

BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC KINH TẾ HỒ CHÍ MINH
KHOA CÔNG NGHỆ THÔNG TIN KINH DOANH



**ĐỒ ÁN MÔN HỌC
XỬ LÝ NGÔN NGỮ TỰ NHIÊN**

Đề tài: NHẬN DIỆN TIN NHẮN RÁC

Thành viên: Phạm Phú Khánh 31211025006
Nguyễn Trịnh Hiếu Kiên 31211027199

Giảng viên: TS.Đặng Ngọc Hoàng Thành

Thành phố Hồ Chí Minh , ngày 18 tháng 11 năm 2023

Mục Lục

Chương I: Tổng Quan Đề Tài	1
Chương II: Cơ Sở Lý Thuyết	1
2.1. Tiền xử lý dữ liệu	1
2.1.1. Text Mining :	1
2.2. Các mô hình học máy	2
2.2.1. Naïve Bayes	2
2.2.2. Support Vector Machine (SVM)	3
2.2.3. Decision Tree	5
2.2.4. Logistic Regression	6
2.2.5. Long Short Term Memory (LSTM)	7
Chương III: Tiền xử lý dữ liệu	8
3.1. Tổng quan bộ dữ liệu	8
3.2. Tiền xử lý dữ liệu	9
3.2.1. Kiểm tra dữ liệu bị thiếu	9
3.2.2. Xử lý dữ liệu thiếu và trùng lặp	9
3.3. Khai phá văn bản (Text Mining)	11
3.3.1. Porter Stemmer	11
3.3.1.1 CountVectorize	13
3.3.1.2 TFIDF Vectorize	13
3.3.2 Lemma Stemmer	13
3.3.2.1. CountVectorize	15
3.3.2.2. TFIDF Vectorize	16
3.3.3. So sánh và đánh giá	17
Chương IV: EDA	17
4.1. Full Target	18
4.2. Spam Target	19
4.3. Ham Target	21
Chương V: Máy học	24
5.1. Prepare the Data	24
5.2. Machine Learning Model	24
5.2.1 Multinomial Naïve Bayes	24
5.2.2. SVM (Support Vector Machine)	27
5.2.3. Decision Tree	30
5.2.4. Logistic Regression	33
5.2.5. Kết luận	36
5.3. Maxent Model	37
5.4. Deep Learning Model	38
5.4.1 Lemma Stemmer_CountVectorizer	39
5.4.2 Porter Stemmer_CountVectorizer	40
5.5. Evaluation Machine Learning Model	42

5.5.1. Lemma_CountVectorization	42
5.5.2. Lemma_TfidVectorization	44
5.5.3. Porter_CountVectorization	46
5.5.4. Porter_TfidVectorization	48
5.5.5. Kết luận	50
5.6. Evaluation Deep Learning Model	50
<i>Chương VI: Kết Luận</i>	52

Chương I: Tổng Quan Đề Tài

Trong bối cảnh thời đại công nghệ 4.0 không chỉ mở ra những cơ hội mới mẻ cho doanh nghiệp trong việc quảng bá, truyền thông đến khách hàng. Với sự phát triển mạnh mẽ của các phương tiện truyền thông số, các doanh nghiệp có thể dễ dàng tiếp cận đến khách hàng tiềm năng cũng như là dễ dàng gửi đi thông điệp trên hàng triệu khách hàng mới thông qua các kênh truyền thông này, đặc biệt qua tin nhắn. Chính vì sự dễ dàng và những thông điệp có giá trị, có nhiều doanh nghiệp đã tận dụng một cách vô tội vạ, có rất nhiều tin nhắn rác được gửi đi mà không được thông qua sự đồng ý của khách hàng, gây nên tình trạng quá tải thông tin và ảnh hưởng tiêu cực đến trải nghiệm đến khách hàng. Nhận thấy rằng việc nhận quá nhiều tin nhắn rác, nhóm đã nghiên cứu và phát triển đề tài “**Phát hiện tin nhắn rác**” để có thể phần nào cải thiện những tin nhắn spam làm phiền đến người dùng.

Bài toán phát hiện tin nhắn Spam trong bối cảnh hiện tại là một đề tài nghiên cứu cần thiết trong cuộc sống. Bài toán trên không chỉ là một thách thức trong lĩnh vực công nghệ thông tin mà là còn một vấn đề quản lý và ngăn chặn các hành vi lạm dụng công nghệ để làm phiền đến đời sống. Mục tiêu của bài toán là phát triển các mô hình học máy có khả năng phân biệt chính xác giữ tin nhắn hợp pháp(ham) và tin nhắn rác(spam), từ đó giúp người dùng có thể tập trung vào những tin nhắn quan trọng và tránh bị làm phiền bởi các tin nhắn nhận được không mong muốn.

Chương II: Cơ Sở Lý Thuyết

2.1. Tiền xử lý dữ liệu

a. Khái Niệm

Tiền xử lý là quá trình áp dụng các biến đổi cho dữ liệu trước khi đưa vào thuật toán. Được thực hiện để chuyển đổi dữ liệu thô thành một bộ dữ liệu sạch, tiền xử lý dữ liệu là kỹ thuật quan trọng giúp tối ưu hóa hiệu suất của thuật toán.

b. Tại sao cần tiền xử lý dữ liệu

Với mục tiêu đạt được kết quả tốt nhất từ mô hình áp dụng trong các dự án Machine Learning, dữ liệu cần phải được làm sạch và chuẩn hóa. Dữ liệu thô thường chứa nhiều lỗi và bất thường, có thể ảnh hưởng đến hiệu suất của mô hình.

Tiền xử lý dữ liệu giúp loại bỏ các lỗi và bất thường này, cải thiện chất lượng dữ liệu và giúp mô hình học tập chính xác hơn. Tiền xử lý dữ liệu là một bước quan trọng trong quá trình xây dựng mô hình Machine Learning. Bằng cách làm sạch và chuẩn hóa dữ liệu, chúng ta có thể cải thiện chất lượng dữ liệu và giúp mô hình học tập chính xác hơn.

2.1.1. Text Mining :

a. Khái niệm

Text Mining, còn được gọi là khai phá văn bản, là quá trình chuyển đổi ngôn ngữ tự nhiên thành một hình thức có thể được thao tác, lưu trữ và phân tích bởi máy móc.

Văn bản là một trong các loại dữ liệu phổ biến. Trong thế giới đa dạng của dữ liệu, văn bản đóng vai trò quan trọng như một nguồn thông tin phong phú.

Các kỹ thuật khai phá dữ liệu thường được sử dụng như :

-Information retrieval(Truy xuất thông tin) : Các công cụ khai phá văn bản nhận một truy vấn và tìm kiếm thông tin cụ thể trong nhiều văn bản, sau đó trả về dữ liệu mong muốn. Ví dụ, các phương pháp truy xuất thông tin thường được triển khai trong các công cụ tìm kiếm như Google. Một số nhiệm vụ con trong IR phổ biến bao gồm:

- Tokenization (Tách từ): Phân chia các văn bản dài thành các "tokens", có thể là từng từ, câu hoặc cụm từ.
 - Stemming (Tổng hợp từ): Đưa từ về dạng gốc của nó bằng cách loại bỏ hậu tố và tiền tố.
 - Trích xuất thông tin (Information Extraction) : Trích xuất thông tin liên quan đến việc rút trích thông tin có cấu trúc từ văn bản tự do. Các kỹ thuật này có thể trích xuất các thực thể quan trọng, mối quan hệ và thuộc tính, sau đó tổ chức chúng thành một định dạng dễ truy cập.
- Các công việc con phổ biến của IE:
- Chọn đặc trưng (Feature selection) : Mô tả các thuộc tính quan trọng.
 - Trích xuất đặc trưng (Feature extraction): Tính chi tiết bằng cách trích xuất một tập con của mỗi đặc trưng có liên quan.
 - Nhận diện thực thể được đặt tên (Named-entity recognition): Xác định các thực thể như tên người, địa điểm, v.v., trong văn bản.
- Xử lý ngôn ngữ tự nhiên (NLP)

Đây là một kỹ thuật tiên tiến dựa trên trí tuệ nhân tạo, ngôn ngữ học và khoa học dữ liệu, giúp máy tính "hiểu" ngôn ngữ của con người.

Các công việc con phổ biến nhất của NLP trong khai phá văn bản bao gồm:

- Tóm tắt (Summarization): Cung cấp tóm tắt ngắn gọn của đoạn văn dài, chẳng hạn là bài viết lớn hoặc thậm chí là sách.
- Phân loại văn bản (Text Categorization): Gán nhãn cho dữ liệu không có cấu trúc. Ví dụ, có thể phân loại tài liệu văn bản vào các danh mục đã định trước hoặc phân loại đánh giá của khách hàng dựa trên sản phẩm họ đề cập đến.
- Phân tích tâm trạng (Sentiment Analysis): Xác định tâm trạng tích cực, trung tính, và tiêu cực trong văn bản.

c. Ứng dụng

- Tăng trải nghiệm khách hàng : phát hiện điểm đau khách hàng thông qua đánh giá của khách hàng,...
- Tìm kiếm tự động : tổng hợp thông tin tự động từ nguồn văn bản lớn
- Thị trường thông minh : theo dõi hiệu suất làm việc và cập nhật của đối thủ cạnh tranh
- Chăm sóc sức khỏe: Các kỹ thuật khai thác văn bản ngày càng có giá trị đối với các nhà nghiên cứu trong lĩnh vực y sinh, đặc biệt là để phân cụm thông tin. Điều tra thủ công các nghiên cứu y học có thể tốn kém và mất thời gian; khai thác văn bản cung cấp một phương pháp tự động hóa để trích xuất thông tin có giá trị từ tài liệu y khoa.

2.2. Các mô hình học máy

2.2.1. Naïve Bayes

a. Khái niệm

Thuật toán Naïve Bayes classifiers là thuật toán máy học có giám sát, được sử dụng cho các bài toán phân loại như phân loại văn bản,... Thuật toán **Naïve Bayes classifiers** được xây dựng dựa trên định lý Bayes. Phản ánh mối quan hệ giữa xác suất của một biến cố mà không quan tâm các yếu tố khác với xác suất của biến cố đó sau khi một biến cố khác đã xảy ra. Định lý Bayes cho phép tính xác suất xảy ra của một sự kiện ngẫu nhiên A khi biết sự kiện liên quan B đã xảy ra. Xác suất này được ký hiệu là $P(A|B)$, và đọc là "xác suất của A nếu có B". Đại lượng này được gọi là xác suất có điều kiện hay xác suất hậu nghiệm vì nó được rút ra từ giá trị được cho của B hoặc phụ thuộc vào giá trị đó.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \frac{\text{likelihood} * \text{prior}}{\text{normalizing_constant}}$$

b. Các loại của mô hình Naïve Bayes classifiers

Mô hình Naïve Bayes classifiers có 3 loại chính bao gồm:

- Gaussian Naïve Bayes (GaussianNB): Đây là một biến thể của bộ phân loại Naïve Bayes, được sử dụng với các bản phân phối Gaussian—tức là, phân phối chuẩn—và các biến liên tục. Mô hình này được trang bị bằng cách tìm giá trị trung bình và độ lệch chuẩn của từng lớp.
- Multinomial Naïve Bayes (MultinomialNB): Loại trình phân loại Naïve Bayes này giả định rằng các tính năng là từ các phân phối đa thức. Biến thể này hữu ích khi sử dụng dữ liệu rời rạc, chẳng hạn như số lần xuất và thường được áp dụng trong các trường hợp sử dụng xử lý ngôn ngữ tự nhiên, như phân loại thư rác.
- Bernoulli Naïve Bayes (BernoulliNB): Đây là một biến thể khác của trình phân loại Naïve Bayes, được sử dụng với các biến Boolean—nghĩa là các biến có hai giá trị, chẳng hạn như Đúng và Sai hoặc 1 và 0.

c. Ưu điểm và nhược điểm

- Ưu điểm
 - Độ phức tạp thấp: So với các thuật toán phân loại khác trong máy học, Naïve Bayes được phân loại là thuật toán có độ phức tạp thấp vì có các tham số dễ ước tính. Và đây là một trong những thuật toán nền tảng cho lĩnh vực học máy.
 - Cần ít dữ liệu để train: Khi so sánh với các thuật toán học máy khác như cây quyết định hoặc mạng thần kinh, Naive Bayes cần ít dữ liệu đào tạo hơn. Điều này là do số lượng tham số phải được ước tính từ dữ liệu bị giảm do nó được xác định dựa trên ý tưởng về tính độc lập của đặc tính.
 - Thực hiện tốt việc phân loại văn bản: Naive Bayes là một thuật toán phổ biến cho các tác vụ phân loại văn bản, như phân tích cảm tính hoặc lọc thư rác. Điều này là do Naive Bayes có khả năng xử lý dữ liệu nhiều chiều và hoạt động tốt với dữ liệu phân loại, cả hai đều phổ biến trong xử lý ngôn ngữ tự nhiên.
- Nhược điểm
 - Giả định độc lập: Thuật toán đưa ra giả định rằng tất cả các đặc điểm đều độc lập với nhau, điều này thường sai trong các ứng dụng thực tế. Nếu các đặc điểm có mối tương quan với nhau, điều này có thể dẫn đến kết quả phân loại không chính xác.
 - Độ nhạy cảm với outlier: Naive Bayes rất nhạy cảm với các ngoại lệ hoặc giá trị cực đoan trong dữ liệu, điều này có thể có tác động đáng kể đến xác suất ước tính và tạo ra kết quả phân loại không chính xác.
 - Thiêng vị các giá trị có tần số cao: Naive Bayes dễ có xu hướng thiêng về các giá trị phổ biến trong tập dữ liệu huấn luyện. Điều này có thể trở thành vấn đề nếu bỏ sót một số giá trị ít phổ biến hơn nhưng quan trọng.

2.2.2. Support Vector Machine (SVM)

a. Khái niệm

Thuật toán Support Vector Machine được phân loại thuật toán máy học có giám sát. SVM là một trong những phương pháp mạnh mẽ dùng để phân loại, hồi quy và phát hiện outlier. Nguyên tắc chính của SVM là tìm ra siêu phẳng tốt nhất để phân tách các điểm dữ liệu của các

lớp khác nhau sao cho biên (khoảng cách giữa siêu phẳng và các điểm dữ liệu gần nhất từ mỗi lớp) là tối đa.

Siêu phẳng (Hyperplane): là ranh giới phân chia giữa các điểm dữ liệu có nhãn lớp khác nhau. Bộ phân loại SVM (Support Vector Machine) tách biệt các điểm dữ liệu bằng cách sử dụng một mặt phẳng quyết định với khoảng cách biên lớn nhất có thể. Mặt phẳng này được gọi là mặt phẳng biên tối đa (maximum margin hyperplane) và bộ phân loại tuyến tính mà nó định nghĩa được gọi là bộ phân loại biên tối đa (maximum margin classifier).

Vector hỗ trợ(Support Vectors): là những điểm dữ liệu mẫu nằm gần nhất với mặt phẳng quyết định (hyperplane). Những điểm dữ liệu này giúp xác định rõ ràng hơn đường phân cách hoặc siêu phẳng bằng cách tính toán khoảng cách biên.

Biên (Margin): là khoảng cách phân tách giữa hai đường ở các điểm dữ liệu gần nhất. Nó được tính là khoảng cách vuông góc từ đường tới các vector hỗ trợ hoặc điểm dữ liệu gần nhất. Trong SVM (Máy Vector Hỗ Trợ), chúng ta cố gắng tối đa hóa khoảng cách phân tách này để có được biên lớn nhất có thể.

b. Các loại của Support Vector Machine (SVM)

- SVM Phân loại (SVM Classification):
 - Linear SVM: Dùng cho dữ liệu có thể phân tách tuyến tính. Mục tiêu là tìm ra một đường biên (hyperplane) tuyến tính phân chia dữ liệu thành các lớp.
 - Non-linear SVM: Khi dữ liệu không phân tách tuyến tính, SVM sử dụng các hàm nhân (kernel functions) như RBF (Radial Basis Function), đa thức (polynomial), sigmoid, v.v., để chuyển dữ liệu sang không gian đa chiều nơi chúng có thể được phân tách tuyến tính.
- SVM Hồi quy (SVM Regression): Trong SVR, mục tiêu không phải là tìm một đường biên phân chia dữ liệu thành các lớp, mà là tìm một đường biên sao cho càng nhiều điểm dữ liệu nằm trong lề của đường biên này càng tốt, đồng thời giảm thiểu sai số dự đoán.
- One-Class SVM: Được sử dụng trong các bài toán phát hiện ngoại lệ (anomaly detection), nơi mà mục tiêu là phân biệt giữa các điểm dữ liệu bình thường và các điểm dữ liệu ngoại lệ.
- Nu-SVM: Một biến thể của SVM, nơi mà tham số 'nu' thay thế cho tham số C (trong soft-margin SVM). Tham số 'nu' đại diện cho một hạn chế trên phần trăm của các vector hỗ trợ và một hạn chế dưới của phần trăm các điểm dữ liệu sai lệch.
- Kernel SVM: Đây không phải là một loại SVM riêng biệt, nhưng là một thuật ngữ dùng để mô tả việc sử dụng các hàm nhân trong SVM, đặc biệt trong trường hợp của non-linear SVM.

c. Ưu điểm và nhược điểm

- Ưu điểm
 - Xử lý dữ liệu phi tuyến tính hiệu quả: SVM có thể xử lý dữ liệu phi tuyến tính một cách hiệu quả bằng thủ thuật Kernel.
 - Giải quyết cả vấn đề Phân loại và Hồi quy: SVM có thể được sử dụng để giải quyết cả vấn đề phân loại và hồi quy. SVM được sử dụng cho các vấn đề phân loại trong khi SVR (Support Vector Regression) được sử dụng cho các vấn đề hồi quy.
 - Tính ổn định: Một thay đổi nhỏ đối với dữ liệu không ảnh hưởng lớn đến siêu phẳng và do đó ảnh hưởng đến SVM. Như vậy mô hình SVM ổn định.
 - Tránh bị Overfitting: SVM có L2 Regularization. Vì vậy, nó có khả năng khai thác hóa tốt giúp ngăn chặn tình trạng Overfitting.

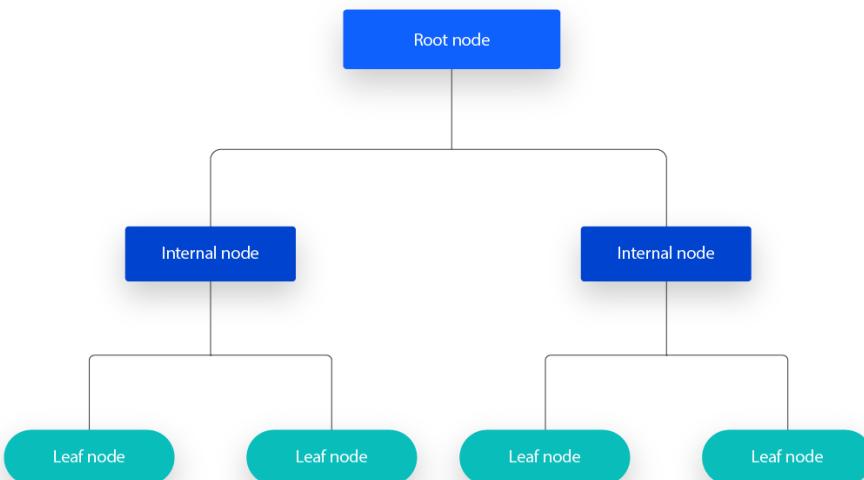
- **Nhược điểm**

- Lựa chọn một hàm Kernel phù hợp rất khó: Việc lựa chọn một hàm Kernel phù hợp (để xử lý dữ liệu phi tuyến tính) không phải là một việc dễ dàng. Nó có thể khó khăn và phức tạp. Trong trường hợp sử dụng Kernel có kích thước cao, bạn có thể tạo ra quá nhiều vector hỗ trợ, điều này làm giảm đáng kể tốc độ đào tạo.
- Yêu cầu bộ nhớ lớn: Độ phức tạp của thuật toán và yêu cầu bộ nhớ của SVM rất cao. Bạn cần rất nhiều bộ nhớ vì bạn phải lưu trữ tất cả các vector hỗ trợ trong bộ nhớ và con số này tăng đột ngột theo kích thước của tập dữ liệu huấn luyện.
- Khó diễn giải: Mô hình SVM phức tạp vì vậy rất khó hiểu và khó giải thích.

2.2.3. Decision Tree

a. Khái niệm

Decision Tree được phân loại là thuật toán học tập có giám sát không tham số, được sử dụng để phân loại và hồi quy. Decision Tree có cấu trúc mô hình cây phân cấp bao gồm một nút gốc, các nhánh, nút nội bộ và nút lá.



b. Các loại Decision Tree

- ID3: “Iterative Dichotomiser 3.” Thuật toán này tận dụng entropy và thông tin thu được làm các chỉ số để đánh giá các điểm chia tiềm năng.
- C4.5: Thuật toán này được coi là bản cải thiện của ID3. Nó có thể sử dụng thông tin thu được hoặc tỷ lệ thu được để đánh giá các điểm chia trong các cây quyết định.
- CART: Thuật ngữ CART là viết tắt của “classification and regression trees” (cây phân loại và hồi quy) và được giới thiệu bởi Leo Breiman. Thuật toán này thường sử dụng độ không tinh khiết Gini để xác định thuộc tính lý tưởng để chia. Độ không tinh khiết Gini đo lường mức độ thường xuyên một thuộc tính được chọn ngẫu nhiên bị phân loại sai.

c. Ưu điểm và nhược điểm

- **Ưu điểm**

- Dễ hiểu: Bản chất phân cấp của một cây quyết định cũng giúp dễ dàng nhận biết các thuộc tính quan trọng nhất, điều này không phải lúc nào cũng rõ ràng với các thuật toán khác, như mạng nơ-ron.
- Ít hoặc không cần chuẩn bị dữ liệu: Cây quyết định có thể xử lý các loại dữ liệu khác nhau - tức là các giá trị rời rạc hoặc liên tục, và các giá trị liên tục có thể

được chuyển đổi thành các giá trị phân loại thông qua việc sử dụng nguõng. Ngoài ra, nó cũng có thể xử lý các giá trị bị thiếu, điều này có thể gây vấn đề cho các bộ phân loại khác, như Naïve Bayes.

- Linh hoạt hơn: Cây quyết định có thể được sử dụng cho cả nhiệm vụ phân loại và hồi quy, làm cho nó linh hoạt hơn so với một số thuật toán khác. Nó cũng không nhạy cảm với mối quan hệ cơ bản giữa các thuộc tính; điều này có nghĩa là nếu hai biến có mối tương quan cao, thuật toán chỉ chọn một trong các thuộc tính để chia.

- **Nhược điểm**

- Dễ bị Over Fitting: Các cây quyết định phức tạp có xu hướng quá khớp và không tổng quát hóa tốt trên dữ liệu mới. Tình huống này có thể được tránh thông qua các quá trình tia trước hoặc tia sau.
- Ước lượng có biến động cao: Những biến đổi nhỏ trong dữ liệu có thể tạo ra một cây quyết định rất khác. Phương pháp tổng hợp, hay trung bình các ước lượng, có thể là một cách giảm biến động của cây quyết định. Tuy nhiên, cách tiếp cận này có hạn chế vì nó có thể dẫn đến các dự đoán có mối tương quan cao.
- Tốn kém hơn: Vì cây quyết định áp dụng cách tiếp cận tìm kiếm tham lam trong quá trình xây dựng, chúng có thể tốn kém hơn để đào tạo so với các thuật toán khác.

2.2.4. Logistic Regression

a. Khái niệm

Logistic Regression là một thuật toán học máy có giám sát chủ yếu được sử dụng cho các nhiệm vụ phân loại, nơi mục tiêu là dự đoán xác suất mà một thực thể thuộc về một lớp cụ thể. Nó được sử dụng cho các thuật toán phân loại với tên gọi là hồi quy Logistic. Nó được gọi là hồi quy bởi vì thuật toán này lấy đầu ra của hàm hồi quy tuyến tính làm đầu vào và sử dụng hàm sigmoid để ước lượng xác suất cho lớp cụ thể. Sự khác biệt giữa hồi quy tuyến tính và hồi quy Logistic là hồi quy tuyến tính cho ra kết quả là giá trị liên tục có thể là bất kỳ giá trị nào, trong khi hồi quy Logistic dự đoán xác suất mà một thực thể thuộc về một lớp cụ thể hoặc không.

b. Các loại Logistic Regression

Có ba loại mô hình Logistic Regression, được xác định dựa trên phản hồi phân loại.

- Hồi quy Logistic nhị phân (Binary logistic regression): Trong phương pháp này, biến phản hồi hoặc biến phụ thuộc có bản chất song phân - tức là nó chỉ có hai kết quả có thể xảy ra.
- Hồi quy Logistic đa thức (Multinomial logistic regression): Trong loại mô hình hồi quy Logistic này, biến phụ thuộc có ba kết quả hoặc nhiều hơn; tuy nhiên, những giá trị này không có thứ tự cụ thể.
- Hồi quy Logistic thứ tự (Ordinal logistic regression): Loại mô hình hồi quy Logistic này được sử dụng khi biến phản hồi có ba kết quả hoặc nhiều hơn, nhưng trong trường hợp này, những giá trị này có thứ tự xác định.

c. Ưu điểm và nhược điểm

- **Ưu điểm**

- Hiệu quả với Dữ liệu Phân Tách Tuyến Tính: Hồi quy Logistic phát huy hiệu quả khi tập dữ liệu có thể được phân tách một cách tuyến tính.
- Hạn chế Over fitting: Hồi quy Logistic ít có nguy cơ bị quá khớp, nhưng trong các tập dữ liệu có số chiều lớn, nguy cơ này có thể tăng lên. Trong

những trường hợp như vậy, việc áp dụng các kỹ thuật Regularization (L1 và L2) là cần thiết để ngăn chặn quá khớp.

- Cung cấp Thông tin Chi tiết: Ngoài việc cung cấp thông tin về mức độ quan trọng của một dự báo (qua kích thước hệ số), hồi quy Logistic còn cho biết hướng liên kết (tích cực hoặc tiêu cực) của dự báo đó.
- Dễ triển khai và hiệu quả: Hồi quy Logistic không chỉ dễ dàng trong việc triển khai và giải thích mà còn rất hiệu quả trong quá trình đào tạo.

- **Nhược điểm**

- Giả định Tuyến Tính: Hạn chế lớn của hồi quy Logistic là nó giả định mối quan hệ tuyến tính giữa biến phụ thuộc và các biến độc lập, trong khi trên thực tế, dữ liệu hiếm khi tuyến tính hoàn toàn. Phần lớn dữ liệu thường phức tạp và không rõ ràng.
- Hạn chế với Số Lượng Quan Sát Ít: Nếu số lượng quan sát thấp hơn số lượng đặc trưng, việc sử dụng hồi quy Logistic có thể dẫn đến hiện tượng Over fitting.
- Hạn chế trong Dự đoán: Hồi quy Logistic chỉ phù hợp với việc dự đoán các hàm rời rạc. Do đó, biến phụ thuộc trong hồi quy Logistic bị giới hạn trong tập số rời rạc, điều này tạo ra hạn chế trong việc dự đoán dữ liệu liên tục.

2.2.5. Long Short Term Memory (LSTM)

a. Khái niệm

Long Short-Term Memory (LSTM) là một loại Mạng Nơ-ron Hồi Quy (RNN) được thiết kế đặc biệt để xử lý dữ liệu tuần tự, như dữ liệu chuỗi thời gian, giọng nói và văn bản. Các mạng LSTM có khả năng học các phụ thuộc lâu dài trong dữ liệu tuần tự, làm cho chúng phù hợp với các nhiệm vụ như dịch thuật ngôn ngữ, nhận dạng giọng nói và dự báo chuỗi thời gian. LSTM giải quyết vấn đề về sự phụ thuộc lâu dài của RNN, trong đó RNN không thể dự đoán từ lưu trữ trong bộ nhớ dài hạn nhưng có thể đưa ra dự đoán chính xác hơn từ thông tin gần đây. Khi khoảng cách thời gian tăng lên, hiệu suất của RNN không còn hiệu quả. LSTM có thể mặc định giữ thông tin trong một khoảng thời gian dài.

b. Ưu điểm và nhược điểm

- **Ưu điểm**

- Năm Bắt Phụ Thuộc Lâu Dài: Mạng LSTM có khả năng bắt giữ phụ thuộc lâu dài trong dữ liệu. Chúng sở hữu tế bào nhớ có khả năng lưu trữ thông tin trong thời gian dài.
- Ghi Nhớ Ngữ Cảnh Quan Trọng: LSTM cho phép mô hình nắm bắt và nhớ ngữ cảnh quan trọng, ngay cả khi có khoảng cách thời gian lớn giữa các sự kiện liên quan trong chuỗi. Điều này rất quan trọng trong các ứng dụng như dịch máy.

- **Nhược điểm**

- Chi Phí Tính Toán Cao: So với các kiến trúc đơn giản hơn như mạng nơ-ron tiến hóa, mạng LSTM đòi hỏi chi phí tính toán cao hơn. Điều này có thể hạn chế khả năng mở rộng của chúng với các tập dữ liệu lớn hoặc trong môi trường có hạn chế.
- Đào Tạo Mất Nhiều Thời Gian: Do độ phức tạp tính toán, quá trình đào tạo mạng LSTM có thể mất nhiều thời gian hơn so với các mô hình đơn giản. Do đó, đào

tạo LSTM thường yêu cầu nhiều dữ liệu hơn và thời gian đào tạo lâu hơn để đạt được hiệu suất cao.

Chương III: Tiền xử lý dữ liệu

3.1. Tổng quan bộ dữ liệu

Dữ liệu gồm 2 biến và 5572 dòng

```
: data = pd.read_csv('SMSSpamCollection.txt', sep='\t', header=None, names=['Target', 'Text'])  
data.head()
```

	Target	Text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

```
: print(f'Số dòng của bộ dữ liệu là:{data.shape[0]} dòng')  
print(f'Số cột của bộ dữ liệu là:{data.shape[1]} cột')
```

Số dòng của bộ dữ liệu là:5572 dòng
Số cột của bộ dữ liệu là:2 cột

Giải thích các biến :

Biến Text : là dữ liệu tin nhắn được thu nhập từ nhiều nguồn khác nhau

Biến Target : có 2 giá trị lần lượt là ham và spam (tin nhắn bình thường và tin rác)

```
data.columns
```

```
Index(['Target', 'Text'], dtype='object')
```

```
# Kiểm tra xem có phải cột Target chỉ có 2 giá trị là ham và spam hay không  
unique_values = data['Target'].unique()  
print(unique_values)
```

```
['ham' 'spam']
```

3.2. Tiền xử lý dữ liệu

3.2.1. Kiểm tra dữ liệu bị thiếu

```
missing_values = data.isnull().sum()  
print(missing_values)
```

```
Target          0  
Text           0  
Lemma_Text_Process 0  
dtype: int64
```

Tiến hành kiểm tra dữ liệu Missing Value bị thiếu trong dữ liệu. Dữ liệu không có Missing Value.

3.2.2. Xử lý dữ liệu thiếu và trùng lặp

Ta tiến hành in ra các dữ liệu thiếu và trùng lặp. Ở bộ dữ liệu này không có giá trị thiếu và giá trị null. Tuy nhiên xuất hiện giá trị trùng lặp.

```
: data.info()  
  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 5572 entries, 0 to 5571  
Data columns (total 2 columns):  
 #   Column  Non-Null Count  Dtype    
---  --    
 0   Target    5572 non-null  object  
 1   Text      5572 non-null  object  
dtypes: object(2)  
memory usage: 87.2+ KB
```

```
: # Kiểm tra nếu có bất kỳ giá trị nào thiếu trong toàn bộ DataFrame  
missing_any = data.isnull().any().sum()  
  
# Kiểm tra nếu có bất kỳ dòng nào trùng lặp trong toàn bộ DataFrame  
duplicate_any = data.duplicated().any()  
  
print("Có giá trị thiếu:", missing_any)  
print("Có dòng trùng lặp:", duplicate_any)
```

```
Có giá trị thiếu: 0  
Có dòng trùng lặp: True
```

Ta tiếp tục in ra các dòng bị trùng lặp và quan sát kỹ hơn

```

: # Kiểm tra dòng trùng lặp
duplicate_mask = data.duplicated()
# Lọc và in ra các dòng trùng lặp
duplicate_rows = data[duplicate_mask]
print("Các dòng trùng lặp:")
duplicate_rows

```

Các dòng trùng lặp:

	Target	Text
103	ham	As per your request 'Melle Melle (Oru Minnamin...
154	ham	As per your request 'Melle Melle (Oru Minnamin...
207	ham	As I entered my cabin my PA said, " Happy B'd...
223	ham	Sorry, I'll call later
326	ham	No calls..messages..missed calls
...
5524	spam	You are awarded a SiPix Digital Camera! call 0...
5535	ham	I know you are thinkin malaria. But relax, chi...
5539	ham	Just sleeping..and surfing
5553	ham	Hahaha..use your brain dear
5558	ham	Sorry, I'll call later

403 rows × 2 columns

Đếm số lượng trùng lặp của từng dòng, và in ra số lượng của các dòng xuất hiện 1 lần (để kiểm tra thật sự các dòng này có bị trùng lặp hay không).

```
: duplicate_rows['count'] = duplicate_rows.groupby('Text')['Text'].transform('count')
duplicate_rows[duplicate_rows['count'] == 1]
```

	Target	Text	count
357	spam	Congratulations ur awarded 500 of CD vouchers ...	1
533	ham	Gudnite....tc...practice going on	1
825	ham	Have a good evening! Ttyl	1
850	spam	Today's Offer! Claim ur £150 worth of discount...	1
900	spam	Your free ringtone is waiting to be collected....	1
...
5471	ham	Yup	1
5497	spam	SMS SERVICES. for your inclusive text credits,...	1
5510	ham	I went to project centre	1
5524	spam	You are awarded a SiPix Digital Camera! call 0...	1
5535	ham	I know you are thinkin malaria. But relax, chi...	1

210 rows × 3 columns

Có 210 dòng chỉ xuất hiện 1 lần (không trùng lặp) nhưng hàm duplicate lại in ra tới 403 dòng trùng lặp, vì vậy nhóm quyết định không xóa những giá trị trùng lặp này.

3.3. Khai phá văn bản (Text Mining)

3.3.1. Porter Stemmer

Đầu tiên , chúng ta tạo ra hàm expand_abbreviations để mở rộng các ký tự bị viết tắt, như từ “u” sẽ được trở về dạng gốc của nó là từ “you ” . Do thời gian có hạn và chưa tìm ra cách tổng quát để biến đổi các từ viết tắt này, nhóm chỉ biến đổi một vài từ phổ biến.

```
def expand_abbreviations(senti):
    abbreviation_dict = {
        " u ":" you ",
        "dun": "do not",
        "don": "do not",
        "cant": "can not",
        "pl": "please",
        "dont": "do not"
    }
    for abbr, expanded in abbreviation_dict.items():
        senti = senti.replace(abbr, expanded)
    return senti
```

Sau đó chúng ta tiến hành xử lý ngôn ngữ tự nhiên trong cột Text trong văn bản bằng các bước như sau

- Tạo tập stop words: Một tập hợp các từ ngưng (stop words) trong tiếng Anh được tạo bằng cách sử dụng thư viện nltk với stopwords.words('english'). Tập này sẽ được sử dụng để loại bỏ các từ không quan trọng trong quá trình xử lý văn bản.
- Xử lý từng câu trong dữ liệu:
 - Vòng lặp for chạy qua mỗi câu trong cột 'Text' của DataFrame data.
 - re.sub('[^A-Za-z]', ' ', sen): Loại bỏ các ký tự không phải là chữ cái, thay thế chúng bằng khoảng trắng.
 - senti = senti.lower(): Chuyển đổi tất cả các ký tự thành chữ thường để đồng nhất.
 - words = word_tokenize(senti): Tách câu thành các từ.
 - word = [stem.stem(i) for i in words if i not in stop_words]: Loại bỏ stop words và thực hiện stemming (chuyển từ về dạng gốc) cho mỗi từ còn lại trong câu bằng Porter
 - senti = ''.join(word): Ghép các từ lại thành câu sau khi đã xử lý.
 - senti = expand_abbreviations(senti): Mở rộng các viết tắt trong câu bằng cách sử dụng hàm expand_abbreviations.

Câu được xử lý cuối cùng được thêm vào danh sách sentis.

```
stem = PorterStemmer()
stop_words = set(stopwords.words('english'))

# Preprocess and expand abbreviations for each sentence
sentis = []
for sen in data['Text']:
    senti = re.sub('[^A-Za-z]', ' ', sen)
    senti = senti.lower()
    words = word_tokenize(senti)
    word = [stem.stem(i) for i in words if i not in stop_words]
    senti = ''.join(word)
    senti = expand_abbreviations(senti)
    sentis.append(senti)
```

Và kết quả được lưu vào cột PorterStemmer_Text_Process

	Target	Text	PorterStemmer_Text_Process
0	ham	Go until jurong point, crazy.. Available only ...	go jurong point crazi avail bugi n great world...
1	ham	Ok lar... Joking wif u oni...	ok lar joke wif you oni
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	free entri wkli comp win fa cup final tkt st m...
3	ham	U dun say so early hor... U c already then say...	u do not say earli hor you c alreadi say
4	ham	Nah I don't think he goes to usf, he lives aro...	nah think goe usf live around though

3.3.1.1 CountVectorize

Sau đó chúng ta lần lượt vector hóa để làm đầu vào cho mô hình Máy học bằng CountVectorize :

```
cv=CountVectorizer(max_features=5000)
features=cv.fit_transform(data['PorterStemmer_Text_Process'])
features=features.toarray()
```

Kết quả thu được :

```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=int64)
```

3.3.1.2 TFIDF Vectorize

Tương tự như trên, sau khi tiến hành xử lý ngôn ngữ tự nhiên, ta cũng vecto hóa bằng TDIF :

```
tfidf=TfidfVectorizer(max_features=5000)
features=tfidf.fit_transform(data['PorterStemmer_Text_Process'])
features=features.toarray()
```

Và kết quả thu được :

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

3.3.2 Lemma Stemmer

Nhóm sẽ đi tạo các hàm xử lý văn bản:

```

def expand_abbreviations(text):
    abbreviation_dict = {
        " u ": " you ",
        "dun": "do not",
        "don": "do not",
        "cant": "can not",
        "pl": "please",
        "dont": "do not"
    }
    for abbr, expanded in abbreviation_dict.items():
        text = text.replace(abbr, expanded)
    return text

```

Hàm trên giúp xử lý các từ viết tắt như các ví dụ trên

```

def remove_non_alphabetic(text):
    return re.sub('[^A-Za-z]', ' ', text)

```

Hàm trên xóa các giá trị không phải là chữ cái viết hoa và viết thường

```

def convert_to_lowercase(text):
    return text.lower()

```

Hàm ‘convert_to_lowercase’ giúp chuyển đổi những chữ viết hoa thành chữ thường

```

def tokenize_lemmatize_remove_stopwords(text, lemma):
    words = word_tokenize(text)
    words = [lemma.lemmatize(word) for word in words if word not in stopwords.words('english')]
    return ' '.join(words)

```

Hàm tokenize_lemmatize_remove_stopwords nhận vào chuỗi văn bản text và một đối tượng lemmatizer lemma, thực hiện ba bước xử lý chính: tách từ văn bản thành các từ đơn lẻ (tokenization) bằng word_tokenize(text), chuyển mỗi từ về dạng gốc của nó (lemmatization) thông qua lemma.lemmatize(word), và loại bỏ những stop words (các từ thường gặp nhưng không mang nhiều ý nghĩa trong ngữ cảnh NLP) sử dụng điều kiện word not in stopwords.words('english')). Cuối cùng, hàm trả về chuỗi văn bản với các từ đã qua xử lý được

nối lại thành một chuỗi duy nhất, giúp tăng hiệu quả cho các quá trình xử lý ngôn ngữ tự nhiên tiếp theo.

```
def preprocess(sentences):
    lemma = WordNetLemmatizer()
    processed_sentences = []

    for text in sentences:
        # Expand abbreviations
        text = expand_abbreviations(text)

        # Remove non-alphabetic characters
        text = remove_non_alphabetic(text)

        # Convert to lowercase
        text = convert_to_lowercase(text)

        # Tokenize, lemmatize, and remove stopwords
        processed_sentence = tokenize_lemmatize_remove_stopwords(text, lemma)

        processed_sentences.append(processed_sentence)

    return processed_sentences
```

Cuối cùng tạo hàm preprocess() để tổng hợp tất cả các hàm xử lý văn bản.

```
data['Lemma_Text_Process'] = preprocess(data['Text'])

data.head()
```

	Target	Text	Lemma_Text_Process
0	ham	Go until jurong point, crazy.. Available only ...	go jurong point crazy available bugis n great ...
1	ham	Ok lar... Joking wif u oni...	ok lar joking wif oni
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	free entry wkly comp win fa cup final tkts st ...
3	ham	U dun say so early hor... U c already then say...	u say early hor u c already say
4	ham	Nah I don't think he goes to usf, he lives aro...	nah think go usf life around though

Sau khi đã hoàn thành việc xử lý văn bản, tạo 1 cột có giá trị Lemma_Text_Process để lưu những giá trị văn bản đã xử lý.

3.3.2.1. CountVectorize

Vector hóa bằng phương thức CountVectorize

```

start_time = time.time()

cv = CountVectorizer(max_features=5000)
features = cv.fit_transform(data['Lemma_Text_Process']).toarray()
print(features)
end_time = time.time()
execution_time = end_time - start_time

print(f"Thời gian xử lý vector hóa bằng CountVectorizer: {execution_time:.5f} giây")

```

Thu được kết quả như sau:

```

[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]

...
[0 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]]

```

3.3.2.2. TFIDF Vectorize

Vector hóa bằng phương thức TFIDF Vectorize

```

start_time = time.time()

tfidf=TfidfVectorizer(max_features=5000)
features=tfidf.fit_transform(data['Lemma_Text_Process']).toarray()
print(features)
end_time = time.time()
execution_time = end_time - start_time

print(f"Thời gian xử lý vector hóa bằng TDIFVectorizer: {execution_time:.5f} giây")

```

Kết quả thu được

```

[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]

...
[0 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]]

```

3.3.3. So sánh và đánh giá

Sau khi đã xử lý văn bản bằng 2 cách là PorterStemmer và Lemmatizer và Vector hóa bằng 2 phương pháp CountVectorizer và TfidfVectorizer. Nhóm ra được kết quả sau:

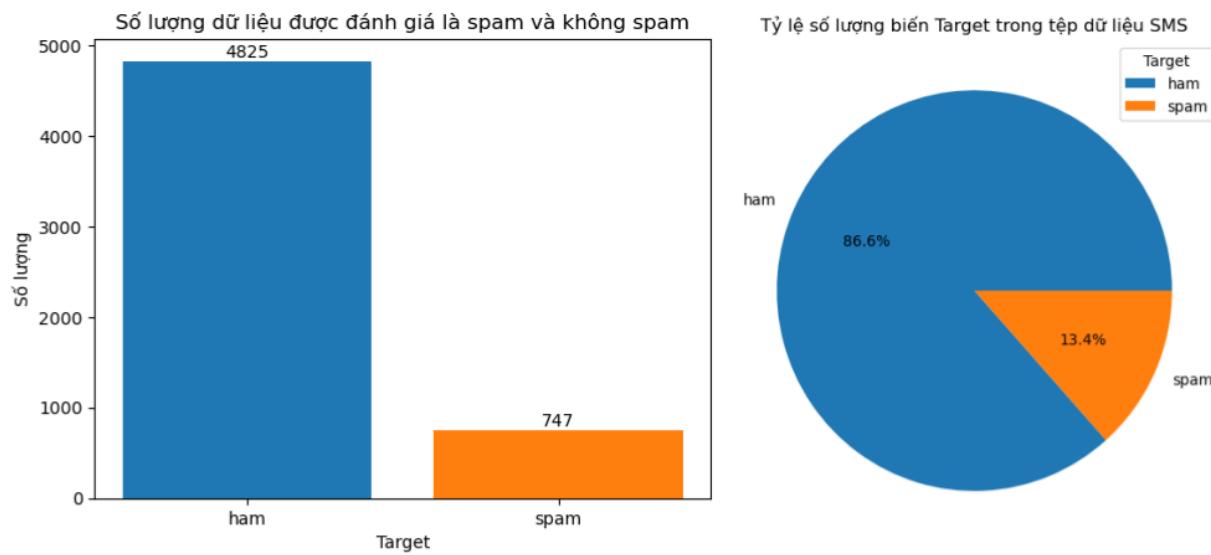
	PorterStemmer	Lemmatizer
CountVectorizer	1.13828 giây	11.71651 giây
TfidfVectorizer	1.07027 giây	10.94792 giây

Sau khi đã xử lý dữ liệu phù hợp để đưa vào mô hình máy học dùng để dự đoán tin nhắn rác. Nhóm nhận thấy rằng xử lý bằng phương pháp PorterStemmer sẽ nhanh hơn nhiều so với phương pháp Lemmatizer. Và Vectorize bằng CountVectorizer sẽ tốn thời gian hơn phương pháp TfidfVectorizer tuy nhiên không đáng kể.

Kết luận về mặt thời gian nhóm nhận thấy rằng kết hợp phương pháp xử lý văn bản bằng PorterStemmer và Vector hóa bằng phương pháp TfidfVectorizer sẽ có thời gian nhanh nhất. Về mặt độ chính xác nhóm sẽ đánh giá các phương pháp bằng độ chính xác của các mô hình học máy ở chương V.

Chương IV: EDA

Về tổng quan bộ dữ liệu sẽ được biểu diễn sau đây:

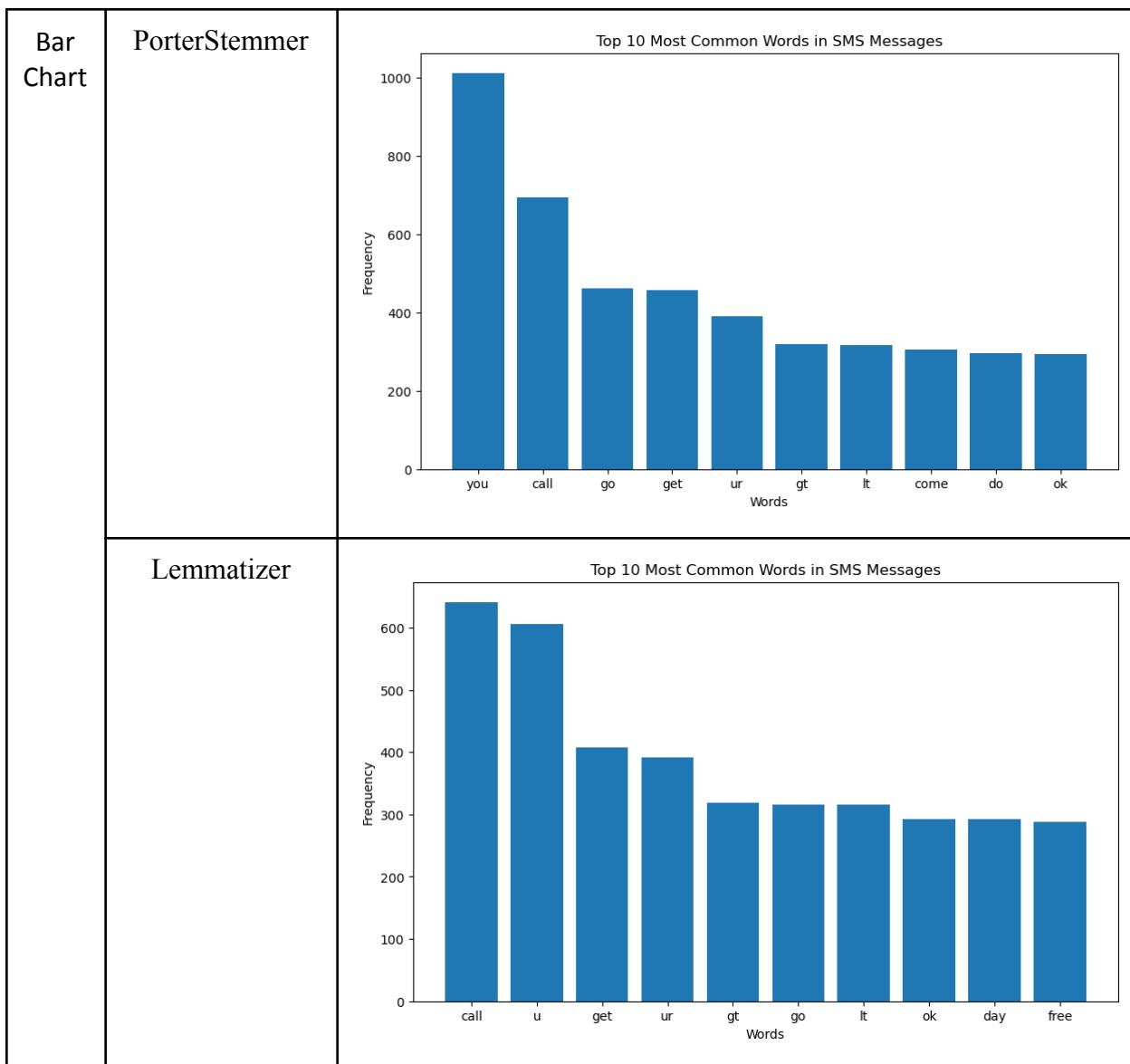


Có tất cả 4825(86,6%) tin nhắn được phân loại Ham và 747(13,4%) tin nhắn được phân loại Spam trong cột Target.

Tiếp theo ta sẽ đi xem xét sau khi xử lý văn bản bằng các phương pháp PorterStemmer và Lemmatizer sẽ cho ra kết quả các như thế nào.

4.1. Full Target

Word Cloud	PorterStemmer	
	Lemmatizer	



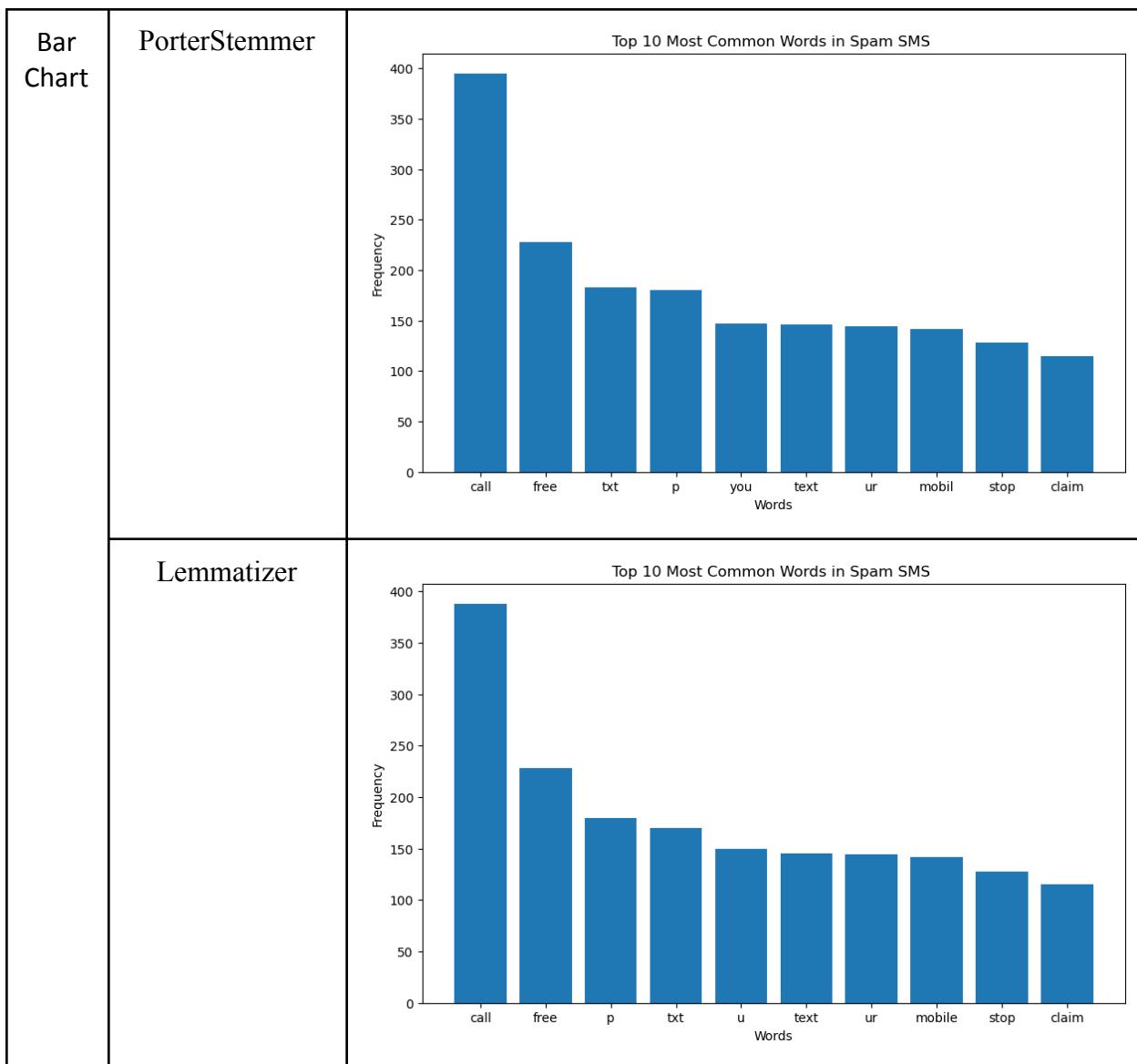
Tổng quan bộ dữ liệu sau khi xử lý theo 2 phương thức PorterStemmer và Lemmatizer sẽ cho ra được kết quả:

Với phương thức xử lý bằng PorterStemmer sẽ cho ra các từ được xuất hiện nhiều lần bao gồm: you, can, go, get, ur.

Với phương thức xử lý bằng PorterStemmer sẽ cho ra các từ được xuất hiện nhiều lần bao gồm: call, u, get ,ur, gt.

4.2. Spam Target

Word Cloud	PorterStemmer	
	Lemmatizer	



Dữ liệu có biến Target theo Spam sau khi xử lý theo 2 phương thức PorterStemmer và Lemmatizer sẽ cho ra được kết quả:

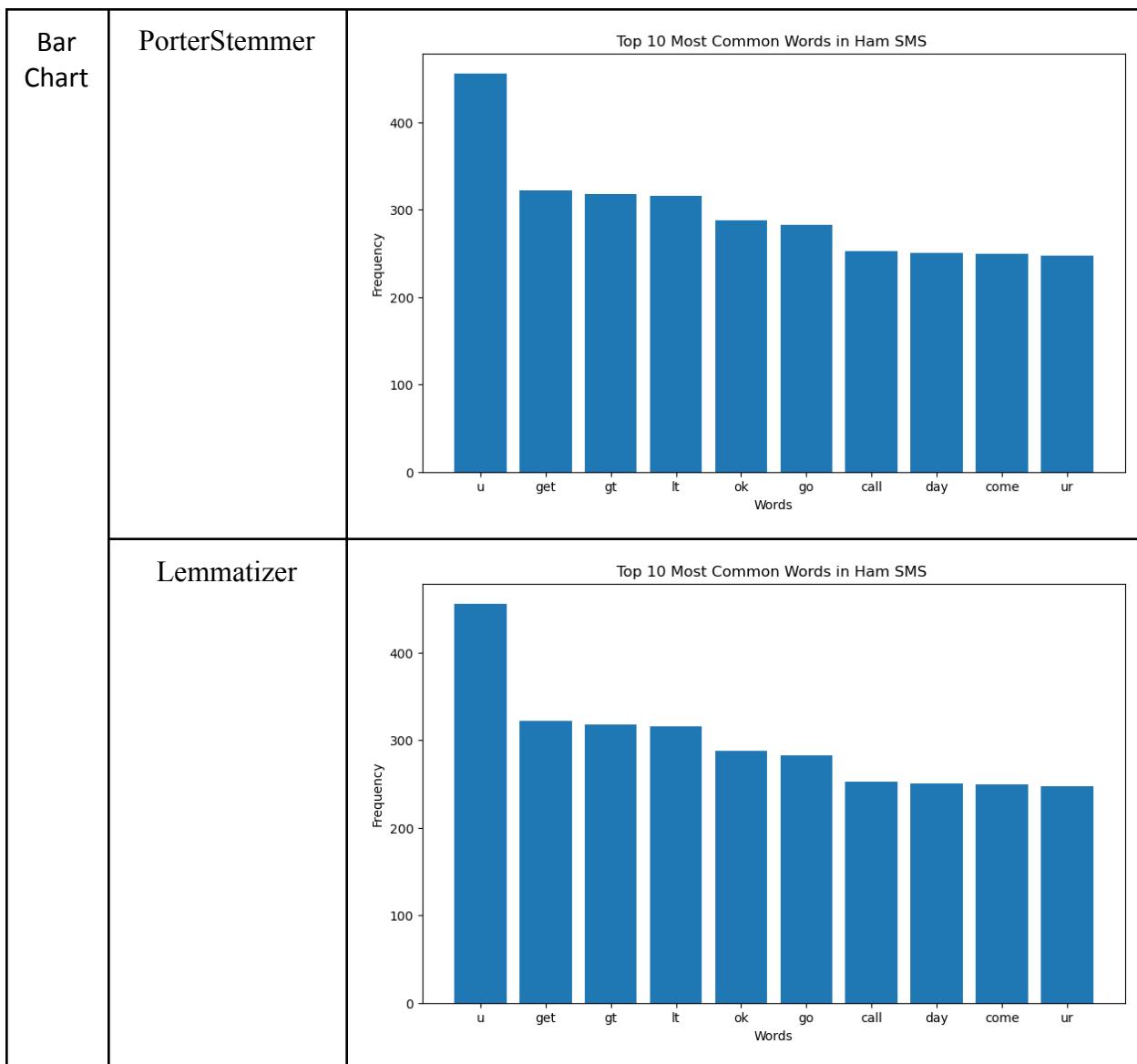
Với phương thức xử lý bằng PorterStemmer sẽ cho ra các từ được xuất hiện nhiều lần bao gồm: call, free, txt, p, you.

Với phương thức xử lý bằng PorterStemmer sẽ cho ra các từ được xuất hiện nhiều lần bao gồm: call, free, p, txt, u.

Điểm chung cả 2 phương thức đều phát hiện những từ call và free với biến Target là Spam. Đây là những từ xuất hiện nhiều trong các tin nhắn rác.

4.3. Ham Target

Word Cloud	PorterStemmer	
	Lemmatizer	



Dữ liệu có biến Target theo Ham sau khi xử lý theo 2 phương thức PorterStemmer và Lemmatizer sẽ cho ra được kết quả:

Với phương thức xử lý bằng PorterStemmer sẽ cho ra các từ được xuất hiện nhiều lần bao gồm: u, get, git, it, ok.

Với phương thức xử lý bằng PorterStemmer sẽ cho ra các từ được xuất hiện nhiều lần bao gồm: u, get, git, it, ok.

Điểm chung cả 2 phương thức đều phát hiện những từ call và free với biến Target là Spam. Đây là những từ xuất hiện nhiều trong các tin nhắn rác.

Kết luận: Nhìn chung 2 phương thức xử lý văn bản PorterStemmer và Lemmatizer cho ra kết quả khá tương đồng nhau. Tuy nhiên vẫn có những trường hợp khác nhau. Điều này tạo nên sự khác biệt của mỗi phương thức. Để có thể đánh giá được độ hiệu quả nhóm sẽ đưa vào sau khi đã xử lý vào các mô hình máy học để đánh giá độ hiệu quả của các phương pháp xử lý văn bản cũng như Vector hóa.

Chương V: Máy học

5.1. Prepare the Data

```
X = features
y = data['Target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, shuffle = True)
```

Nhóm đặt biến X là các biến Features và Y là biến Target trong mô hình Machine Learning. Và tập test_size chiếm 20% bộ dữ liệu.

```
X_train_shape = X_train.shape
X_test_shape = X_test.shape
y_train_shape = (len(y_train),)
y_test_shape = (len(y_test),)

X_train_shape, X_test_shape, y_train_shape, y_test_shape

((4457, 5000), (1115, 5000), (4457,), (1115,))
```

Đây là kích thước của các tập X_train, X_test, Y_train và Y_test.

5.2. Machine Learning Model

5.2.1 Multinomial Naïve Bayes

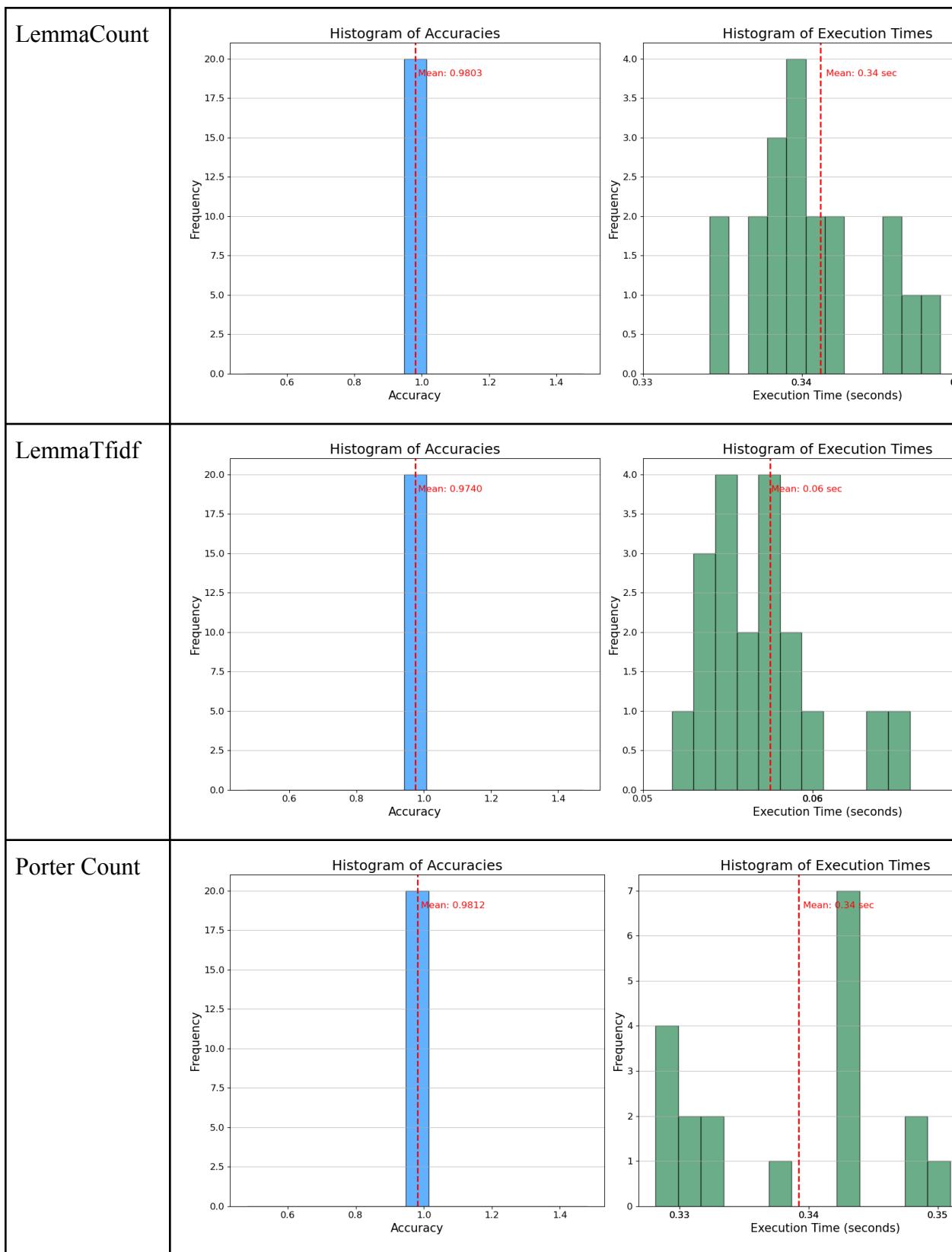
Tổng quan về mô hình **Multinomial Naïve Bayes** nhóm áp dụng vào bài toán:

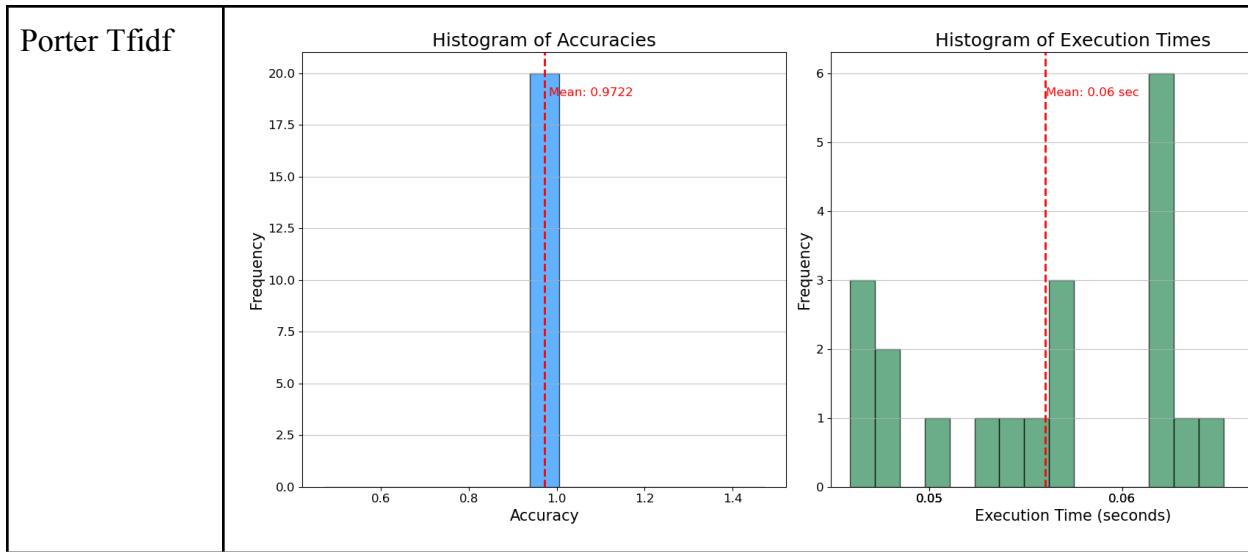
Đầu tiên nhóm sẽ để tham số mặc định trong mô hình học máy **Multinomial Naïve Bayes** để có thể xem kết quả của từng phương pháp xử lý văn bản Porter_TfidfVectorization, Porter_CountVectorization, Lemma_TfidfVectorization, Lemma_CountVectorization. Dưới đây là độ chính xác của thường phương pháp sau khi qua 1 lần train.

	PorterStemmer	Lemmatizer
CountVectorizer	Accuracy: 98.21%	Accuracy: 98.03%
TfidfVectorizer	Accuracy: 97.22%	Accuracy: 97.40%

Với 20 lần train các phương pháp sẽ cho được phân phối thời gian và độ chính xác sau đây:

Phương pháp	Phân phối (20 lần) của mô hình Multinomial Naïve Bayes	
	Accuracy	Time

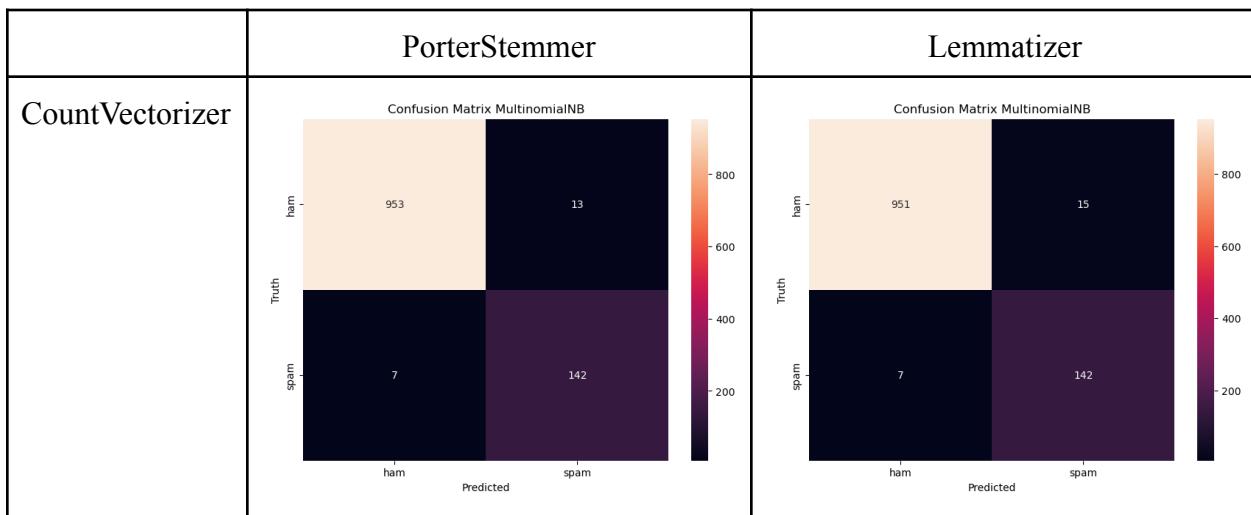


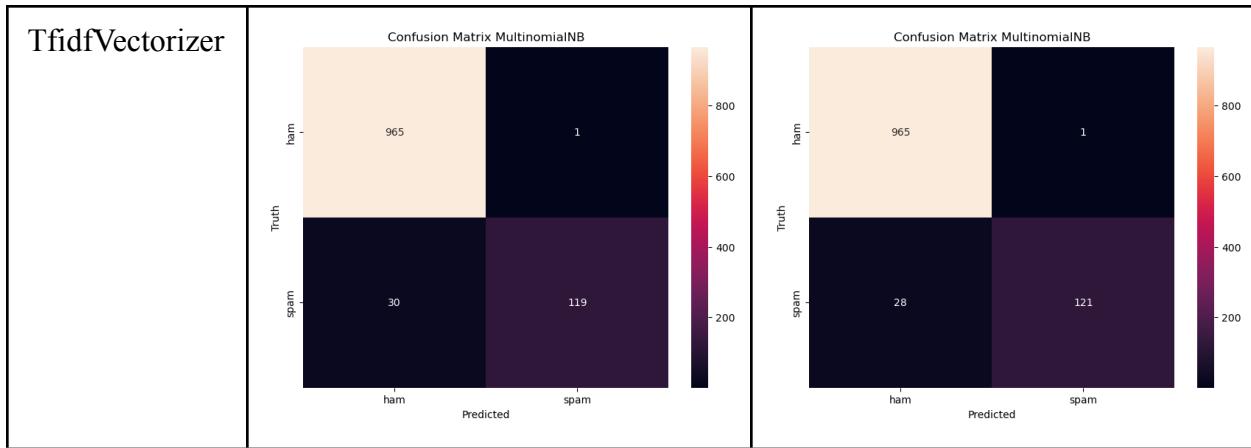


Nhìn chung với mô hình phân lớp **Multinomial Naïve Bayes**. Độ chính của từng phương pháp khá cao. Thấp nhất là phương pháp Porter_Tfidf với Accuracy 97,22% và cao nhất là Porter_Count với Accuracy 98,12%.

Về mặt thời gian mô hình phân lớp **Multinomial Naïve Bayes** cho thời gian training bộ dữ liệu khá là nhanh. Cả 4 phương pháp đều có thời gian training dưới 1 giây và đặt biệt với phương pháp áp dụng TfidfVectorizer có thời gian training là 0,06 giây.

Ma trận nhầm lẫn





5.2.2. SVM (Support Vector Machine)

Tổng quan về mô hình **SVM** nhóm áp dụng vào bài toán:

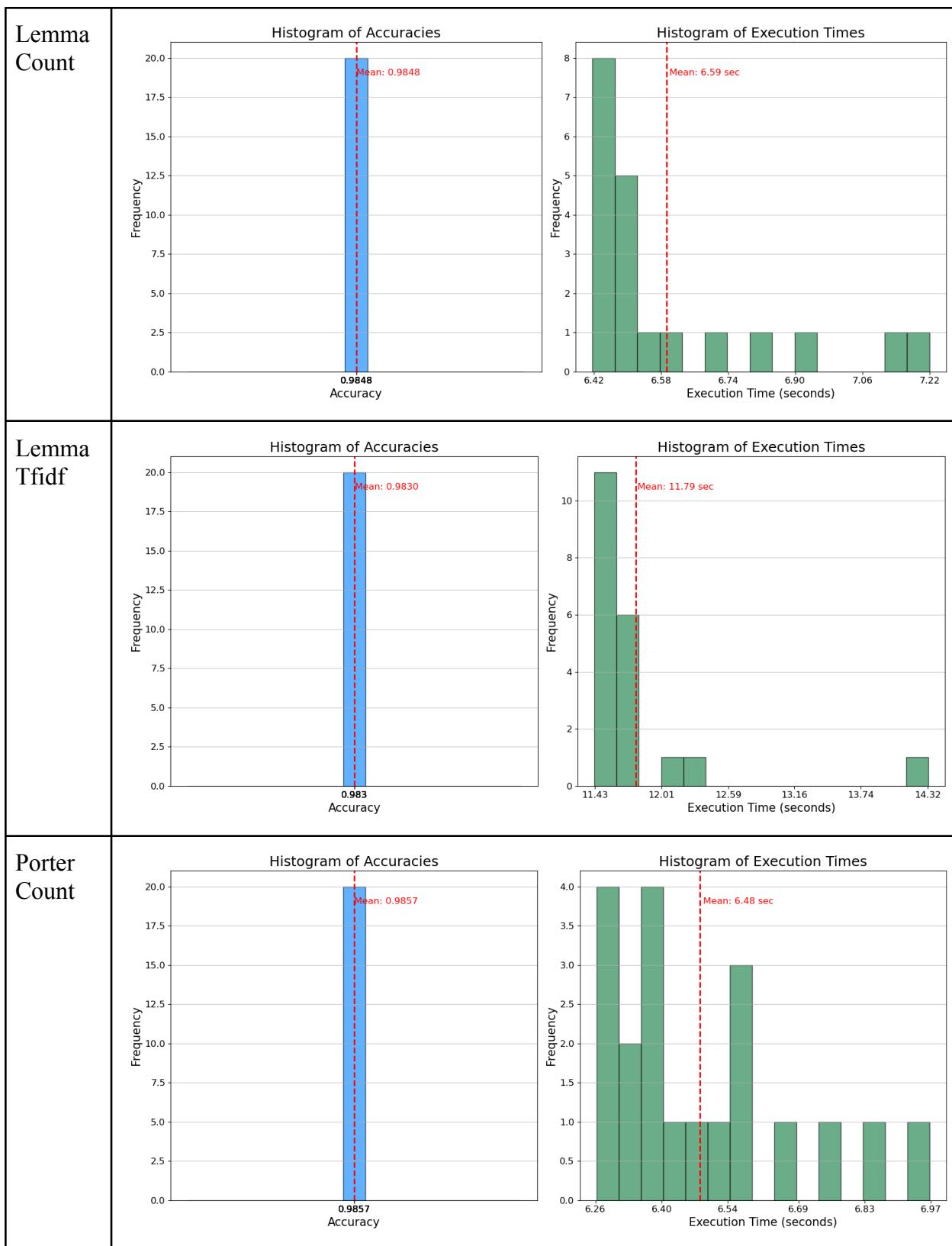
Đầu tiên nhóm sẽ để tham số mặc định trong mô hình học máy **SVM** để có thể xem kết quả của từng phương pháp xử lý văn bản Porter_TfidfVectorization, Porter_CountVectorization, Lemma_TfidfVectorization, Lemma_CountVectorization.

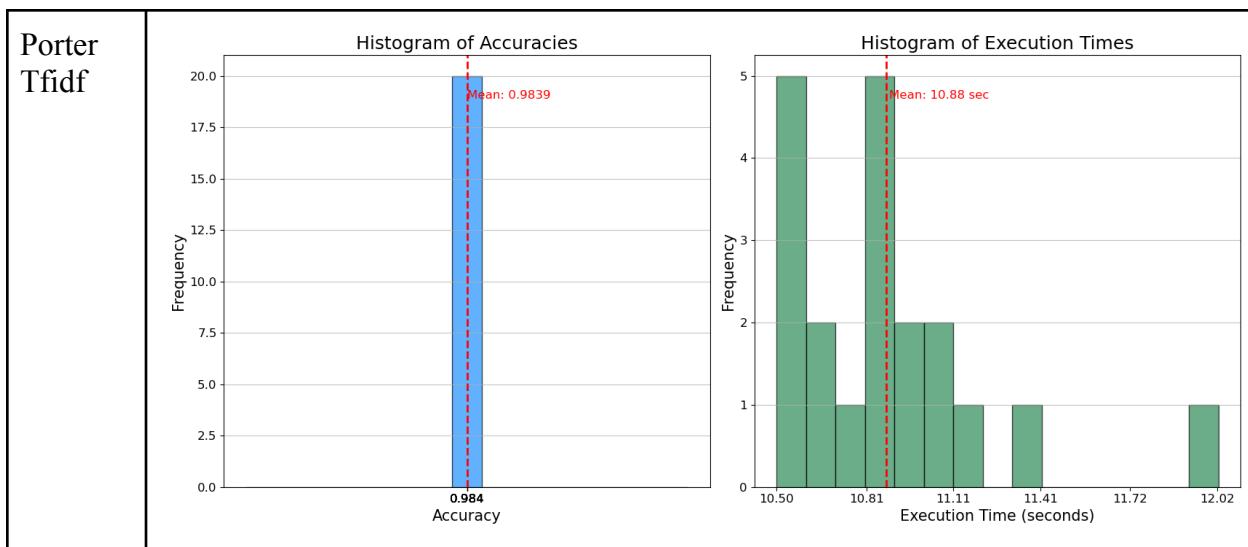
Dưới đây là độ chính xác của thường phương pháp sau khi qua 1 lần train.

	PorterStemmer	Lemmatizer
CountVectorizer	Accuracy: 98.57%	Accuracy: 98.48%
TfidfVectorizer	Accuracy: 98.39%	Accuracy: 98.30%

Với 20 lần train các phương pháp sẽ cho được phân phối thời gian và độ chính xác sau đây:

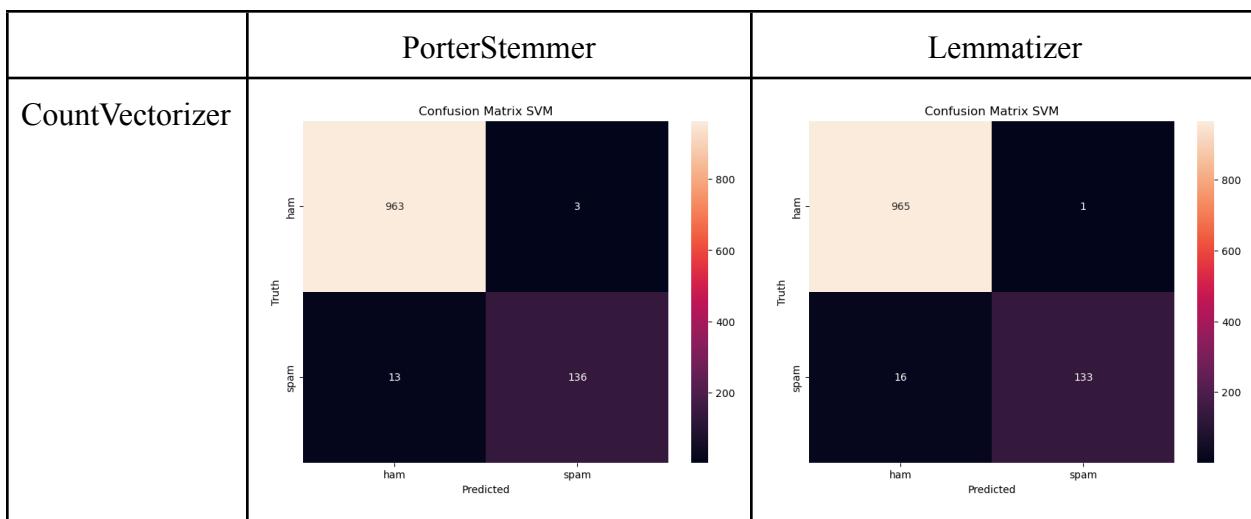
Tên	Phân phối (20 lần) của mô hình SVM	
	Accuracy	Time

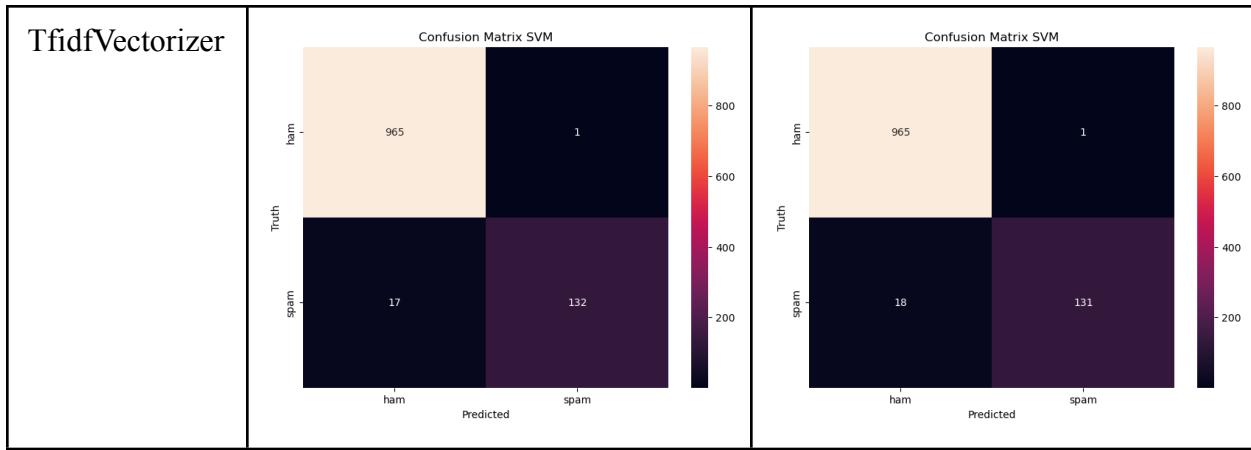




Nhìn chung với mô hình phân lớp **SVM**. Độ chính của từng phương pháp khá cao đều trên 98%. Về mặt thời gian mô hình phân lớp **SVM** được chia thành 2 phần với phương pháp CountVectorizer có thời gian trung bình training trong khoảng 6 giây và với phương pháp TfifdVectorize có thời gian training set từ 10 đến 12 giây.

Ma trận nhầm lẫn





5.2.3. Decision Tree

Tổng quan về mô hình **Decision Tree** nhóm áp dụng vào bài toán:

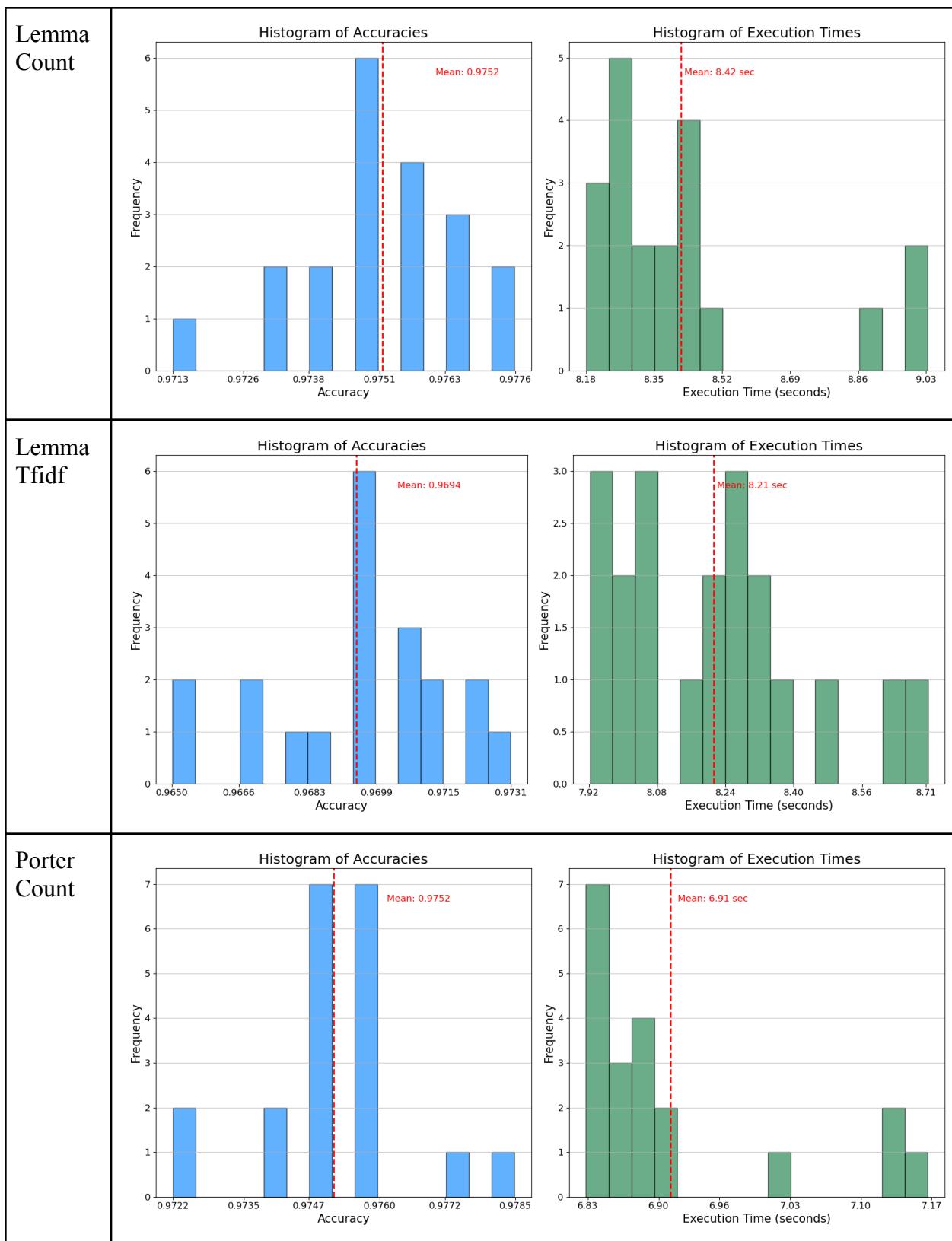
Đầu tiên nhóm sẽ để tham số mặc định trong mô hình học máy **Decision Tree** để có thể xem kết quả của từng phương pháp xử lý văn bản Porter_TfidfVectorization, Porter_CountVectorization, Lemma_TfidfVectorization, Lemma_CountVectorization.

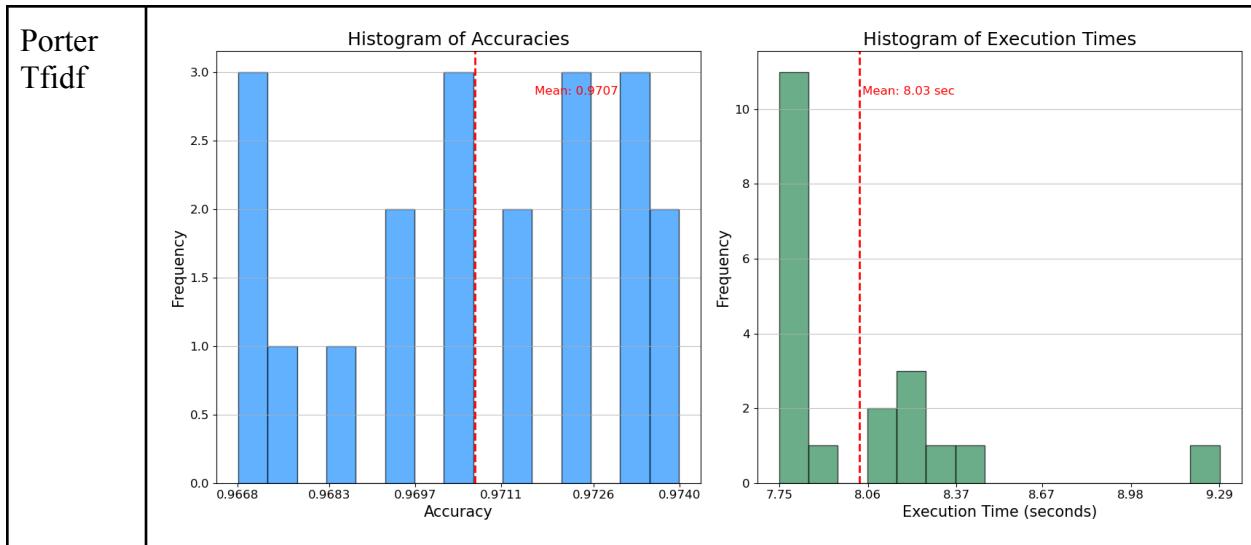
Dưới đây là độ chính xác của thường phương pháp sau khi qua 1 lần train.

	PorterStemmer	Lemmatizer
CountVectorizer	Accuracy: 97.49%	Accuracy: 97.49%
TfidfVectorizer	Accuracy: 96.95%	Accuracy: 97.04%

Với 20 lần train các phương pháp sẽ cho được phân phối thời gian và độ chính xác sau đây:

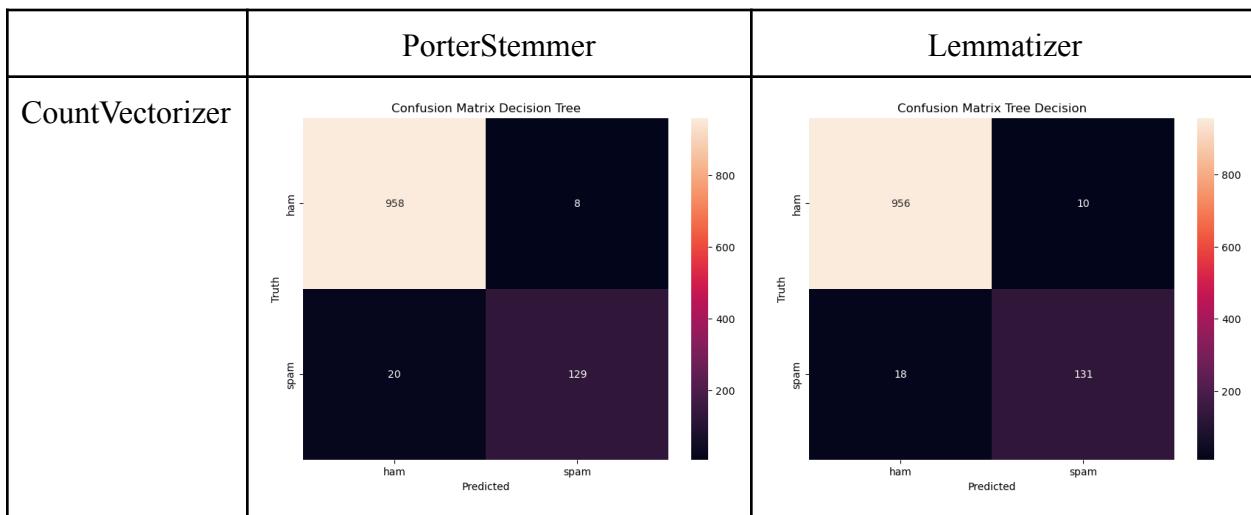
Tên	Phân phối (20 lần) của mô hình Decision Tree	
	Accuracy	Time

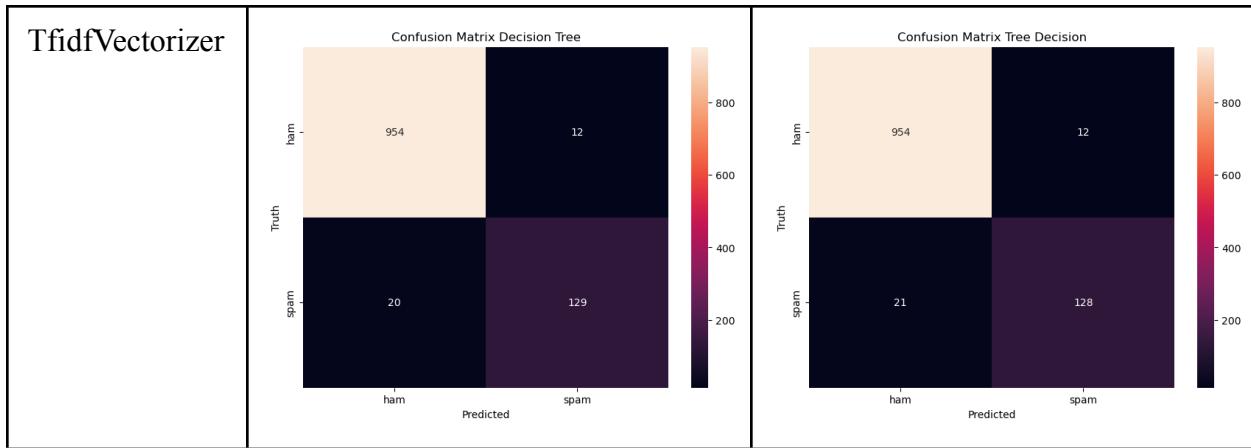




Khác với 2 mô hình phân lớp Multinomial Naïve Bayes và SVM. Mô hình Decision Tree sau khi chạy 20 lần sẽ được kết quả Accuracy phân phối như bảng trên. Tuy nhiên phương sai Accuracy của mô hình rất nhỏ và độ chính xác nằm trong khoảng 97%
Về thời gian phương pháp Porter_Count có thời gian chạy nhanh nhất với thời gian trung bình là 6,09 giây.

Ma trận nhầm lẫn





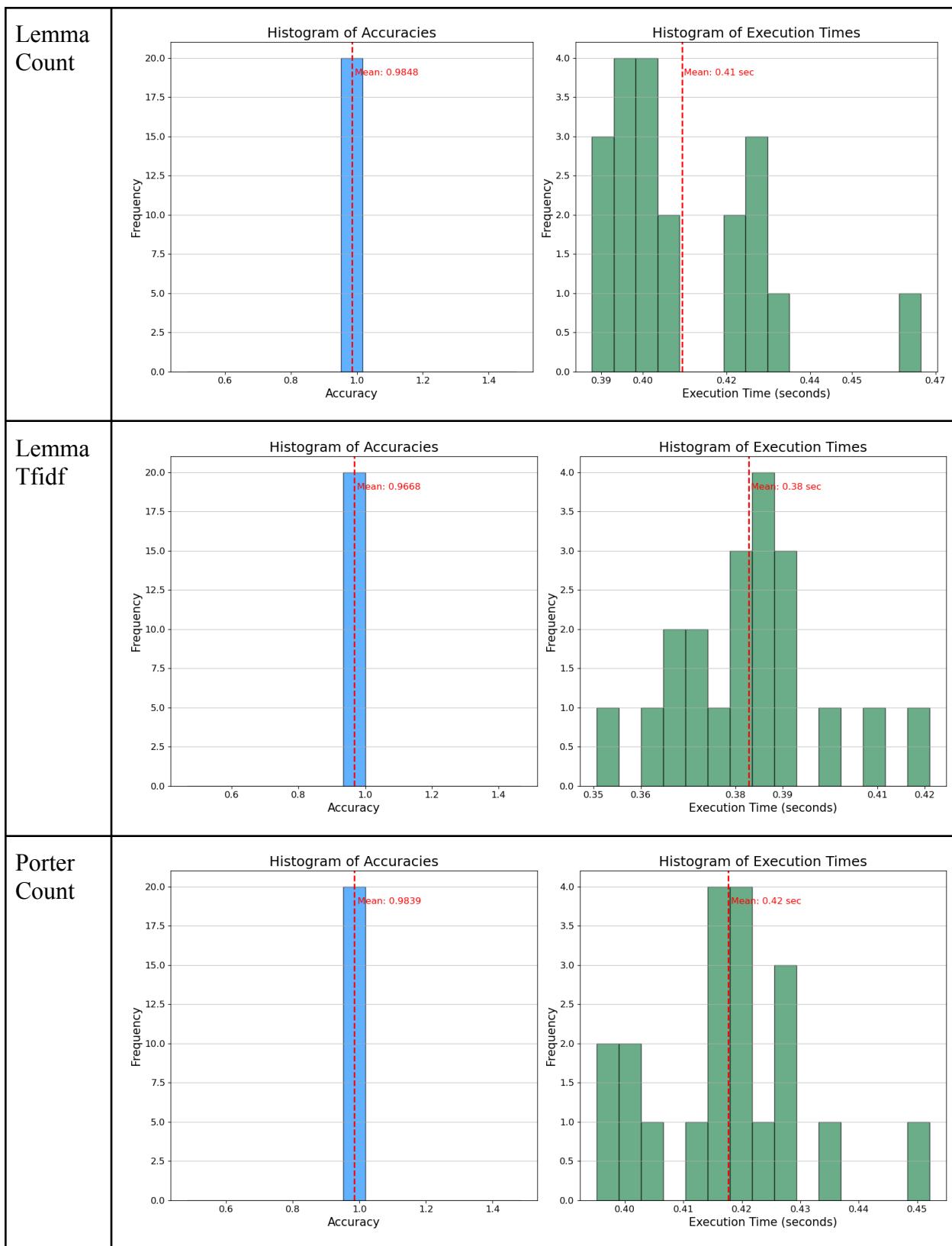
5.2.4. Logistic Regression

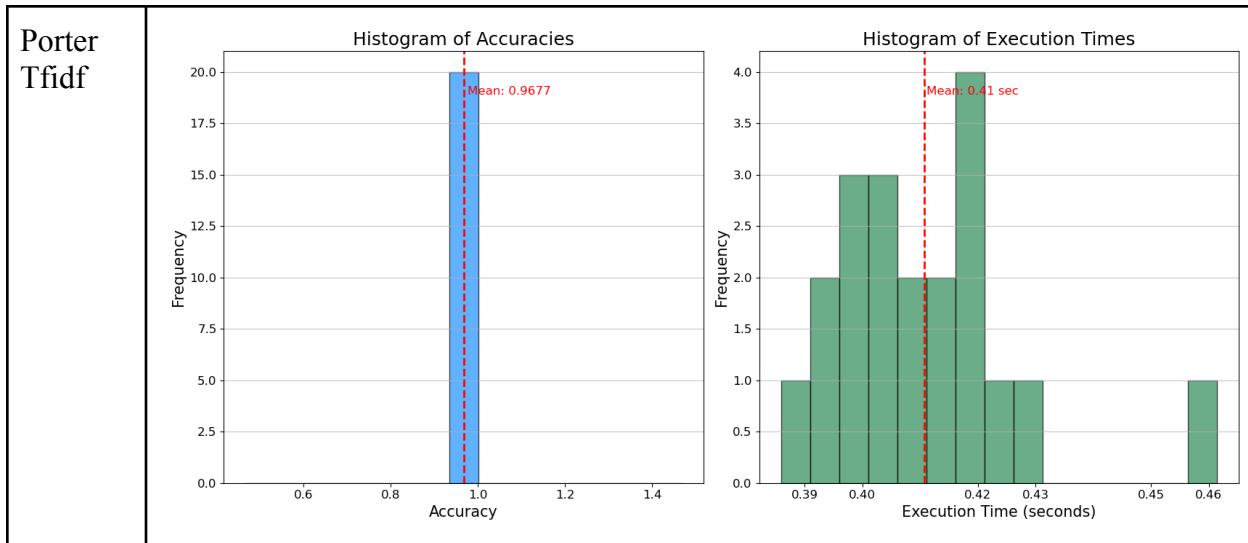
Tổng quan về mô hình **Logistic Regression** nhóm áp dụng vào bài toán:

Đầu tiên nhóm sẽ để tham số mặc định trong mô hình học máy **Logistic Regression** để có thể xem kết quả của từng phương pháp xử lý văn bản Porter_TfidfVectorization, Porter_CountVectorization, Lemma_TfidfVectorization, Lemma_CountVectorization. Dưới đây là độ chính xác của thường phương pháp sau khi qua 1 lần train.

	PorterStemmer	Lemmatizer
CountVectorizer	Accuracy: 98.39%	Accuracy: 98.48%
TfidfVectorizer	Accuracy: 96.77%	Accuracy: 96.68%

Tên	Phân phối(20 lần) của mô hình Logistic Regression	
	Accuracy	Time

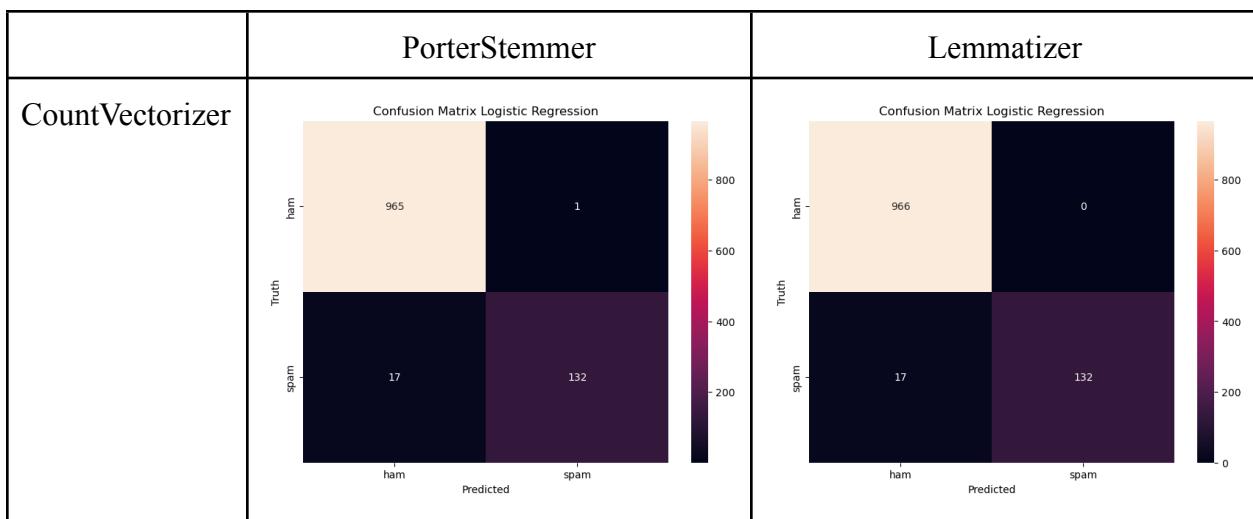


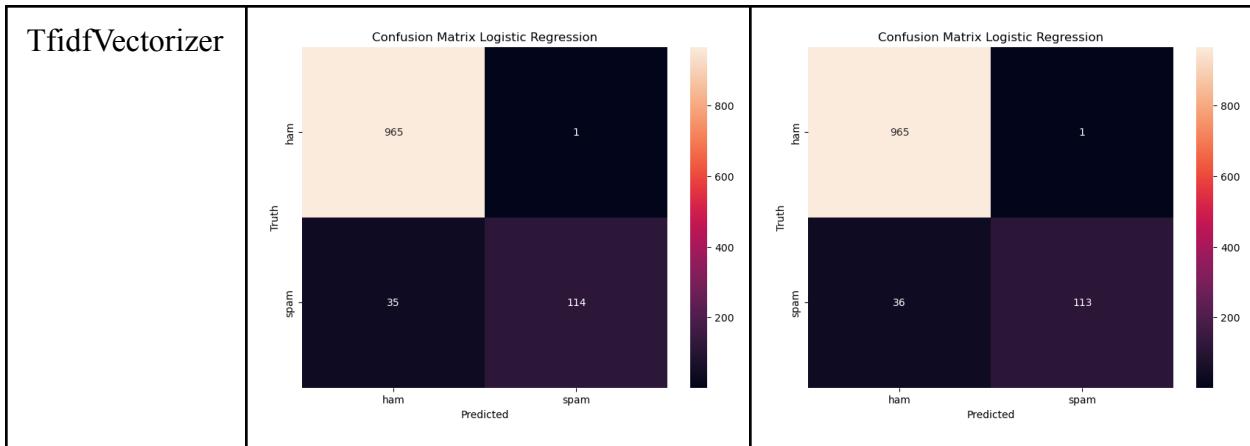


Về độ hiệu quả Accuracy của 4 phương pháp, các phương pháp Vectorize bằng TfIdfVectorize có độ chính xác thấp hơn so với phương pháp Vectorize bằng CountVectorize.

Về mặt thời gian thì 4 phương pháp xử lý văn bản trong mô hình **Logistic Regression** đều có thời gian training khá là ngắn (dưới 1s) trong tập dữ liệu.

Ma trận nhầm lẫn





5.2.5. Kết luận

Nhóm sẽ ưu tiên độ chính xác hơn về mặt thời gian sau đây nhóm sẽ đưa ra được phương pháp xử lý văn bản nào và mô hình nào đặt độ chính xác cao nhất

	MultinomialNB	SVM	Decision Tree	Logistic Regression
Lemma Count	98,03%	98,48%	97,52%	98,48%
Lemma Tfifdf	97,4%	98,3%	96,94%	96,68%
Porter Count	98,12%	98,57%	97,52%	98,39%
Porter Tfifdf	97,22%	98,39%	97,07%	96,77%

Qua bảng kết quả trên nhận thấy rằng với phương pháp xử lý văn bản bằng Porter_CountVectorization được sử dụng trong mô hình SVM (thông số mặc định) có độ chính xác cao nhất 98,57%. Tuy nhiên so với các phương pháp và các mô hình còn lại thì chỉ chênh lệch 1 chút ít.

Nhìn chung dù có xử lý văn bản bằng 1 trong 4 phương pháp nhóm áp dụng vào 4 mô hình học máy nhóm đã thực nghiệm đều cho ra độ chính xác cao.

5.3. Maxent Model

```
class NLTKMaxentClassifier:  
    def __init__(self, algorithm='GIS', max_iter=10):  
        self.algorithm = algorithm  
        self.max_iter = max_iter  
        self.classifier = None  
  
    def fit(self, X_train, y_train):  
        train_data = [{f'word_{i}': feature for i, feature in enumerate(x)}, y]  
        for x, y in zip(X_train, y_train)]  
        self.classifier = MaxentClassifier.train(train_data,  
                                                algorithm=self.algorithm,  
                                                max_iter=self.max_iter)  
  
    def predict(self, X_test):  
        test_data = [{f'word_{i}': feature for i, feature in enumerate(x)}  
                    for x in X_test]  
        return [self.classifier.classify(features) for features in test_data]
```

Nhóm sử dụng thuật toán GIS “Generalized Iterative Scaling”, một thuật toán sử dụng trong việc huấn luyện mô hình dựa trên Maximum Entropy (MaxEnt) hay Maximum Entropy Markov Models (MEMMs) trong xử lý ngôn ngữ tự nhiên và học máy. Thuật toán này được sử dụng để tối ưu hóa các trọng số của mô hình dựa trên nguyên lý cực đại entropy, tức là mô hình sẽ cố gắng tối đa hóa sự không chắc chắn (entropy) khi dự đoán dữ liệu.

```
mx_model = NLTKMaxentClassifier(max_iter=5)  
mx_model.fit(X_train, y_train)  
y_pred = mx_model.predict(X_test)  
accuracy = accuracy_score(y_test, y_pred)  
  
print('-----Maxent-----')  
print(classification_report(y_test, y_pred))  
print(f'Accuracy: {accuracy*100:.2f}%')
```

Sau khi train mô hình nhóm thu được kết quả sau đây:

```
==> Training (5 iterations)  
  
Iteration      Log Likelihood      Accuracy  
-----  
    1          -0.69315          0.134  
    2          -0.26397          0.866  
    3          -0.26337          0.866  
    4          -0.26298          0.866  
Final          -0.26259          0.866  
  
-----Maxent-----  
precision      recall      f1-score      support  
  
ham           0.87       1.00       0.93       966  
spam          0.00       0.00       0.00      149  
  
accuracy          0.87       0.87       0.87      1115  
macro avg      0.43       0.50       0.46      1115  
weighted avg    0.75       0.87       0.80      1115  
  
Accuracy: 86.64%
```

Điều cần lưu ý ở đây là cả 4 phương pháp xử lý văn bản đều cho kết quả tương tự, chứng tỏ rằng mô hình rất ổn định.

Nhóm sẽ đi phân tích kết quả.

Mô hình chỉ có thể dự đoán đúng được những biến có giá trị “Ham” và không có trường hợp nào dự đoán đúng giá trị “Spam”.

Mục tiêu của bài toán là phát hiện tin nhắn rác, từ đó nâng cao trải nghiệm người dùng cũng như giúp người dùng tránh gặp phải phiền nhiễu khi nhận tin nhắn Spam. Việc mô hình không thể nhận diện được tin nhắn rác không phù hợp với mục tiêu bài toán vì vậy nhóm không đánh giá cao Maxent Model trong bài toán phát hiện tin nhắn rác.

5.4. Deep Learning Model

Vì thời gian dành cho đề tài có giới hạn, nhóm chỉ chọn ra 2 phương pháp đại diện là Lemma Stemmer_CountVectorizer và PorterStemmer_CountVectorizer để chạy mô hình Deep Learning. Trước khi bỏ vào mô hình Deep Learning, ta cần tiến hành One-hot encoding để làm đầu vào cho mô hình như sau :

```
: one_hot_sent = [one_hot(i, 10000) for i in data['PorterStemmer_Text_Process']]
max_length = max(len(sent) for sent in one_hot_sent)
pad_sequences_result = pad_sequences(one_hot_sent, maxlen=80)
pad_sequences_result

: array([[ 0,  0,  0, ..., 5204, 7548, 8272],
       [ 0,  0,  0, ..., 2894, 1445, 5680],
       [ 0,  0,  0, ..., 5670, 4676, 5550],
       ...,
       [ 0,  0,  0, ..., 457, 4297, 1862],
       [ 0,  0,  0, ..., 193, 182, 6743],
       [ 0,  0,  0, ..., 3513, 8894, 6770]])
```

Và tương tự cho Lemma_Stemmer :

```
: one_hot_sent = [one_hot(i, 10000) for i in data['LemmaStemmer_Text_Process']]
max_length = max(len(sent) for sent in one_hot_sent)
pad_sequences_result = pad_sequences(one_hot_sent, maxlen=80)
pad_sequences_result

: array([[ 0,  0,  0, ..., 6311, 7243, 7834],
       [ 0,  0,  0, ..., 2431, 3004, 5243],
       [ 0,  0,  0, ..., 5692, 458, 5912],
       ...,
       [ 0,  0,  0, ..., 8417, 8491, 4907],
       [ 0,  0,  0, ..., 2463, 3004, 8895],
       [ 0,  0,  0, ..., 7064, 2298, 4466]])
```

Tiếp theo ta tiến hành xây dựng model :

```

: model=Sequential()
model.add(Embedding(10000,64,input_length=80))
model.add(LSTM(100))
model.add(Dense(2,activation='sigmoid'))
model.compile(loss='binary_crossentropy',metrics=['accuracy'],optimizer='adam')

: model.summary()

Model: "sequential"

Layer (type)                 Output Shape              Param #
=====
embedding (Embedding)        (None, 80, 64)          640000
lstm (LSTM)                  (None, 100)             66000
dense (Dense)                (None, 2)               202
=====
Total params: 706202 (2.69 MB)
Trainable params: 706202 (2.69 MB)
Non-trainable params: 0 (0.00 Byte)

```

Sau khi xây dựng được model, ta tiến hành fit model với dữ liệu :

```
model.fit(x_train,y_train,epochs=20,validation_data=(x_test,y_test))
```

Hình bên dưới là kết quả khi chạy fit model với từng phương pháp

5.4.1 Lemma Stemmer_CountVectorizer

```

Epoch 1/20
140/140 [=====] - 8s 45ms/step - loss: 0.2125 - accuracy: 0.9271 - val_loss: 0.0493 - val_accuracy: 0.
9857
Epoch 2/20
140/140 [=====] - 7s 50ms/step - loss: 0.0407 - accuracy: 0.9888 - val_loss: 0.0466 - val_accuracy: 0.
9865
Epoch 3/20
140/140 [=====] - 7s 51ms/step - loss: 0.0164 - accuracy: 0.9962 - val_loss: 0.0476 - val_accuracy: 0.
9892
Epoch 4/20
140/140 [=====] - 7s 52ms/step - loss: 0.0077 - accuracy: 0.9975 - val_loss: 0.0489 - val_accuracy: 0.
9874
Epoch 5/20
140/140 [=====] - 7s 53ms/step - loss: 0.0042 - accuracy: 0.9989 - val_loss: 0.0496 - val_accuracy: 0.
9865
Epoch 6/20
140/140 [=====] - 8s 54ms/step - loss: 0.0024 - accuracy: 0.9996 - val_loss: 0.0510 - val_accuracy: 0.
9892
Epoch 7/20
140/140 [=====] - 7s 53ms/step - loss: 0.0037 - accuracy: 0.9991 - val_loss: 0.0513 - val_accuracy: 0.
9865
Epoch 8/20
140/140 [=====] - 7s 49ms/step - loss: 0.0018 - accuracy: 0.9998 - val_loss: 0.0533 - val_accuracy: 0.
9883
Epoch 9/20
140/140 [=====] - 7s 52ms/step - loss: 7.5985e-04 - accuracy: 0.9998 - val_loss: 0.0611 - val_accuracy: 0.
9848
Epoch 10/20
140/140 [=====] - 7s 53ms/step - loss: 3.2148e-04 - accuracy: 1.0000 - val_loss: 0.0648 - val_accuracy: 0.
9883

```

```

-----
Epoch 11/20
140/140 [=====] - 7s 52ms/step - loss: 4.8064e-04 - accuracy: 0.9998 - val_loss: 0.0665 - val_accuracy: 0.9883
Epoch 12/20
140/140 [=====] - 7s 52ms/step - loss: 2.4196e-04 - accuracy: 1.0000 - val_loss: 0.0680 - val_accuracy: 0.9857
Epoch 13/20
140/140 [=====] - 7s 53ms/step - loss: 1.3422e-04 - accuracy: 1.0000 - val_loss: 0.0763 - val_accuracy: 0.9874
Epoch 14/20
140/140 [=====] - 7s 53ms/step - loss: 1.3062e-04 - accuracy: 1.0000 - val_loss: 0.0735 - val_accuracy: 0.9857
Epoch 15/20
140/140 [=====] - 7s 53ms/step - loss: 9.4932e-05 - accuracy: 1.0000 - val_loss: 0.0757 - val_accuracy: 0.9874
Epoch 16/20
140/140 [=====] - 8s 54ms/step - loss: 6.0453e-05 - accuracy: 1.0000 - val_loss: 0.0807 - val_accuracy: 0.9857
Epoch 17/20
140/140 [=====] - 8s 56ms/step - loss: 4.7694e-05 - accuracy: 1.0000 - val_loss: 0.0831 - val_accuracy: 0.9857
Epoch 18/20
140/140 [=====] - 7s 53ms/step - loss: 4.1615e-05 - accuracy: 1.0000 - val_loss: 0.0888 - val_accuracy: 0.9865
Epoch 19/20
140/140 [=====] - 7s 50ms/step - loss: 3.5162e-05 - accuracy: 1.0000 - val_loss: 0.0887 - val_accuracy: 0.9857
Epoch 20/20
140/140 [=====] - 7s 53ms/step - loss: 2.9301e-05 - accuracy: 1.0000 - val_loss: 0.0897 - val_accuracy: 0.9857

```

5.4.2 Porter Stemmer_CountVectorizer

```

model.fit(X_train,y_train,epochs=20,validation_data=(X_test,y_test))

Epoch 1/20
140/140 [=====] - 10s 56ms/step - loss: 0.1915 - accuracy: 0.9365 - val_loss: 0.0413 - val_accuracy: 0.9901
Epoch 2/20
140/140 [=====] - 7s 53ms/step - loss: 0.0375 - accuracy: 0.9888 - val_loss: 0.0369 - val_accuracy: 0.9919
Epoch 3/20
140/140 [=====] - 7s 53ms/step - loss: 0.0149 - accuracy: 0.9962 - val_loss: 0.0383 - val_accuracy: 0.9919
Epoch 4/20
140/140 [=====] - 7s 53ms/step - loss: 0.0094 - accuracy: 0.9975 - val_loss: 0.0466 - val_accuracy: 0.9892
Epoch 5/20
140/140 [=====] - 8s 56ms/step - loss: 0.0058 - accuracy: 0.9987 - val_loss: 0.0444 - val_accuracy: 0.9901
Epoch 6/20
140/140 [=====] - 8s 56ms/step - loss: 0.0068 - accuracy: 0.9982 - val_loss: 0.0403 - val_accuracy: 0.9910
Epoch 7/20
140/140 [=====] - 8s 56ms/step - loss: 0.0038 - accuracy: 0.9991 - val_loss: 0.0543 - val_accuracy: 0.9901
Epoch 8/20
140/140 [=====] - 8s 56ms/step - loss: 0.0022 - accuracy: 0.9993 - val_loss: 0.0579 - val_accuracy: 0.9901
Epoch 9/20
140/140 [=====] - 8s 56ms/step - loss: 0.0036 - accuracy: 0.9987 - val_loss: 0.1075 - val_accuracy: 0.9713
Epoch 10/20
140/140 [=====] - 8s 56ms/step - loss: 0.0099 - accuracy: 0.9980 - val_loss: 0.0562 - val_accuracy: 0.9883

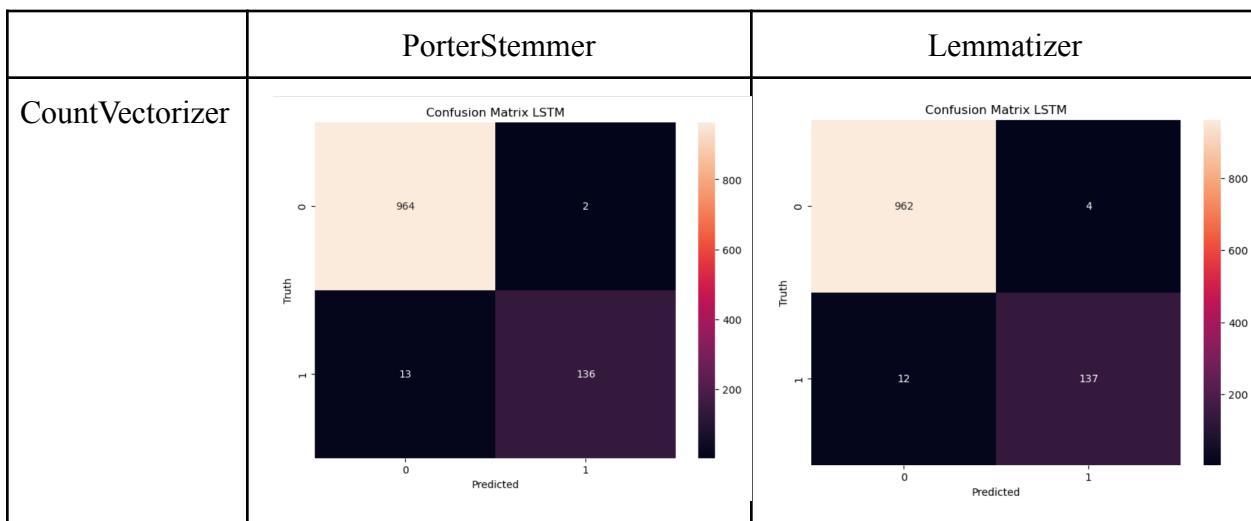
```

```

Epoch 11/20
140/140 [=====] - 8s 58ms/step - loss: 0.0023 - accuracy: 0.9993 - val_loss: 0.0683 - val_accuracy: 0.9865
Epoch 12/20
140/140 [=====] - 8s 56ms/step - loss: 9.6222e-04 - accuracy: 0.9998 - val_loss: 0.0732 - val_accuracy: 0.9865
Epoch 13/20
140/140 [=====] - 8s 55ms/step - loss: 6.9488e-04 - accuracy: 0.9998 - val_loss: 0.0750 - val_accuracy: 0.9865
Epoch 14/20
140/140 [=====] - 8s 56ms/step - loss: 3.5444e-04 - accuracy: 1.0000 - val_loss: 0.0761 - val_accuracy: 0.9883
Epoch 15/20
140/140 [=====] - 8s 56ms/step - loss: 2.2753e-04 - accuracy: 1.0000 - val_loss: 0.0782 - val_accuracy: 0.9874
Epoch 16/20
140/140 [=====] - 8s 57ms/step - loss: 1.5645e-04 - accuracy: 1.0000 - val_loss: 0.0812 - val_accuracy: 0.9883
Epoch 17/20
140/140 [=====] - 8s 59ms/step - loss: 1.1785e-04 - accuracy: 1.0000 - val_loss: 0.0855 - val_accuracy: 0.9874
Epoch 18/20
140/140 [=====] - 8s 56ms/step - loss: 8.7105e-05 - accuracy: 1.0000 - val_loss: 0.0881 - val_accuracy: 0.9874
Epoch 19/20
140/140 [=====] - 8s 54ms/step - loss: 7.3503e-05 - accuracy: 1.0000 - val_loss: 0.0901 - val_accuracy: 0.9874
Epoch 20/20
140/140 [=====] - 7s 53ms/step - loss: 5.5349e-05 - accuracy: 1.0000 - val_loss: 0.0937 - val_accuracy: 0.9865

```

Kết quả ma trận nhầm lẫn của 2 phương pháp



Độ chính xác của 2 phương pháp

	PorterStemmer	Lemmatizer

CountVectorizer	Độ chính xác của mô hình là 98.65%	Độ chính xác của mô hình là 98.57%
-----------------	------------------------------------	------------------------------------

Nhận xét :

5.5. Evaluation Machine Learning Model

Tại phần này nhóm sẽ đi tìm ra các giá trị hyperparameter cũng như parameter trong từng mô hình để được độ chính xác cao nhất.

Nhóm sử dụng phương pháp GridSearch là một kỹ thuật tìm kiếm siêu tham số (hyperparameter) phổ biến trong machine learning. Nó giúp tự động tìm ra tổ hợp tham số tối ưu nhất cho một mô hình machine learning thông qua việc thử nghiệm tất cả các tổ hợp có thể từ một không gian tham số đã cho.

5.5.1. Lemma_CountVectorization

Với phương pháp xử lý văn bản Lemma_CountVectorization, nhóm sẽ làm như sau:

Multinomial Naïve Bayes

```

param_grid = {
    'alpha': [0.1, 0.5, 1.0, 1.5, 2.0], # Thay đổi giá trị của alpha
    'fit_prior': [True, False], # Thử nghiệm với fit_prior là True và False
    'class_prior': [None, [0.2, 0.8], [0.5, 0.5]] # Thử nghiệm với các giá trị khác nhau cho class_prior
}

# Tạo một đối tượng GridSearchCV để tìm kiếm siêu tham số tốt nhất
grid_search = GridSearchCV(estimator=MultinomialNB(), param_grid=param_grid, scoring='accuracy', cv=5)
grid_search.fit(X_train, y_train)

# In ra siêu tham số tốt nhất
print("Best Parameters: ", grid_search.best_params_)

# Sử dụng mô hình với siêu tham số tốt nhất để đánh giá trên tập kiểm tra
best_nb_model = grid_search.best_estimator_
y_pred_tuned = best_nb_model.predict(X_test)

# In ra báo cáo phân loại và độ chính xác
print('-----Tuned MultinomialNB-----')
print(classification_report(y_test, y_pred_tuned))
accuracy_tuned = accuracy_score(y_test, y_pred_tuned)
print(f'Accuracy (Tuned): {accuracy_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters: {'alpha': 2.0, 'class_prior': None, 'fit_prior': True}
```

SVM

```

param_grid_svm_linear = {
    'C': [0.1, 1, 10],           # Thay đổi giá trị của hằng số ống với Lè
    'kernel': ['linear'],       # Chỉ sử dụng kernel Linear
    'gamma': ['scale', 'auto'],
    'class_weight': [None, 'balanced'], # Trọng số cho các lớp, có thể là None hoặc 'balanced'
    'shrinking': [True, False],     # Thuật toán thu nhỏ Lè
}

# Create a GridSearchCV object to find the best hyperparameters
grid_search_svm = GridSearchCV(estimator=SVC(), param_grid=param_grid_svm_linear, scoring='accuracy', cv=5)
grid_search_svm.fit(X_train, y_train)

# Print the best parameters
print("Best Parameters (SVM): ", grid_search_svm.best_params_)

# Use the model with the best parameters to evaluate on the test set
best_svm_model = grid_search_svm.best_estimator_
y_pred_svm_tuned = best_svm_model.predict(X_test)

# Print classification report and accuracy
print('-----Tuned Support Vector Machine-----')
print(classification_report(y_test, y_pred_svm_tuned))
accuracy_svm_tuned = accuracy_score(y_test, y_pred_svm_tuned)
print(f'Accuracy (Tuned): {accuracy_svm_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters (SVM): {'C': 0.1, 'class_weight': 'balanced', 'gamma': 'scale', 'kernel': 'linear', 'shrinking': True}
```

Decision Tree

```

tree_model = DecisionTreeClassifier()

# Định nghĩa các giá trị thử nghiệm cho các tham số của Decision Tree
param_grid_tree = {
    'criterion': ['gini', 'entropy'], # Loại độ đo chất lượng tách
    'splitter': ['best', 'random'],   # Chiến lược chọn điểm tách
    'max_depth': [None, 10, 20, 30], # Độ sâu tối đa của cây
    'min_samples_split': [2, 5, 10], # Số lượng mẫu tối thiểu để tách một nút
    'min_samples_leaf': [1, 2, 4]    # Số lượng mẫu tối thiểu ở lá
}

# Tạo một đối tượng GridSearchCV để tìm kiếm siêu tham số tốt nhất
grid_search_tree = GridSearchCV(estimator=tree_model, param_grid=param_grid_tree, scoring='accuracy', cv=5)
grid_search_tree.fit(X_train, y_train)

# In ra siêu tham số tốt nhất
print("Best Parameters (Decision Tree): ", grid_search_tree.best_params_)

# Sử dụng mô hình với siêu tham số tốt nhất để dự đoán trên tập kiểm tra
best_tree_model = grid_search_tree.best_estimator_
y_pred_tree_tuned = best_tree_model.predict(X_test)

# In ra báo cáo phân loại và độ chính xác
print('-----Tuned Decision Tree-----')
print(classification_report(y_test, y_pred_tree_tuned))
accuracy_tree_tuned = accuracy_score(y_test, y_pred_tree_tuned)
print(f'Accuracy (Tuned): {accuracy_tree_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters (Decision Tree): {'criterion': 'gini', 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 10, 'splitter': 'random'}
```

Logistic Regression

```

lr_model = LogisticRegression()

# Define the hyperparameter grid for Logistic regression
param_grid_lr = {
    'penalty': ['l1', 'l2'],
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'solver': ['liblinear', 'saga']
}

# Create a GridSearchCV object
grid_search_lr = GridSearchCV(estimator=lr_model, param_grid=param_grid_lr, scoring='accuracy', cv=5)
grid_search_lr.fit(X_train, y_train)

# Print the best parameters
print("Best Parameters (Logistic Regression): ", grid_search_lr.best_params_)

# Use the model with the best parameters to predict on the test set
best_lr_model = grid_search_lr.best_estimator_
y_pred_lr_tuned = best_lr_model.predict(X_test)

# Print classification report and accuracy
print('-----Tuned Logistic Regression-----')
print(classification_report(y_test, y_pred_lr_tuned))
accuracy_lr_tuned = accuracy_score(y_test, y_pred_lr_tuned)
print(f'Accuracy (Tuned): {accuracy_lr_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters (Logistic Regression): {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
```

5.5.2. Lemma_TfidVectorization

Với phương pháp xử lý văn bản Lemma_TfidVectorization, nhóm sẽ làm như sau:

Multinomial Naïve Bayes

```

param_grid = {
    'alpha': [0.1, 0.5, 1.0, 1.5, 2.0], # Thay đổi giá trị của alpha
    'fit_prior': [True, False], # Thử nghiệm với fit_prior là True và False
    'class_prior': [None, [0.2, 0.8], [0.5, 0.5]] # Thử nghiệm với các giá trị khác nhau cho class_prior
}

# Tạo một đối tượng GridSearchCV để tìm kiếm siêu tham số tốt nhất
grid_search = GridSearchCV(estimator=MultinomialNB(), param_grid=param_grid, scoring='accuracy', cv=5)
grid_search.fit(X_train, y_train)

# In ra siêu tham số tốt nhất
print("Best Parameters: ", grid_search.best_params_)

# Sử dụng mô hình với siêu tham số tốt nhất để đánh giá trên tập kiểm tra
best_nb_model = grid_search.best_estimator_
y_pred_tuned = best_nb_model.predict(X_test)

# In ra báo cáo phân loại và độ chính xác
print('-----Tuned MultinomialNB-----')
print(classification_report(y_test, y_pred_tuned))
accuracy_tuned = accuracy_score(y_test, y_pred_tuned)
print(f'Accuracy (Tuned): {accuracy_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters: {'alpha': 0.1, 'class_prior': None, 'fit_prior': True}
```

SVM

```

param_grid_svm_linear = {
    'C': [0.1, 1, 10],           # Thay đổi giá trị của hằng số ống với Lè
    'kernel': ['linear'],       # Chỉ sử dụng kernel Linear
    'gamma': ['scale', 'auto'],
    'class_weight': [None, 'balanced'], # Trọng số cho các lớp, có thể là None hoặc 'balanced'
    'shrinking': [True, False],     # Thuật toán thu nhỏ Lè
}

# Create a GridSearchCV object to find the best hyperparameters
grid_search_svm = GridSearchCV(estimator=SVC(), param_grid=param_grid_svm_linear, scoring='accuracy', cv=5)
grid_search_svm.fit(X_train, y_train)

# Print the best parameters
print("Best Parameters (SVM): ", grid_search_svm.best_params_)

# Use the model with the best parameters to evaluate on the test set
best_svm_model = grid_search_svm.best_estimator_
y_pred_svm_tuned = best_svm_model.predict(X_test)

# Print classification report and accuracy
print('-----Tuned Support Vector Machine-----')
print(classification_report(y_test, y_pred_svm_tuned))
accuracy_svm_tuned = accuracy_score(y_test, y_pred_svm_tuned)
print(f'Accuracy (Tuned): {accuracy_svm_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters (SVM): {'C': 1, 'class_weight': None, 'gamma': 'scale', 'kernel': 'linear', 'shrinking': True}
```

Decision Tree

```

tree_model = DecisionTreeClassifier()

# Định nghĩa các giá trị thử nghiệm cho các tham số của Decision Tree
param_grid_tree = {
    'criterion': ['gini', 'entropy'], # Logi độ đo chất lượng tách
    'splitter': ['best', 'random'],   # Chiến lược chọn điểm tách
    'max_depth': [None, 10, 20, 30], # Độ sâu tối đa của cây
    'min_samples_split': [2, 5, 10], # Số lượng mẫu tối thiểu để tách một nút
    'min_samples_leaf': [1, 2, 4]    # Số lượng mẫu tối thiểu ở lá
}

# Tạo một đối tượng GridSearchCV để tìm kiếm siêu tham số tốt nhất
grid_search_tree = GridSearchCV(estimator=tree_model, param_grid=param_grid_tree, scoring='accuracy', cv=5)
grid_search_tree.fit(X_train, y_train)

# In ra siêu tham số tốt nhất
print("Best Parameters (Decision Tree): ", grid_search_tree.best_params_)

# Sử dụng mô hình với siêu tham số tốt nhất để dự đoán trên tập kiểm tra
best_tree_model = grid_search_tree.best_estimator_
y_pred_tree_tuned = best_tree_model.predict(X_test)

# In ra báo cáo phân loại và độ chính xác
print('-----Tuned Decision Tree-----')
print(classification_report(y_test, y_pred_tree_tuned))
accuracy_tree_tuned = accuracy_score(y_test, y_pred_tree_tuned)
print(f'Accuracy (Tuned): {accuracy_tree_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters (Decision Tree): {'criterion': 'gini', 'max_depth': 30, 'min_samples_leaf': 1, 'min_samples_split': 10, 'splitter': 'random'}
```

Logistic Regression

```

lr_model = LogisticRegression()

# Define the hyperparameter grid for Logistic regression
param_grid_lr = {
    'penalty': ['l1', 'l2'],
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'solver': ['liblinear', 'saga']
}

# Create a GridSearchCV object
grid_search_lr = GridSearchCV(estimator=lr_model, param_grid=param_grid_lr, scoring='accuracy', cv=5)
grid_search_lr.fit(X_train, y_train)

# Print the best parameters
print("Best Parameters (Logistic Regression): ", grid_search_lr.best_params_)

# Use the model with the best parameters to predict on the test set
best_lr_model = grid_search_lr.best_estimator_
y_pred_lr_tuned = best_lr_model.predict(X_test)

# Print classification report and accuracy
print('-----Tuned Logistic Regression-----')
print(classification_report(y_test, y_pred_lr_tuned))
accuracy_lr_tuned = accuracy_score(y_test, y_pred_lr_tuned)
print(f'Accuracy (Tuned): {accuracy_lr_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters (Logistic Regression): {'C': 100, 'penalty': 'l2', 'solver': 'liblinear'}
```

5.5.3. Porter_CountVectorization

Với phương pháp xử lý văn bản Porter_CountVectorization, nhóm sẽ làm như sau:

Multinomial Naïve Bayes

```

param_grid = {
    'alpha': [0.1, 0.5, 1.0, 1.5, 2.0], # Thay đổi giá trị của alpha
    'fit_prior': [True, False], # Thử nghiệm với fit_prior là True và False
    'class_prior': [None, [0.2, 0.8], [0.5, 0.5]] # Thử nghiệm với các giá trị khác nhau cho class_prior
}

# Tạo một đối tượng GridSearchCV để tìm kiếm siêu tham số tốt nhất
grid_search = GridSearchCV(estimator=MultinomialNB(), param_grid=param_grid, scoring='accuracy', cv=5)
grid_search.fit(X_train, y_train)

# In ra siêu tham số tốt nhất
print("Best Parameters: ", grid_search.best_params_)

# Sử dụng mô hình với siêu tham số tốt nhất để đánh giá trên tập kiểm tra
best_nb_model = grid_search.best_estimator_
y_pred_tuned = best_nb_model.predict(X_test)

# In ra báo cáo phân loại và độ chính xác
print('-----Tuned MultinomialNB-----')
print(classification_report(y_test, y_pred_tuned))
accuracy_tuned = accuracy_score(y_test, y_pred_tuned)
print(f'Accuracy (Tuned): {accuracy_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters: {'alpha': 2.0, 'class_prior': None, 'fit_prior': True}
```

SVM

```

param_grid_svm_linear = {
    'C': [0.1, 1, 10],           # Thay đổi giá trị của hằng số ống với Lè
    'kernel': ['linear'],       # Chỉ sử dụng kernel Linear
    'gamma': ['scale', 'auto'],
    'class_weight': [None, 'balanced'], # Trọng số cho các lớp, có thể là None hoặc 'balanced'
    'shrinking': [True, False],     # Thuật toán thu nhỏ Lè
}

# Create a GridSearchCV object to find the best hyperparameters
grid_search_svm = GridSearchCV(estimator=SVC(), param_grid=param_grid_svm_linear, scoring='accuracy', cv=5)
grid_search_svm.fit(X_train, y_train)

# Print the best parameters
print("Best Parameters (SVM): ", grid_search_svm.best_params_)

# Use the model with the best parameters to evaluate on the test set
best_svm_model = grid_search_svm.best_estimator_
y_pred_svm_tuned = best_svm_model.predict(X_test)

# Print classification report and accuracy
print('-----Tuned Support Vector Machine-----')
print(classification_report(y_test, y_pred_svm_tuned))
accuracy_svm_tuned = accuracy_score(y_test, y_pred_svm_tuned)
print(f'Accuracy (Tuned): {accuracy_svm_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters (SVM): {'C': 0.1, 'class_weight': 'balanced', 'gamma': 'scale', 'kernel': 'linear', 'shrinking': True}
```

Decision Tree

```

tree_model = DecisionTreeClassifier()

# Định nghĩa các giá trị thử nghiệm cho các tham số của Decision Tree
param_grid_tree = {
    'criterion': ['gini', 'entropy'], # Logi độ đo chất lượng tách
    'splitter': ['best', 'random'],   # Chiến lược chọn điểm tách
    'max_depth': [None, 10, 20, 30], # Độ sâu tối đa của cây
    'min_samples_split': [2, 5, 10], # Số lượng mẫu tối thiểu để tách một nút
    'min_samples_leaf': [1, 2, 4]    # Số lượng mẫu tối thiểu ở lá
}

# Tạo một đối tượng GridSearchCV để tìm kiếm siêu tham số tốt nhất
grid_search_tree = GridSearchCV(estimator=tree_model, param_grid=param_grid_tree, scoring='accuracy', cv=5)
grid_search_tree.fit(X_train, y_train)

# In ra siêu tham số tốt nhất
print("Best Parameters (Decision Tree): ", grid_search_tree.best_params_)

# Sử dụng mô hình với siêu tham số tốt nhất để dự đoán trên tập kiểm tra
best_tree_model = grid_search_tree.best_estimator_
y_pred_tree_tuned = best_tree_model.predict(X_test)

# In ra báo cáo phân loại và độ chính xác
print('-----Tuned Decision Tree-----')
print(classification_report(y_test, y_pred_tree_tuned))
accuracy_tree_tuned = accuracy_score(y_test, y_pred_tree_tuned)
print(f'Accuracy (Tuned): {accuracy_tree_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters (Decision Tree): {'criterion': 'gini', 'max_depth': 30, 'min_samples_leaf': 1, 'min_samples_split': 5, 'splitter': 'random'}
```

Logistic Regression

```

lr_model = LogisticRegression()

# Define the hyperparameter grid for Logistic regression
param_grid_lr = {
    'penalty': ['l1', 'l2'],
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'solver': ['liblinear', 'saga']
}

# Create a GridSearchCV object
grid_search_lr = GridSearchCV(estimator=lr_model, param_grid=param_grid_lr, scoring='accuracy', cv=5)
grid_search_lr.fit(X_train, y_train)

# Print the best parameters
print("Best Parameters (Logistic Regression): ", grid_search_lr.best_params_)

# Use the model with the best parameters to predict on the test set
best_lr_model = grid_search_lr.best_estimator_
y_pred_lr_tuned = best_lr_model.predict(X_test)

# Print classification report and accuracy
print('-----Tuned Logistic Regression-----')
print(classification_report(y_test, y_pred_lr_tuned))
accuracy_lr_tuned = accuracy_score(y_test, y_pred_lr_tuned)
print(f'Accuracy (Tuned): {accuracy_lr_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters (Logistic Regression): {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
```

5.5.4. Porter_TfidVectorization

Với phương pháp xử lý văn bản Porter_TfidVectorization, nhóm sẽ làm như sau:

Multinomial Naïve Bayes

```

param_grid = {
    'alpha': [0.1, 0.5, 1.0, 1.5, 2.0], # Thay đổi giá trị của alpha
    'fit_prior': [True, False], # Thử nghiệm với fit_prior là True và False
    'class_prior': [None, [0.2, 0.8], [0.5, 0.5]] # Thử nghiệm với các giá trị khác nhau cho class_prior
}

# Tạo một đối tượng GridSearchCV để tìm kiếm siêu tham số tốt nhất
grid_search = GridSearchCV(estimator=MultinomialNB(), param_grid=param_grid, scoring='accuracy', cv=5)
grid_search.fit(X_train, y_train)

# In ra siêu tham số tốt nhất
print("Best Parameters: ", grid_search.best_params_)

# Sử dụng mô hình với siêu tham số tốt nhất để đánh giá trên tập kiểm tra
best_nb_model = grid_search.best_estimator_
y_pred_tuned = best_nb_model.predict(X_test)

# In ra báo cáo phân loại và độ chính xác
print('-----Tuned MultinomialNB-----')
print(classification_report(y_test, y_pred_tuned))
accuracy_tuned = accuracy_score(y_test, y_pred_tuned)
print(f'Accuracy (Tuned): {accuracy_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters: {'alpha': 0.1, 'class_prior': None, 'fit_prior': True}
```

SVM

```

param_grid_svm_linear = {
    'C': [0.1, 1, 10],           # Thay đổi giá trị của hằng số ống với Lè
    'kernel': ['linear'],       # Chỉ sử dụng kernel Linear
    'gamma': ['scale', 'auto'],
    'class_weight': [None, 'balanced'], # Trọng số cho các lớp, có thể là None hoặc 'balanced'
    'shrinking': [True, False],     # Thuật toán thu nhỏ Lè
}

# Create a GridSearchCV object to find the best hyperparameters
grid_search_svm = GridSearchCV(estimator=SVC(), param_grid=param_grid_svm_linear, scoring='accuracy', cv=5)
grid_search_svm.fit(X_train, y_train)

# Print the best parameters
print("Best Parameters (SVM): ", grid_search_svm.best_params_)

# Use the model with the best parameters to evaluate on the test set
best_svm_model = grid_search_svm.best_estimator_
y_pred_svm_tuned = best_svm_model.predict(X_test)

# Print classification report and accuracy
print('-----Tuned Support Vector Machine-----')
print(classification_report(y_test, y_pred_svm_tuned))
accuracy_svm_tuned = accuracy_score(y_test, y_pred_svm_tuned)
print(f'Accuracy (Tuned): {accuracy_svm_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters (SVM): {'C': 10, 'class_weight': 'balanced', 'gamma': 'scale', 'kernel': 'linear', 'shrinking': True}
```

Decision Tree

```

tree_model = DecisionTreeClassifier()

# Định nghĩa các giá trị thử nghiệm cho các tham số của Decision Tree
param_grid_tree = {
    'criterion': ['gini', 'entropy'], # Logi độ đo chất lượng tách
    'splitter': ['best', 'random'],   # Chiến lược chọn điểm tách
    'max_depth': [None, 10, 20, 30], # Độ sâu tối đa của cây
    'min_samples_split': [2, 5, 10], # Số lượng mẫu tối thiểu để tách một nút
    'min_samples_leaf': [1, 2, 4]    # Số lượng mẫu tối thiểu ở lá
}

# Tạo một đối tượng GridSearchCV để tìm kiếm siêu tham số tốt nhất
grid_search_tree = GridSearchCV(estimator=tree_model, param_grid=param_grid_tree, scoring='accuracy', cv=5)
grid_search_tree.fit(X_train, y_train)

# In ra siêu tham số tốt nhất
print("Best Parameters (Decision Tree): ", grid_search_tree.best_params_)

# Sử dụng mô hình với siêu tham số tốt nhất để dự đoán trên tập kiểm tra
best_tree_model = grid_search_tree.best_estimator_
y_pred_tree_tuned = best_tree_model.predict(X_test)

# In ra báo cáo phân loại và độ chính xác
print('-----Tuned Decision Tree-----')
print(classification_report(y_test, y_pred_tree_tuned))
accuracy_tree_tuned = accuracy_score(y_test, y_pred_tree_tuned)
print(f'Accuracy (Tuned): {accuracy_tree_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

```
Best Parameters (Decision Tree): {'criterion': 'gini', 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 10, 'splitter': 'random'}
```

Logistic Regression

```

lr_model = LogisticRegression()

# Define the hyperparameter grid for Logistic regression
param_grid_lr = {
    'penalty': ['l1', 'l2'],
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'solver': ['liblinear', 'saga']
}

# Create a GridSearchCV object
grid_search_lr = GridSearchCV(estimator=lr_model, param_grid=param_grid_lr, scoring='accuracy', cv=5)
grid_search_lr.fit(X_train, y_train)

# Print the best parameters
print("Best Parameters (Logistic Regression): ", grid_search_lr.best_params_)

# Use the model with the best parameters to predict on the test set
best_lr_model = grid_search_lr.best_estimator_
y_pred_lr_tuned = best_lr_model.predict(X_test)

# Print classification report and accuracy
print('-----Tuned Logistic Regression-----')
print(classification_report(y_test, y_pred_lr_tuned))
accuracy_lr_tuned = accuracy_score(y_test, y_pred_lr_tuned)
print(f'Accuracy (Tuned): {accuracy_lr_tuned * 100:.2f}%')

```

Kết quả thu được các giá trị tham số sau đây:

Best Parameters (Logistic Regression): {'C': 100, 'penalty': 'l2', 'solver': 'liblinear'}

5.5.5. Kết luận

Bảng kết quả độ chính xác của Machine Learning Model

	MultinomialNB	SVM	Decision Tree	Logistic Regression
Lemma Count	98,21%	98,65%	97,49%	98,65%
Lemma Tfifd	98,48%	98,3%	97,49%	98,03%
Porter Count	98,03%	98,76	97,22%	98,65%
Porter Tfifd	98,21%	98,03%	96,86%	97,76%

Sau khi Fine Tuning các mô hình, nhóm nhận thấy rằng có sự cải thiện về độ chính xác, rất nhỏ không đáng kể. Phương pháp xử lý văn bản Porter_CountVectorization được áp dụng vào mô hình SVM vẫn là mô hình có độ chính xác cao nhất.

5.6. Evaluation Deep Learning Model

Để cải thiện độ chính xác cho mô hình LSTM, ta thực hiện Evaluation cho mô hình này như sau :

```

from keras.layers import Dropout
from keras.optimizers import Adam

# Tạo mô hình Sequential
model = Sequential()

# Thêm Lớp Embedding
model.add(Embedding(10000, 64, input_length=80))

# Thêm Lớp LSTM với dropout
model.add(LSTM(100, dropout=0.1, recurrent_dropout=0.2))

# Thêm Lớp Dropout giữa LSTM và Dense
model.add(Dropout(0.1))

# Thêm Lớp Dense với activation là sigmoid
model.add(Dense(2, activation='sigmoid'))

# Tạo optimizer với learning rate
optimizer = Adam(learning_rate=0.001)

# Compile mô hình
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=optimizer)

```

Đã có sự thay đổi với model đầu tiên mà chúng ta tạo ra :

- bổ sung lớp Dropout trong cấu hình của lớp LSTM và giữa lớp LSTM và lớp Dense : nhằm giảm hiện tượng overfitting
- điều chỉnh learning rate trong optimizer Adam : cho phép kiểm soát tốc độ học của mô hình một cách chính xác hơn, góp phần cải thiện khả năng hội tụ và hiệu suất tổng thể của mô hình.

Kết quả của mô hình LSTM_PorterStemmer sau khi cải tiến là :

```

y_pred=model.predict(X_test)
y_pred_=[np.argmax(i,axis=0) for i in y_pred]
accuracy = accuracy_score(y_test_orig,y_pred_)
print(f'Dộ chính xác của mô hình là {accuracy*100:.2f}%')

```

35/35 [=====] - 1s 21ms/step

Độ chính xác của mô hình là 98.92%

Và kết quả của mô hình LSTM_LemmaStemmer sau khi cải tiến là :

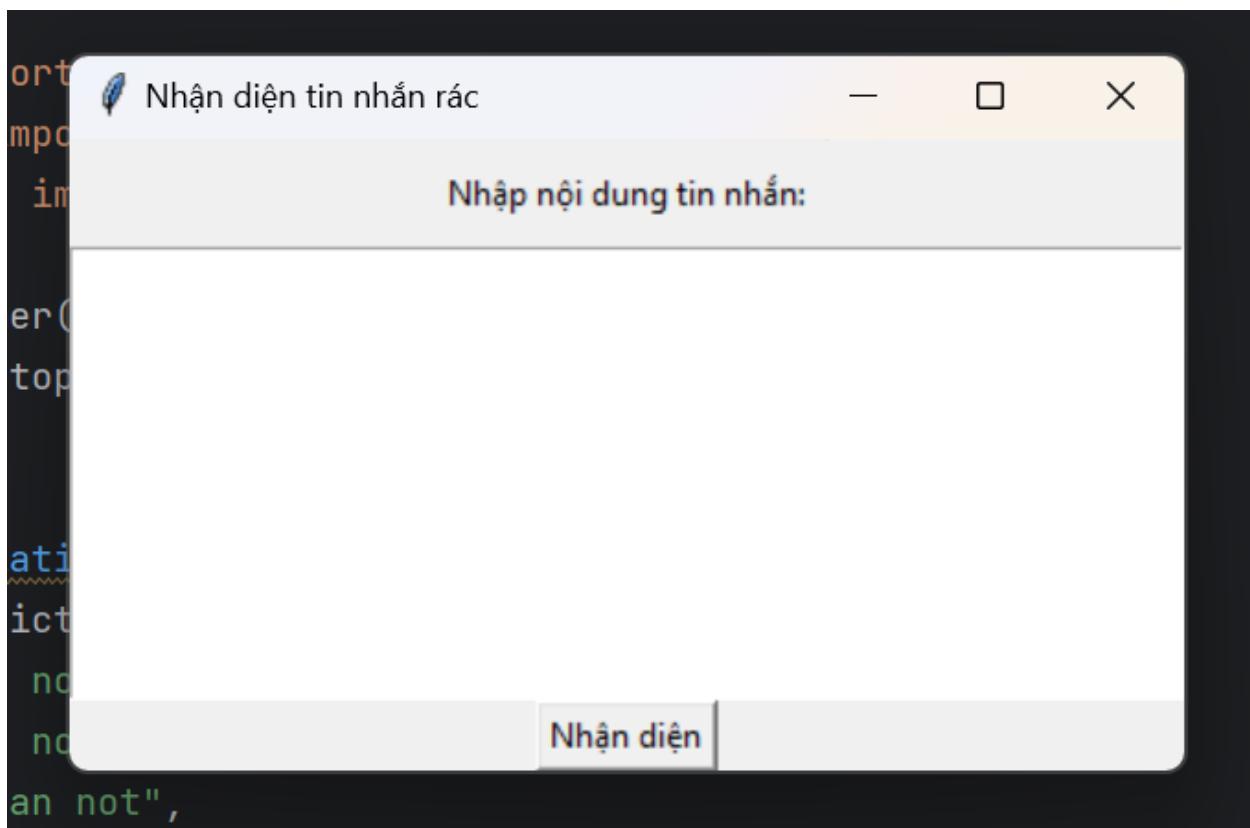
```
: y_pred=model.predict(x_test)
y_pred_=[np.argmax(i,axis=0) for i in y_pred]
accuracy = accuracy_score(y_test_orig,y_pred_)
print(f'Độ chính xác của mô hình là {accuracy*100:.2f}%')
```

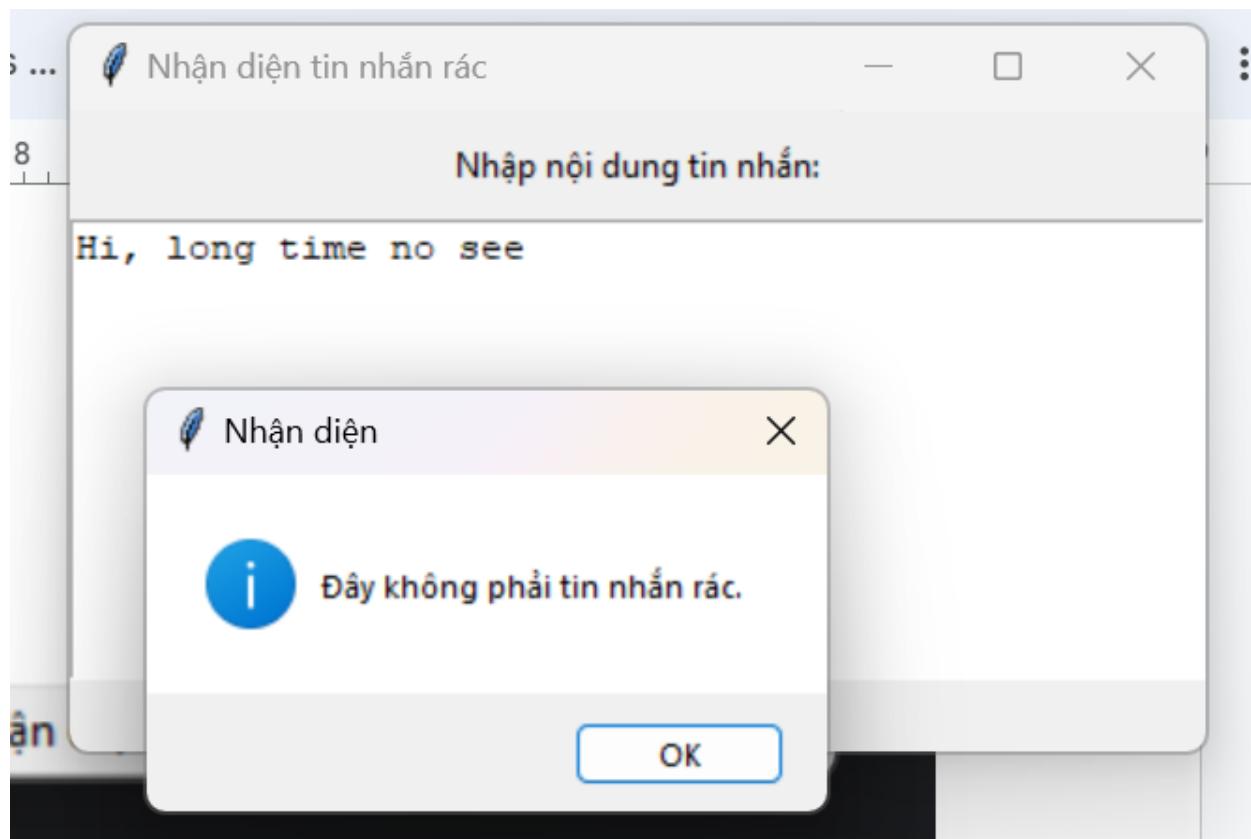
```
35/35 [=====] - 1s 11ms/step
Độ chính xác của mô hình là 99.10%
```

Chương VI: Kết Luận

Để có thể giải quyết bài toán phát hiện tin nhắn rác, nhóm thực hiện TextMining bằng 2 phương pháp bao gồm Porter Stemmer và Lemma Stemmer và Vector hóa CountVectorization và TfidfVectorization từ đó ra được tổ hợp 4 phương thức xử lý bao gồm: Lemma_CountVectorization, Lemma_TfidfVectorization, Porter_CountVectorization và Porter_TfidfVectorization. Tiếp theo đó nhóm sử dụng các mô hình phân lớp cơ bản như Multinomial Naïve Bayes, SVM (Support Vector Machine), Decision Tree, Logistic Regression, mô hình phân lớp thuộc nhóm Maxent và LSMT (Long Short Term Memory là mô hình thuộc Deep Learning) để giải quyết bài toán.

Giao diện





Sinh viên	Nhiệm vụ	Dánh giá
Phạm Phú Khánh	Text Mining và chạy các mô hình phân lớp theo các phương pháp Porter_CountVectorization, Porter_TfidfVectorization, EDA, làm giao diện	100%
Nguyễn Trịnh Hiếu Kiên	Text Mining và chạy các mô hình phân lớp theo các phương pháp Lemma_CountVectorization, Lemma_TfidfVectorization, Evaluation	100%