

PROJET N°7
RAPPORT

Approfondissement des notions de NLP focus sur BERT

Étudiant :
Fayz EL RAZAZ

Enseignant :
Amine HADJ-YOUCER

13 octobre 2022

Table des matières

1	Introduction	2
2	État de l'art	2
2.1	Les transformers	2
2.1.1	Les couches d'attention	5
2.1.2	L'encodage et les encodeurs	5
2.1.3	Le décodage et les décodeurs	5
2.2	L'algorithme BERT et ses déclinaisons	6
3	Jeu de données & objectif	7
3.1	Stack Overflow	7
3.2	Objectifs	7
4	Modèle implémenté & performances	7
4.1	Modèle & performances	7
4.1.1	BERT - Première implémentation	8
4.1.2	BERT - Deuxième implémentation	8
4.1.3	BERT - Dernière implémentation	9
5	Conclusion	10

1 Introduction

Le traitement du langage naturel constitue un pan important de la science des données. Il s'agit de façon globale, de comprendre, manipuler et générer du langage "humain", indépendamment de la langue. Le traitement du langage naturel, aussi appelé NLP (pour natural language processing), constitue la part importante de ce qui est appelé généralement, l'informatique sémantique.

On y retrouve les problématiques de traduction automatique, de l'analyse de sentiment, les chatbots ou encore la classification de texte.

Dans le cadre du parcours Ingénieur machine learning d'Openclassrooms, nous avons été amené à travailler sur un sujet de NLP (Projet n°5) que nous avons souhaité approfondir ici. Pour ce faire, nous avons réalisé un état de l'art du domaine que nous exposerons dans ce document. Nous présenterons les méthodes, récemment mise en place dans différents laboratoire de recherche, et présenterons leur implémentation en reprenant le jeu de données sur lequel nous avons travaillé auparavant. Nous représenterons ce jeu de données après avoir exposé les résultats importants et avant de présenter la méthode baseline qui avait été retenu et les performances du nouveau modèle.

2 État de l'art

Le traitement du langage naturel est un domaine de l'apprentissage automatique se concentrant sur la compréhension de la langue humaine. Nous reprenons dans cette partie, les notions importantes qui ont été développées ces dernières années pour améliorer et mettre en place les algorithmes de traitement de NLP.

2.1 Les transformers

Une part importante des technologies développées autour de la NLP fonctionne grâce à la notion de transformers. Les transformers sont utilisés pour résoudre toute sorte de tâches de NLP.

L'architecture Transformer a été présentée en juin 2017 [9]. Le but initial au moment de sa présentation, portait sur la problématique de traduction. Elle a été suivie par l'introduction de plusieurs modèles influents qui se sont construit à partir de celle ci (Bert, GPT, DistilBERT, ou encore BART).

Tous les transformers mentionnés ci-dessus (GPT, BERT, BART, DistilBERT) ont été entraînés sur une large quantité de textes de façon autosupervisée. L'apprentissage autosupervisé est un type d'entraînement dans lequel l'objectif est automatiquement calculé à partir des entrées du modèle. Il n'est donc pas nécessaire d'étiqueter les données à la main.

Ce type de modèle développe une compréhension statistique de la langue sur laquelle il a été entraîné, mais n'est pas utilisable en l'état. Il est alors nécessaire d'effectuer un apprentissage par transfert. Le modèle est finetuné de manière supervisée (en utilisant des étiquettes annotées à la main) pour une tâche donnée.

Un exemple de tâche consiste classifier un texte donné en lui associant différents tags (cela peut consister en une émotion particulière, il s'agit alors de l'analyse de sentiment, ou alors d'une thématique particulière, et on est alors en train de faire du topic modeling).

Les transformers ont modifié le paysage de la recherche en NLP en améliorant de façon importante les résultats jusqu'alors obtenues.

Un inconvénient est cependant que les transformers sont des modèles très conséquent, et que pour améliorer leur performance, l'une des seules stratégies est d'augmenter encore la quantité de paramètres du modèle.

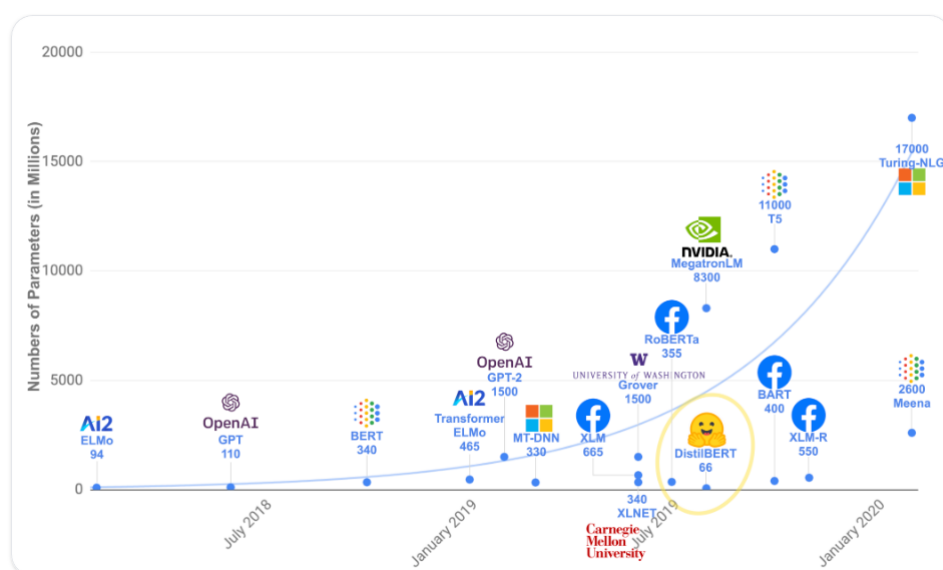


FIGURE 1 – Évolution du nombre de paramètres cf [1]

Le nombre important de paramètre implique une quantité de calcul très importante, ce qui a pour effet d'avoir un impact sur le plan écologique comme le montre le graphe suivant :

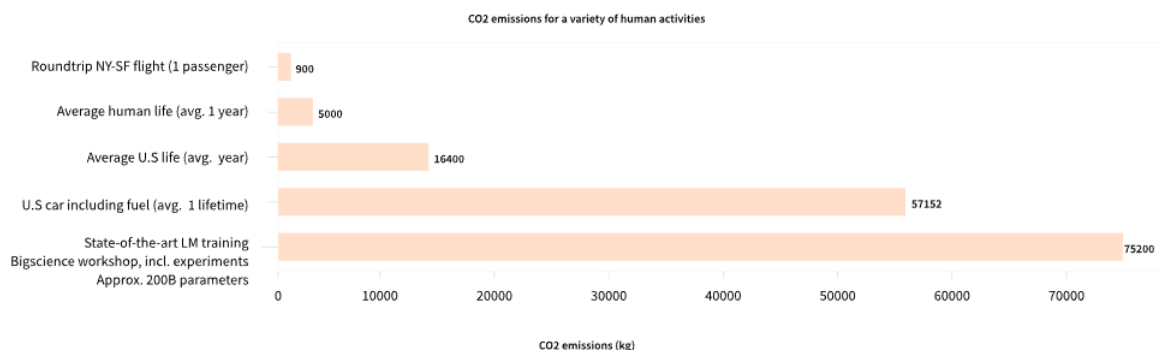


FIGURE 2 – Émission de CO2 de différentes activités humaines cf [1]

Bien que des efforts soient réalisés pour réduire l'impact écologique de l'entraînement de modèle de machine learning, on constate que une émission de CO2 importante. C'est pourquoi, les modèles sont pré-entraînés, puis partagé. La communauté peut alors utiliser cette base

de poids d'entraînement commun et l'adapter à une problématique propre tout en réduisant l'empreinte carbone.

Le pré-entraînement consiste à partir d'un modèle totalement vierge, et des poids aléatoire.

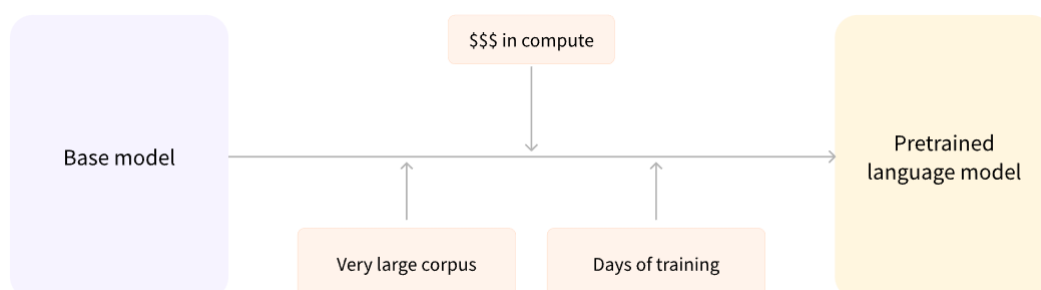


FIGURE 3 – Phase de pré-entraînement[1]

Une fois cette phrase terminée, on récupère ce jeu de donnée et on l'adapte à notre problème en faisant du finetuning.

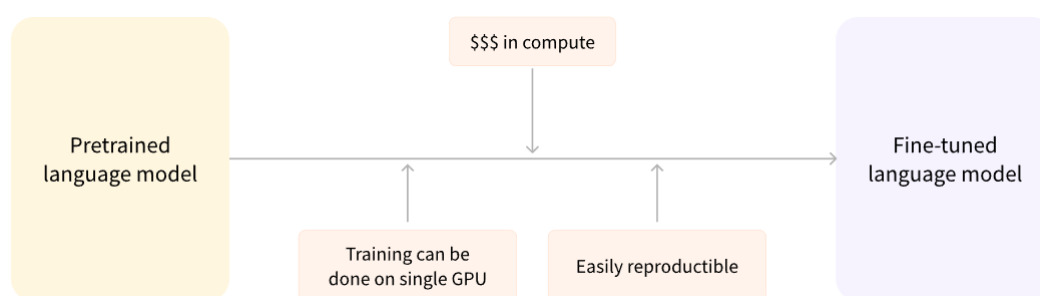


FIGURE 4 – Fine-tuning de modèle [1]

Il est donc primordial, tant du point de vue des performances que du point de vue écologique de partir d'un modèle pré-entraîné pour travailler sur une problématique en NLP.

L'architecture des transformers est composé de deux éléments importants : les encodeurs, et les décodeurs.

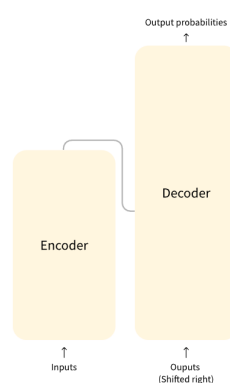


FIGURE 5 – Architecture d'un transformers [1]

Nous en donnons le détail dans les parties qui suivent. Nous nous arrêtons d'abord sur la notion de couche d'attention, primordiale pour cerner l'architecture du transformers.

2.1.1 Les couches d'attention

Les couches d'attention sont une caractéristique incontournable des transformers. Lors du traitement des mots (phase d'encodage que l'on verra à la partie suivante), la couche d'attention indique au modèle ce sur quoi il est le plus important de prêter attention. Le papier qui a introduit l'architecture transformers [9], s'intitule *Attention is all you need!*, faisant référence à ces couches d'attention. Ces couches d'attention sont ce qui permet au modèle de donner un sens précis aux mots, et permet donc un encodage efficace. En effet, rappelons que l'architecture transformers a été mise en place initialement pour des tâches de traduction. La traduction d'un même mot pouvant être différente d'une phrase à une autre (si on traduit l'expression au féminin ou au masculin par exemple, ou si l'on doit conjuguer un verbe qui s'écrit toujours de la même façon dans une langue mais pas dans l'autre), la traduction ne peut pas être réduite au simple remplacement de mot d'une langue à une autre. Il est nécessaire de considérer le contexte dans lequel se place la phrase. C'est pourquoi ont été développées ces couches d'attention, permettant de considérer de façon plus fine les problématiques de contexte, ou de règles grammaticales complexes.

On retrouve ces couches d'attentions dans les encodeurs comme dans les décodeurs.

2.1.2 L'encodage et les encodeurs

L'encodage L'encodage constitue la partie des transformers où l'on va transformer nos entrées (phrases ou textes, transformées en mots par la phase de tokenisation) en vecteur. Cependant, on retrouve dans cette partie, une couche d'attention qui considère l'ensemble des mots précédents de la phrase ou du texte à étudier, à chaque étape, à chaque nouveau mot. Un même mot peut alors avoir un encodage différent selon le contexte où on le retrouve.

Les encodeurs Les modèles basés sur l'encodeur utilisent l'encodeur d'un transformer et seulement celui-ci. À toutes les étapes, les couches d'attention accèdent à tous les mots de la phrase ou du texte donné. Ces modèles sont souvent caractérisés comme ayant une attention bidirectionnelle et sont souvent appelés modèles d'auto-encodage (comme BERT que l'on retrouvera ensuite).

Le pré-entraînement de ces modèles se fait généralement sur la modification d'une phrase donnée (en masquant des mots dans celle-ci) et en calibrant les poids du modèle en demandant de trouver ou de reconstruire la phrase initiale.

Ces modèles sont adaptés aux tâches qui requièrent une compréhension intégrale de la phrase, telles que la classification de phrases ou la reconnaissance d'entités nommées.

On retrouvera parmi les encodeurs, BERT, DistilBERT, ALBERT, ou encore le modèle ELECTRA.

2.1.3 Le décodage et les décodeurs

Le décodage Le décodage consiste à associer un mot à un vecteur donné. Comme pour l'encodeur, celui-ci dispose de couches d'attention. Cependant, il y a ici une spécificité. Lors de la phase de décodage, le décodeur a accès à l'ensemble de l'entrée de l'encodeur via les couches d'attention (et pas seulement les $n-1$ premiers mots de l'encodeur si on décode le n -ème mot),

et a aussi accès aux mots décodés avant le n -ième considéré, mais ne peut avoir accès aux suivants.

Les décodeurs Les modèles basés sur le décodeur utilisent uniquement le décodeur d'un transformer. À chaque étape, pour un mot donné, les couches d'attention n'accèdent qu'aux mots situés avant dans la phrase. Ces modèles sont souvent appelés modèles autorégressifs.

Le pré-entraînement des modèles basés sur le décodeur se concentre généralement sur la prédiction du prochain mot dans la phrase.

Ces modèles sont vraiment adaptés aux tâches qui impliquent la génération de texte.

Parmi les modèles célèbres ne s'appuyant que sur le décodage, on retrouvera CTRL, GPT ou TransformerXL.

2.2 L'algorithme BERT et ses déclinaisons

Le modèle BERT, pour *Bidirectional Encoder Representations from Transformers* [8] a été introduit en Mai 2019, et est le fruit d'un travail de chercheur de Google AI Language. Ce modèle, selon la version, a été entraîné avec 110 millions de paramètres ou 340 millions de paramètres [8].

Le modèle a été entraîné de façon différentes des modèles entraînés jusqu'alors. On l'a entraîné en masquant certains mots des coprus sur lesquels on l'a entraîné, en calibrant les poids des paramètres sur le fait de retrouver les bons mots.

De plus, comme indiqué dans son nom, le modèle est entraîné de façon bidirectionnelle.

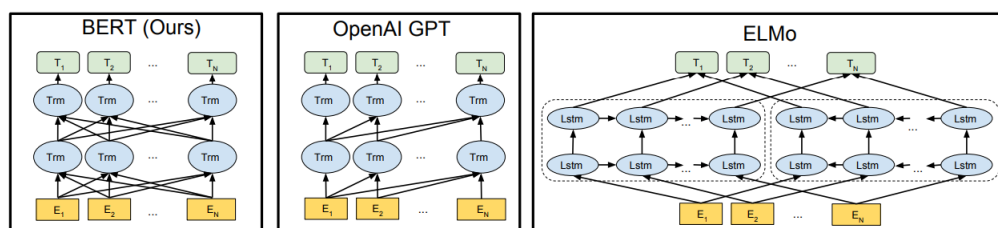


FIGURE 6 – Encodage bidirectionnel [8]

En effet, une part importante des modèles pour de la compréhension de texte permettant de l'analyse de sentiment ou de la classification de texte était entraîné dans le sens traditionnel de lecture, de gauche à droite. Or, ici, le modèle est entraîné dans les deux sens, et l'encodage s'effectue dans les deux sens, ce qui permet une compréhension des textes plus approfondies. Enfin, lors de la phase d'embedding, le modèle bert considère trois couches d'embeddings pour l'encodage des mots, la couche d'embedding lié au token, la couche d'embedding lié à la phrase, et enfin la couche lié à la position du token.

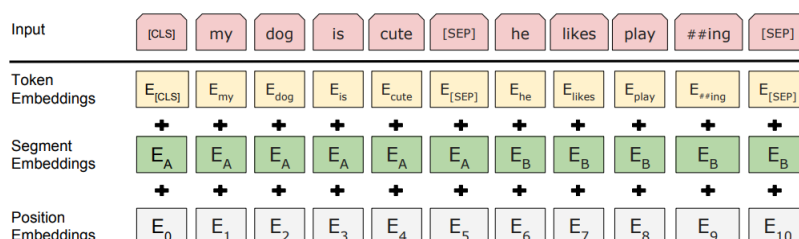


FIGURE 7 – Plongement d'un token avec BERT [8]

En pratique, on récupère un modèle pré-entraîné que l'on adapte à notre problématique, comme expliqué dans la partie précédente.

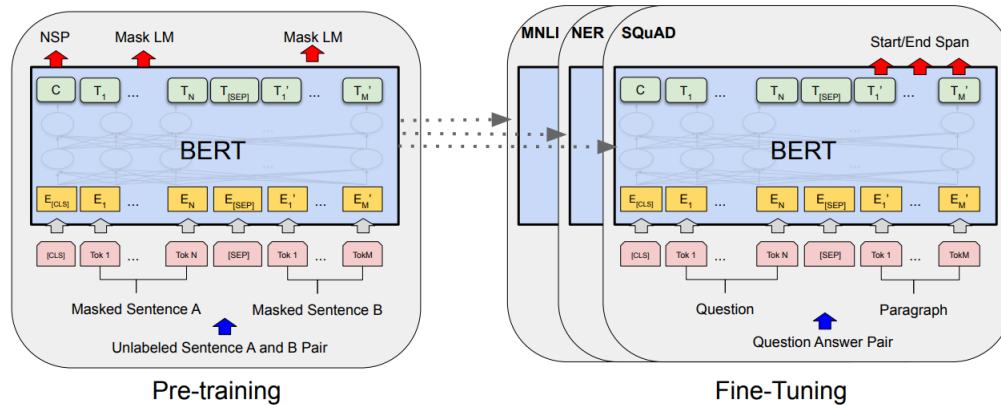


FIGURE 8 – Tuning de BERT [8]

Une fois BERT en place, une quantité importante d'algorithmes, basés sur la même technologie, ont été créés pour répondre aux problématiques des différentes langues, BERT ayant été entraîné en anglais.

Nous retrouverons par exemple CamemBERT ou FlauBERT pour la langue française.

3 Jeu de données & objectif

3.1 Stack Overflow

Pour illustrer la méthode étudiée, nous avons repris le jeu de données étudié lors du projet n°5, où l'on avait pour missions de catégoriser automatiquement les questions du célèbre site de développement informatique, stackoverflow.

3.2 Objectifs

Nous avons ici gardé l'objectif du projet n°5, suggérer des tags pertinents avec le sujet abordé par l'utilisateur lors du dépôt de sa question.

4 Modèle implémenté & performances

Ce projet a été une opportunité pour implémenter tout notre code en utilisant la bibliothèque PyTorch. Nous présentons ici, les résultats obtenus en utilisant BERT.

4.1 Modèle & performances

Nous nous sommes cantonnés dans ce projet à l'implémentation d'un modèle BERT pour répondre à notre problématique : proposer des tags à un utilisateur postant une question sur StackOverflow.

Nous avons implémentés 3 codes différentes présentant des différences, en terme de modèle comme en terme de paramètres.

4.1.1 BERT - Première implémentation

Notre première implémentation à été une reprise du projet initial. Nous avons utilisé la classe BertModel, de la bibliothèque *transformers* et BertTokenizer pour la tokenisation de nos textes. Nous avons effectué notre apprentissage sur les **200** tags les plus récurrents (comme pour le premier projet). Nous avons de plus, utilisé les hyperparamètres suivants :

- MAXLEN = 300 (on limite à 300 mots les textes étudiés, nous avons au delà eu des problématiques de mémoire GPU)

- TRAIN_BATCH_SIZE = 4

- VALID_BATCH_SIZE = 4

- EPOCHS = 2

- LEARNING_RATE = 1e-05 Nous n'avons pas tracé de courbes d'évolution de la fonction de perte durant l'apprentissage car celui-ci a été effectué sur deux époques.

Le temps d'apprentissage étant relativement long, nous avons calculé la valeur moyenne du F1 score (calculé sur chacun des vecteurs prédits et réels des tags) du modèle sur le jeu de test.

Nous avons obtenu un F1 score moyen de **0.12** pour ce modèle.

Voilà quelques sorties de test du modèle :

```

Texte test :
check string valid windows directory folder path trying determine whether string input user valid representing p
ath folder valid mean formatted properly. application folder represents installation destination provided folder
path valid want determine folder exists create not. currently using code io.directory.exists string path code fi
nd works fine except user format string properly when happens method return false indicates folder exist but pro
blem n't able create folder afterwards. from googling found suggestion use regular expression check format prope
r experience regular expressions wondering viable approach here found pre code regex new regex ^\\ return r.isma
tch path code pre would regular expression test combination code directory.exists code give good enough method c
heck path valid whether exists know vary factors program targeted strong windows strong users only.
Tags réels
['c#', 'windows', 'validation']
Tags Prédit :
['c++', 'java', '.net', 'c#']

```

FIGURE 9 – Texte test n°1

```

Texte test :
javascript jquery mean new javascript jquery 've learning make functions lot functions cropped brackets let show
mean pre code .click function something code pre always appears function n't even use value often
Tags réels
['javascript', 'jquery', 'function']
Tags Prédit :
['css', 'html', 'jquery', 'javascript']


```

FIGURE 10 – Texte test n°2

4.1.2 BERT - Deuxième implémentation

Pour cette deuxième implémentation, nous avons utilisé la classe *AutoModelForSequence-Classification* de la bibliothèque *transformers*. Cette classe, qui implémente un modèle BERT, adapté aux problématiques de classification, s'est avéré plus efficace en terme de durée d'apprentissage et de prédictions de tags.

Nous avons comme pour le premier modèle, donné 200 tags pour l'apprentissage. Nous avons obtenus les métriques suivantes :

 [16380/16380 1:40:30, Epoch 10/10]

Epoch	Training Loss	Validation Loss	F1	Accuracy
1	0.058700	0.058443	0.987984	0.987984
2	0.056600	0.053911	0.987984	0.987984
3	0.048500	0.045682	0.989527	0.989527
4	0.041400	0.041008	0.989686	0.989686
5	0.037500	0.038274	0.990061	0.990061
6	0.034900	0.036295	0.990520	0.990520
7	0.032800	0.034998	0.990574	0.990574
8	0.030700	0.034176	0.990696	0.990696
9	0.030000	0.033643	0.990853	0.990853
10	0.029900	0.033464	0.990867	0.990867

FIGURE 11 – Performances du deuxième modèle

Nous avons ici obtenu des résultats très performantes avec des prédictions juste dans un nombre très importants de cas. La seule limitation du modèle était celle de la quantité de mot à donner au texte qui sature rapidement la mémoire (nous avons coupé les textes étudiés à 256 mots). Il a été intéressant de remarquer, que lorsque les tags n'étaient pas ceux donné par un utilisateur, ils n'étaient néanmoins pas incohérent avec le texte donné. Avec un second GPU, doté d'une plus grande quantité de mémoire, nous pourrions donner ce même modèle en augmentant la taille des textes à étudier, et on pourrait s'attendre à une amélioration des performances.

4.1.3 BERT - Dernière implémentation

Pour ce dernier modèle, nous avons repris la dernière architecture, en modifiant le nombre de tags dans l'apprentissage. En effet, lors du premier travail sur ce projet, nous avons noté qu'en réduisant la quantité de tags pour l'apprentissage, le modèle était en mesure d'améliorer ses performances, ce qui s'explique notamment par la moindre quantité de variables à prédire.

Nous avons alors obtenus les résultats suivants :

Epoch	Training Loss	Validation Loss	F1	Accuracy
1	0.086800	0.084122	0.980559	0.980559
2	0.068900	0.064037	0.983532	0.983532
3	0.056700	0.054515	0.984600	0.984600
4	0.047900	0.049649	0.985281	0.985281
5	0.042700	0.046157	0.985910	0.985910
6	0.038900	0.044185	0.986386	0.986386
7	0.035400	0.042519	0.986755	0.986755
8	0.033000	0.041603	0.987167	0.987167
9	0.031600	0.041138	0.987179	0.987179
10	0.030300	0.040985	0.987253	0.987253

FIGURE 12 – Performances du dernier modèle

Comme pour le modèle précédent, le modèle est performant, mais légèrement moins perforant que le précédent. Le modèle a donc mieux appris avec une quantité plus importante de tags.

On pourrait prolonger ce travail en généralisant notre code et en recherchant la valeur de tags optimales pour les meilleurs prédictions.

5 Conclusion

Nous avons effectué un état de l'art du NLP. Nous nous sommes familiarisé avec la notion de transformers, incontournable dans l'ensemble des problématiques de NLP contemporaines. Nous avons implémentés le model BERT, en utilisant différentes classe de la bibliothèque *transformers*, et obtenus des résultats satisfaisant et meilleurs que lors du premier projet. Nous avons donc pu mener à bien notre POC, et dans un cadre industriel, nous aurions pu poursuivre le projet et passer en production ce nouveau modèle qui performe mieux.

Une poursuite de ce travail est à mener en utilisant les variants d'algorithme BERT qui sont récemment sorties (nous pensons notamment à roBERTa, ou DistilBERT).

Références

- [1] In : (). URL : <https%20://huggingface.co/course/fr/chapter1/4%20?fw=pt>.
- [2] In : (). URL : <https://www.thepythoncode.com/code/finetuning-bert-using-huggingface-transformers-python>.
- [3] In : (). URL : https://www.youtube.com/watch?v=u--UVvH-LIQ&t=921s&ab_channel=HuggingFace.
- [4] In : (). URL : https://huggingface.co/docs/transformers/v4.23.1/en/main_classes/trainer#transformers.Trainer.
- [5] In : (). URL : https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertForSequenceClassification.
- [6] In : (). URL : <https://huggingface.co/docs/transformers/training>.
- [7] In : (). URL : [https://colab.research.google.com/github/NielsRogge/Transformers-Tutorials/blob/master/BERT/Fine_tuning_BERT_\(and_friends\)_for_multi_label_text_classification.ipynb#scrollTo=D0McCtJ8HRJY](https://colab.research.google.com/github/NielsRogge/Transformers-Tutorials/blob/master/BERT/Fine_tuning_BERT_(and_friends)_for_multi_label_text_classification.ipynb#scrollTo=D0McCtJ8HRJY).
- [8] Jacob DEVLIN et al. *BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2018. DOI : 10 . 48550 / ARXIV . 1810 . 04805. URL : <https://arxiv.org/abs/1810.04805>.
- [9] Ashish VASWANI et al. "Attention Is All You Need". In : (2017). DOI : 10 . 48550 / ARXIV . 1706 . 03762. URL : <https://arxiv.org/abs/1706.03762>.