

Rapport Final du Projet d'IA

Modèle d'IA permettant la
détection de messages
toxiques dans un Webchat

Projet réalisé par :

BERGONZI Vinicius	ISI 3A - Groupe 5
LENING Steve	
MASI Alessio	
THEUBO Ghislain	

Table des matières

I. Projet choisi :	3
II. Description de l'application :	3
III. Jeux de données étudiés	3
IV. Description détaillée du jeu de données choisi	4
V. Architecture globale de l'application	6
VI. Description des traitements mis en place	7
1. Prétraitement du jeu de données	7
2. Entraînement du modèle	9
3. Test du modèle	12
VII. Description du WebChat	13
1. Structure du repository :	13
2. Architecture du chat :	14
3. Comment lancer l'application :	14
4. Description du fonctionnement Chat	15
VIII. Les limites de notre solution :	18
IX. Les impacts environnementaux et sociétaux :	18
1. Impact social de l'application	18
2. Impact environnemental immédiat de l'application	19
3. Impact environnemental à long terme	20
Sources	21

I. Projet choisi :

Projet de développement d'un modèle d'intelligence artificielle permettant la détection et la signalisation de messages toxiques dans une application de messagerie.

II. Description de l'application :

Il s'agit d'une application web du style système de messagerie dans laquelle chaque message est analysé automatiquement par un modèle d'Intelligence Artificielle et signalé si le taux de "toxicité" est supérieur à un certain seuil (0,6/1 par exemple).

L'analyse de message est faite par une api extérieure qui est en fait un modèle d'intelligence artificielle entraîné avec un jeu de données adapté. Ce modèle d'intelligence artificielle constitue le cœur de notre projet.

L'objectif étant d'éviter les situations de harcèlement dans un système de messagerie, il faut bien s'assurer qu'un message ou une série de messages est effectivement péjoratif(ve) à l'égard d'une personne. On ne peut donc pas confier la décision finale au modèle car ce dernier pourrait faire une erreur d'interprétation.

Si la toxicité d'un message M envoyé par un utilisateur X est supérieure au seuil fixé (0,6/1), son nom et le message sont transférés à un administrateur qui pourra déterminer s'il doit être sanctionné ou pas. Le fait que la décision finale revienne à un administrateur qui est un être humain est dû au fait que notre modèle pourrait ne pas bien interpréter certains messages, il est alors primordial qu'une personne capable d'interpréter une critique constructive, de l'humour ou encore le second degré dans une conversation puisse analyser le message avant de trancher en faveur ou en défaveur de l'individu concerné.

III. Jeux de données étudiés

Pour l'entraînement de notre modèle, nous avons entrepris de consulter et d'analyser plusieurs jeux de données avant d'en retenir le plus adapté à notre projet.

Les datasets analysés contenaient un ensemble de commentaires toxiques envoyés sur diverses plateformes comme Youtube, Wikipedia ou encore Chat messages. Ces datasets proviennent majoritairement de Kaggle [\[1\]](#) et Hugging Face [\[2\]](#) qui sont des sites proposant des datasets fiables et libres de droit.

Kaggle

- <https://www.kaggle.com/datasets/nursyahrina/chat-sentiment-dataset>
- <https://www.kaggle.com/datasets/reihanenamdari/youtube-toxicity-data>
- <https://www.kaggle.com/datasets/get2jawa/toxic-comments-train>
- <https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/data>

Hugging Face

- <https://huggingface.co/datasets/ifmain/text-moderation-410K>

IV. Description détaillée du jeu de données choisi

Parmi les différents jeux de données que nous avons parcouru, nous avons décidé d'utiliser celui de Hugging Face accessible via le lien suivant :

<https://huggingface.co/datasets/ifmain/text-moderation-410K>

Nous avons décidé d'utiliser ce jeu de données en particulier, car parmi tous ceux que nous avons analysés, c'est celui qui offre une disponibilité de données plus complète et plus hétérogène. En particulier, ce jeu de données contient un score continu pour chaque catégorie qu'il inclut, ce qui signifie qu'il est adapté non seulement à la classification binaire, mais également à une classification continue, avec un score de confiance attribué à chaque catégorie. Au moment du choix, cela constituait la principale raison, car nous souhaitons développer un modèle capable de produire de tels scores et de les analyser par la suite.

Par la suite, nous avons décidé de passer à une classification binaire plus simple, étant donné que la façon dont l'application et le système fonctionnent actuellement consiste simplement à envoyer une alerte à un administrateur si un message est toxique, indépendamment de la catégorie dans laquelle il l'est. Cependant, nous avons choisi de conserver ce jeu de données en pensant à la pérennité du projet, car nous l'avons déjà étudié et nous serons en mesure de l'utiliser pour améliorer l'application plus facilement.

En ce qui concerne l'analyse et la qualité du jeu de données, nous pensons qu'il est plus adapté au type d'application que nous souhaitons créer (messages plus courts, utilisation d'argot, comportement humain). De plus, le jeu de données contient de nombreux échantillons (environ 410K) avec divers formats et structures, ce qui est préférable lors de la construction (entraînement et test) d'un modèle.

Pour chaque échantillon de données (ligne du jeu de données), il est fourni le commentaire/message proprement dit, et les classifications binaires pour chaque

type de catégorie toxique, ainsi que les scores de 0 à 1 pour chacune d'entre elles. Voici les différentes métriques dont nous disposons pour chaque ligne du dataset :

- **harassment**
- **harassment_threatening**
- **hate**
- **hate_threatening**
- **self_harm**
- **self_harm_instructions**
- **self_harm_intent**
- **sexual**
- **sexual_minors**
- **violence**
- **violence_graphic**

Voici par exemple un aperçu des critères et des scores pour un élément du dataset utilisé.

```
"You were right to be embarrassed, casual."

{
  "categories": {
    "harassment": false,
    "harassment_threatening": false,
    "hate": false,
    "hate_threatening": false,
    "self_harm": false,
    "self_harm_instructions": false,
    "self_harm_intent": false,
    "sexual": false,
    "sexual_minors": false,
    "violence": false,
    "violence_graphic": false
  },
  "category_scores": {
    "harassment": 0.020891988649964333,
    "harassment_threatening": 0.0000451470295956824,
    "hate": 0.00011699183232849464,
    "hate_threatening": 8.157169162359423e-9,
    "self_harm": 4.6329483893714496e-7,
    "self_harm_instructions": 5.776672296065044e-8,
    "self_harm_intent": 2.0666092837018368e-7,
```

```
"sexual": 0.0007244459702633321,  
"sexual_minors": 0.00000897045811143471,  
"violence": 0.0005294209695421159,  
"violence_graphic": 0.000011841298146464396  
},  
"flagged": false  
}
```

V. Architecture globale de l'application

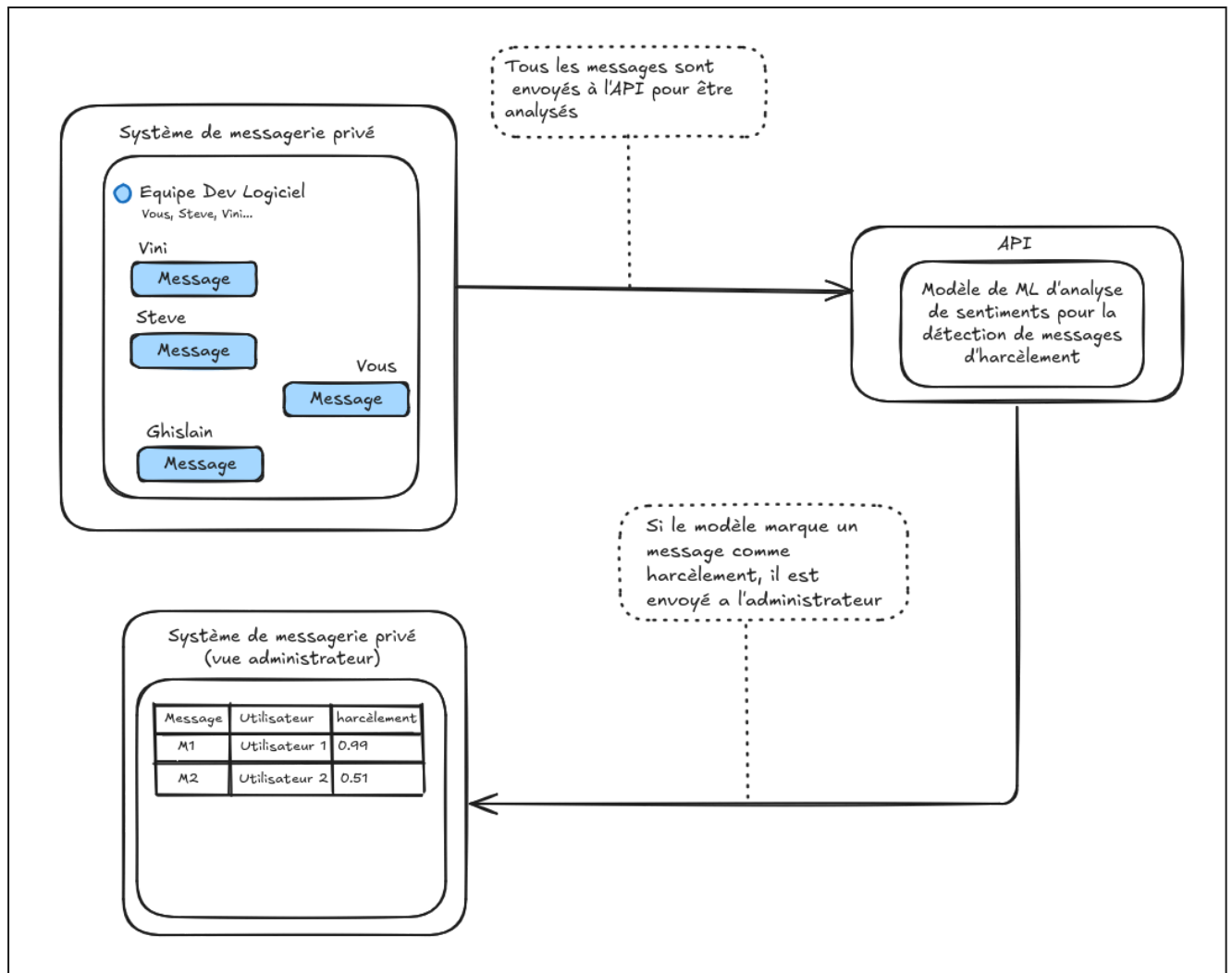
Le système est composé de :

- ❖ Un système de messagerie privée (vue utilisateur)
- ❖ Un système de messagerie privée (vue administrateur)
- ❖ Une API contenant le modèle d'intelligence artificielle utilisé pour la détection de messages toxiques,

Le tout dans une architecture Client-Serveur.

Lorsqu'un nouveau message est envoyé dans le WebChat, le modèle vérifie automatiquement s'il s'agit oui ou non d'un message toxique. Si la valeur retournée est > 0.6 , il s'agit potentiellement d'un message toxique et pour le confirmer, on envoie ce type de messages (appelés **messages suspects**) à un administrateur qui va trancher et effectuer une action qui pourrait consister à supprimer le message du Chat par exemple ou à le signaler. On espère ainsi avoir un modèle dont le résultat soit le plus cohérent possible, quoique le modèle ne pourra toujours pas identifier des situations comme les critiques constructives ou le second degré, d'où le choix de laisser la décision finale à un humain.

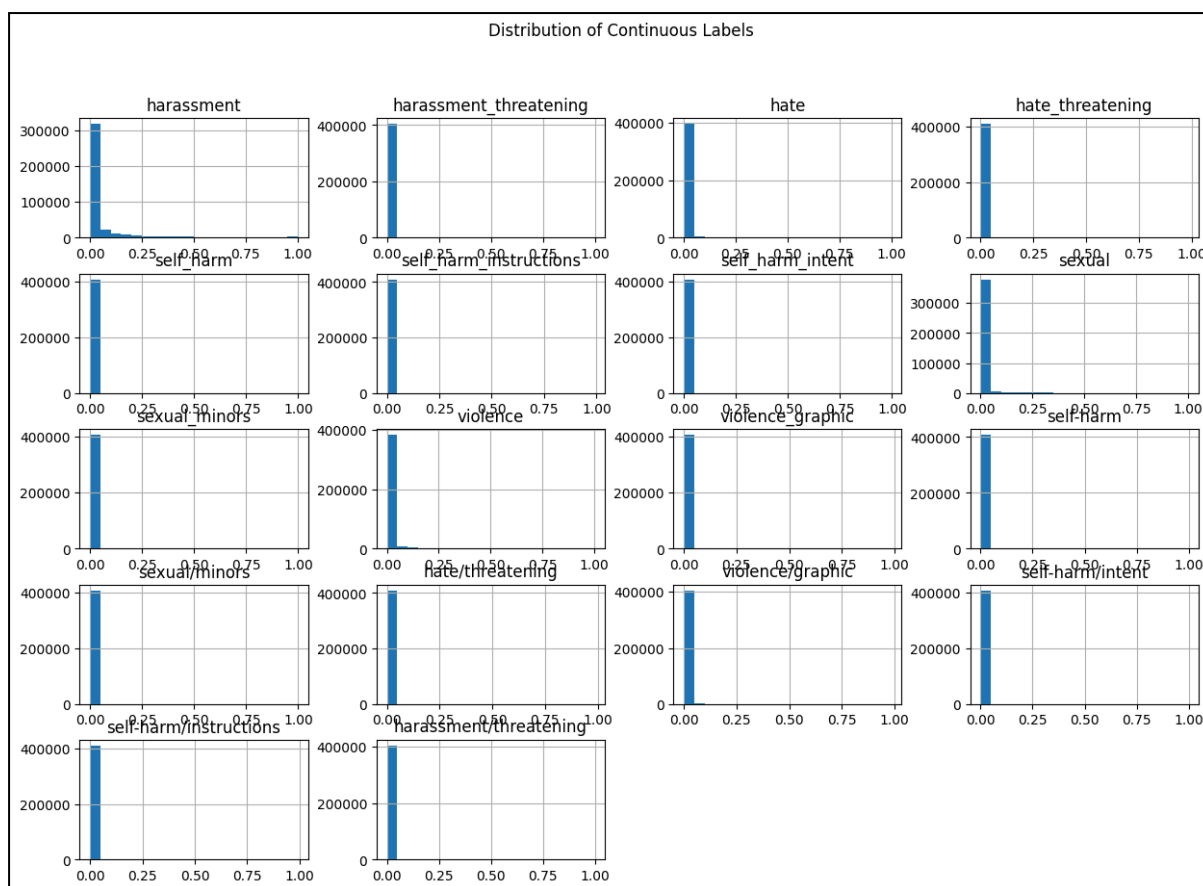
Voici l'architecture globale de notre application :



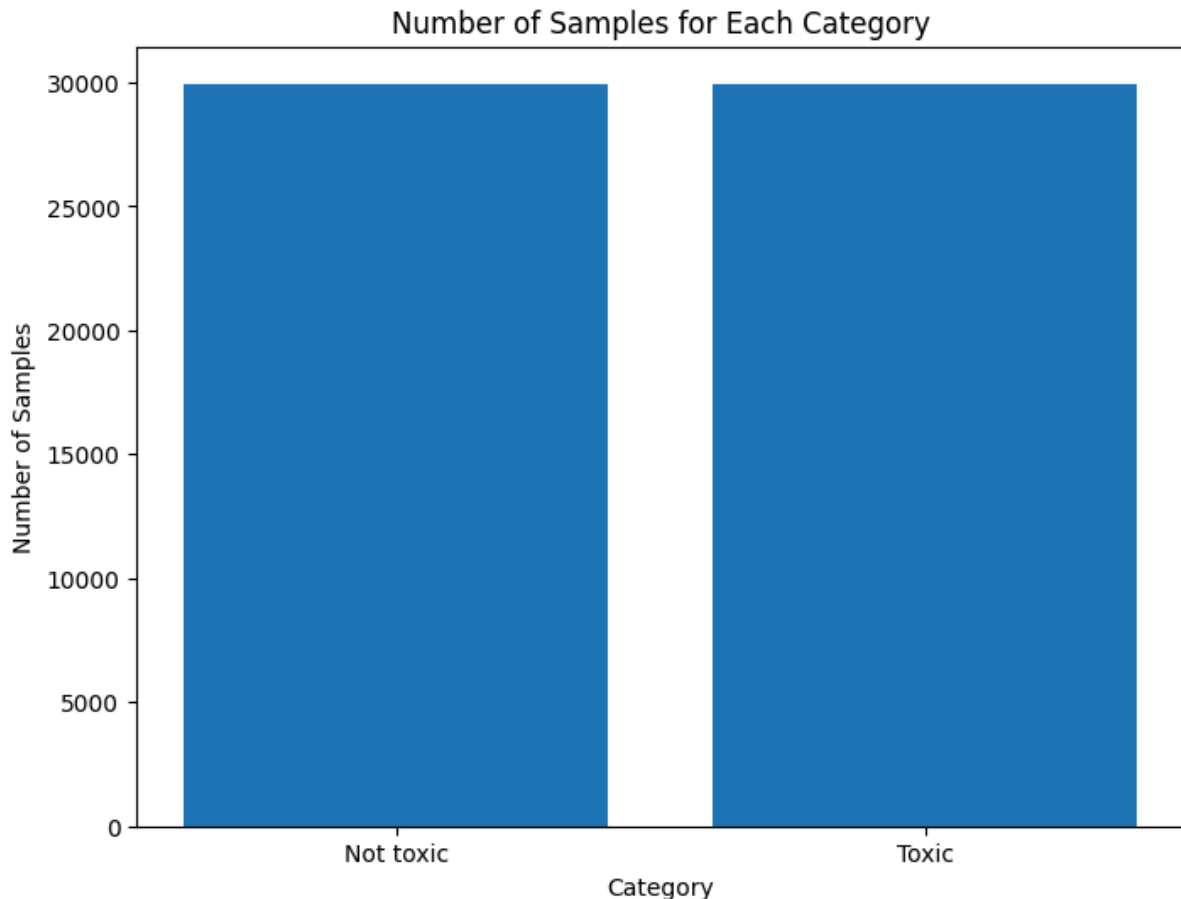
VI. Description des traitements mis en place

1. Prétraitement du jeu de données

La première étape a été d'analyser le jeu de données et de décider quel type de prétraitement effectuer. Nous avons remarqué que le jeu de données original et complet était fortement déséquilibré en faveur des données proches de 0, ce qui signifie qu'il contient davantage d'échantillons non toxiques que de messages toxiques. Cela est clairement visible sur le [graphique](#) ci-dessous, qui met en évidence ce déséquilibre.



Comme discuté dans la [section IV](#), nous avons finalement décidé de ne pas effectuer de classification multi-étiquettes, mais de tout regrouper dans une seule colonne. Pour ce faire, nous avons vérifié pour chaque entrée (chaque message) s'il existait au moins une colonne avec une valeur supérieure à un seuil fixé à 0,6. Ainsi, nous avons obtenu le jeu de données utilisé pour l'entraînement, où un échantillon est marqué comme 0 si sa valeur est inférieure ou égale à 0,6, et comme 1 dans le cas contraire. Étant donné que nous avons constaté un déséquilibre dans le jeu de données original, nous avons décidé de l'équilibrer afin d'obtenir la même quantité de données pour les deux catégories. Ainsi, le jeu de données utilisé pour l'entraînement contient environ **29000 échantillons** par catégorie.



2. Entraînement du modèle

Choix du modèle

Le modèle que nous souhaitons développer appartient au domaine du NLP (Natural Language Processing) ou TALN (Traitement Automatique du Langage Naturel) en français. Dans ce domaine, de nombreuses options sont disponibles, et notre idée était d'utiliser les transformers pour accomplir cette tâche. L'objectif était d'utiliser un modèle BERT [7] comme base et de réaliser un fine-tuning avec le jeu de données sélectionné. BERT est un modèle très puissant et largement utilisé dans le domaine du NLP, reposant sur une architecture basée sur les transformers.

Pour ce projet, nous avons choisi le modèle **DeBERTa v3 Small** [8][9] comme base pour notre modèle. DeBERTa (Decoding-enhanced BERT with Disentangled Attention) est un modèle de langage de pointe basé sur les transformers, conçu pour améliorer l'architecture BERT originale. Il introduit des améliorations clés telles que l'attention désentrelacée et un décodeur de masques amélioré, ce qui lui permet de mieux capturer les relations contextuelles dans le texte.

La version "v3" de DeBERTa intègre des avancées supplémentaires, notamment le pré-entraînement avec le Gradient-Enhanced Masked Language Modeling (**GEM**).

Cette technique permet au modèle de tirer parti des informations de gradient pendant le pré-entraînement, améliorant ainsi l'efficacité et la précision pour les tâches en aval.

La variante "Small" de la famille DeBERTa v3 était particulièrement adaptée à notre projet en raison de son équilibre entre performance et efficacité computationnelle. Elle comporte un nombre réduit de paramètres par rapport aux modèles plus grands de la famille, ce qui la rend plus rapide à entraîner tout en maintenant une précision compétitive. Ce facteur était essentiel compte tenu des contraintes de nos ressources disponibles et de la nécessité d'un modèle pratique mais robuste pour notre cas d'utilisation.

Configuration de l'entraînement

Nous avons défini une classe de jeu de données personnalisée, **HarassmentDataset**, pour prétraiter les données textuelles et les étiquettes. Cette classe utilise un tokenizer pour encoder les entrées textuelles, en appliquant une troncature et un padding pour des longueurs d'entrée uniformes, avec une longueur maximale de 128 tokens. Les données encodées, ainsi que les étiquettes correspondantes, sont stockées sous forme de tenseurs PyTorch pour un traitement en lots efficace. Cette conception garantit une compatibilité avec l'API Hugging Face Trainer.

Configuration de l'entraînement :

Paramètre	Valeur
Taux d'apprentissage	1e-5
Taille de lot	16 (entraînement et évaluation)
Nombre d'époques	5
Décroissance de poids	0.01
Stratégie d'évaluation	À la fin de chaque époque
Métrique principale	Score F1 (stratégie maximisante)
Étapes de journalisation	Toutes les 10 étapes
Point de contrôle du modèle	Activé, basé sur le meilleur score F1 (limité à deux checkpoints)
Répartition des données	0,7/0,15/0,15 (entraînement, évaluation, test)

Évaluation des performances

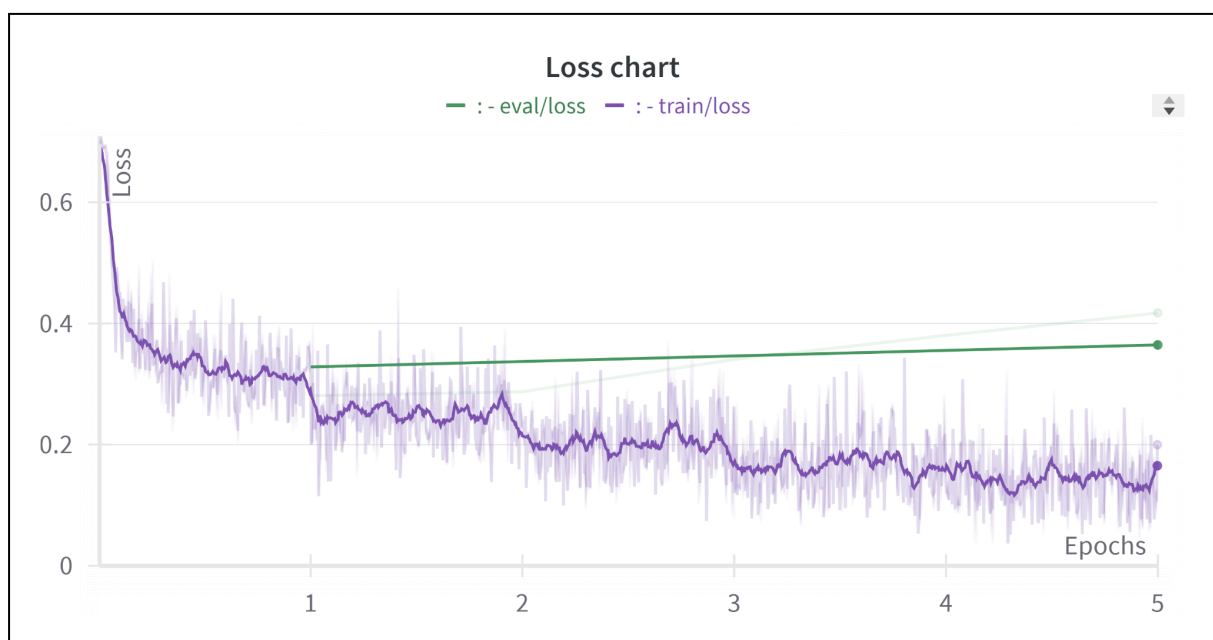
Les performances d'entraînement et de validation ont été évaluées à l'aide de métriques personnalisées définies dans la fonction `compute_metrics`. Les métriques clés incluait :

- Précision
- Recall
- Score F1
- Accuracy

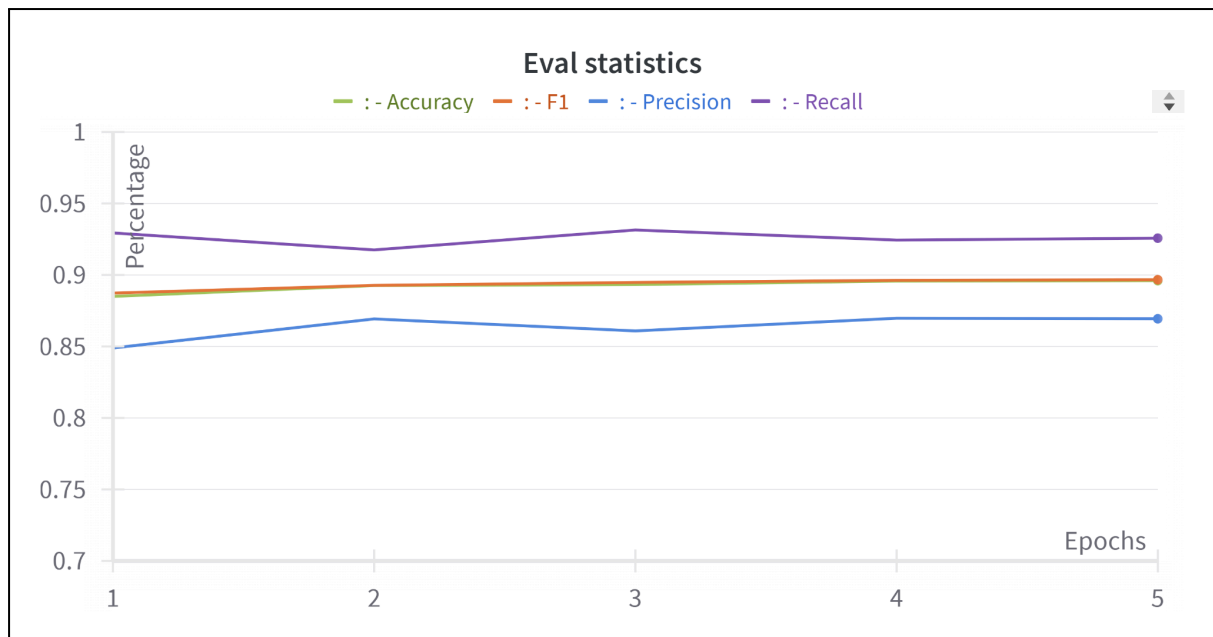
Pendant l'entraînement, la perte d'entraînement (**training loss**) a progressivement diminué au fil des époques, comme le montre le graphique ci-dessous, indiquant que le modèle a bien appris des données d'entraînement. Cependant, la perte d'évaluation (**evaluation loss**) a commencé à augmenter après les premières époques, suggérant la présence d'un surapprentissage (overfitting).

Nous avons envisagé deux explications potentielles à ce comportement :

1. **Simplicité de la tâche** : La tâche de classification binaire, avec une séparation claire entre les classes, permet au modèle d'atteindre rapidement de bonnes performances sur le jeu de données d'entraînement, mais peut entraîner un surapprentissage si le modèle "mémorise" les données au lieu de généraliser les motifs.
2. **Capacité limitée du modèle** : Le modèle DeBERTa v3 Small a une capacité suffisante pour gérer la complexité de la tâche. Il est donc peu probable que le surapprentissage observé soit dû à une saturation du modèle, car il montre de bonnes performances initiales et reste efficace dans l'ensemble.



Malgré ce surapprentissage, les métriques finales d'évaluation montrent que le modèle atteint des performances satisfaisantes, avec des scores F1 et une précision adaptés au déploiement dans le cadre de ce projet.

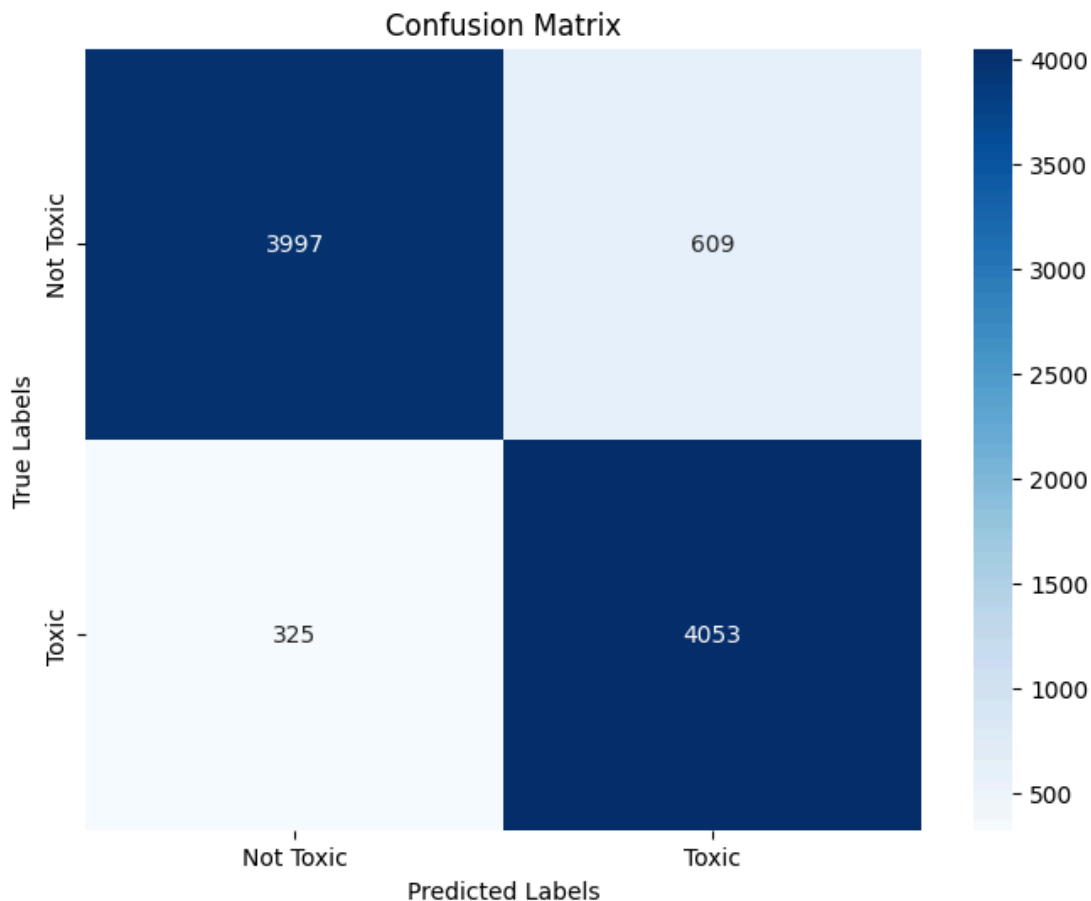


3. Test du modèle

L'évaluation du modèle sur le jeu de données de test a donné des métriques de performance solides:

- **Accuracy** : 0,90
- **Score F1** : 0,90
- **Precision** : 0,87
- **Recall** : 0,93

Ces résultats soulignent la capacité du modèle à classifier avec précision les messages toxiques et non toxiques, trouvant un excellent équilibre entre l'identification du contenu toxique et la minimisation des faux positifs. La matrice de confusion ci-dessous illustre davantage ces performances :



Le recall élevé (**0,93**) démontre que le modèle identifie avec succès la grande majorité des contenus toxiques, ce qui est une exigence essentielle pour un outil de modération où l'absence de messages nuisibles pourrait avoir des implications graves. En parallèle, la précision (**0,87**) garantit que les messages signalés sont majoritairement exacts, limitant ainsi les interventions inutiles. De plus, le score F1 équilibré (**0,90**) reflète l'harmonie globale entre la précision et le recall, montrant la capacité du modèle à se généraliser efficacement sur les deux classes. Le nombre relativement faible de faux positifs et de faux négatifs, comme indiqué dans la matrice de confusion, renforce la fiabilité du modèle, tant pour la détection du contenu toxique que pour le maintien de l'exactitude sur les cas non toxiques.

VII. Description du WebChat

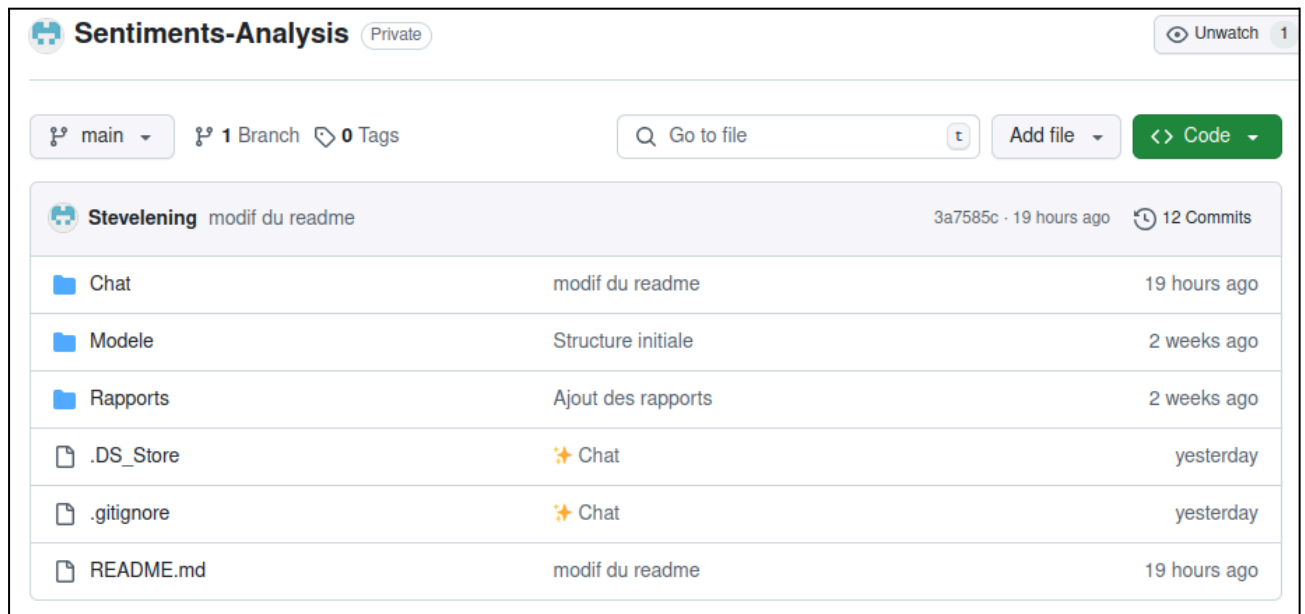
Nous avons mis en place un repository GitHub pour la création et la collaboration des membres de l'équipe sur ce projet.

1. Structure du repository :

- ❖ Un répertoire **Chat/** qui contient l'essentiel de l'application web de chat

- ❖ Un répertoire **Modele/** qui contient le modèle d'IA que nous utilisons pour l'analyse et la classification des messages échangés dans le chat. Il contient également le code que nous avons utilisé pour l'entraînement du modèle dans un fichier **Model_classification_training.ipynb**
- ❖ Un répertoire **Rapports/** qui contient tous les rapports aux formats word et pdf ainsi que la diapo du dernier pitch.
- ❖ Un fichier **README.md** permettant une prise en main rapide de l'application par les utilisateurs.

Voici un aperçu de la structure de notre repository GitHub :



2. Architecture du chat :

- **Frontend** : HTML5, CSS3 et JavaScript
- **Backend** : NodeJs et Express
- **Base de Données** : SQLite3

3. Comment lancer l'application :

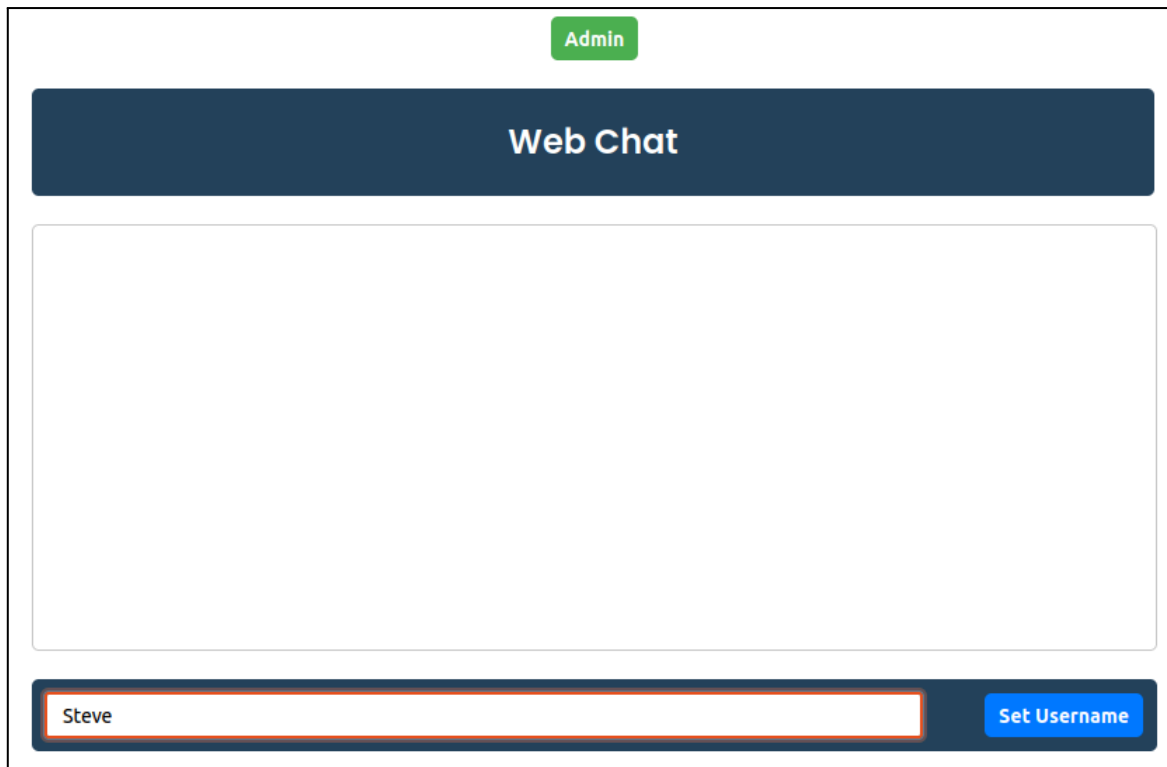
- ❖ Se déplacer dans le répertoire **Chat/** en exécutant la commande `cd Chat`
- ❖ Exécuter la commande `npm install` pour installer les dépendances nécessaires à l'exécution du serveur.
- ❖ Exécuter la commande `node index.js` pour lancer le serveur express. Si tout se passe bien, vous verrez apparaître le message **Server running at http://localhost:3000** dans votre terminal.
- ❖ Dans le navigateur, aller à l'adresse `http://localhost:3000`.

- ❖ Télécharger le dossier `model_save` à l'adresse suivante [model_folder \[6\]](#) et le mettre dans le dossier `Modele/`
- ❖ Dans une autre fenêtre du terminal, accéder au dossier `Modele/` avec la commande `cd Modele/` et lancer le serveur python qui va faire l'évaluation des messages avec la commande `python server.py`
- ❖ Vous pouvez maintenant revenir dans le navigateur et utiliser le Chat.
- ❖ Pour vous connecter en tant qu'administrateur, veuillez cliquer sur le bouton `Admin` en haut de la page et utiliser les identifiants `admin` qui se trouvent dans le fichier `Chat/config.js`.

4. Description du fonctionnement Chat

Le webchat est l'infrastructure que le modèle va utiliser pour collecter les données à évaluer auprès des utilisateurs (les messages transmis dans le webchat). L'idée est de fournir une interface graphique aux utilisateurs interagissant les uns avec les autres par échanges de messages dans l'application, tout en exécutant le modèle en arrière-plan, sans aucune interférence avec les actions des utilisateurs (l'exécution du modèle se fait de façon complètement transparente pour les utilisateurs du Chat). Nous utilisons une architecture **Client/Serveur** et les technologies HTML5, CSS3 et JavaScript pour l'interface qui est **le client**, ainsi que Express et NodeJS pour le backend qui est **le serveur**. Nous utilisons SQLite3 comme **base de données** afin de stocker les messages des utilisateurs.

Ci-dessous se trouve un aperçu de la première interface de l'application, où l'utilisateur peut choisir son propre nom d'utilisateur qui sera stocké dans la session du navigateur et affiché une fois que le message arrivera à destination. Afin de simuler un environnement réel pour cette application, il est nécessaire d'utiliser deux navigateurs différents pour créer des clients différents, puisque le nom de l'utilisateur est stocké dans la session du navigateur et que pour un même navigateur et une même adresse, on ne peut pas avoir deux sessions différentes.

A screenshot of a web chat interface. At the top center, there is a green button labeled 'Admin'. Below it is a dark blue header bar with the text 'Web Chat' in white. The main area is a large, empty white rectangle. At the bottom, there is a dark blue footer bar. On the left of this bar is a text input field with the text 'Steve' inside. On the right of the bar is a blue button labeled 'Set Username'.

Lorsque l'utilisateur renseigne son nom d'utilisateur et clique sur le bouton `Set Username`, un utilisateur portant ce nom est créé dans la base de données et ajouté à la session du navigateur. Cet utilisateur est ensuite redirigé vers la page ci-dessous où il peut échanger avec les autres utilisateurs du Chat.

Si le serveur est configuré et opérationnel, une fois que l'utilisateur envoie un message à un autre client, l'utilisateur final doit recevoir le message, le nom de l'utilisateur qui l'a envoyé et un indicateur indiquant si le message a déjà été évalué par le modèle ou si cette évaluation est en attente. Si le message a déjà été évalué, l'interface va indiquer le niveau de toxicité trouvé par le modèle.

Le serveur est conçu suivant une architecture **RestAPI** qui propose deux endpoints principaux pour la collecte et l'évaluation des données (messages utilisateur) : `/messages` et `/evaluate`. Le premier se charge de fournir tous les messages qui n'ont pas encore été évalués, cela peut se faire en ajoutant simplement un flag « **evaluated** » sur chaque message. Le second reçoit un **messageId** et un **toxicity** qui est en fait le score de toxicité et qui contient la valeur calculée par le modèle dans la base de données (valeur comprise entre 0 et 1). Cette valeur est ensuite affichée à l'utilisateur. Si le message n'est pas encore évalué, l'utilisateur verra un indicateur (`Not evaluated`) devant le message.

Admin

Web Chat

Steve: Hi Vini (Toxicity: 0.0004)

Steve: How are you ? (Toxicity: 0.0003)

Vini: Hi Steve (Toxicity: 0.0002)

Vini: I'm fine and you ? (Toxicity: 0.001)

Vini: I will punch you if you don't stop kidding on me. (Toxicity: 0.9973)

Steve: Come on mother fucker, I will beat you (Toxicity: 0.9974)

Vini: You too mother fucker, calm down, I was just joking. (Toxicity: 0.9965)

Send

Logout

On peut clairement voir sur la capture d'écran précédente que le modèle a su identifier les propos violents et les injures échangés par les deux interlocuteurs. De son côté, l'administrateur a accès à tous les messages dont la toxicité est supérieure ou égale à 0,6. Il peut les analyser et agir en conséquence. Pour tester, vous pouvez accéder aux identifiants de connexion administrateur dans le fichier **Chat/config.js**.

Welcome admin

Liste des messages suspects

Name	Message	Toxicity
Vini	I will punch you if you don't stop kidding on me.	0.9973
Steve	Come on mother fucker, I will beat you	0.9974
Vini	You too mother fucker, calm down, I was just joking.	0.9965

Retour
Logout

VIII. Les limites de notre solution :

❖ La performance pratique du modèle

On constate que les modèles de la classification de sentiments ont du mal à analyser l'humour, le second degré ou même juste une critique constructive lors d'un débat. Le nôtre ne fait pas exception, c'est pourquoi nous avons ajouté un administrateur pour prendre la décision finale : le modèle permet donc de faire une première sélection et l'administrateur peut ainsi se focaliser uniquement sur les messages suspects. Par ailleurs, lors des tests, nous avons constaté que notre modèle peut être mis en difficulté par des variations subtiles dans le texte, comme des lettres échangées ou modifiées dans des phrases offensantes. Malgré les très bonnes métriques que nous avons obtenues (Score F1 de 0,90 et Accuracy de 0,90), nous ne pouvons pas être sûrs à 100% de la classification effectuée par le modèle.

❖ La scalabilité du serveur python

Le serveur python qui est responsable de l'évaluation des messages n'a pas été pensé pour supporter la montée en charge, c'est-à-dire l'augmentation trop importante du nombre de connexions simultanées. Il faudrait mettre en place une architecture plus élaborée afin de permettre aux utilisateurs d'avoir la même qualité de service même lorsque le nombre de connexions explose.

IX. Les impacts environnementaux et sociétaux :

1. Impact social de l'application

La finalité de notre application est d'être utilisée en tant que système de messagerie dans les entreprises afin de traquer le harcèlement en milieu professionnel. Cependant, elle pourrait conduire à :

❖ Une augmentation du stress chez les employés :

Etant donné qu'une application de messagerie est d'abord un endroit où on peut échanger librement entre collègues et apporter des critiques constructives pour améliorer la qualité du travail, il est primordial que cet outil ne devienne pas une source de stress pour les employés. Ce qui pourrait arriver si chacun de leurs messages est scruté à la loupe pour détecter le moindre problème.

❖ Une utilisation détournée de l'application :

Les gérants d'une entreprise pourraient par exemple utiliser notre application pour établir une politique de communication trop stricte au sein de leur entreprise, interdisant ainsi l'humour et certains types de messages qui détendent l'atmosphère et améliorent les conditions de vie au travail.

2. Impact environnemental immédiat de l'application

Nous avons évalué les émissions de carbone et les coûts en électricité liés à l'entraînement du modèle et voici les résultats que nous avons obtenus :

- ❖ **Temps total d'entraînement** : $12\text{h } 39\text{m } 27\text{s} = 43200 + 2340 + 27 = 45567\text{s}$
- ❖ **Consommation moyenne d'énergie pendant l'entraînement** : 63W
- ❖ **Énergie totale**: $45567\text{s} \times 63\text{W} = 2\,870\,721$ Joules, soit 0,797 kWh

Le modèle a été entraîné en utilisant le datacenter Google Colab suivant :

```
{
  "ip": "34.139.159.179",
  "hostname": "179.159.139.34.bc.googleusercontent.com",
  "city": "North Charleston",
  "region": "South Carolina",
  "country": "US",
  "loc": "32.8546,-79.9748",
  "org": "AS396982 Google LLC",
  "postal": "29415",
  "timezone": "America/New_York",
  "readme": "https://ipinfo.io/missingauth"
}
```

En utilisant les informations fournies par Google [4], il est possible d'estimer les émissions totales de carbone. Pour le datacenter de **Caroline du Sud** que nous avons utilisé, l'intensité moyenne en carbone du réseau électrique est de **560 gCO₂/kWh**.

Nous pouvons donc calculer les émissions totales de carbone de la manière suivante :

Émissions totales de carbone: $560 \text{ gCO}_2/\text{kWh} \times 0.797\text{kWh} = 446.32 \text{ gCO}_2$

Maintenant que le modèle est entraîné, il faudra inférer là-dessus, c'est-à-dire évaluer les différents messages envoyés dans le WebChat. Nous avons donc également évalué les émissions liées à l'inférence sur le modèle obtenu :

Estimation des émissions pour l'inférence

- ❖ **Median inference time**: 1,85s
- ❖ **Consommation moyenne d'énergie**: 56W
- ❖ **Énergie totale**: $1,85\text{s} \times 56\text{W} = 103,6$ Joules, soit 0.0000288 kWh pour un seul message.

Cela signifie que si **27 674 messages** étaient évalués, les coûts énergétiques des inférences dépasseraient les coûts d'entraînement du modèle.

En évaluant les scénarios d'inférence, nous pouvons supposer plusieurs tailles d'entreprises, échangeant un nombre différent de messages chaque jour, en supposant le même datacenter qu'auparavant.

Company size	Messages/day	Total energy used	Carbon used
Small	1 thousand	0,0288 kWh	16,12 gCO ₂
Medium	30 thousand	0,864 kWh	483,84 gCO ₂
Big	250 thousand	7,2 kWh	4,032 kgCO ₂
World emails (2022) [5]	333.2 billion	9596,1 tWh	5373849,6 tCO ₂

3. Impact environnemental à long terme

- ❖ L'application actualise la liste des messages chaque 1000 millisecondes (1 seconde) pour pouvoir évaluer les potentiels nouveaux messages non évalués et ça implique un nombre important de requêtes. Cela conduit naturellement à un impact écologique plus important. Il faudrait trouver un compromis entre le fait de vouloir évaluer les nouveaux messages le plus rapidement possible et le nombre de requêtes que cela implique. En optimisant cette fréquence d'actualisation du site, on pourra réduire l'impact de l'application.
- ❖ Si l'application est utilisée à grande échelle, la multiplication des requêtes pour analyser les messages entraînera une consommation énergétique plus importante. Nous n'avons pas dimensionné le serveur pour qu'il soit en mesure de gérer une charge de travail trop importante.
- ❖ Le WebChat étant un système de messagerie, il s'agit d'une application temps réelle qui doit être activée en continu pour garantir une réponse rapide aux requêtes des utilisateurs. Cette activité constante peut entraîner une utilisation prolongée des ressources informatiques, augmentant l'empreinte environnementale de notre application [\[3\]](#). Il faudrait idéalement penser à une architecture dans laquelle les ressources ne sont utilisées que lorsque cela est vraiment nécessaire. Les fournisseurs de Cloud de nos jours proposent ce genre de solution en allouant en temps réel juste la quantité de ressources nécessaire et dans la limite d'une quantité maximale.

- ❖ L'amélioration ou le remplacement du modèle pour maintenir des performances optimales peut nécessiter un réentraînement fréquent du modèle, augmentant ainsi la consommation de ressources à long terme de notre solution qui serait alors déployée en ligne sur une architecture Cloud par exemple [4].

Sources

1. Kaggle, une plateforme web contenant entre autres des datasets fiables et libres. Accessible sur <https://www.kaggle.com/datasets>
2. Hugging Face, une plateforme contenant entre autres des datasets sur la classification de texte. Accessible sur <https://huggingface.co/datasets>
3. Google Cloud Sustainability. (n.d.). *Carbon-aware computing*. Accessible sur <https://sustainability.google>
4. Google Cloud Data Center Metrics. (n.d.). *Energy and carbon impact of AI training*. Accessible sur <https://cloud.google.com/sustainability/region-carbon>
5. DemandSage (2024). How Many Emails Are Sent Per Day?. Accessible sur <https://www.demandsage.com/how-many-emails-are-sent-per-day/>
6. Modèle final téléchargeable pour le lancement du serveur. Accessible sur https://drive.google.com/drive/folders/13rzgQqhpOP6GMd-pG7Dhslygs12Ygo_hV?usp=sharing
7. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Accessible sur <https://arxiv.org/abs/1810.04805>
8. DeBERTaV3: Improving DeBERTa using ELECTRA-Style Pre-Training with Gradient-Disentangled Embedding Sharing. Accessible sur : <https://arxiv.org/abs/2111.09543>
9. Deberta-v3-small. Disponible sur : <https://huggingface.co/microsoft/deberta-v3-small>