# Model Vietnamese Correction

Link project : <https://github.com/MTfixbug/Vietnamese-correct>

Link model :

model 1 : <https://huggingface.co/Trinity2105/modelkerasVNcorect>

model 2 : <https://huggingface.co/Trinity2105/finalprojectVNcorect>

Link dataset 1: <https://www.kaggle.com/datasets/tanphuc/trainplusdataset>

Link dataset 2: https://huggingface.co/datasets/Trinity2105/datasetmodel

**I.Ý tưởng** : Kết hợp LSTM và PhoBART để Tăng Độ Chính Xác của Mô Hình Sửa Lỗi Chính Tả

Cụ thể ý tưởng này là xây dựng một pipeline sửa lỗi chính tả kết hợp giữa hai mô hình học sâu khác nhau – một mô hình LSTM và một mô hình PhoBART đã được tinh chỉnh. Pipeline này có thể cải thiện độ chính xác bằng cách tận dụng sức mạnh của cả hai loại mô hình.

#### **Quy trình Hoạt động của Pipeline**

1. **Chuẩn bị dữ liệu đầu vào:** Chuỗi văn bản đầu vào được chuẩn hóa (chuyển thành chữ thường, loại bỏ các ký tự không mong muốn, tách từ viết hoa nếu cần) để đảm bảo tính nhất quán.
2. **Bước đầu tiên - Mô hình LSTM:**
   1. Đầu vào văn bản sẽ đi qua mô hình LSTM. Mô hình LSTM hoạt động trên các n-gram (các cụm từ có kích thước cố định) của câu, thực hiện dự đoán từ hoặc cụm từ dựa trên ngữ cảnh.
   2. Hàm batch\_predict chuyển đổi văn bản đầu vào thành một định dạng ma trận để mô hình LSTM dự đoán. Kết quả đầu ra là một danh sách các từ/cụm từ đã được sửa lỗi.
3. **Ghép kết quả của LSTM:**
   1. Từ những từ dự đoán bởi LSTM, hệ thống ghép các từ lại thành một câu hoàn chỉnh bằng cách chọn từ có xác suất cao nhất từ các dự đoán của mô hình LSTM.
   2. Đoạn văn bản này là kết quả đầu ra từ mô hình LSTM và sẽ được chuyển đến mô hình PhoBART.
4. **Bước thứ hai - Mô hình PhoBART tinh chỉnh:**
   1. Văn bản đầu ra từ LSTM được dùng làm đầu vào cho mô hình PhoBART. PhoBART sẽ tinh chỉnh lại các từ và cụm từ để đảm bảo ngữ cảnh và ngữ nghĩa tốt hơn.
   2. Mô hình PhoBART đã được tinh chỉnh cho các nhiệm vụ như sửa lỗi chính tả hoặc tạo văn bản trong tiếng Việt, giúp tăng cường độ chính xác của văn bản đầu ra.
5. **Kết quả cuối cùng:**
   1. Mô hình PhoBART trả về văn bản sau khi đã được tinh chỉnh, đây là kết quả cuối cùng từ pipeline sửa lỗi.
   2. Kết quả cuối cùng được gửi lại dưới dạng phản hồi trong API.

#### **Lợi Ích của Pipeline LSTM - PhoBART**

* **Khả năng xử lý ngữ cảnh:** LSTM giúp cải thiện tính chính xác trên cấp độ ngữ cảnh ngắn (các n-gram) và PhoBART củng cố độ chính xác tổng thể của văn bản.
* **Tối ưu hóa quá trình sửa lỗi:** Bằng cách xử lý văn bản qua hai mô hình, hệ thống giảm thiểu sai sót và cải thiện chất lượng văn bản đầu ra so với chỉ dùng một mô hình.
* **Tích hợp linh hoạt trong API:** Toàn bộ quá trình được đóng gói thành một API sử dụng FastAPI, dễ triển khai và tích hợp với các hệ thống khác.

#### **Triển khai**

* **FastAPI** cho phép triển khai API dễ dàng để thực hiện các bước xử lý và trả về kết quả chỉnh sửa cho người dùng cuối.
* **Ngrok** cung cấp một đường dẫn công khai để kiểm tra và dùng thử API trong môi trường phát triển.

**II.Mô hình Fine-tune BARTpho**

## **1. Giới thiệu**

BARTpho được VinAI Research giới thiệu vào năm 2021 với hai phiên bản: BARTpho-syllable và BARTpho-word. BARTpho tận dụng kiến trúc và các trọng số đã huấn luyện trước của mô hình BART, điều này giúp mô hình phù hợp với nhiều nhiệm vụ tạo ngôn ngữ tự nhiên khác nhau, đặc biệt là trong tiếng Việt. Mô hình BARTpho thực hiện phương pháp tiền huấn luyện của BART gồm hai bước chính: (i) làm nhiễu văn bản đầu vào bằng cách áp dụng một hàm gây nhiễu và (ii) học cách khôi phục lại văn bản gốc, từ đó tối ưu hóa entropy chéo giữa đầu ra của bộ giải mã và văn bản ban đầu.

## **2. Kiến trúc**

Cả hai mô hình BARTpho-syllable và BARTpho-word đều được xây dựng trên kiến trúc mã hóa-giải mã với 12 lớp, được huấn luyện theo quy trình gây nhiễu hai bước của BART. Cụ thể, BARTpho sử dụng hai loại hàm gây nhiễu: làm đầy văn bản (text infilling) và hoán vị câu (sentence permutation). Đối với làm đầy văn bản, một số đoạn văn bản có độ dài khác nhau, được lấy mẫu từ phân phối Poisson (λ=3.5), sẽ được thay thế bằng ký hiệu đặc biệt `<mask>`. Đối với hoán vị câu, các câu liên tiếp được nhóm thành các khối 512 ký tự và sau đó được xáo trộn ngẫu nhiên trong từng khối. BARTpho cũng thêm một lớp chuẩn hóa ở cả các lớp mã hóa và giải mã, dựa trên mô hình mBART.

## **3. Triển khai**

### **a. Cài đặt môi trường và phụ thuộc**

Tập lệnh yêu cầu cài đặt các thư viện như 'datasets', 'evaluate', 'accelerate', 'sacrebleu', và 'jiwer' để quản lý dữ liệu, đánh giá mô hình, và tối ưu hóa quá trình huấn luyện. Ngoài ra, nó sử dụng 'huggingface\_hub' để kết nối với Hugging Face Hub và các thư viện quan trọng như 'torch', 'transformers', và 'numpy'.

### **b. Tải dữ liệu và Token hóa**

Tập lệnh tải một tập dữ liệu có tên 'Trinity2105/datasetmodel' từ Hugging Face Hub, chứa dữ liệu văn bản để huấn luyện và đánh giá. Sử dụng 'AutoTokenizer' với mô hình 'vinai/bartpho-syllable' (một mô hình tiếng Việt đã huấn luyện trước) để thực hiện quá trình token hóa. Các hàm tiền xử lý cặp văn bản đầu vào và đầu ra dưới dạng token, giới hạn mỗi chuỗi không quá 512 token (MAX\_LENGTH).

### **c. Hàm tiền xử lý và Token hóa**

Hàm 'preprocess\_function' thực hiện token hóa các cặp văn bản đầu vào-đầu ra, giới hạn độ dài để tránh chuỗi quá dài. Hàm này được áp dụng theo lô để tối ưu hóa tốc độ token hóa cho toàn bộ tập dữ liệu.

### **d. Thiết lập mô hình**

Tập lệnh khởi tạo mô hình Seq2Seq, cụ thể là 'AutoModelForSeq2SeqLM', với cấu hình bartpho-syllable, tối ưu hóa cho các tác vụ chuyển đổi văn bản thành văn bản trong tiếng Việt.

### **e. Hàm đánh giá và chỉ số**

Tập lệnh sử dụng 'sacrebleu' làm chỉ số đánh giá, cho phép so sánh chất lượng dự đoán với văn bản tham chiếu. Hàm 'compute\_metrics' tính toán điểm BLEU và thực hiện một số bước xử lý trước (như loại bỏ khoảng trắng).

# **III.Mô Hình LSTM-Based**

## **1. Giới thiệu**

Mục tiêu chính của mô hình này là học cách sửa lỗi chính tả tiếng Việt một cách tự động, tập trung vào việc sửa lỗi ở cấp độ bigram (cặp hai từ liên tiếp). Điều này có nghĩa là mô hình được huấn luyện để xác định và sửa các lỗi chính tả trong bigram bằng cách dự đoán bigram đúng từ một bigram bị lỗi.  
   
- Lý do chọn cấp độ bigram: Lỗi chính tả trong tiếng Việt có thể bị ảnh hưởng bởi ngữ cảnh từ liền kề. Việc dùng bigram cho phép mô hình xem xét mối liên kết giữa các từ trong ngữ cảnh gần, giúp sửa lỗi chính xác hơn so với xử lý ở cấp độ từ đơn lẻ.  
 - Kiến trúc: Kiến trúc được sử dụng là mạng encoder-decoder với LSTM hai chiều. Bộ mã hóa LSTM nhận đầu vào là bigram bị lỗi, trong khi bộ giải mã Bidirectional LSTM sử dụng thông tin ngữ cảnh xung quanh để tái tạo bigram đúng.  
   
**2. Kiến trúc**

### **a. Encoder (LSTM)**

- Đầu vào:  
 - Bigram bị lỗi được mã hóa dưới dạng vector one-hot cho mỗi ký tự.  
 - Mỗi ký tự trong bigram được biểu diễn bằng một vector one-hot có chiều dài 199, tương ứng với số lượng ký tự trong bảng chữ cái tiếng Việt (bao gồm cả các ký tự dấu và dấu câu).  
 - Bigram được padding bằng ký tự null ('\x00') để đạt độ dài cố định là 40 ký tự.  
   
- LSTM Layer:  
 - Lớp LSTM với 256 units được sử dụng để xử lý chuỗi đầu vào tuần tự.  
 - Ở mỗi bước thời gian, LSTM nhận một vector one-hot đại diện cho một ký tự trong bigram và cập nhật trạng thái ẩn của nó.  
 - return\_sequences=True đảm bảo rằng LSTM trả về chuỗi các trạng thái ẩn cho tất cả các bước thời gian. Điều này quan trọng vì ta muốn bảo toàn toàn bộ thông tin ngữ cảnh của bigram, thay vì chỉ lấy trạng thái cuối cùng.  
   
- Đầu ra:  
 - Một chuỗi các vector ẩn với chiều dài 256 tại mỗi bước thời gian, chứa đặc trưng của bigram đầu vào. Chuỗi này được chuyển tiếp cho bộ giải mã.  
   
  
**b. Decoder (Bidirectional LSTM)**

- Đầu vào: Chuỗi các vector ẩn từ encoder, cung cấp thông tin ngữ cảnh toàn diện về bigram bị lỗi.  
   
- Bidirectional LSTM Layer:  
 - Lớp Bidirectional LSTM gồm hai lớp LSTM riêng biệt:  
 - Một LSTM xử lý chuỗi theo chiều xuôi (từ trái qua phải).  
 - Một LSTM xử lý chuỗi theo chiều ngược (từ phải qua trái).  
 - Đầu ra của hai LSTM này được kết hợp lại (thường là bằng cách nối các vector) để tạo ra biểu diễn ngữ cảnh phong phú hơn.  
 - Việc sử dụng Bidirectional LSTM giúp mô hình nắm bắt thông tin từ trước và sau mỗi ký tự trong bigram, giúp cải thiện độ chính xác trong việc dự đoán các từ trong ngữ cảnh đầy đủ.  
   
- Dropout:  
 - Một lớp Dropout với dropout=0.2 được thêm vào để ngăn chặn overfitting. Trong quá trình huấn luyện, 20% các unit trong LSTM sẽ bị "tắt" ngẫu nhiên, giúp mô hình học các đặc trưng phân tán hơn và giảm thiểu sự phụ thuộc vào một số unit cụ thể.  
   
- Đầu ra: Chuỗi các vector ẩn với chiều dài 512 (256 từ LSTM xuôi và 256 từ LSTM ngược), chứa đặc trưng của bigram bị lỗi.  
   
  
**c. TimeDistributed Dense Layers**

- TimeDistributed(Dense(256), activation='relu'):  
 - Lớp Dense này với 256 units và hàm kích hoạt ReLU được áp dụng cho mỗi bước thời gian của chuỗi đầu ra từ decoder.  
 - Việc sử dụng hàm ReLU giúp tăng tính phi tuyến cho mô hình, làm cho mô hình học được các đặc trưng phức tạp hơn của ngữ cảnh.  
   
- TimeDistributed(Dense(199), activation='softmax'):  
 - Tương tự, lớp Dense với 199 units (bằng với số ký tự trong bảng chữ cái tiếng Việt) và hàm kích hoạt softmax được áp dụng cho mỗi bước thời gian.  
 - Hàm softmax chuyển đổi đầu ra thành một phân phối xác suất trên 199 ký tự, giúp mô hình dự đoán ký tự nào có khả năng xuất hiện nhất tại mỗi vị trí trong bigram đúng.  
   
**3. Cài đặt**

### **a. Tiền xử lý dữ liệu**

- latin\_extract: Hàm này lọc dữ liệu để chỉ giữ lại các câu chứa ký tự Latin và có độ dài lớn hơn 2 từ. Điều này giúp loại bỏ những câu không phù hợp và tối ưu hóa dữ liệu cho mô hình.  
   
- \_extract\_phrases: Trích xuất các cụm từ từ dữ liệu đã lọc. Quá trình này giúp chuẩn bị các đoạn văn bản ngắn chứa ngữ cảnh cho quá trình tạo bigram.  
   
- generate\_bi\_grams: Tạo các bigram từ các cụm từ đã trích xuất. Các bigram có độ dài vượt quá giá trị MAXLEN sẽ bị loại bỏ, đảm bảo tính đồng nhất trong dữ liệu đầu vào của mô hình.  
   
- add\_noise: Đây là hàm quan trọng nhất trong việc tạo các lỗi chính tả nhân tạo. Hàm này sử dụng các kiểu lỗi thường gặp như lỗi gõ, lỗi vùng miền và teencode để tạo ra các bigram bị lỗi. Tham số pivot1 và pivot2 cho phép điều chỉnh tỷ lệ lỗi được thêm vào, tạo ra dữ liệu huấn luyện phong phú và sát với thực tế.

### **b. Data Generator (generate\_data)**

- generate\_data: Hàm này tạo ra các batch dữ liệu dùng cho quá trình huấn luyện và validation.  
 - Hàm sử dụng add\_noise để tạo ra bigram bị lỗi và encoder\_data để mã hóa bigram bị lỗi và bigram đúng thành dạng one-hot.  
 - Generator giúp tránh việc load toàn bộ dữ liệu vào bộ nhớ, đặc biệt hữu ích khi xử lý tập dữ liệu lớn, đồng thời tối ưu hiệu suất trong quá trình huấn luyện.  
   
**c. Huấn luyện mô hình**

- model.fit: Quá trình huấn luyện mô hình sử dụng dữ liệu từ generator, giúp mô hình học cách sửa các lỗi trong bigram thông qua nhiều epoch.  
   
- steps\_per\_epoch: Xác định số lượng batch dữ liệu được sử dụng trong mỗi epoch. Tham số này rất quan trọng để đảm bảo tính hiệu quả và tốc độ của quá trình huấn luyện.  
   
- validation\_steps: Xác định số lượng batch dữ liệu validation được sử dụng trong mỗi epoch, giúp đánh giá hiệu suất của mô hình trên dữ liệu chưa nhìn thấy.  
   
- callbacks=[checkpointer]: Callback này cho phép lưu lại mô hình sau mỗi epoch, giúp duy trì trạng thái tốt nhất của mô hình trong quá trình huấn luyện và dễ dàng khôi phục khi cần thiết.