Écouter pour comprendre : l'IA au service des Urgences Médicales

Module HAI823I

Projet TER

Janvier 2025 - Mai 2025

Encadrant de projet :
Pascal PONCELET

Participants :
Sonia KECHAOU
Rim LEMRABOTT
Fannie RUIZ

Sommaire

1 Introduction

Dans un contexte d'urgence médicale, chaque seconde compte. Les professionnels de santé doivent souvent poser un premier diagnostic en s'appuyant uniquement sur les informations verbales fournies par le patient, parfois au cours d'un simple appel téléphonique. Or, dans ces situations critiques, certains mots ou expressions clés peuvent orienter de manière décisive la prise en charge : mention d'une allergie, antécédents familiaux, symptômes respiratoires, etc. Détecter automatiquement ces éléments dans un discours pourrait considérablement améliorer la rapidité, la fiabilité du triage médical et libérer du temps qui peut s'avérer précieux à nos médecins.

Ce rapport s'inscrit dans le cadre d'un projet de recherche appliquée mené en collaboration avec le service des urgences du CHU de Montpellier et le laboratoire IES. L'objectif est de développer un système intelligent capable d'identifier, à partir d'un enregistrement audio, les mots et expressions pertinents pour un diagnostic médical. Cette approche repose sur des techniques avancées d'analyse audio, de traitement automatique du langage (NLP) et d'apprentissage profond, en particulier via le Fine-tuning.
Ce travail se veut à la fois une exploration technique du potentiel de l'IA dans le domaine de la santé, et une contribution concrète à l'amélioration des outils de diagnostic d'urgence.

Notre rapport est organisé de la manière suivante : nous présentons en premier lieu les concepts et connaissances nécessaires à la compréhension de notre rapport. Par la suite, nous détaillons la mise en place et l'évolution des modèles que nous avons élaborés, de l'ingénierie des données au dernier modèle retenu. Nous continuons en exposant le processus de Fine-Tuning que nous avons réalisé sur un modèle basé sur BERT. Enfin, nous illustrons l'organisation du projet, et une conclusion vient terminer ce rapport et offrir une perspective d'amélioration.

2 Concepts et outils utilisés

Afin de mieux comprendre et de nous préparer à la tâche que nous avons à réaliser, nous nous sommes renseignées et documentées à l'aide des articles suivants :

- Deep learning with word embeddings improves biomedical named entity recognition[3]
- Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling[5]

2.1 Concepts techniques fondamentaux

Machine learning

Le Machine Learning (ML), ou apprentissage automatique, est un domaine de l'Intelligence Artificielle dont le but est de procurer à un système informatique des capacités de prédiction, qu'il acquiert en se basant sur une grande quantité de données qui doivent lui être fournies. Il y a trois grands types de Machine Learning : l'apprentissage supervisé, l'apprentissage non-supervisé et l'apprentissage par renforcement [6]. Nous allons nous concentrer sur l'apprentissage supervisé.

Apprentissage Supervisé

Le principe est de fournir au modèle des données étiquetées, c'est-à-dire avec la réponse correcte à prédire. Les données sont nommées la matrice **X**, les étiquettes sont un vecteur nommé **y** (voir figure 1).

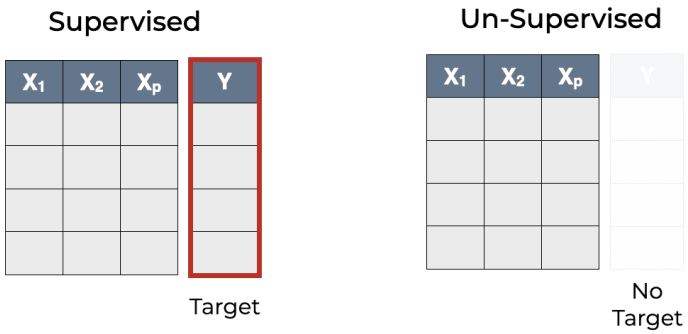

FIGURE 1 – Les données en apprentissage supervisé et non-supervisé

En général nous coupons en 2 nos données : une majorité est assignée à l'apprentissage par le modèle, le reste est dédié à tester ses performances. La fonction train_test_split, de la bibliothèque scikit-learn, permet de diviser automatiquement un jeu de données en deux parties : les données d'entraînement (train) pour apprendre, et les données de test pour évaluer si le modèle généralise bien à de nouvelles données. Cela permet de s'assurer que le modèle sera bon sur des cas réels.

Deep Learning et Réseau de Neurones

Le Deep Learning est un sous-domaine du Machine Learning utilisant des réseaux neuronaux artificiels apprenant à partir de très grandes quantités de données. Les réseaux de neurones sont conçus pour fonctionner comme le cerveau humain : ils sont

composés d'unités computationnelles, appelées neurones, organisées en dizaines voire centaines de couches successives, qui ont pour fonction de communiquer de couche en couche grâce à des liens, pondérés par des poids appris lors de l'entraînement. Il y a une couche d'entrée où sont déposées les données, des couches cachées réalisant le travail de raisonnement, et une couche de sortie où seront disponibles les prédictions du modèle.

Afin d'apprendre les poids de ses liens, un réseau de neurones a besoin de faire un "aller-retour" sur ses couches de neurones avec des données. L'"aller" est appelé forward propagation : chaque neurone apprend et propage des poids à ses successeurs. Il s'ensuit un "retour", nommé back propagation : une prédiction est réalisée et un taux d'erreur est calculé sur la couche de sortie. Ce taux d'erreur est ensuite propagé de la couche sortie à la couche entrée en "marche arrière" afin de déterminer par des calculs la direction que le modèle doit prendre lorsqu'il voudra améliorer ses poids.

Pour en savoir plus sur le fonctionnement du Deep Learning, nous vous conseillons ce site [1].

Epoch

Un passage complet, donc un forward et un back propagation, sur l'ensemble du jeu de données d'entraînement est appelé une "epoch". Il représente le nombre de fois que l'algorithme parcourt les données pendant la phase d'entraînement.

Évaluation

Pour évaluer les capacités d'un modèle, nous nous reposons sur des métriques simples : la précision globale (accuracy), la précision et le rappel.
La précision globale (Accuracy) mesure le pourcentage de prédictions correctes parmi toutes les prédictions effectuées. Elle est définie par :

$$Accuracy = \frac{TruePositive + TrueNegative}{TruePositive + TrueNegative + FalsePositive + FalseNegative}$$

La précision mesure la proportion de prédictions positives correctes parmi toutes les prédictions positives faites par le modèle. Elle est définie par :

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

Le rappel, aussi appelé sensibilité, mesure la capacité du modèle à récupérer correctement les individus positifs parmi tous les réels positifs. Il est défini comme :

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

Validation Croisée

Afin de se reposer sur des mesures d'évaluation plus robustes et fiables, il est nécessaire d'effectuer une validation croisée en K-Folds, au lieu de simplement couper nos données en 2 par train_test_split. Faire des K-Folds consiste à diviser les données en K blocs de tailles approximativement égales, et réaliser un cycle entier d'apprentissage et

1. `https://datascientest.com/deep-learning-definition` (dernière consultation : 23/05/2025)

d'évaluation sur chaque bloc : à chaque itération, un bloc sert d'ensemble de test, tandis que les K−1 autres blocs sont utilisés pour l'entraînement. L'ordre des blocs n'est pas pris en compte. La figure 2 illustre ce processus pour une validation croisée à 5 plis (5-fold cross-validation). Chaque colonne représente une itération, avec les blocs utilisés pour l'entraînement en bleu (80% du jeu de données) et ceux utilisés pour le test en rouge (20% du jeu de données).

Ainsi, le modèle est évalué sur plusieurs tentatives, ce qui permet de pondérer les mesures d'évaluation par un écart-type et de se rendre compte si le modèle est instable.

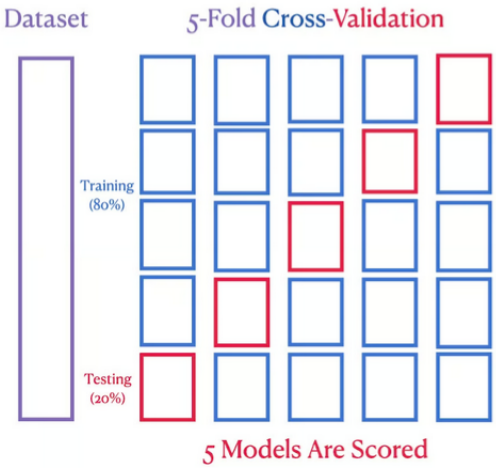

FIGURE 2 – Exemple illustrant le fonctionnement de la validation croisée en K-Folds (source :)

NER

NER (Named Entity Recognition), ou Reconnaissance d'Entités Nommées en français, est une tâche consistant à identifier les entités nommées dans un texte. L'objectif de NER est de détecter automatiquement des mots ou expressions appartenant à des catégories spécifiques comme les personnes, les lieux, les organisations, les dates ou heures, les montants ou devises, etc.

Réglage fin ou ajustement fin (Fine-Tuning en anglais)

Il consiste à prendre un modèle déjà existant et entraîné et à l'adapter dans de nouveaux contextes d'utilisation, tout en veillant à conserver ses hautes capacités de prédiction sur les tâches précédentes.

2.2 Outils utilisés

Pour cette tâche de Deep Learning, les données utilisées proviennent du site Kaggle, et nous avons rédigé nos codes en Python, sur la plateforme Colab de Google. L'outil de communication et d'organisation principal fut l'application Discord. Nous avons rédigé notre rapport en Latex sur le site PLMlatex du CNRS.

3 Modèles de Détection d'Entités Nommées

Comme première grande partie de ce projet, notre défi est de créer nous même un modèle de Deep Learning capable de réaliser de la Reconnaissance d'Entités Nommées (NER) (voir section 2.1), et ce grâce aux nombreux outils et bibliothèques Python de manipulation de données, de Machine Learning et de Deep Learning. Ici, notre but n'est pas de reconnaître des éléments propres à la médecine ou aux urgences médicales.

Nous allons dans un premier temps détailler et analyser nos données, puis nous présenterons 4 modèles, de plus en plus performants, que nous avons conçu pour cette tâche de NER.

3.1 Données et protocoles

3.1.1 Analyse des données

Nous avons choisi de prendre nos données d'entraînements sur le site Kaggle, où nous avons sélectionné le fichier intitulé "Named Entity Recognition (NER) Corpus" [2], qui est un dataset issu de la Groningen Meaning Bank [8]. Il se compose de 143877 phrases en Anglais et des Tags associés à chaque mot. Notre fichier de données comporte les attributs suivants :

- *ID* : le numéro de la phrase
- *Sentence* : la phrase en anglais
- *Tag* : les étiquettes correspondant aux mots de la phrase

D'autres caractéristiques utiles de nos phrases : la plus grande est de 104 mots, la plus petite 1, avec une moyenne de 22 mots par phrase et une médiane à 21 mots.

Concernant les Tags, en voici la liste exhaustive :

- *O* : un mot que l'on ne souhaite pas reconnaître.
- *B-org* et *I-org* : une organisation (ex : American Diabetes Association)
- *B-gpe* et *I-gpe* : une entité politique (ex : American)
- *B-per* et *I-per* : une personne (ex : Nancy-Amelia Collins)
- *B-eve* et *I-eve* : un événement (ex : 2012 Summer Olympics)
- *B-nat* et *I-nat* : un phénomène naturel (ex : H1N1)
- *B-art* et *I-art* : un artefact (Baghdad International Airport)
- *B-geo* et *I-geo* : une entité géographique (ex : Egypt)
- *B-tim* et *I-tim* : une indication temporelle (ex : January)

Dans un mot ou un groupe de mots, le premier sera étiqueté avec une première lettre *B* et les suivants avec une première lettre *I*. Ainsi, 'American Diabetes Association' donnera donc ['B-org', 'I-org', 'I-org'].

2. https ://www.kaggle.com/datasets/naseralqaydeh/named-entity-recognition-ner-corpus (dernière consultation : 03/04/2025)

Pour un exemple complet tiré de notre dataset, la phrase :
"British police arrested Ramda in 1995."
a les Tags suivants :
['B-gpe', 'O', 'O', 'B-per', 'O', 'B-tim', 'O']

La figure 3 montre l'occurrence de chacun de ses Tags. Comme nous pouvons le voir le Tag 'O' est nettement sur-représenté.

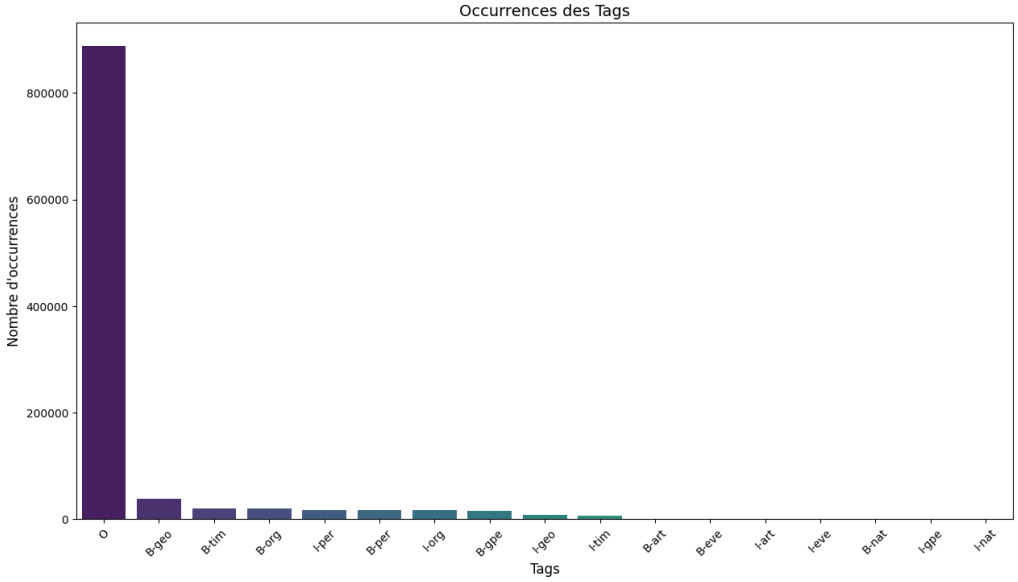

FIGURE 3 – Occurrences des Tags du dataset

Afin de mieux visualiser la distribution des Tags nous avons supprimé le Tag 'O' du graphique. Le résultat est la figure 4.

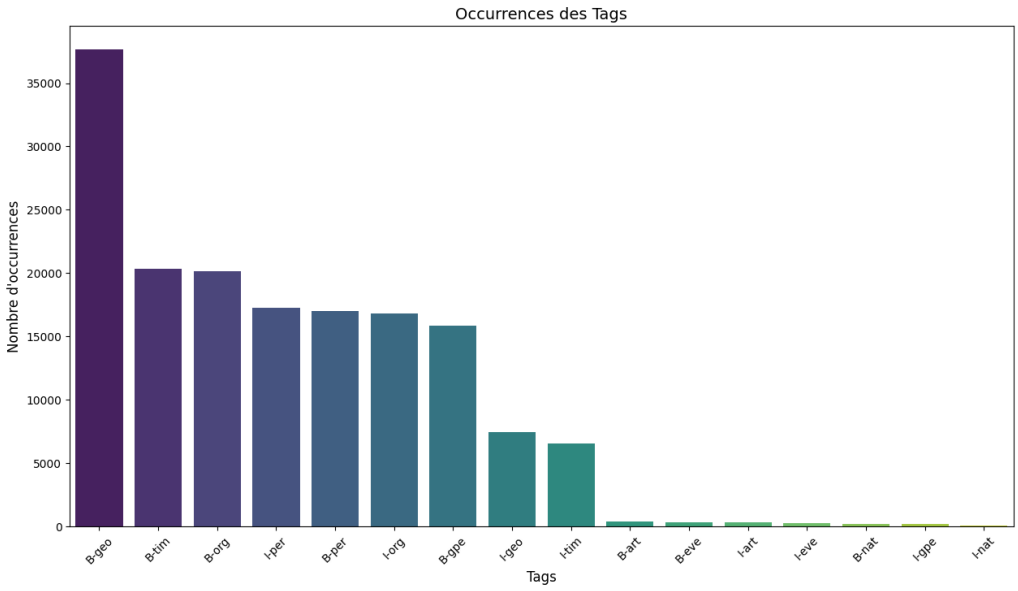

FIGURE 4 – Occurrences des Tags du dataSet sans le Tag 'O'

Le nombre précis de chaque Tag dans notre dataset :

Tag	O	B-geo	B-tim	B-org	I-per	B-per	I-org	B-gpe
Count	37644	20333	20143	17251	16990	16784	15870	7414

Tag	I-geo	I-tim	B-art	B-eve	I-art	I-eve	B-nat	I-gpe	I-nat
Count	887908	6528	402	308	297	253	201	198	51

Des statistiques plus précises :

	Nombre de Tags différents par ligne	**Nombre de mots par ligne**
Maximum	11	104
Minimum	1	1
Moyen	3,39	22
Médian	3	21
Total	17 Tags différents	x

3.1.2 Prétraitement des données

Afin de transformer ces données textuelles en données numériques exploitables par des modèles d'intelligence artificielle, nous devons dès le début réaliser certains prétraitements.

One-Hot Encoding

Le premier prétraitement nécessaire pour nos modèles est le One-Hot Encoding. Il transforme un vecteur de taille N en une matrice de taille N x (nombre de Tags présents dans le dataset).

Reprenons l'exemple précédent : *"British police arrested Ramda in 1995."* qui donnait *['B-gpe', 'O', 'O', 'B-per', 'O', 'B-tim', 'O']*

Sachant que nous avons associé chacun des 17 Tags à un nombre et que 'O' a l'indice 0, 'B-gpe' a l'indice 3, 'B-per' a l'indice 5 et 'B-time' a l'indice 15, notre vecteur devient : [3 , 0 , 0 , 5 , 0 , 15 , 0]. Ici, N = 7.

Le One-Hot Encoding crée une nouvelle ligne pour chaque valeur de cette liste de nombres, soit 7 lignes, et autant de colonnes qu'il y a de Tags, soit 17 colonnes. Puis, pour chaque ligne, il met un 1 dans la colonne qui correspond à la valeur, et 0 ailleurs :

[
[0, 0, 0, **1**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[**1**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[**1**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, **1**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[**1**, 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, **1**, 0],
[**1**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]

Dans notre cas, nous choisissons de mettre en pratique cette technique pour transformer nos vecteurs de données comportant des valeurs croissantes en données numériques binaires, préférées par les réseaux de neurones. En effet, ces algorithmes font des calculs de distances avec les valeurs numériques, ce qui fait qu'ils donneront aux indices un poids et un ordre qui n'existent pas réellement. Cette conversion permet d'effacer cette interprétation abusive par le modèle.

Tokenisation

La tokenisation est un processus qui consiste à découper la phrase en mots ou sous-mots, appelés tokens, et à les associer à un nombre. Ceci est nécessaire car les algorithmes de Machine Learning ne sont pas conçus pour comprendre le langage naturel humain, mais seulement les nombres. Par ailleurs, tous les tokenizers ne découpent pas une phrase de la même manière.

En reprenant notre exemple, avec une tokenisation par mot (en utilisant la couche Tokenizer de Keras [3]), la phrase *"British police arrested Ramda in 1995."* deviendra, après tokenisation, *['British', 'police', 'arrested', 'Ramda', 'in', '1995', '.']*, puis, après encodage, *[15, 851, 4, 59, 87, 54, 568]*.

Padding

Pour qu'un réseau de neurones fonctionne correctement, toutes les entrées doivent être de la même taille. Nous devons donc forcer une taille unique à toutes nos phrases, en complétant toutes celles trop courtes avec un Tag neutre. Cette complétion s'appelle un padding et la taille unique est un paramètre nommé max_length.

Comme nous avons pu le voir lors de l'analyse de nos données (voir section 3.1.1), la moyenne et la médiane de la longueur des phrases sont très proches, ce qui démontre que nous n'avons pas de valeurs aberrantes, comme une dizaine de phrases à un mot et une seule à 1000 mots. De plus les tailles de nos phrases se situent dans un intervalle assez petit, donc nous prenons la décision d'initialiser max_length à celle de la plus grande taille dans notre dataset.

Par exemple, pour un max_length égal à 10, *"British police arrested Ramda in 1995."* qui avait les Tags suivants : *['B-gpe', 'O', 'O', 'B-per', 'O', 'B-tim', 'O']*
aura maintenant les Tags : *['B-gpe', 'O', 'O', 'B-per', 'O', 'B-tim', 'O','O','O','O']*.

3. `https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer` (dernière consultation : 22/04/2025)

3.2 Un premier modèle naïf

Nous avons commencé par réaliser un premier prototype très simple, sans technologie permettant de lier les mots entre eux et une partition simple de nos données par train_test_split (voir section 2.1). Nous avons également réduit la taille du dataset à 1000 phrases pour que le temps d'exécution soit plus court.

Après 10 epochs (voir section 2.1 pour une définition), voici ce que nous obtenons :

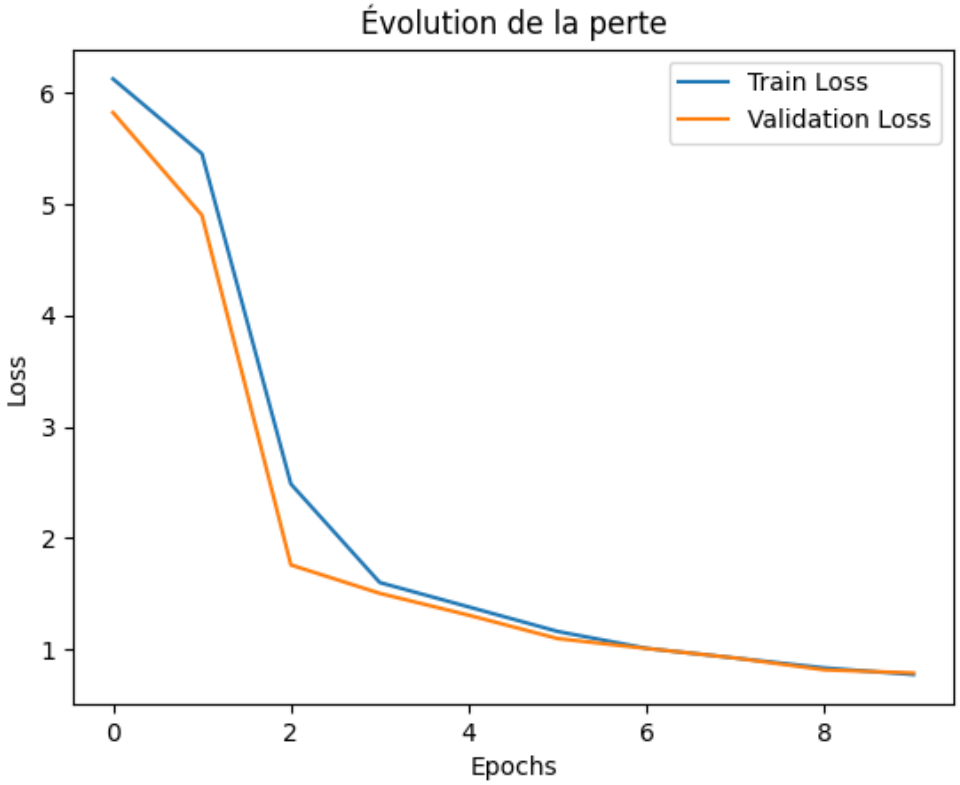

FIGURE 5 – Évolution de la perte (Loss) pour notre premier modèle, sur 10 epoch
Score d'accuracy : 0.9259

La courbe bleue correspond à la "Train Loss", ou autrement dit la courbe de perte d'apprentissage. Elle quantifie la marge d'erreur entre les prédictions du modèle et les données réelles sur chaque epoch sur les données d'entraînement. La courbe orange correspond à la "Validation Loss", ou autrement dit la courbe de perte de validation. Elle est calculée de la même manière, mais cette fois sur les données de test. En résumé, la courbe bleue montre le taux d'erreurs du modèle pendant qu'il apprend les données, et la courbe orange montre le taux d'erreurs du modèle après avoir appris, lorsqu'il est évalué sur des données qu'il n'a pas apprises.

Les courbes de la Train Loss et celle de la Validation Loss sont très proches, ce qui est positif. Cependant, elles restent toutes les deux au-dessus de 1, ce qui est beaucoup trop élevé. Notre modèle ne comprend pas assez bien nos données.

Nos principaux problèmes

Nous constatons que nous n'avons pas assez de données dans notre dataset, ce qui explique la valeur élevée de la Loss : notre modèle n'a simplement pas assez de données pour prédire assez précisément. De plus, comme observé plus haut, nous n'avons aucune technologie permettant de lier les mots entre eux, ils sont donc chacun pris indépendamment, sans le contexte de la phrase ni informations sémantiques.

3.3 Un modèle LSTM simple

Pour pallier à ces problèmes, nous choisissons tout d'abord d'augmenter le volume de nos données de 1 000 à 47 959 phrases, en prenant simplement le dataset récupéré en entier.

Ensuite, nous remplaçons notre simple train_test_split par un K-Fold de 8 (voir section 2.1 pour une définition) pour une cross-validation, rendant nos résultats plus fiables. Nous souhaitions ajouter du K-Fold stratified comme nos classes sont déséquilibrées. Cependant, nous nous sommes heurtées à la dimension du y (les Tags) qui était trop importante.

Enfin, nous ajoutons une couche de LSTM et une couche d'Embeddings (appris pendant l'entraînement de notre modèle), afin d'ajouter une compréhension du contexte et de la sémantique.

Les LSTM

Les LSTM[1] (Long Short-Term Memory) sont une forme avancée de réseaux de neurones récurrents (RNN)[4], spécialement conçus pour gérer les données séquentielles, comme du texte et de l'audio. Les réseaux de neurones classiques ne retiennent pas d'informations sur ce qu'ils ont vu auparavant. Ceci est problématique lorsque nous traitons des séquences, où le contexte précédent influence le sens.
Par exemple : *"Le patient a déclaré ne pas avoir de fièvre."* Le mot *"pas"* change tout le sens de la phrase. Sans mémoire, le modèle pourrait mal interpréter.
Nous utilisons des LSTM pour améliorer les performances d'un modèle de NER. Grâce à leur capacité à prendre en compte le contexte d'un mot dans une phrase, les LSTM sont mieux adaptés que les modèles naïfs, qui traitent chaque mot indépendamment.

Les Embeddings

Quand nous travaillons avec du texte, les ordinateurs ne comprennent pas directement les mots. Ils ont besoin qu'on les transforme en nombres. Un Embedding est une façon de représenter un mot par un vecteur de nombres, de manière à capturer son sens ou ses usages dans la langue. Chaque mot est transformé en une liste de nombres (vecteur), souvent de taille 50, 100, 300, etc. Mais à la différence du One-Hot Encoding (où chaque mot est unique mais sans lien avec les autres), les Embeddings placent les mots dans un espace où les mots proches en sens sont proches en vecteurs. Par exemple : "docteur", "médecin" et "chirurgien" auront des vecteurs proches, "banane" sera éloigné de "urgence".

Les Embeddings donnent au modèle une compréhension du langage plus fine et la

capacité de généraliser sur des mots qu'il n'a jamais vus (s'il a vu des mots similaires). C'est une base pour des tâches comme la classification, la traduction automatique, ou la reconnaissance d'entités nommées.

Ces représentations vectorielles sont généralement apprises à partir de grands textes. Des algorithmes comme GloVe ou des modèles plus récents comme BERT analysent comment les mots apparaissent ensemble dans des phrases pour comprendre leur signification par contexte. Nous utilisons des Embeddings (notamment GloVe) pour transformer les mots de nos phrases en vecteurs compréhensibles par le modèle, avant de les envoyer dans un réseau de neurones. Cela améliore fortement la performance, car le modèle ne part pas de zéro.

Voici nos résultats :

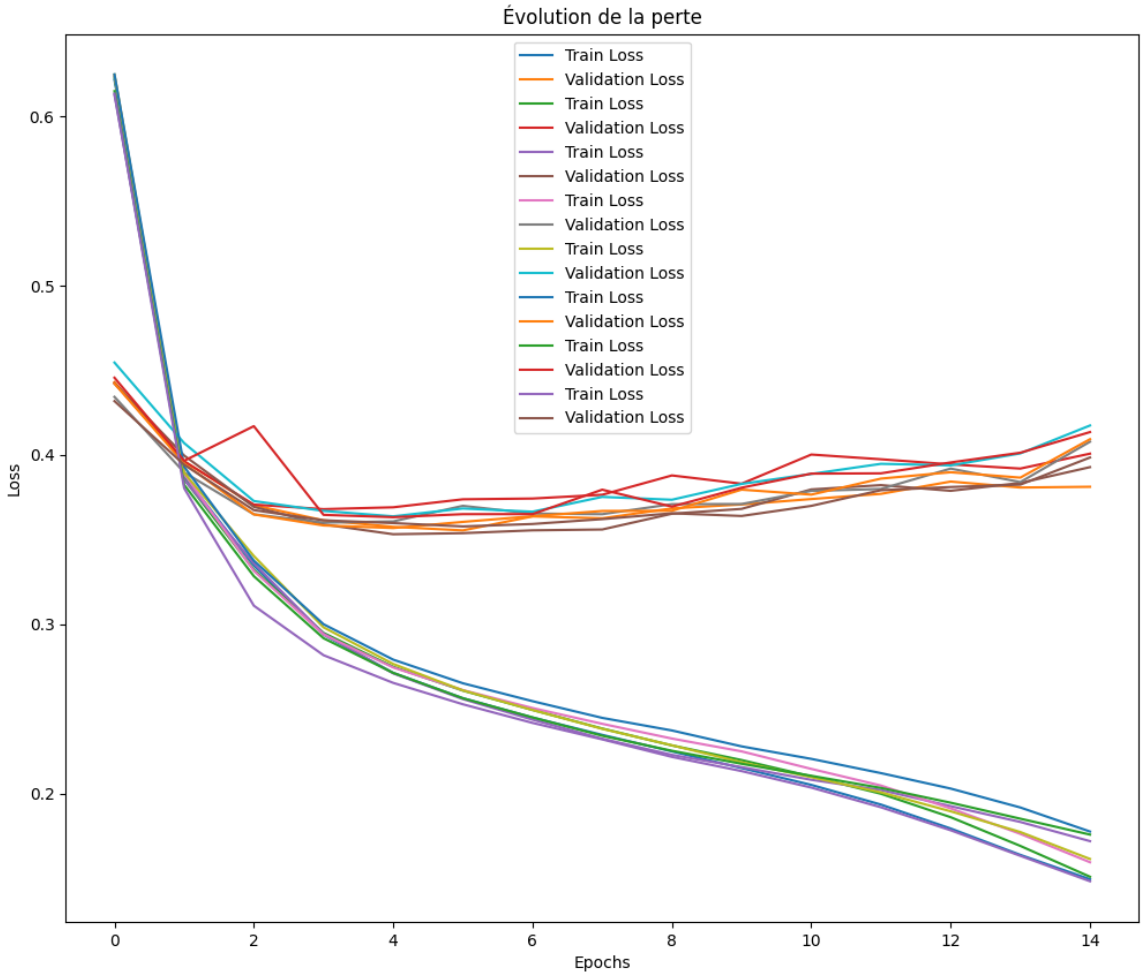

FIGURE 6 – Évolution de la perte pour notre modèle LSTM simple.
Score d'accuracy moyen : 0.9255 ± 0.0034

La figure 6 montre 8 couples de courbes Train Loss et Validation Loss, qui correspondent à nos 8 K-Folds. Nous remarquons un écart croissant entre les courbes de Validation, se situant en haut vers les 0,4 de Loss, et les courbes de Train, se situant en bas et étant décroissantes. Même si la Validation Loss à baissé de 1 à 0,4, nous remarquons que plus notre modèle est bon sur les données de Train, plus il est mauvais sur celles de

Test. Ceci est révélateur d'un sur-apprentissage : notre modèle apprend par coeur nos données de Train et les traite très bien. Il n'est donc pas généralisable à nos données de Test sur lesquelles il se trompe beaucoup. Le score d'accuracy de 0.9255 montre que notre modèle n'est pas très performant, mais l'écart-type de 0.0034 témoigne de sa stabilité.

Nos principaux problèmes

Notre priorité est de pallier au sur-apprentissage. Par ailleurs, certaines phrases vides, c'est-à-dire ne comportant que des Tags 'O', nous semblaient aussi poser problème : en effet, comme nous pouvons le remarquer dans la figure 3, il y a un énorme déséquilibre entre le Tag 'O' et tous les autres Tags. Ainsi, notre modèle attribue trop de Tags 'O', et reconnaît très mal les autres Tags. Enfin, nous nous sommes rendues compte que notre batch_size de 2 est beaucoup trop petit, puisqu'il ne donne que très peu de données à la fois à notre modèle, et ce à une haute fréquence. Autrement dit, le modèle doit très souvent modifier ses poids avec de nouvelles données, qui sont en trop petites quantités pour qu'il n'apprenne vraiment des poids pertinents.

3.4 Un modèle LTSM plus complexe

À la suite des résultats obtenus avec notre premier modèle LSTM, nous avons observé un clair sur-apprentissage : la perte (Loss) sur les données d'entraînement diminuait fortement tandis que celle sur les données de validation gravitait autour de 0,4 (voir figure 6). Ce comportement indique que notre modèle s'adapte trop bien aux données qu'il connaît, mais généralise mal les phrases jamais vues. De plus, il y a un large surnombre de Tags 'O' dans notre dataset ce qui amplifie le problème, en poussant le modèle à prédire majoritairement cette classe par défaut.

Pour remédier à cela, la première amélioration consiste à nettoyer notre jeu de données. Nous supprimons toutes les phrases dont plus de 80% des mots portaient le Tag 'O'. Cela nous permet de réduire le déséquilibre entre les classes.

Par la suite, afin de limiter le sur-apprentissage, nous appliquons un dropout de 20% sur la couche LSTM, c'est-à-dire qu'à chaque itération d'entraînement, 20% des neurones de cette couche sont temporairement désactivés, ce qui oblige le réseau à apprendre des représentations plus robustes, moins dépendantes de certains neurones. Cette technique est particulièrement efficace dans les modèles profonds ou récurrents, comme les LSTM.

Dans le modèle précédent, les Embeddings (représentations vectorielles des mots) étaient apprises durant l'entraînement, à partir de nos seules données. Nous préférons ici utiliser GloVe, un ensemble d'Embeddings pré entraînés sur Wikipédia, ce qui nous permet de partir avec une compréhension plus complète du vocabulaire.

Nous avons ajusté également la taille du batch à 32. Le batch_size contrôle la quantité d'exemples utilisés à chaque étape d'apprentissage. Un bon réglage permet de rendre l'apprentissage plus efficace, plus stable, et moins sujet au sur-apprentissage. Ce changement permet d'avoir un meilleur compromis entre la vitesse d'entraînement et un apprentissage plus général, car les poids sont mis à jour sur des données plus représentatives, et évite les instabilités observées avec un batch_size de 2.

Enfin, nous modifions notre tokenizer pour qu'il corresponde à celui utilisé lors de

la création du dataset (via la couche TextVectorization de Keras [4]). Cette modification fait suite à certaines observations que nous avons pu faire, où certains Tags semblaient décalés par rapport aux mots qu'ils devaient représenter, ce qui faussait totalement les résultats. Après investigation, nous avons conclus que la façon dont les phrases ont été découpées dans le dataset et la façon dont nous le faisions ne correspondaient pas. Le changement de tokenizer à finalement été la solution.

Notre modèle LSTM complexe comporte de nombreux hyperparamètres : taille des couches, taux d'apprentissage, dropout, batch_size, etc. Les ajuster manuellement aurait été long et inefficace. Nous avons donc utilisé Optuna[7], une bibliothèque open-source d'optimisation automatique d'hyperparamètres, pour trouver automatiquement une meilleure combinaison de valeurs de ces hyperparamètres. L'algorithme explore semi-aléatoirement l'espace des hyperparamètres et évalue chaque configuration en fonction de sa performance sur les données de validation.

Enfin, nous avons intégré deux fonctions de rappel (callbacks) essentielles à l'entraînement : l'EarlyStopping et le ModelCheckpoint. EarlyStopping arrête automatiquement l'apprentissage lorsque la performance sur l'ensemble de validation ne s'améliore plus après un certain nombre d'epochs, ce nombre définissant la patience. Cela évite le surapprentissage et réduit le temps de calcul. ModelCheckpoint enregistre automatiquement le modèle ayant donné la meilleure performance sur la validation. Cela garantit que nous conservions par la suite la version la plus performante du modèle, même si l'entraînement se dégrade ensuite.

Dans la figure 7 présentant nos résultats, nous pouvons constater que l'écart entre les deux courbes a nettement diminué, ce qui démontre que nous n'avons plus de surapprentissage. De plus la Loss est meilleure que le précédent modèle que ce soit pour l'apprentissage ou la validation. Par ailleurs, le score d'accuracy moyen a augmenté à 0.9440, ce qui est satisfaisant, et l'écart-type a diminué à 0.0011, montrant une meilleure stabilité. Ces résultats sont une nette amélioration par rapport au modèle précédent, et nous avons à présent un modèle performant.

4. `https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization`

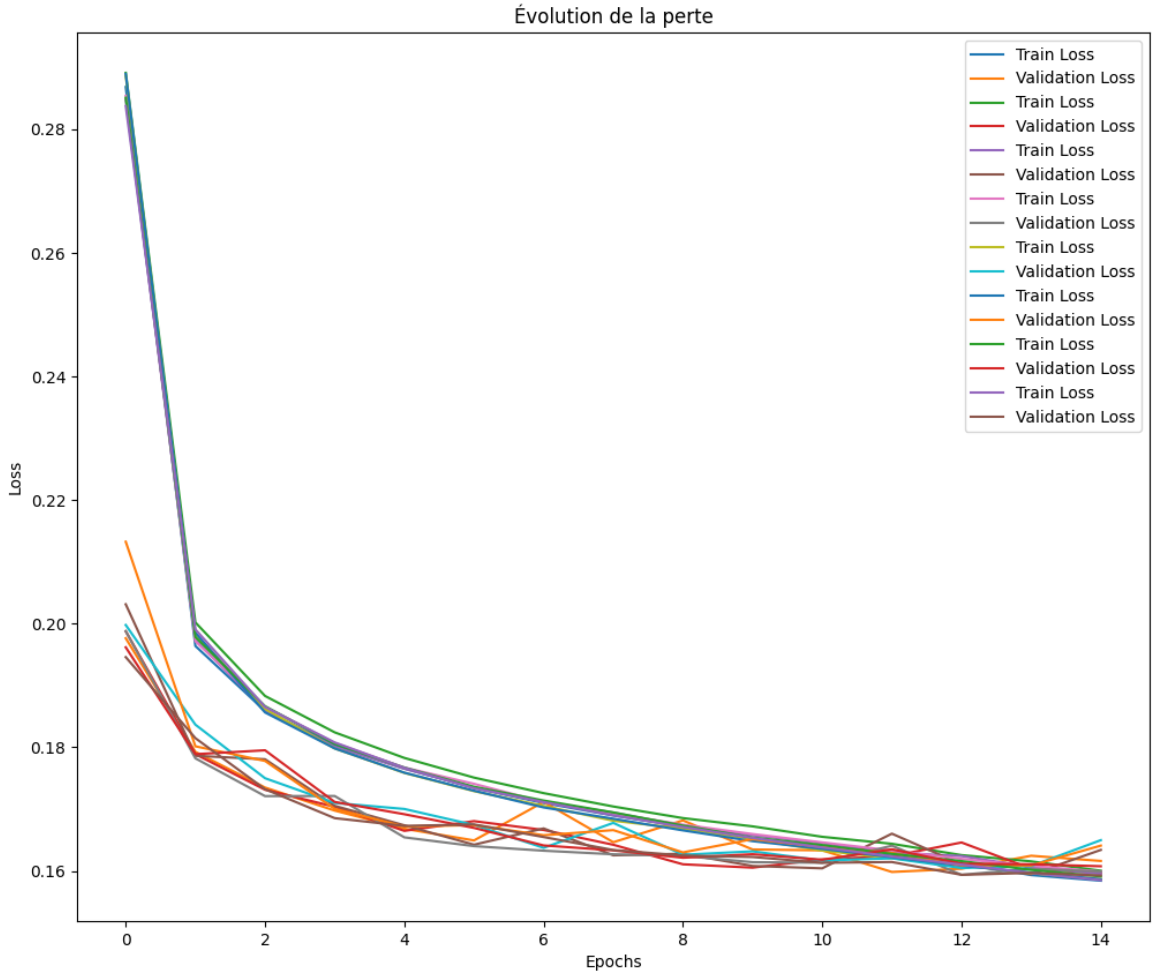

FIGURE 7 – Évolution de la perte pour notre modèle complexe
Score d'accuracy moyen : 0.9440 ± 0.0011

3.5 Un modèle avec mécanisme d'Attention

Notre dernier modèle réalisé (voir section 3.4) est efficace, mais nous faisons le choix de pousser une peu plus loin et d'élaborer un modèle avec un mécanisme d'Attention afin de permettre au modèle de mieux se concentrer sur les parties pertinentes des phrases.

L'Attention

Le mécanisme d'Attention (Attention Mechanism) permet au modèle de se focaliser sur les différentes parties d'une entrée. Contrairement à un LSTM classique qui compresse toute la séquence en un seul vecteur de contexte, l'attention permet d'associer un poids d'importance à chaque mot de la séquence. Cela améliore la capacité du modèle à gérer les longues phrases et à capturer le contexte pertinent pour chaque mot.

Dans notre modèle, nous remplaçons le LSTM unidirectionnel par un LSTM bidirectionnel (BiLSTM), afin de prendre en compte le contexte à la fois de gauche à droite et de droite à gauche dans la phrase. Nous appliquons ensuite une couche d'attention afin d'ajuster automatiquement l'importance de chaque mot selon son contexte.

Par ailleurs, pour permettre de bien visualiser l'évolution du modèle, nous avons augmenté le nombre d'epochs de 15 à 40. Cela a pour objectif de s'assurer que le modèle ait suffisamment de temps pour apprendre.

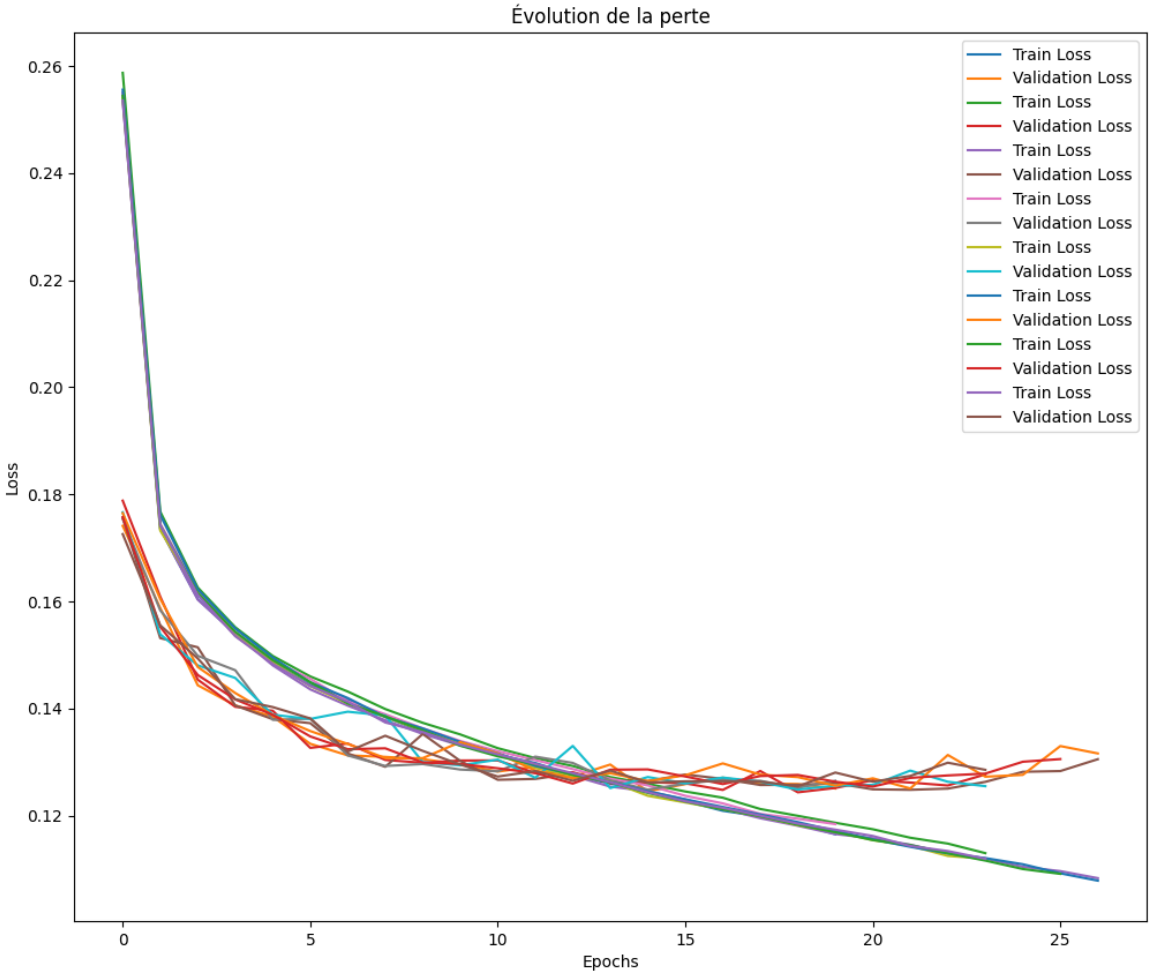

FIGURE 8 – Évolution de la perte pour notre modèle avec Attention
Score accuracy moyen : 0.9559 ± 0.00059

Le modèle avec Attention a obtenu un score d'accuracy moyen de 0.9559 ± 0.00059, ce qui est une amélioration des performances et de la stabilité par rapport au LSTM seul. La meilleure prise en charge du contexte permet donc comme prévu de mieux reconnaître la nature de chaque mot. Nous remarquons par ailleurs qu'en s'arrêtant vers les 15 epochs, nous ne tombons pas dans le sur-apprentissage et gardons des performances optimales.

4 Affinement "Fine-tunning" de modèle de Détection d'Entités Nommées

Le fine-tuning consiste à adapter un modèle de langage pré-entraîné à une tâche spécifique. Les modèles comme BERT ou CamemBERT ont initialement été entraînés sur des corpus très vastes, contenant des milliards de mots[9]. Cette phase d'entraînement leur permet d'acquérir une compréhension générale du langage. Le fine-tuning, ou ajustement fin, sert alors à continuer l'entraînement de ce modèle sur un jeu de données plus petit, mais spécifique à une tâche particulière (ici la Reconnaissance d'Entités Nommées). Ce processus permet de bénéficier à la fois de la puissance du modèle pré-entraîné et de son adaptation à un contexte ou domaine particulier.

4.1 Données

4.1.1 Transformer le son en texte

Les données fournies par le CHU sont des fichiers audios sous format .mp3 d'environ 1 minute et 30 secondes. Nous devons donc réaliser un convertisseur de l'audio vers le texte afin de pouvoir utiliser ces données pour une tâche de NER.

Pour cela, nous utilisons de nombreuses bibliothèques Python. La bibliothèque os est utile pour la gestion des fichiers et des répertoires, celle speech_recognition traduit nos fichiers .wav en texte, et la bibliothèque pydub transforme les fichiers MP3 en WAV. Des dépendances supplémentaires comme ffmpeg sont requises pour assurer la conversion audio.

En ce qui concerne le fonctionnement du script, nous parcourons le dossier contenant les fichiers audio. Puis pour chaque fichier nous commençons par parcourir le dossier contenant les fichiers audio. Nous convertissons les fichiers MP3 en WAV et nous supprimons le fichier MP3 après conversion. Nous utilisons ensuite la bibliothèque speech_recognition pour transformer l'audio en texte. Nous finissons par sauvegarder le texte dans un fichier ayant le même nom que l'audio, mais avec l'extension .txt.

4.1.2 Création et prétraitement des données

Malheureusement, les données fournies par le CHU sont inutilisables dans notre contexte : une bonne partie n'est pas anonymisée donc non transmise, elles contiennent beaucoup de bruit et très peu de contenu utile pour la tâche de NER. De plus, elles ne sont pas étiquetées avec les Tags nécessaires, processus très lent et laborieux que nous ne pouvons réaliser dans le cadre de ce projet.

Nous faisons donc le choix de créer des données, basées sur d'autres déjà existantes et adaptées à notre tâche, grâce à l'aide de l'IA générative. Naturellement, nous repassons sur chaque mot généré et corrigeons ceux qui le nécessitent.

Nous avons donc 162 phrases avec leurs Tags associés. La figure 9 montre la fréquence de chaque Tag dans notre jeu de données.

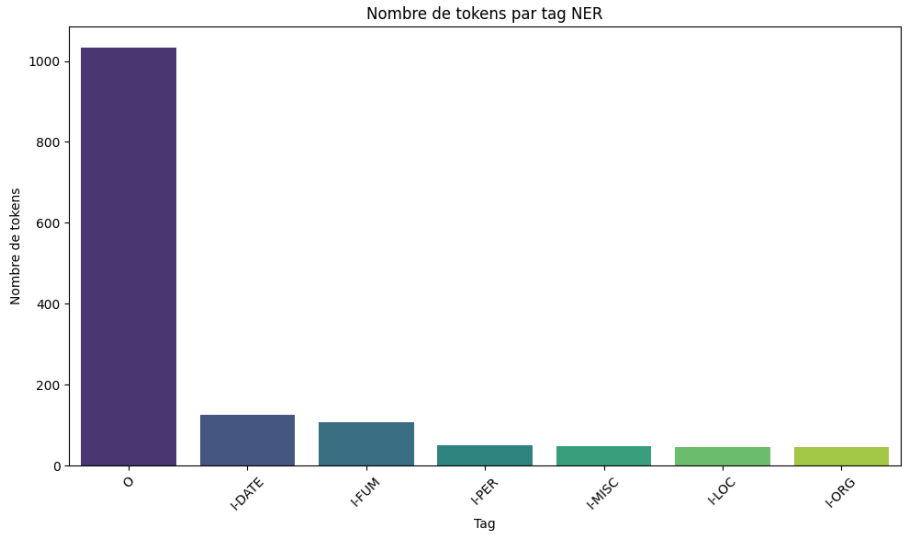

FIGURE 9 – Occurrences des Tags du dataSet

Afin de réaliser un fine-tuning efficace, un prétraitement des données est nécessaire. Dans notre cas, les données sont constituées de phrases, découpées en tokens, et annotées avec des étiquettes correspondant aux entités. Nous attribuons à chaque token un label de type O (pour "hors entité"), ou encore B-ORG, I-PER, etc., s'il fait partie d'une entité nommée. Le format des données suit la structure clasique utilisée en NER, avec deux colonnes principales : les tokens et les ner_tags. Par exemple, une phrase comme : "Elle a commencé à vapoter en 2020." pourra être associée aux étiquettes suivantes : [12, 12, 12, 12, 1, 12, 0, 12], où chaque entier correspond à un label d'entité. Pour plus de détails, se référer à la section 3.1.1.

Les données sont ensuite divisées en deux parties à l'aide d'un split entraînement/test, avec 20% des exemples alloués à l'évaluation. Cette étape permet de mesurer les performances du modèle de manière plus fiable.

L'étape suivante est la tokenisation. Puisque le modèle CamemBERT découpe les mots en sous-tokens (par exemple, le mot "université" peut être décomposé en "univer", "sité"), il est essentiel d'aligner correctement les étiquettes (ner_tags) avec ces sous-tokens. Seul le premier sous-token d'un mot reçoit l'étiquette correspondante, tandis que les autres reçoivent la valeur spéciale -100, ce qui permet au modèle d'ignorer ces positions dans le calcul de la fonction de perte. Cette étape permet de ne pas introduire de biais dans l'apprentissage.

4.2 Fine-tuning de CamemBERT

Le modèle utilisé pour cette tâche est camembert-base fourni par Hugging Face [5], une version francophone du modèle RoBERTa, entraînée sur de très grands corpus français. Nous utilisons l'architecture AutoModelForTokenClassification, qui permet d'ajouter automatiquement une couche de classification linéaire en sortie du modèle pour assigner à chaque token une étiquette parmi les suivantes : O, I-PER, I-ORG, I-LOC, I-MISC,

5. `https://huggingface.co/almanach/camembert-base` (dernière consultation : 19/05/2025)

I-DATE, et I-FUM.

Pour entraîner le modèle, nous utilisons la classe Trainer [6] de la bibliothèque transformers de Hugging Face. Ce composant intègre de nombreux éléments essentiels à l'entraînement, notamment l'optimiseur AdamW, un taux d'apprentissage ou encore des sauvegardes périodiques du modèle.

Nous avons également constaté un déséquilibre important entre les classes, comme on peut le voir sur la figure 9. La majorité des tokens sont annotés comme O, ce qui signifie qu'ils ne font pas partie d'une entité. Ce déséquilibre affaiblit l'apprentissage du modèle, qui peut tenter de prédire systématiquement O. Pour corriger ce biais, nous avons calculé des poids de classe à l'aide de la fonction compute_class_weight de scikit-learn [7], afin de pénaliser davantage les erreurs sur les classes rares. Nous avons également mit en place un système d'early stopping (EarlyStoppingCallback) pour éviter le sur-apprentissage.

Malgré ces ajustements, les performances du modèle sont restées limitées. Cela peut s'expliquer principalement par la taille réduite du dataset d'entraînement. Les tâches de reconnaissance d'entités nommées nécessitent généralement un grand volume de données annotées pour obtenir de bons résultats. Nous avons donc pu mettre en œuvre l'ensemble des étapes de fine-tuning pour une tâche de NER en français avec CamemBERT, en appliquant les bonnes pratiques standards du domaine. Mais, les résultats montrent que des améliorations sont possibles, notamment en augmentant la quantité et la diversité des données, et en appliquant des techniques d'augmentation automatique. Nous obtenons un score d'accuracy de 0.1598. Ce score ne comporte pas d'écart-type puisque la validation croisée n'est pas appliquée dans un contexte de Fine-tuning.

Afin d'améliorer les performances médiocres du modèle, nous introduisons une étape d'optimisation automatique des hyperparamètres en utilisant la bibliothèque Optuna. Avant cette optimisation, le score moyen du modèle reflétait notamment la difficulté du modèle à reconnaître correctement les entités nommées dans un contexte déséquilibré et avec un jeu de données de taille limitée.

Grâce à Optuna, nous pouvons ajuster automatiquement plusieurs paramètres cruciaux tels que le taux d'apprentissage (learning_rate), le batch_size ou le nombre d'époques (num_train_epochs). Après cette phase d'optimisation ainsi qu'un ajustement manuel et réfléchi de ces paramètres, le modèle atteint un score d'accuracy de 0.7018, soit une amélioration significative des performances. On voit ainsi l'importance de la phase d'optimisation des hyperparamètres dans les tâches de fine-tuning. Même avec un dataset de taille réduite, une configuration adaptée peut permettre au modèle d'exploiter plus efficacement les données disponibles.

Nous réalisons une nouvelle tentative d'amélioration des performances de notre modèle en augmentant la taille du dataset. Nous constatons une amélioration notable des performances, le score d'accuracy étant passé à 0.7222.

6. `https://huggingface.co/docs/transformers/en/main_classes/trainer` (dernière consultation : 19/05/2025)

7. `https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html` (dernière consultation : 19/05/2025)

4.3 Fine-tuning de CamemBERT-NER

Par volonté de partir du modèle le plus spécifique possible, nous choisissons de Fine-tune une version de CamemBERT déjà spécialisée dans la tâche de NER : CamemBERT-NER[2], issu du site Hugging Face également.

Ce modèle possède donc déja la capacité de reconnaître certains Tags : O, I-PER, I-ORG, I-LOC et I-MISC. Comme nous le remarquons, ce modèle diffère de ses prédécesseurs dans le fait qu'il ne possède que des Tags débutant par la lettre *I*. Notre tâche de Fine-tuning consiste ici à permettre à ce modèle de reconnaître 2 nouveaux Tags : I-DATE, visant les dates comme "1er janvier 2000", et I-FUM, visant le tabagisme de manière générale et permettant de reconnaître une question suivie d'une réponse sur les habitudes tabagiques d'une personne, comme par exemple "Fumez-vous ? Oui". Cet ajout doit se faire sans qu'il ne perde ses capacités sur les Tags qu'il connaît déjà.

Nous utilisons les mêmes données que dans le Fine-tuning précédent, les classes restent donc déséquilibrées par rapport au Tag neutre 'O', mais assez similaires autrement. De plus, les traitements que nous réalisons sur ce modèle sont très similaires à ceux que nous avons réalisé sur le Fine-tuning précédent.

En ce qui concerne le partitionnement des données en Train et Test et compte tenu de nos données, nous réalisons une fonction personnalisée de partitionnement : le principe est de, pour chaque Tag, assigner environ 80% des Tags à la partie Train et 20% à la partie Test. Les phrases comportant plusieurs Tags peuvent poser problème : nous choisissons de les assigner aléatoirement à un ensemble ou à l'autre. Nous arrivons à un partitionnement très satisfaisant comme montré au tableau 1 :

Table 1 – Répartition des tags entre les ensembles d'entraînement et de test

Tag	Train (%)	Test (%)
O	87.0	13.0
I-FUM	85.9	14.1
I-DATE	88.1	11.9
I-LOC	86.2	13.8
I-PER	77.3	22.7
I-ORG	86.4	13.6
I-MISC	78.3	21.7

Après avoir transformé les données pour qu'elles soient compatibles avec le modèle, initialisé ce dernier avec des hyperparamètres classiques, et choisi les mesures pour l'évaluer, nous obtenons des résultats extrêmement satisfaisants : une accuracy de 0.9948 et une Validation Loss de 0.1280. De plus, le score de F1 est de 0.9947, ce qui conforte les hautes performances de ce modèle.

Nous concluons qu'il est probable que le fait qu'une couche de classification de NER soit déjà présente et performante sur ce deuxième modèle nous permette d'avoir des très bons résultats avec si peu de données, contrairement au premier modèle.

5 Organisation

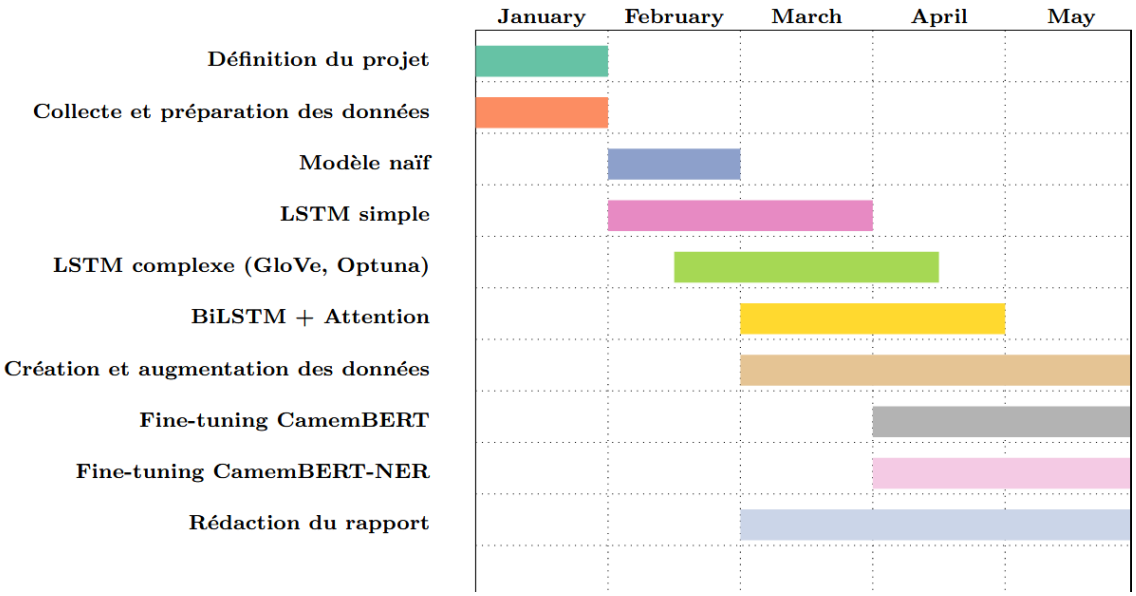

FIGURE 10 – Diagramme de Gantt détaillant la planification du projet

La figure 10 montre le diagramme de Gantt relatif à l'organisation de notre projet. Tous les membres du projet ont principalement participé à toutes les tâches présentes dans ce diagramme, car notre but était avant tout d'apprendre de nouvelles compétences et connaissances, et non la performance de nos modèles.

Nous avions des réunions hebdomadaires au minimum et avons fonctionné par sprint. Chaque réunion commençait par un récapitulatif du travail et des avancées réalisées par chacune. Nous décidions ensuite des prochaines étapes à réaliser en fonction de cela, les grandes étapes du projet ayant été décidées à l'avance. Nous avions également une réunion quasi hebdomadaire avec notre encadrant afin d'évaluer notre avancée et de nous conseiller.

6 Conclusion

Durant ce projet, nous avons développé et comparé plusieurs modèles de reconnaissance d'entités nommées (NER) dans le but de construire un modèle capable d'identifier, à partir d'un enregistrement audio, les mots et expressions pertinents pour un diagnostic médical.

Notre travail a débuté par la mise en œuvre d'un modèle naïf servant de référence de base. Nous avons ensuite développé un modèle LSTM simple, en explorant différentes méthodes de validation (train_test_split, validation croisée) et l'intégration d'Embeddings. Cette première phase a permis de mieux comprendre les processus d'apprentissage et de résoudre les principaux défis que nous pouvions rencontrer lors des tâches de NER.

Dans une seconde étape, nous avons construit un modèle LSTM plus complexe, avec une couche d'embeddings pré entraînés (GloVe) ainsi que des mécanismes d'optimisation et de callbacks pour un meilleur contrôle de l'apprentissage. Nous avons ensuite testé une architecture plus avancée combinant un BiLSTM avec un mécanisme d'Attention, permettant au modèle de mieux exploiter le contexte bidirectionnel et de se focaliser sur les éléments importants des phrases.

Enfin, nous avons ajusté finement deux modèles basés sur BERT : le modèle CamemBERT, pré entraîné sur un grand corpus francophone, et sa sur-couche spécialisée dans la tâche de NER, CamemBERT-NER. Le premier n'a malheureusement pas pu atteindre les meilleurs résultats en termes de précision comparé à son prédécesseur, principalement par manque de données. Le second en revanche a fourni d'excellents résultats sur les mêmes données, étant donnée sa spécialisation.

Nos modèles montrent une progression claire des performances à chaque étape du projet, montrant l'intérêt d'une approche itérative combinant la création de notre propre modèle et le fine-tuning d'un autre (ici CamemBERT). Ils montrent ainsi l'intérêt de l'utilisation de modèles pré entraînés, avec une amélioration des résultats grâce à l'augmentation du dataset et à des architectures plus complexes.

En dépit de ces avancées, nous avons rencontré certaines limitations. Notamment, nous n'avons pas pu tester nos modèles sur des enregistrements audio réels issus des services d'urgence, en raison du processus long et complexe d'anonymisation de ces données, nécessaire au respect des contraintes éthiques et légales en vigueur. Nous avons donc eu des limites imposées par la taille réduite du jeu de données médicales annotées que nous avons créé.

Références

[1] **A Review of Recurrent Neural Networks : LSTM Cells and Network Architectures.** neural comput 2019; 31 (7) : 1235–1270. https ://doi.org/10.1162/neco_a_01199.

[2] **camembert-ner : model fine-tuned from camemBERT for NER task** https ://huggingface.co/jean-baptiste/camembert-ner. (dernière consultation le 23/05/2025).

[3] **Deep learning with word embeddings improves biomedical named entity recognition** https ://academic.oup.com/bioinformatics/article/33/14/i37/3953940 ?login=false. (dernière consultation le 26/05/2025).

[4] **Les RNN** https ://www.ibm.com/fr-fr/think/topics/recurrent-neural-networks. (dernière consultation le 22/05/2025).

[5] **Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling** https ://static.googleusercontent.com/media/research.google.com/fr//pubs/archive/43905.pdf. (dernière consultation le 26/05/2025).

[6] **Qu'est-ce que le Machine Learning** https ://www.lecepe.fr/articles/qu-est-ce-que-le-machine-learning/. (dernière consultation le 23/05/2025).

[7] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. **Optuna : A Next-generation Hyperparameter Optimization Framework**. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[8] Johan Bos, Valerio Basile, Kilian Evang, Noortje Venhuizen, and Johannes Bjerva. **The Groningen Meaning Bank**.

[9] Yichu Zhou and Vivek Srikumar. **A Closer Look at How Fine-tuning Changes BERT.** https ://arxiv.org/abs/2106.14282. 2022.