AI VIET NAM
@aivietnam.edu.vn

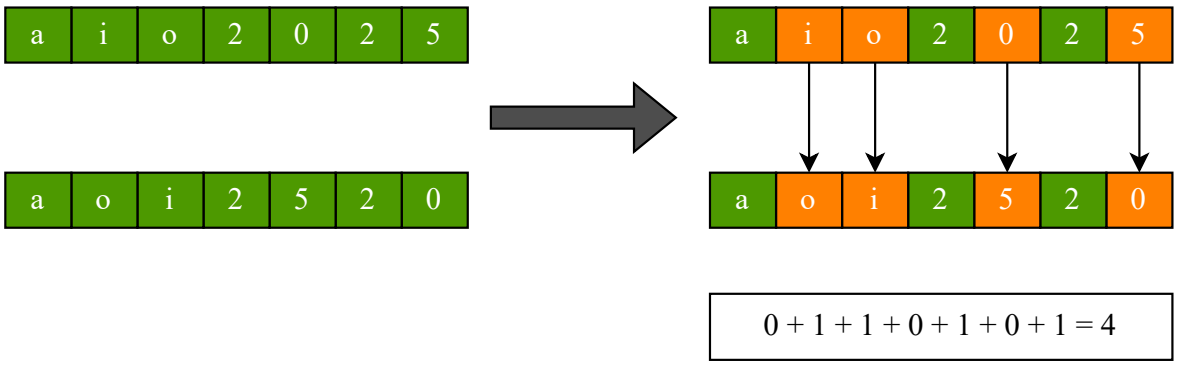
AI VIET NAM – AI COURSE 2025

Levenshtein Distance for Spell Checking

Duc-Huy Tran Phuc-Thinh Nguyen Quang-Vinh Dinh

I. Giới thiệu

Trong bối cảnh chuyển đổi số đang diễn ra mạnh mẽ ngày nay, sự toàn vẹn và chính xác của dữ liệu đã trở thành yếu tố then chốt và đôi khi còn là yếu tố sống còn trong một số lĩnh vực. Đặc biệt trong lĩnh vực y tế, nơi mỗi thông tin đều có thể tác động trực tiếp đến sức khỏe và tính mạng con người, vấn đề này càng trở nên quan trọng. Quá trình nhập liệu thủ công, dù là một phần không thể thiếu trong việc số hóa hồ sơ bệnh án, lại là nguồn gốc của những sai sót đáng lo ngại.



Hình 1: Minh họa về cách tính đơn giản khoảng cách Levenshtein.

Một nghiên cứu được công bố trên Thư viện Y khoa Quốc gia Hoa Kỳ (PMC) đã chỉ ra rằng tỉ lệ lỗi nhập liệu thủ công trung bình trong các hệ thống hồ sơ bệnh án điện tử là **2.8%** trên tổng số các trường dữ liệu, với tỉ lệ ở các trường dữ liệu riêng biệt dao động từ **0.5%** đến **6.4%** [1]. Với việc nhập liệu sai sót này, dù chỉ là lỗi chính tả do gõ nhầm (typographical error) cũng hoàn toàn có thể dẫn đến việc ghi nhận sai tên thuốc, sai lệch thông tin chẩn đoán, gây ra những rủi ro tiềm tàng và hậu quả nghiêm trọng cho bệnh nhân. Do đó, việc xây dựng một cơ chế hỗ trợ sửa lỗi chính tả thông minh và hiệu quả ngay tại khâu nhập liệu là một bài toán không chỉ mang ý nghĩa về mặt công nghệ mà còn có giá trị nhân văn sâu sắc.

Để giải quyết vấn đề này, khoảng cách Levenshtein từ lâu đã được công nhận là một thuật toán nền tảng và mạnh mẽ. Bằng cách tính toán số lượng thao tác tối thiểu (thêm, xóa, thay thế) để

biến một chuỗi ký tự này thành một chuỗi ký tự khác, Levenshtein cung cấp một thước đo định lượng về sự tương đồng giữa các từ. Mặc dù hiệu quả, phương pháp này tồn tại một hạn chế cố định: nó coi tất cả các thao tác thay thế ký tự có chi phí ngang bằng nhau. Điều này không phản ánh chính xác bản chất lỗi gõ phím của con người, nơi việc gõ nhầm một ký tự liền kề (ví dụ: "o" thành "p") xảy ra thường xuyên hơn nhiều so với việc gõ nhầm một ký tự ở xa (ví dụ: "o" thành "q").

Nhằm khắc phục nhược điểm trên và đề xuất các từ sửa lỗi gần gũi hơn với thói quen người dùng, bài viết này đề xuất một phương pháp cải tiến kết hợp hai tầng. Đầu tiên, hệ thống sẽ sử dụng thuật toán Levenshtein để nhanh chóng xác định và đề xuất 5 từ trong từ điển có khoảng cách ngắn nhất so với từ bị gõ sai. Tiếp theo, trong trường hợp có nhiều từ ứng viên cùng sở hữu khoảng cách Levenshtein bằng nhau, một mô hình chi phí dựa trên vị trí vật lý của các phím trên bàn phím QWERTY (Keyboard Cost Model) sẽ được áp dụng làm tiêu chí phân định. Theo đó, những từ mà sự sai khác ký tự tương ứng với các phím gần nhau sẽ được ưu tiên hơn. Sự kết hợp này hướng tới việc tạo ra một danh sách gợi ý không chỉ chính xác về mặt toán học mà còn thông minh và trực quan hơn.

Mục tiêu chính của bài viết là trình bày chi tiết việc thiết kế, xây dựng và thử nghiệm hệ thống gợi ý sửa lỗi chính tả dựa trên phương pháp lai ghép này. Các phần tiếp theo sẽ đi sâu vào cơ sở lý thuyết của khoảng cách Levenshtein, mô tả kiến trúc và cách triển khai mô hình chi phí bàn phím, trình bày các thực nghiệm được tiến hành để đánh giá hiệu quả của phương pháp đề xuất, và cuối cùng đưa ra những kết luận cho bài viết.

Mục lục

I. Giới thiệu **1**

II. Sửa lỗi chính tả bằng khoảng cách Levenshtein **4**

II.1. Giới thiệu 4

II.2. Tập dữ liệu và Từ điển 5

II.3. Phương pháp 6

III. Cải tiến với khoảng cách Levenshtein có trọng số bàn phím **8**

III.1. Ma trận khoảng cách bàn phím 8

III.2. Hàm khoảng cách cải tiến 12

III.3. Kết quả thực nghiệm 14

IV. Kết luận **15**

V. Tài liệu tham khảo **16**

A Phụ lục **16**

A1. Thông tin về words (Unix) 16

A2. Thông tin về Birkbeck Spelling Error Corpus 16

II. Sửa lỗi chính tả bằng khoảng cách Levenshtein

Để giải quyết bài toán sửa lỗi chính tả một cách có hệ thống, câu hỏi cơ bản nhất cần trả lời là: "Làm thế nào để đo lường mức độ 'gần gũi' hay 'tương đồng' giữa từ bị gõ sai và từ đúng trong từ điển?". Một cách tiếp cận trực quan và hiệu quả là coi từ bị sai là kết quả của một vài "lỗi" chỉnh sửa nhỏ từ một từ gốc. Các lỗi này thường rơi vào một trong ba trường hợp phổ biến của người dùng:

1. Bỏ sót một ký tự (ví dụ: gõ "helo" thay vì "hello").
2. Thêm một ký tự thừa (ví dụ: gõ "helllo" thay vì "hello").
3. Gõ nhầm một ký tự (ví dụ: gõ "hella" thay vì "hello").

Một trong những thuật toán kinh điển và mạnh mẽ nhất để lượng hóa chính xác các "lỗi" này là **khoảng cách Levenshtein (Levenshtein distance)**. Thuật toán này cung cấp một thước đo định lượng, phản ánh đúng bản chất của các lỗi gõ phím. Bằng cách tính toán số lượng thao tác chỉnh sửa tối thiểu để biến một chuỗi này thành chuỗi khác, chúng ta có thể tìm ra những từ trong từ điển có khả năng là từ đúng nhất bằng khoảng cách Levenshtein nhỏ nhất so với từ bị sai.

II.1. Giới thiệu

Khoảng cách Levenshtein là thước đo số thao tác đơn ký tự cần thiết (chèn, xóa, thay thế) để biến chuỗi này thành chuỗi kia. Giữa chuỗi S và chuỗi T là số bước ít nhất biến chuỗi S thành chuỗi T thông qua 3 phép biến đổi là:

- **Thêm (Insert)**: chi phí bằng $C_{\text{insert}} = 1.0$.
- **Xoá (Delete)**: chi phí bằng $C_{\text{delete}} = 1.0$.
- **Thay thế (Substitute)**: chi phí bằng 1 với kí tự khác nhau và bằng 0 với kí tự giống nhau $C_{\text{substitute}}(a, b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases}$.

Ví dụ khoảng cách giữa "kitten" và "sitting" là 3:

1. **k**itten → **s**itten (thay "k" thành "s")
2. sitt**e**n → sitt**i**n (thay "e" thành "i")
3. sittin → sittin**g** (chèn "g")

Hàm quy hoạch động để tính khoảng cách Levenshtein được viết như sau:

Tính khoảng cách Levenshtein

```python
def levenshtein_distance(token1, token2):
    distances = [[0] * (len(token2) + 1) for _ in range(len(token1) + 1)]

    for t1 in range(len(token1) + 1):
        distances[t1][0] = t1

    for t2 in range(len(token2) + 1):
        distances[0][t2] = t2

    for t1 in range(1, len(token1) + 1):
        for t2 in range(1, len(token2) + 1):
            if token1[t1 - 1] == token2[t2 - 1]:
                distances[t1][t2] = distances[t1 - 1][t2 - 1]
            else:
                a = distances[t1][t2 - 1]      # insert
                b = distances[t1 - 1][t2]      # delete
                c = distances[t1 - 1][t2 - 1]  # replace
                distances[t1][t2] = min(a, b, c) + 1

    return distances[len(token1)][len(token2)]
```

II.2. Tập dữ liệu và Từ điển

Để có thể đánh giá một cách khách quan hiệu quả của phương pháp đề xuất, việc lựa chọn và xây dựng bộ dữ liệu phù hợp là yếu tố tiên quyết. Trong phạm vi bài viết này, chúng ta sử dụng hai dữ liệu chính: một bộ từ điển tiếng Anh làm cơ sở tham chiếu và một tập kiểm thử lỗi chính tả tiêu chuẩn bằng tiếng Anh.

1. **Từ điển Tham chiếu (Reference Vocabulary)**: Từ điển tham chiếu đóng vai trò là ground truth cho các từ tiếng Anh hợp lệ. Hệ thống sẽ dựa vào đây để xác định một từ có khả năng bị sai chính tả (nếu từ đó không tồn tại trong từ điển). Đồng thời, dựa vào danh sách từ này, ta sẽ tạo ra một không gian các từ ứng viên (candidate words) để tìm kiếm và đề xuất từ sửa lỗi.

   Trong phạm vi bài viết này, chúng ta sử dụng bộ từ điển có sẵn từ corpus `words` của thư viện **Natural Language Toolkit (NLTK)** trong Python. Wordlist được tải xuống ở đoạn code dưới đây là tập hợp các từ tiếng Anh có phân biệt chữ viết hoa và viết thường, có cả chữ in hoa kí từ đầu (Allen, Monday...) và toàn in hoa (DNA) nhưng không có dấu câu ở giữa (Dr.). Ngoài ra nếu muốn phần mềm bên dưới chạy nhanh hơn, ta có thể sử dụng bộ "en-basic" với 850 chữ cái thường xuất hiện nhất trong tiếng Anh.

Tải bộ từ điển tiếng anh từ NLTK

```
!pip install nltk

import nltk
from nltk.corpus import words

nltk.download('words')

def load_english_vocabulary(name='en'):
    % vocabulary = set(words.words("en-basic"))
    vocabulary = set(words.words(name))
    print(f"Vocabulary loaded with {len(vocabulary)} words.")
    return vocabulary
```

2. **Corpus lỗi chính tả (Error Corpus)**: Để đánh giá chính xác khả năng sửa lỗi của hệ thống, chúng ta cần một tập dữ liệu gồm các cặp (từ_sai, từ_đúng). Thay vì tự tạo một cách thủ công, chúng ta sử dụng **Birkbeck Spelling Error Corpus**, một bộ dữ liệu tiêu chuẩn và được công nhận rộng rãi trong cộng đồng nghiên cứu về kiểm tra chính tả.

II.3. Phương pháp

Phương pháp đề xuất trong bài dựa trên một tiếp cận đơn giản nhưng hiệu quả, sử dụng khoảng cách Levenshtein để gợi ý sửa lỗi chính tả. Cụ thể, với mỗi từ được nhập vào (nghi ngờ có lỗi), hệ thống sẽ thực hiện các bước như sau:

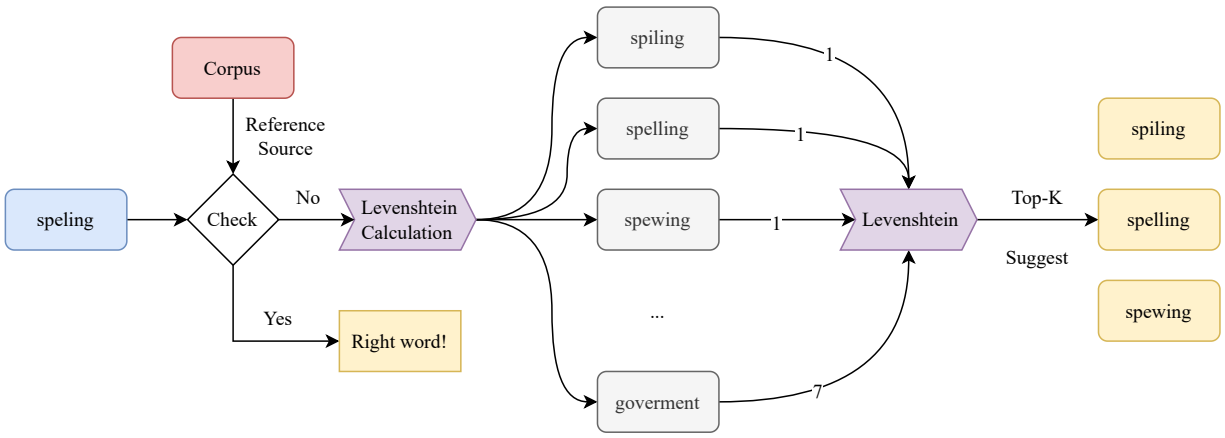
1. Với mỗi từ nghi ngờ w, tính khoảng cách Levenshtein đến tất cả các từ c trong từ điển tiếng Anh.

2. Lọc ra tập từ ứng viên C^* sao cho mỗi từ $c \in C^*$ có khoảng cách Levenshtein không vượt quá một ngưỡng cho trước (≤ 2).

3. Sắp xếp các từ ứng viên theo khoảng cách tăng dần và đề xuất k từ có khoảng cách nhỏ nhất (ví dụ: $k = 5$) làm kết quả gợi ý.

Hàm thực hiện đề xuất từ sửa lỗi

```python
def suggest_corrections(token,
                        vocabulary,
                        max_suggestions=5,
                        max_distance=2):
    token_lower = token.lower()
    if token_lower in vocabulary:
        return [token]

    # Return candidates with small Levenshtein distance
    candidates = []
    for word in vocabulary:
        # Calculate Levenshtein distance only if the \
            # length difference is within the allowed range
        # This helps to reduce the number of comparisons
        if abs(len(token_lower) - len(word)) <= max_distance:
            dist = levenshtein_distance(token_lower, word)
            if dist <= max_distance:
                candidates.append((word, dist))

    candidates.sort(key=lambda x: x[1])

    return [word for word, dist in candidates[:max_suggestions]]
```



Hình 2: Minh họa quy trình tìm từ đúng bằng khoảng cách Levenshtein.

Phương pháp trên xử lý khá tốt các lỗi gõ nhầm đơn giản. Chẳng hạn, với từ "wate" và ngưỡng Levenshtein $d = 1$, ta thu được các ứng viên "date", "late", "water", "wave". Nhưng phương pháp này chưa đủ để gợi ý các từ mà có nhiều phiên bản gần nhau, ví dụ như từ "speling" có 6 từ với $d = 1$ và > 200 từ với $d = 2$ trong bộ từ điển tiếng Anh.

Bảng 1: Kết quả sửa lỗi với khoảng cách Levenshtein.

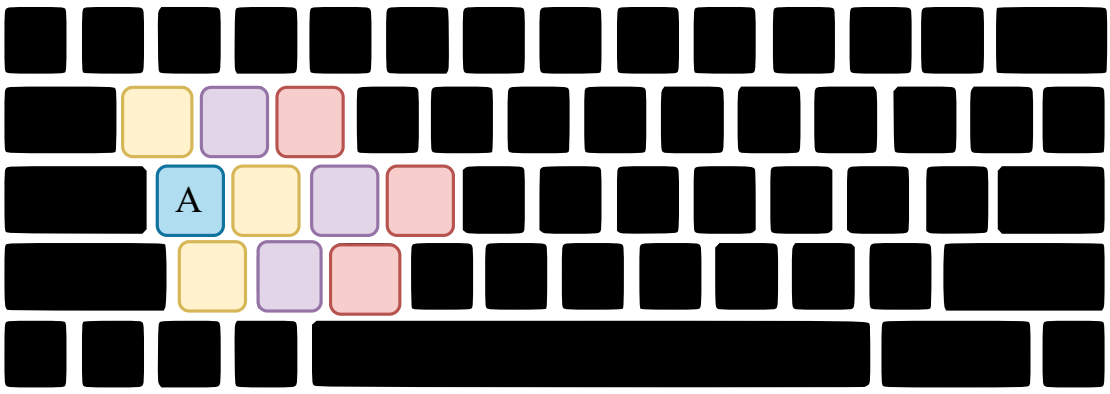
Từ sai	Ứng viên (d=1)	Top-1 đúng?
speling	spelling, spewing	Không
receeve	receive, recede	Có
happe	happy, happen, haste	Không

III. Cải tiến với khoảng cách Levenshtein có trọng số bàn phím

III.1. Ma trận khoảng cách bàn phím

Trong trường hợp có nhiều từ ứng viên cùng sở hữu khoảng cách Levenshtein bằng nhau so với từ gõ sai, việc chỉ dựa vào độ dài thao tác sẽ không đủ để lựa chọn từ gợi ý hợp lý nhất. Để giải quyết vấn đề này, bài viết đề xuất một phương pháp kết hợp, bổ sung thêm một mô hình chi phí dựa trên khoảng cách giữa các phím trên bàn phím vật lý QWERTY. Cách tiếp cận này cho phép hệ thống ưu tiên các từ có lỗi đánh máy gần nhau về mặt vị trí tay gõ.

Đầu tiên, bàn phím QWERTY được biểu diễn dưới dạng một đồ thị vô hướng, trong đó mỗi ký tự là một nút (node), và các cạnh nối giữa những phím liền kề theo hàng ngang hoặc hàng dọc. Hàm `build_keyboard_graph()` tạo ra đồ thị bàn phím bằng cách duyệt qua từng hàng trong bố cục QWERTY tiêu chuẩn, sau đó nối các ký tự liền kề nhau thành các cặp đỉnh có liên kết trực tiếp. Ví dụ, ký tự 'd' sẽ được nối với 's', 'f', và có thể là ký tự cùng cột ở hàng trên (nếu có).



Hình 3: Minh họa các khoảng cách bàn phím từ phím 'A '. Các phím với màu vàng mang khoảng cách 1 so với phím A, tăng dần với các phím màu tím và đỏ.

Hàm sinh ma trận kề của bàn phím

```python
from collections import deque
import pandas as pd

def build_keyboard_graph():
    """
    Create a graph (adjacency list) for the QWERTY keyboard.
    Each key is a node, and its value is a set of adjacent keys.

    Return:
        dict: A dictionary representing the keyboard graph.
    """
    layout = [
        "qwertyuiop",
        "asdfghjkl",
        "zxcvbnm"
    ]

    graph = {char: set() for row in layout for char in row}
    rows = len(layout)

    for r_idx, row in enumerate(layout):
        cols = len(row)
        for c_idx, char in enumerate(row):
            # Add neighbors in the above row
            if r_idx > 0 and c_idx < len(layout[r_idx - 1]):
                neighbor = layout[r_idx - 1][c_idx]
                graph[char].add(neighbor)
                graph[neighbor].add(char)

            # Add neighbors in the left
            if c_idx > 0:
                neighbor = layout[r_idx][c_idx - 1]
                graph[char].add(neighbor)
                graph[neighbor].add(char)

    return graph
```

Sau khi đồ thị được xây dựng, khoảng cách từ mỗi phím đến tất cả các phím còn lại được tính toán bằng thuật toán Breadth-First Search (BFS). Hàm `bfs_from_start_node()` thực hiện tìm kiếm theo chiều rộng từ một ký tự cho trước, và trả về một bảng ánh xạ giữa ký tự gốc với khoảng cách ngắn nhất đến từng ký tự khác. Để đảm bảo tính nhất quán của kết quả, các hàng xóm của mỗi phím được sắp xếp theo thứ tự chữ cái trước khi đưa vào hàng đợi.

Hàm tính khoảng cách bàn phím

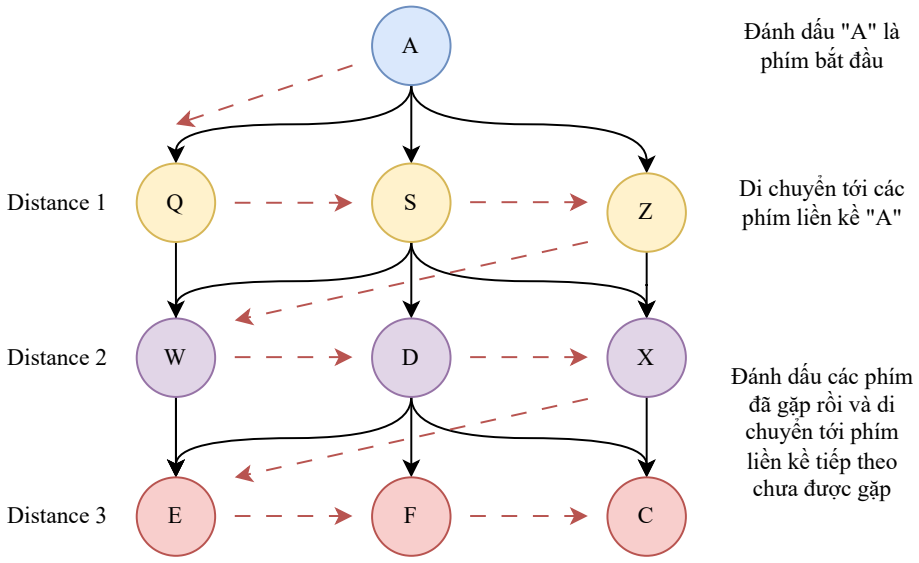
```python
def bfs_from_start_node(graph, start_node):
    """
    Run BFS from a start node to find distances to all other nodes.

    Input:
        graph (dict): The keyboard graph.
        start_node (str): The key from which to start the BFS.

    Return:
        dict: A dictionary with distances from the start node to all other nodes.
    """
    # Initialize distances with -1 (unvisited)
    distances = {node: -1 for node in graph}

    # Set the distance to the start node as 0
    distances[start_node] = 0

    queue = deque([start_node])

    while queue:
        current_node = queue.popleft()

        for neighbor in sorted(list(graph[current_node])):
            if distances[neighbor] == -1: # If node is unvisited
                distances[neighbor] = distances[current_node] + 1
                queue.append(neighbor)

    return distances
```

Tiếp theo, hàm `generate_distance_matrix()` sẽ lặp qua toàn bộ các phím có mặt trên bàn phím, và thực hiện BFS từ từng phím một. Kết quả là một ma trận hai chiều, trong đó mỗi dòng tương ứng với một ký tự xuất phát, và mỗi cột biểu diễn khoảng cách đến các ký tự còn lại. Ví dụ: khoảng cách giữa 'f' và 'g' sẽ là 1, nhưng khoảng cách giữa 'q' và 'm' sẽ lớn hơn nhiều.

Cuối cùng, ma trận này được lưu dưới dạng `DataFrame` của thư viện `pandas`, sắp xếp lại thứ tự hàng và cột theo thứ tự bảng chữ cái để tiện truy cập và sử dụng sau này. Ma trận sau đó được ghi ra file CSV để phục vụ cho việc tính toán chi phí thay thế có trọng số ở các bước tiếp theo. Mỗi ô trong file CSV tương ứng với `distances[char1][char2]`, là khoảng cách bàn phím giữa hai ký tự.

Hàm tính sinh ma trận khoảng cách và lưu file

```python
def generate_distance_matrix():
    """
    Generate a distance matrix for all keys on the keyboard.
    """
    keyboard_graph = build_keyboard_graph()
    all_chars = sorted(keyboard_graph.keys())

    distance_matrix = {}

    for char in all_chars:
        # Chạy BFS từ mỗi phím để lấy một hàng của ma trận
        distance_matrix[char] = bfs_from_start_node(keyboard_graph, char)

    return distance_matrix

final_matrix = generate_distance_matrix()

# Save the distance matrix to a csv file
df = pd.DataFrame(final_matrix)

# Sort the DataFrame by index (characters)
df = df.reindex(sorted(df.index))

df.to_csv("keyboard_distances.csv", index_label='char')
print("Distance matrix saved to 'keyboard_distances.csv'")
```



Hình 4: Minh họa cách thuật toán BFS tính khoảng của phím "A" trên bàn phím. Mũi tên tô đen là các phím liền kề nhau trên ma trận kề. Mũi tên đỏ là hướng di chuyển của thuật toán BFS.

Ma trận khoảng cách giữa các ký tự được xây dựng và lưu trữ dưới dạng file CSV, trong đó mỗi

ô biểu thị khoảng cách giữa hai ký tự trên bàn phím. Ma trận này được nạp vào dưới dạng cấu trúc `hashmap hai cấp`, sử dụng định danh dạng `distances[char1][char2]`.

Hàm load ma trận khoảng cách

```python
def load_distances_from_csv(filename="keyboard_distances.csv"):
    distances = {}
    with open(filename, 'r', newline='', encoding='utf-8') as f:
        reader = csv.reader(f)
        header = next(reader)[1:] # Skip the first column (character names)

        for row in reader:
            char1 = row[0]
            char1_distances = {}
            for i, dist_str in enumerate(row[1:]):
                char2 = header[i]
                char1_distances[char2] = int(dist_str)
            distances[char1] = char1_distances

    print(f"Keyboard distances loaded from {filename}.")
    return distances
```

III.2. Hàm khoảng cách cải tiến

Chi phí của các phép biến đổi được định nghĩa như sau:

- **Thêm**: chi phí cố định $C_{\text{insert}} = 1.0$
- **Xoá**: chi phí cố định $C_{\text{delete}} = 1.0$
- **Thay thế**: chi phí thay đổi tuỳ thuộc vào khoảng cách giữa hai ký tự trên bàn phím. Ta cần chi phí này nằm trong khoảng $[0, 1]$ vì nếu chi phí vượt qua khoảng này, sẽ lấn át hai phép biến đổi thêm và xóa được nói ở trên. Ta thiết lập tham số đơn giản như sau:
  - Khoảng cách $= 1 \Rightarrow C_{\text{sub}} = 0.8$
  - Khoảng cách $= 2 \Rightarrow C_{\text{sub}} = 0.9$
  - Khoảng cách > 2 hoặc không xác định $\Rightarrow C_{\text{sub}} = 1.0$

Để tận dụng lại cấu trúc đã có, phương pháp gợi ý sửa lỗi được tái sử dụng từ hàm `suggest_corrections` đã trình bày trong phần trước. Tuy nhiên, điểm khác biệt duy nhất trong phương pháp cải tiến là thay thế hàm tính khoảng cách Levenshtein tiêu chuẩn bằng phiên bản cải tiến có xét đến khoảng cách bàn phím. Kết quả trả về là danh sách từ được đề xuất có chi phí nhỏ nhất, không chỉ xét về độ giống ký tự mà còn xét cả khả năng xảy ra lỗi bàn phím – giúp cải thiện đáng kể độ chính xác của hệ thống trong thực tế.

Bảng 2: Kết quả sửa lỗi với khoảng cách Levenshtein với trọng số bàn phím.

Từ sai	Ứng viên (top-3)	Top-1 đúng?
Ameraca	America, amerce, araca	Có
bsn	ban, ben, bin	Có
hsllo	hello, hollo, sloo	Có

Hàm tính khoảng cách Levenshtein cải tiến

```python
def keyboard_levenshtein_distance(token1, token2, keyboard_distances):
    COST_INSERT = 1.0
    COST_DELETE = 1.0

    token1 = token1.lower()
    token2 = token2.lower()

    distances = [[0.0] * (len(token2) + 1) for _ in range(len(token1) + 1)]

    for t1 in range(len(token1) + 1):
        distances[t1][0] = float(t1) * COST_DELETE

    for t2 in range(len(token2) + 1):
        distances[0][t2] = float(t2) * COST_INSERT

    for t1 in range(1, len(token1) + 1):
        for t2 in range(1, len(token2) + 1):
            char1 = token1[t1 - 1]
            char2 = token2[t2 - 1]

            if char1 == char2:
                distances[t1][t2] = distances[t1 - 1][t2 - 1]
            else:
                key_dist = keyboard_distances.get(char1, {}).get(char2, 99)

                if key_dist == 1:
                    cost_replace = 0.8
                elif key_dist == 2:
                    cost_replace = 0.9
                else:
                    cost_replace = 1.0

                cost_del = distances[t1 - 1][t2] + COST_DELETE
                cost_ins = distances[t1][t2 - 1] + COST_INSERT
                cost_sub = distances[t1 - 1][t2 - 1] + cost_replace

                distances[t1][t2] = min(cost_del, cost_ins, cost_sub)

    return distances[len(token1)][len(token2)]
```

III.3. Kết quả thực nghiệm

Để đánh giá hiệu quả của phương pháp cải tiến và so sánh với phương pháp cơ sở, chúng ta sẽ tiến hành các thử nghiệm trên tập dữ liệu lỗi chính tả **Birkbeck** và từ điển `en-basic` đã được xây dựng. Các chỉ số đánh giá chính là độ chính xác Top-1 (Top-1 accuracy) và Top-5 (Top-5 accuracy), đại diện cho tỷ lệ các trường hợp từ đúng được đề xuất ở vị trí thứ nhất và trong 5 vị trí đầu tiên tương ứng. Tham số được sử dụng bao gồm khoảng cách tối đa cho gợi ý là 2.

Đọc bộ dữ liệu Birkbeck

```
path = "./missp.dat.txt"

with open(path, 'r', encoding='utf-8') as f:
    text = f.read()

pairs = [] # (misspell, correct)
current_correct = None
for token in text.split():
    if token.startswith('\$'):
        current_correct = token[1:]
    else:
        pairs.append((token.lower(), current_correct.lower()))
```

Đánh giá phương pháp cải tiến

```
from collections import Counter
from tqdm import tqdm

vocabulary = load_english_vocabulary('en-basic')

def evaluate_improve(pairs, suggester, topk):
    hits = 0
    for mis, cor in tqdm(pairs, desc=f"Evaluating Top-{topk}"):
        suggestions = suggester(mis, vocabulary, keyboard_distances,
                                max_suggestions=topk, max_distance=2)
        if cor in suggestions[:topk]:
            hits += 1
    return hits / len(pairs)

top1 = evaluate_improve(pairs, suggest_corrections_improve, 1)
top5 = evaluate_improve(pairs, suggest_corrections_improve, 5)

print(f"Top-1 accuracy: {top1:.3%}")
print(f"Top-5 accuracy: {top5:.3%}")
```

Tiếp theo, chúng ta tiến hành đánh giá phương pháp sửa lỗi chính tả cơ sở (`suggest_corrections`) với cùng các chỉ số và tập dữ liệu để có cơ sở so sánh.

Đánh giá phương pháp cơ sở

```python
def evaluate(pairs, suggester, topk):
    hits = 0
    for mis, cor in tqdm(pairs, desc=f"Evaluating Top-{topk}"):
        suggestions = suggester(mis, vocabulary,
                                max_suggestions=topk, max_distance=2)
        if cor in suggestions[:topk]:
            hits += 1
    return hits / len(pairs)

top1 = evaluate(pairs, suggest_corrections, 1)
top5 = evaluate(pairs, suggest_corrections, 5)

print(f"Top-1 accuracy: {top1:.3%}")
print(f"Top-5 accuracy: {top5:.3%}")
```

Bảng 3: Kết quả độ chính xác của hai phương pháp sửa lỗi chính tả.

Phương pháp	Top-1 Accuracy (%)	Top-5 Accuracy (%)
Phương pháp cải tiến	4.099	5.970
Phương pháp cơ sở	4.173	5.978

Kết quả cho thấy, phương pháp cơ sở đạt **4.173%** độ chính xác Top-1 và **5.978%** độ chính xác Top-5. So sánh với phương pháp cải tiến, phương pháp cơ sở cho thấy hiệu suất nhỉnh hơn một chút về độ chính xác trên cả Top-1 và Top-5. Điều này có thể được lý giải là do tập từ điển "en-basic" chỉ với 850 ký tự không đủ để có thể làm tham chiếu cho toàn bộ quá trình đánh giá. Đồng thời các loại lỗi trong Birkbeck Spelling Error Corpus phần lớn không phải là lỗi gõ phím (typo) mà là lỗi ngữ pháp và từ vựng.

IV. Kết luận

Những sai sót tưởng chừng nhỏ như lỗi chính tả khi nhập liệu không chỉ ảnh hưởng đến chất lượng thông tin mà còn có thể dẫn đến hậu quả nghiêm trọng đối với sức khỏe người dùng. Xuất phát từ thực tiễn đó, bài viết đã đề xuất một giải pháp sửa lỗi chính tả dựa trên phương pháp kết hợp: sử dụng khoảng cách Levenshtein để xác định các từ gần đúng, và tiếp tục sàng lọc các ứng viên này bằng mô hình chi phí thao tác dựa trên khoảng cách thực tế giữa các phím trên bàn phím QWERTY. Cách tiếp cận hai tầng này vừa giữ được tính hiệu quả, đơn giản của Levenshtein truyền thống, vừa tăng độ chính xác nhờ yếu tố hành vi người dùng trong thao tác gõ phím.

V. Tài liệu tham khảo

[1] M. H. et al., "Error rates in a clinical data repository: Lessons from the transition to electronic data transfer—a descriptive study", 2013. [Online]. Available: https://bmjopen.bmj.com/content/3/5/e002406.

[2] P. Norvig, "How to write a spelling corrector", 2007. [Online]. Available: http://norvig.com/spell-correct.html.

[3] C. D. Manning et al., *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999, ISBN: 0262133601.

Source code

Các bạn có thể download source code tại đây

A Phụ lục

A1. Thông tin về words (Unix)

wordlist (thường gọi đơn giản là *"words"* trong Unix) là một danh sách văn bản thuần chứa hàng trăm nghìn mục từ tiếng Anh, mỗi dòng đúng một từ. Thuở đầu, tệp này phục vụ cho các tiện ích dòng lệnh như `spell` hoặc `look` để kiểm tra chính tả và tra cứu nhanh; về sau nó trở thành nguồn than chiếu phổ dụng cho các trò chơi chữ, script xử lý văn bản và nghiên cứu NLP. Tệp không bao gồm tần suất xuất hiện, nhãn ngữ nghĩa hay thông tin ngữ pháp, do đó chủ yếu thích hợp cho các thao tác tra cứu đơn giản, lọc chính tả sơ bộ hoặc khởi tạo bộ từ điển gốc.

A2. Thông tin về Birkbeck Spelling Error Corpus

Birkbeck Spelling Error Corpus gồm các tệp văn bản do học viên tiếng Anh tạo, với mỗi lỗi được chú thích cùng từ đúng tương ứng, là tài nguyên chuẩn để đánh giá công cụ sửa lỗi chính tả.

- Hết -