiện.
Note: Giả sử các từ nhập vào đều có các chữ cái thuộc [a-z] hoặc [A-Z]

Ví dụ:

- **Input:** word = 'baby'
- **Output:** 'b': 2, 'a': 1, 'y': 1

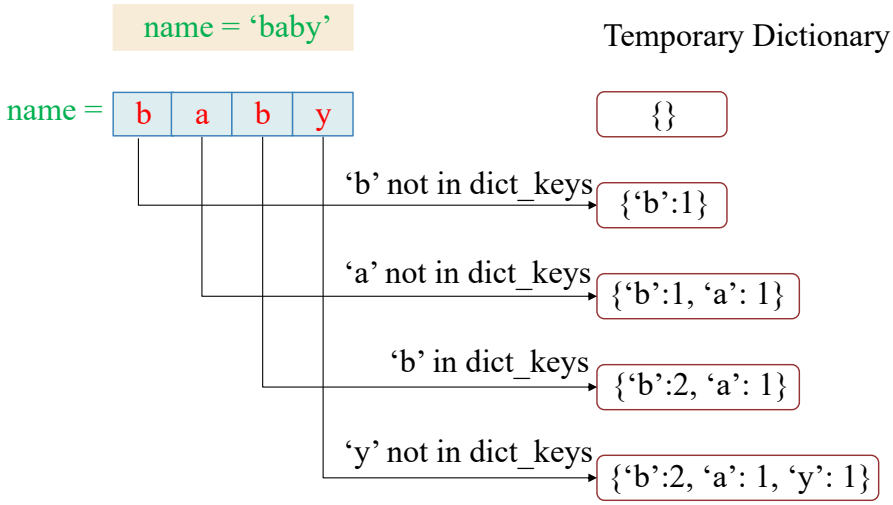
Hãy cùng phân tích lời giải cho bài toán này: Ta bắt đầu bằng cách khởi tạo một dict rỗng, mục tiêu là sẽ dùng nó để lưu từng chữ cái làm key và số lần xuất hiện tương ứng làm value.

Sau đó, ta tiến hành duyệt từng ký tự trong chuỗi đầu vào. Vì yêu cầu chỉ quan tâm đến các chữ cái (regex: `[a-zA-Z]`), ta có thể thêm một bước kiểm tra để bỏ qua những ký tự không phải chữ cái (regex: `[^a-zA-Z]`), nếu cần. Ngoài ra, để xử lý thống nhất, ta có thể chuyển tất cả ký tự về chữ thường.

Khi gặp một ký tự, ta kiểm tra xem nó đã có trong dict chưa:

- Nếu có, ta tăng giá trị tương ứng thêm 1.
- Nếu chưa, tức là đây là lần đầu ký tự đó xuất hiện, ta thêm nó vào dict với giá trị khởi tạo là 1.

Sau khi duyệt hết tất cả ký tự, ta trả về dict chứa kết quả đếm tần suất. Cùng theo dõi hình minh họa dưới để dễ hình dung.



Hình 2: Minh họa về solution trong Character Counting.

Dựa trên những phân tích và hướng dẫn phía trên, chúng ta có thể bắt đầu code để thực hiện bài này.

Hàm character count

```python
def count_character(word):
    # Khởi tạo một dictionary rỗng để chứa thống kê số lần xuất hiện của từng ký tự
    character_statistic = {}

    # Lặp qua từng ký tự trong chuỗi 'word'
    for character in word:
        # Nếu ký tự đã xuất hiện trước đó và hợp lệ, tăng số đếm lên 1
        if character in character_statistic and character.isalpha():
            character_statistic[character] += 1
        # Nếu ký tự chưa xuất hiện, thêm vào dictionary với số đếm bắt đầu từ 1
        else:
            character_statistic[character] = 1

    # Trả về dictionary chứa số lần xuất hiện của từng ký tự
    return character_statistic
```

II.1.3. Cải tiến bằng defaultdict()

Khi sử dụng dict thông thường để đếm ký tự, ta cần kiểm tra xem một ký tự đã tồn tại trong từ điển hay chưa để quyết định có khởi tạo giá trị ban đầu hay không. Điều này khiến đoạn mã trở nên dài dòng và dễ lặp lại các thao tác điều kiện. Python cung cấp một công cụ mạnh mẽ để đơn giản hóa vấn đề này – đó là defaultdict trong module collections.

defaultdict là một lớp con (subclass) của dict, cho phép ta xác định một hàm mặc định (default_factory) để cung cấp giá trị khởi tạo cho bất kỳ khóa (key) nào chưa tồn tại. Tức là khi truy cập một khóa không có trong từ điển, defaultdict sẽ tự động tạo một giá trị mới bằng cách gọi hàm mặc định đó, thay vì báo lỗi như dict thông thường.

Ví dụ, nếu bạn dùng defaultdict(int), thì các khóa mới sẽ có giá trị mặc định là 0. Nếu dùng defaultdict(str), giá trị mặc định sẽ là một chuỗi rỗng "". Điều này giúp viết code gọn hơn và tránh lỗi KeyError.

default_factory	Giá trị mặc định	Ví dụ truy cập d['x']
int	0	d['x'] = 0
list	[]	d['x'] = []
str	""	d['x'] = ""
set	set()	d['x'] = set()

Hình 3: Các giá trị thông thường của default_factory.

Sau khi đã hiểu defaultdict() chúng ta viết lại hàm character count như sau.

Hàm character count

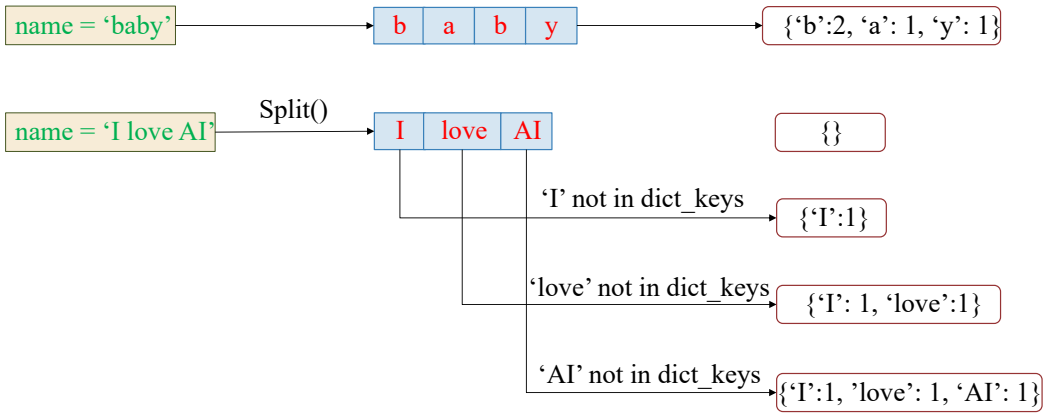
```python
from collections import defaultdict

def count_character(word):
    # Khởi tạo defaultdict với giá trị mặc định là 0
    character_statistic = defaultdict(int)

    # Lặp qua từng ký tự trong chuỗi 'word'
    for character in word:
        if character.isalpha():
            # Không cần kiểm tra tồn tại
            character_statistic[character] += 1

    # Trả về dictionary thông thường nếu không muốn trả defaultdict
    return dict(character_statistic)
```

II.2. Word Counting

II.2.1. Mục đích sử dụng

Sau khi đã tìm hiểu cách sử dụng dict để đếm số lần xuất hiện của từng ký tự, ta có thể mở rộng ý tưởng này để giải quyết một bài toán phổ biến hơn trong xử lý ngôn ngữ tự nhiên: đếm số lần xuất hiện của các từ trong một đoạn văn bản – còn gọi là Word Counting.

Bài toán này là bước tiền xử lý quan trọng trong nhiều kỹ thuật như Bag-of-Words, TF-IDF, hoặc phân tích cảm xúc. Python cung cấp cho chúng ta công cụ dict (và đặc biệt là defaultdict) để thực hiện tác vụ này một cách đơn giản, rõ ràng và hiệu quả. Nó giống như là một bản nâng cấp của Character counting.



Hình 4: Minh họa về Word Couting.

II.2.2. Bài toán cụ thể

Mô tả yêu cầu bài toán

Viết thuật toán đọc các câu trong một file txt, đếm số lượng các từ xuất hiện và trả về một dictionary với key là từ và value là số lần từ đó xuất hiện.
Input: Một từ
Output: Dictionary đếm số lần các từ xuất hiện.
Note: Giả sử các từ trong file txt đều có các chữ cái thuộc [a-z] hoặc [A-Z]. Không cần các thao tác xử lý string phức tạp nhưng cần xử lý các từ đều là viết thường
File: https://drive.google.com/uc?id=1fbeTFfWxO29ipsFxBJFI6xHbXd4ii3FP

Đầu tiên, để làm việc với dữ liệu văn bản lưu trữ trên Google Drive, ta cần tải file .txt. Ta có thể sử dụng thư viện requests hoặc gdown để tự động tải file này. Trong ví dụ dưới đây, sử dụng gdown đơn giản để tải trực tiếp nội dung file trong môi trường GoogleColab

Download và đọc file

```
!gdown --id 1fbeTFfWxO29ipsFxBJFI6xHbXd4ii3FP

with open('/content/P1_data.txt', 'r') as f:
    sentences = f.readlines()
type(sentences)
```

Nội dung trong file là một đoạn văn được chia theo từng câu. Chúng ta cần xử lý như sau:

- Chuyển tất cả ký từ thành chữ thường
- Loại bỏ dấu chấm (.) và dấu chấm phẩy (,)
- Tách câu thành danh sách các từ

Để thực hiện các thao tác trên chúng ta dùng các built-in methods của kiểu dữ liệu str trong python giải quyết.

Text Preprocessing

```
def preprocess_text(sentence):

    sentence = sentence.lower()
    sentence = sentence.replace('.', '').replace(',', '')
    words = sentence.split()
    return words
```

Sau khi đã có list các từ trong file giờ chúng ta sẽ coi từng từ như từng ký tự trong Character Counting và xử lý tương tự. Và ở trong bài trước chúng ta đã sử dụng defaultdict để xử lý task này. Đoạn code được viết như sau:

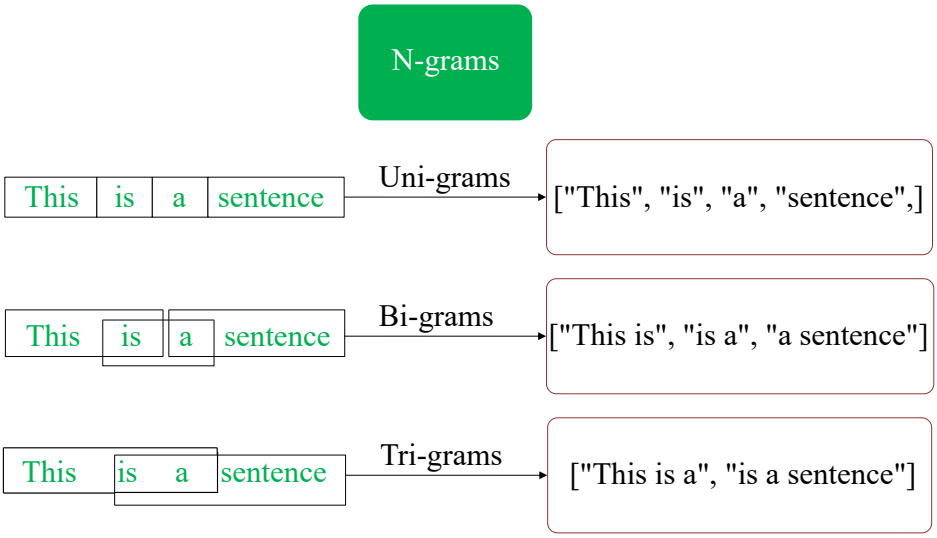
Word Counting

```
from collections import defaultdict

def count_words(sentences):
    """
    Hàm đếm số lần xuất hiện của từng từ trong danh sách các câu.

    Input:
        - sentences: danh sách các câu (list of strings)

    Output:
        - counter: dictionary (defaultdict) với key là từ, value là số lần xuất hiện
    """
    counter = defaultdict(int)
```

```
    for sentence in sentences:
        words = preprocess_text(sentence)
        for word in words:
            counter[word] += 1

    return counter
```

II.3. N-grams

Trong xử lý ngôn ngữ tự nhiên, n-gram là một chuỗi gồm n từ liên tiếp trong văn bản. Đây là một kỹ thuật đơn giản nhưng rất hiệu quả để trích xuất ngữ cảnh cục bộ từ văn bản.



Hình 5: Minh họa cho n-gram

Hình trên minh hoạ cách xây dựng các n-gram từ một câu đơn giản:

- Unigram (n = 1): Mỗi từ được coi là một đơn vị độc lập. List unigram thu được gồm: [“This“, “is“, “a“, “sentence“]
- Bigram (n = 2): Các cặp từ liên tiếp được gom lại thành một đơn vị. Cửa sổ trượt độ dài 2 sẽ tạo ra: [“This is“, “is a“, “a sentence“]
- Trigram (n = 3): Ba từ liên tiếp sẽ được kết hợp thành một cụm. Danh sách trigram là: [“This is a“, “is a sentence“]

Trong xử lý ngôn ngữ, việc phân tích từng từ riêng lẻ (unigram) thường không đủ để nắm bắt ý nghĩa hoặc ngữ cảnh của văn bản. n-gram giúp mô hình nhận diện các cụm từ có ý nghĩa như

"New York" hay "machine learning", thay vì xem chúng là hai từ rời rạc. Nhờ đó, n-gram cung cấp một cách đơn giản nhưng hiệu quả để đưa ngữ cảnh ngắn hạn vào mô hình.

Các ứng dụng cụ thể của n-grams trong nlp gồm:

- Phát hiện cụm từ có ý nghĩa (collocations): Ví dụ: "machine learning", "deep learning", "new year",... không thể tách rời.
- Phân tích phong cách viết (author profiling): Tần suất xuất hiện của một số cụm từ là đặc trưng riêng biệt của từng người viết. Ví dụ, một tác giả có thể dùng nhiều "I think" hoặc "In my opinion" hơn người khác.
- Bổ sung ngữ cảnh ngắn hạn: Thay vì chỉ biết "fox" là một từ, mô hình biết rằng "quick brown fox" là một chuỗi thường gặp - đây là cách đơn giản để đưa ngữ cảnh vào biểu diễn từ.

II.4. Bài toán n-grams cụ thể

Mô tả yêu cầu bài toán

Problem: Xác định 2 văn bản có phải trùng 1 tác giả hay không qua công cụ n-gram và khoảng cách cosine similarity.
Input: Hàm nhận vào ba than số như sau.

- Văn bản chuẩn: Tác phẩm alice.txt của Lewis Carroll.
- Văn bản cần kiểm tra: Một hoặc nhiều văn bản bất kỳ được cung cấp dưới dạng .txt.
- Tham số: Giá trị n của n-gram (thường dùng bigram hoặc trigram). Ngưỡng độ tương đồng (threshold), mặc định là 0.8.

Output: Giá trị Cosine Similarity giữa văn bản cần kiểm tra và văn bản chuẩn (alice.txt). In ra màn hình kết quả.

- Nếu Cosine Similarity > threshold: "Có khả năng cùng tác giả".
- Ngược lại: "Khác tác giả".

Hướng dẫn từng bước

Tiền xử lý văn bản:

- Chuyển về chữ thường (lower()).
- Chuẩn hóa từ gốc bằng lemmatizer.
- Áp dụng stemming để rút gọn từ.

Tính n-gram: Sử dụng hàm count_word để thống kê tần suất xuất hiện các n-gram.
So sánh văn bản:

- Chuyển n-gram profile về vector.

- Tính Cosine Similarity giữa hai vector.

Kết luận: So sánh kết quả với ngưỡng đã đặt.

II.4.1. Số lượng từ tối thiểu trong văn bản để có thể thực hiện bài toán Author Attribution

Trước đây, việc xác định tác giả thường được áp dụng cho các văn bản dài như sách và thư. Một nghiên cứu của Forsyth và Holmes (1996) đã kết luận rằng một văn bản cần có độ dài tối thiểu 250 từ để các đặc điểm về văn phong (stylometric characteristics) có thể được nhận diện một cách rõ ràng.

Tuy nhiên, với sự phát triển của các thuật toán machine learning hiện đại như Support Vector Machines (SVMs) và mạng neural, các nghiên cứu gần đây đã chứng minh khả năng xác định tác giả thành công trên các văn bản rất ngắn. Ví dụ, các kỹ thuật này đã được áp dụng thành công cho các tin nhắn trên Twitter, vốn có độ dài trung bình dưới 25 từ. Trong báo cáo này, các mô hình machine learning đã có thể xác định tác giả của các đoạn trích có độ dài trung bình chỉ 18 từ với độ chính xác khoảng 60% [1], cao hơn 4.5 lần so với việc đoán ngẫu nhiên. Điều này chứng tỏ rằng các kỹ thuật hiện đại đã vượt qua giới hạn 250 từ trước đây, cho phép thực hiện bài toán xác định tác giả trên các văn bản ngắn hơn nhiều.

Bảng 0: Các tiểu thuyết, tác giả và số lượng từ tương ứng trong tập test ở phần II.5..

Author	Novel	Word Count
Louisa May Alcott	*Little Women*	185,662
Jane Austen	*Pride and Prejudice*	121,439
Jane Austen	*Emma*	157,320
Charlotte Brontë	*Jane Eyre*	184,440
Wilkie Collins	*The Woman in White*	244,380
Arthur Conan Doyle	*A Study in Scarlet*	43,192
Arthur Conan Doyle	*The Sign of the Four*	42,918
Arthur Conan Doyle	*The Hound of the Baskervilles*	59,027
L.M. Montgomery	*Anne of Green Gables*	101,982
L.M. Montgomery	*Anne of Avonlea*	90,676
Bram Stoker	*Dracula*	160,486
Mark Twain	*The Adventures of Tom Sawyer*	69,747
Mark Twain	*The Adventures of Huckleberry Finn*	109,933
Lewis Carroll	***Alice in the wonder Land***	**26,470**

II.4.2. Tiền xử lý văn bản

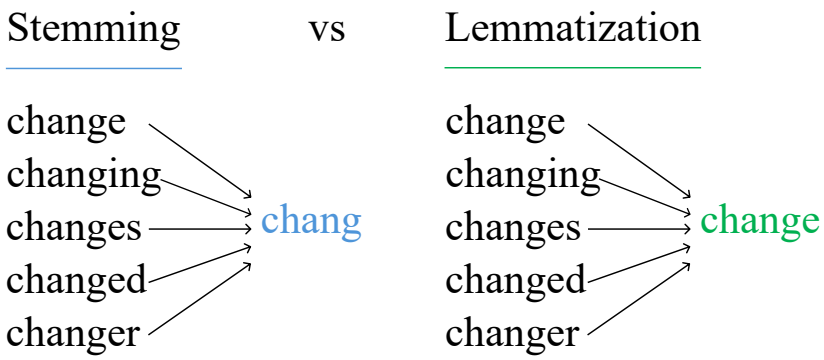
Trước khi xử lý văn bản theo hướng dẫn, chúng ta sẽ đến với hai khái niệm lemmatizer và stemming. Trong ngôn ngữ tự nhiên, một từ có thể có nhiều dạng khác nhau do chia thì, số nhiều, biến thể từ vựng... Cả Lemmatization và Stemming đều là các kỹ thuật giúp:

- Đưa các từ về dạng cơ bản nhất.
- Giảm số lượng từ cần xử lý.
- Giúp mô hình dễ phát hiện được các mẫu (patterns) trong văn bản.

Stemming là kỹ thuật cắt bỏ phần đuôi từ để đưa về gốc. Tuy nhiên Stemming:

- Không đảm bảo tạo thành một từ có nghĩa đầy đủ.
- Đơn giản, nhanh, nhưng có thể không chính xác hoàn toàn.

Lemmatization là kỹ thuật đưa từ về dạng từ điển chính xác nhất, còn gọi là lemma. Lemmatization phân tích cú pháp để đảm bảo kết quả là một từ có nghĩa thực sự.



Hình 6: Minh họa về stemming và lemmatization.

Vậy câu hỏi đặt ra lúc này là tại sao lại phải dùng cả 2 kỹ thuật này ? Câu trả lời như sau:

- Chỉ dùng Stemming: Cắt đuôi từ nhanh nhưng mất nghĩa dễ gây ra các từ sai. Làm giảm tính chính xác khi so sánh giữa các văn bản.
- Chỉ dùng Lemmatization: Tuy chính xác hơn nhưng hoạt động dựa trên từ điển nên không luôn xử lý triệt để mọi biến thể. Với một số dạng từ phức tạp, đặc biệt khi không gán rõ loại từ, Lemmatization có thể bỏ sót những biến đổi cần xử lý. Ngoài ra tốc độ xử lý của Lemmatization chậm hơn nhiều so với stemming.

Quy trình khi kết hợp hai kỹ thuật này sẽ là lemmatization trước để chuẩn hóa các từ về dạng gốc có nghĩa, đảm bảo không bị cắt méo ngữ nghĩa. Sau đó sử dụng stemming tiếp tục rút gọn các dạng từ còn lại, kể cả những từ Lemmatization chưa xử lý triệt để. Quy trình này giúp:

- Giảm thiểu biến thể từ không cần thiết.
- Hạn chế từ bị "méo dạng" do Stemming quá mạnh.
- Đảm bảo profile n-gram rõ ràng, đơn giản, dễ so sánh.
- Giữ lại tương đối chính xác đặc trưng phong cách ngôn ngữ của tác giả, thứ bạn cần để phân tích.

Để thực hiện stemming và lemmatization ta sẽ dùng thư viện **nltk** là một thư viện Python mạnh mẽ và phổ biến nhất dành cho xử lý ngôn ngữ tự nhiên (NLP). Với stemming ta dùng PorterStemmer và với lemmatization thì dùng WordNetLemmatizer

Stemming và lemmatization

```python
def init_nlp_tools():
    """
    Hàm khởi tạo và đảm bảo tài nguyên NLTK đầy đủ.

    Returns:
        tuple: (stemmer, lemmatizer)
    """
    # Tải các tài nguyên cần thiết nếu chưa có
    nltk.download('punkt')
    nltk.download('wordnet')

    # Khởi tạo công cụ
    stemmer = PorterStemmer()
    lemmatizer = WordNetLemmatizer()

    return stemmer, lemmatizer

def stem_lema(text):
    """
    Tiền xử lý văn bản:
    - Chuyển về chữ thường
    - Tokenize (tách từ)
    - Loại bỏ dấu câu
    - Lemmatization
    - Stemming

    Args:
        text (str): Văn bản đầu vào

    Returns:
        list: Danh sách từ đã được xử lý
    """
    text = text.lower()
    tokens = word_tokenize(text)
    processed_tokens = []
```

```
    stemmer, lemmatizer = init_nlp_tools()

    for token in tokens:
        # Bỏ qua token là dấu câu
        if token in string.punctuation:
            continue

        lemma = lemmatizer.lemmatize(token)
        stem = stemmer.stem(lemma)
        processed_tokens.append(stem)

    return processed_tokens
```

II.4.3. Tính n-gram

n-grams

```
def count_word(text, n=2):
    """
    Hàm tính n-gram profile của văn bản sau tiền xử lý.

    Args:
        text (str): Văn bản đầu vào
        n (int): Giá trị n của n-gram (mặc định là bigram)

    Returns:
        Counter: Bộ đếm tần suất các n-gram
    """
    tokens = stem_lema(text)  # Tiền xử lý văn bản
    n_gram_list = list(ngrams(tokens, n))
    profile = Counter(n_gram_list)

    return profile
```

II.4.4. Cosine similarity

Cosine Similarity (Độ tương đồng Cosine) là một thước đo phổ biến trong xử lý ngôn ngữ tự nhiên và khai phá văn bản. Nó dùng để đánh giá mức độ tương đồng giữa hai văn bản hoặc hai vector đặc trưng dựa trên góc giữa chúng trong không gian vector.
Trong bài toán này:

- Mỗi văn bản được biểu diễn dưới dạng vector đặc trưng dựa trên tần suất xuất hiện của các n-gram.
- So sánh vector của văn bản cần kiểm tra với vector của alice.txt (tác phẩm chuẩn của Lewis Carroll).
- Dùng Cosine Similarity để đánh giá mức độ tương đồng phong cách viết.

$$\text{Cosine Similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \times \|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Trong đó:

- $\mathbf{A}$, $\mathbf{B}$ là hai vector đặc trưng của hai văn bản.
- $\mathbf{A} \cdot \mathbf{B}$ là tích vô hướng của hai vector.
- $\|\mathbf{A}\|$, $\|\mathbf{B}\|$ là độ dài (norm) của từng vector.

Giả sử ta có 2 văn bản với bi-grams gồm các cặp: (I, love); (love, AI); (AI, Vietnam). Tần suất xuất hiện của các bi-grams trên ở văn bản thứ nhất lần lượt là: `[0, 1, 0]`, còn văn bản thứ hai lần lượt là: `[1, 1, 1]`. Áp dụng Cosine Similarity cho $\mathbf{A} = [0, 1, 0]$ và $\mathbf{B} = [1, 1, 1]$, ta có độ tương đồng cosine như sau:

- Bước 1: Tính tích vô hướng:

$$\mathbf{A} \cdot \mathbf{B} = 0 \cdot 1 \ + \ 1 \cdot 1 \ + \ 0 \cdot 1 = 1.$$

- Bước 2: Tính độ dài của từng vector:

$$\|\mathbf{A}\| = \sqrt{0^2 + 1^2 + 0^2} = 1, \quad \|\mathbf{B}\| = \sqrt{1^2 + 1^2 + 1^2} = \sqrt{3}.$$

- Bước 3: Thay vào công thức:

$$\text{Cosine}(\mathbf{A}, \mathbf{B}) = \frac{1}{1 \times \sqrt{3}} = \frac{1}{\sqrt{3}} \approx 0.577.$$

Cosine Similarity

```python
def cosine_similarity_sklearn(profile1, profile2):
    """
    Hàm tính Cosine Similarity giữa hai profile n-gram dùng sklearn.

    Args:
        profile1 (Counter): Bộ đếm n-gram của văn bản 1
        profile2 (Counter): Bộ đếm n-gram của văn bản 2

    Returns:
        float: Giá trị Cosine Similarity (0 đến 1)
    """

    # Lấy ra các n-gram giống nhau của 2 profile
    all_ngrams = set(profile1.keys()).union(set(profile2.keys()))

    vec1 = []
    vec2 = []

    for ngram in all_ngrams:
```

```
        vec1.append(profile1.get(ngram, 0))
        vec2.append(profile2.get(ngram, 0))

    # Biến đổi thành mảng 2D theo đúng yêu cầu của sklearn
    vec1 = np.array(vec1).reshape(1, -1)
    vec2 = np.array(vec2).reshape(1, -1)

    # Hàm cosine_similarity trả về 1 ma trận 2 chiều
    # Phần tử 0, 0 của ma trận là sim(vec1, vec2) của ta cần.
    similarity = cosine_similarity(vec1, vec2)[0][0]

    return similarity
```

Compare authors

```
def compare_authors(test_text):
    """
    Hàm chính để so sánh tác giả của hai văn bản.
    - Đọc file alice.txt
    - Nhận văn bản cần kiểm tra từ tham số
    - Tiền xử lý và tính n-gram
    - Tính Cosine Similarity
    - Trả về giá trị similarity
    """
    # Đọc file alice.txt
    with open("data/alice.txt", "r", encoding="utf-8") as f:
        alice_text = f.read()

    # Tiền xử lý và tính n-gram
    alice_tokens = stem_lema(alice_text)
    test_tokens = stem_lema(test_text)

    alice_profile = count_word(alice_tokens, n=2)
    test_profile = count_word(test_tokens, n=2)

    # Tính Cosine Similarity
    similarity = cosine_similarity_sklearn(alice_profile, test_profile)

    return similarity
```

II.4.5. Triển khai lên nền tảng streamlit

Streamlit là một framework mã nguồn mở được phát triển bằng ngôn ngữ Python, cho phép các nhà khoa học dữ liệu, kỹ sư AI/ML, và lập trình viên xây dựng ứng dụng web tương tác một cách nhanh chóng và dễ dàng – chỉ bằng code Python thuần.

Streamlit giúp bạn tạo ra giao diện người dùng chỉ bằng một vài dòng lệnh Python. Điều này cực kỳ hữu ích cho việc trình bày, trực quan hóa, và thử nghiệm mô hình một cách trực tiếp. Tạo giao diện cho streamlit như sauL:

Build streamlit UI

```python
def run_streamlit():
    st.set_page_config(page_title="Author Profiling - Lewis Carroll", layout="centered")

    st.title("Author Profiling - Lewis Carroll")

    st.markdown("""
    *Ứng dụng kiểm tra xem văn bản bạn cung cấp có khả năng cùng tác giả với **Lewis Carroll** hay không,*
    *dựa trên kỹ thuật phân tích văn bản và so sánh đặc trưng n-gram.*
    """)

    st.header("Bước 1: Chọn file văn bản cần kiểm tra")
    uploaded_file = st.file_uploader("Chọn file (.txt)", type="txt")

    if uploaded_file is not None:
        text = uploaded_file.read().decode('utf-8')
        st.text_area("Nội dung văn bản:", text, height=200)

        st.header("Bước 2: Kết quả kiểm tra")

        if st.button("Kiểm tra"):
            similarity = compare_authors(text)

            st.write(f"**Cosine Similarity:** {similarity:.4f}")

            if similarity > THRESHOLD:
                st.success("Kết luận: Có khả năng cùng tác giả Lewis Carroll.")
            else:
                st.warning("Kết luận: Khác tác giả Lewis Carroll.")
```

Hình 7: UI của streamlit khi build.

II.5. Test ứng dụng

Sử dụng ứng dụng

Chúng ta sẽ kiểm tra hiệu quả của ứng dựng với tập dữ liệu văn bản từ một repo trên github:

`https://github.com/gkhayes/author_attribution/tree/master/Books`

Kho dữ liệu này bao gồm nhiều tác phẩm văn học của các tác giả nổi tiếng như:

- Lewis Carroll
- Arthur Conan Doyle
- Jane Austen
- William Shakespeare
- Mark Twain
- Charles Dickens

Các văn bản từ kho dữ liệu trên được dùng để:

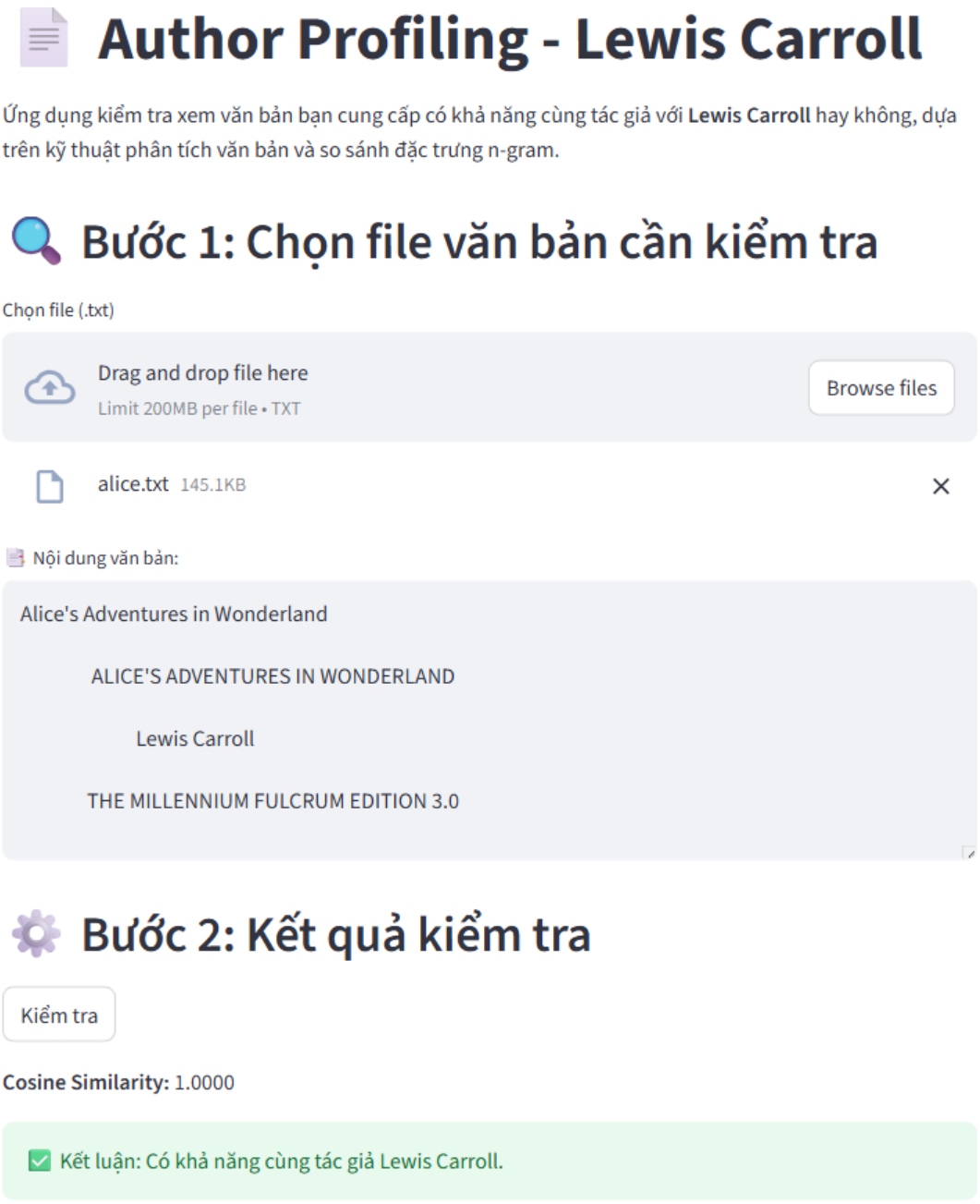
- Làm đầu vào kiểm tra ứng dụng với nhiều phong cách viết khác nhau
- Kiểm tra độ tương đồng Cosine giữa các tác giả khác nhau

- Đánh giá khả năng phân biệt của mô hình giữa văn bản của Lewis Carroll và các tác giả còn lại

Các bước kiểm tra ứng dụng:

- **Tải lên văn bản cần kiểm tra.** Người dùng chuẩn bị một file văn bản định dạng `.txt` (chẳng hạn như đoạn văn, bài viết, hay trích đoạn sách), sau đó chọn file từ máy tính và tải lên giao diện bằng cách sử dụng nút *"Chọn file (.txt)"*.
- **Xem trước nội dung văn bản.** Sau khi tải lên, nội dung file sẽ được hiển thị trong khung văn bản để người dùng kiểm tra lại.
- **Thực hiện kiểm tra tác giả.** Nhấn nút *"Kiểm tra"*, ứng dụng sẽ xử lý văn bản, thực hiện tiền xử lý, sinh n-gram, và tính toán độ tương đồng Cosine với văn bản chuẩn `alice.txt`.
- **Xem kết quả đánh giá.** Hệ thống hiển thị chỉ số **Cosine Similarity** giữa hai văn bản và đưa ra kết luận: liệu văn bản đầu vào có khả năng được viết bởi cùng một tác giả (Lewis Carroll) hay không.

Hình 8: Kết quả hai kiểm tra 2 văn bản tác giả (L.M.Montgomery vs Lewis Carroll).



Hình 9: Kết quả hai kiểm tra 2 văn bản cùng tác giả (Lewis Carroll vs Lewis Carroll).

III. Tài liệu tham khảo

[1] G. Hayes, *Author attribution report*, https://github.com/gkhayes/author_attribution/blob/master/Report.pdf, Accessed on June 27, 2025, 2018.

[2] G. Hayes, *Author attribution dataset - books*, https://github.com/gkhayes/author_attribution/tree/master/Books, Accessed on June 27, 2025, 2018.

Source code

Các bạn có thể download source code tại đây