

# Classification de textes — Analyse de sentiment (MI201 Projet 3)

1<sup>er</sup> Carlos Adrian Meneses Gamboa  
*Programme Ingénieur en STIC*  
ENSTA Paris  
Paris, France  
carlos.meneses@ensta-paris.fr

2<sup>e</sup> Jose Daniel Chacon Gomez  
*Programme Ingénieur en STIC*  
ENSTA Paris  
Paris, France  
jose-daniel.chacon@ensta-paris.fr

3<sup>e</sup> Santiago Florido Gomez  
*Programme Ingénieur en STIC*  
ENSTA Paris  
Paris, France  
santiago.florido@ensta-paris.fr

**Résumé**—Ce travail présente un système d'analyse de sentiment pour de courts textes en anglais et compare des méthodes classiques d'apprentissage automatique à une approche basée sur un transformeur. En utilisant des représentations standard (sac de mots et TF-IDF), plusieurs classificateurs sont entraînés et évalués, puis comparés à un modèle exploitant des embeddings BERT. Les résultats sont rapportés via l'accuracy et le Macro-F1, en mettant en évidence des différences de performance, de robustesse et de coût computationnel. L'étude fournit des repères pratiques pour choisir une chaîne de classification de sentiment adaptée à des contraintes de ressources typiques.

**Index Terms**—analyse de sentiment, TAL, classification de textes, TF-IDF, BERT

## I. INTRODUCTION

L'analyse de sentiment de textes courts devient fondamentalement importante lorsque les perceptions sont considérées comme un actif informationnel critique pour les propriétaires de produits et de services [1]. Ceci est particulièrement pertinent dans le développement de systèmes guidés par les émotions, qui peuvent fournir des informations significatives afin d'améliorer l'expérience utilisateur ou client. Par exemple, ces informations peuvent conduire à des ajustements des stratégies de support client ou à des campagnes marketing plus ciblées [2]. En tant qu'entrée conceptuelle pour de telles améliorations, des systèmes de recherche ou des approches d'analyse de sentiment peuvent être adaptés afin de se concentrer sur les émotions exprimées par la population cible. Dans ce contexte, les réseaux sociaux—and plus spécifiquement des messages courts tels que des tweets et des commentaires sur des plateformes multimédias—figurent parmi les sources les plus couramment utilisées pour conduire ce type d'analyse.

Ce projet se concentre sur l'analyse automatique de sentiment de textes courts en anglais. Tout d'abord, une phase exploratoire est menée au cours de laquelle le contenu du jeu de données est prétraité, et une analyse préliminaire de l'information est réalisée à l'aide de méthodes traditionnelles d'apprentissage automatique. Ensuite, l'étape de classification est effectuée avec des classificateurs standards tels que Naive Bayes, la Régression Logistique et le SVM linéaire, en utilisant plusieurs schémas de représentation du texte, incluant bag-of-words, TF-IDF au niveau des mots et TF-IDF au niveau des caractères. La performance des modèles est

rapportée à l'aide de l'accuracy, du macro-F1 et de métriques complémentaires afin d'assurer une comparaison équitable.

Ensuite, un perceptron multicouche (MLP) entraîné sur du texte vectorisé est évalué, et une alternative basée sur des embeddings BERT est étudiée afin de capturer la sémantique contextuelle. À cette fin, la performance des modèles MLP construits pour chaque approche de vectorisation est comparée à travers quatre architectures de réseau, chacune adaptée à la quantité d'information fournie par le vectoriseur correspondant ou par BERT, et orientée vers une classification finale à trois classes. De plus, une profondeur appropriée est définie en fonction du niveau de détail dans la représentation d'entrée afin de réduire le surapprentissage sur les données d'entraînement. Des couches de dropout sont également incorporées entre les couches cachées afin de contrôler davantage le surapprentissage et le surentraînement.

Enfin, afin d'améliorer la classification des messages, des stratégies basées sur des modèles de langage de grande taille (LLMs) ont été évaluées en utilisant la version API du modèle Gemma 3-4b-it (Gemini) pour comparer sa performance en tant que classifieur de textes courts aux modèles précédemment entraînés. De plus, LoRA a été utilisé pour réaliser un fine-tuning efficace de transformers basés sur BERT [3].

## II. Q0-ANALYSE DU JEU DE DONNÉES AVEC DIFFÉRENTS MODÈLES CLASSIQUES D'APPRENTISSAGE AUTOMATIQUE

Le jeu de données Sentiment Data Analysis auquel l'accès est disponible est composé d'ensembles de tweets courts en anglais qui sont classés en trois catégories selon leur polarité comme positifs, neutres ou négatifs dans la colonne "sentiment". En outre, il existe des informations liées aux métadonnées des tweets en termes du moment de la journée où ils ont été publiés, de l'âge de l'utilisateur et de son pays d'origine dans les colonnes "Age of User" et "country".

Il est initialement proposé d'effectuer un prétraitement du champ texte, en commençant par la suppression des valeurs nulles puis, en utilisant NLTK, en supprimant les stopwords, ce qui permet d'obtenir la colonne de texte traité sans des mots très fréquents en anglais qui n'ont pas une contribution sémantique significative [4].

## A. Analyse exploratoire des données (EDA)

Une analyse de la distribution des classes dans les données d'entraînement est effectuée. Comme le montre la Fig. 1, bien que la présence de données de polarité neutre dépasse les autres classes, il n'existe pas de déséquilibre substantiel dans le jeu de données d'entraînement qui pourrait être associé à un biais dans les classificateurs à développer.

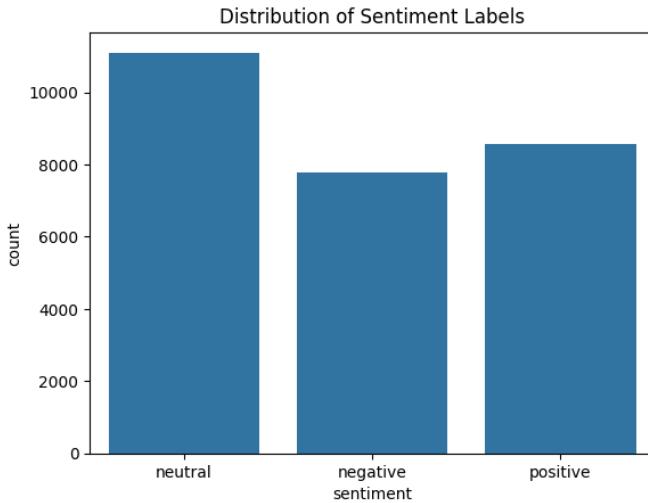


FIGURE 1 – Distribution des sentiments dans le jeu de données d'entraînement.

Il est utile, avant de mettre en œuvre certaines stratégies d'apprentissage automatique, d'examiner le comportement de groupes de termes (bag-of-words) vis-à-vis des classes de polarité, principalement d'un point de vue basé sur la fréquence.

Cela met en évidence l'importance du concept de vecteuriseur, qui est responsable de la conversion de collections de textes—telles que le contenu de la colonne “processed text”—en vecteurs numériques. Cela implique l'obtention d'une représentation creuse de chaque document après application du vectoriseur. Ce processus est précisément connu sous le nom de vectorisation et, selon la manière dont la représentation numérique est construite dans la matrice creuse, il peut être classé en plusieurs types. Dans ce travail, les approches suivantes sont considérées : BoW (Bag of Words), TF-IDF et Char TF-IDF [5].

- **BoW.** Représente un document par l'occurrence de mots en n-grammes, en ignorant les positions qu'ils occupent dans le document. Il construit un dictionnaire/vocabulaire à partir des mots du corpus, attribue un indice à chaque terme, puis compte combien de fois chaque mot apparaît dans le document, en stockant ces comptes dans le vecteur du document. Dans scikit-learn, ceci est implémenté par `CountVectorizer`, qui “convertit une collection de documents textuels en une matrice de comptes de tokens” [5].

- **TF-IDF.** Part d'un principe similaire à BoW dans le sens où il produit également une représentation sous forme de matrice creuse ; cependant, au lieu de ne

s'appuyer que sur des comptages bruts de fréquence, il incorpore le terme d' inverse document frequency (IDF), qui pénalise les termes apparaissant dans de nombreux documents [5]. Conceptuellement, le poids TF-IDF d'un terme  $t$  dans un document  $d$  est calculé comme dans l'Eq. (1), où  $\text{tf}(t, d)$  correspond à la fréquence (brute) du terme dans  $d$ , et  $\text{idf}(t)$  attribue des poids plus faibles aux termes largement distribués dans le corpus.

$$\text{tfidf}(t, d) = \text{tf}(t, d) \times \text{idf}(t). \quad (1)$$

En pratique (et comme implémenté dans des bibliothèques courantes telles que scikit-learn), une version lissée de l>IDF est typiquement utilisée, définie dans l'Eq. (2), où  $n$  est le nombre total de documents dans le corpus et  $\text{df}(t)$  est le nombre de documents contenant le terme  $t$ .

$$\text{idf}(t) = \log \left( \frac{1 + n}{1 + \text{df}(t)} \right) + 1. \quad (2)$$

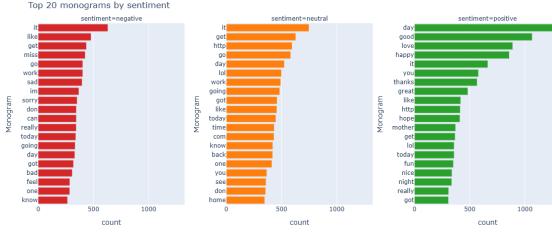
Enfin, après le calcul de TF-IDF, il est courant de normaliser chaque vecteur de document afin de contrôler les différences de longueur des documents et de stabiliser l'échelle des caractéristiques. Dans ce travail, la normalisation  $\ell_2$  est considérée, comme le montre l'Eq. (3), où  $\mathbf{v}$  désigne le vecteur TF-IDF d'un document.

$$\mathbf{v}_{\text{norm}} = \frac{\mathbf{v}}{\|\mathbf{v}\|_2} = \frac{\mathbf{v}}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}}. \quad (3)$$

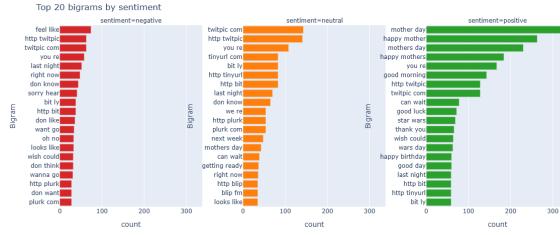
- **Char-IDF.** Consiste à appliquer TF-IDF sur des n-grammes de caractères plutôt que sur des n-grammes de mots, ce qui, dans scikit-learn, peut être contrôlé via le paramètre `analyzer` du vectoriseur [6].

Après avoir défini les schémas de représentation vectorielle des documents, il est proposé d'utiliser BoW pour extraire les 20 monogrammes et bigrammes les plus fréquents au sein de l'ensemble d'entraînement. Dans le cas de TF-IDF, l'objectif est d'afficher les 20 monogrammes et bigrammes avec les poids les plus élevés dans le jeu de données d'entraînement. L'utilisation de n-grammes de caractères n'est pas proposée pour cette étape car, en ne formant pas de mots complets, les caractéristiques résultantes sont moins intelligibles pour l'analyse visée.

En analysant les résultats présentés aux Figs. 2 - 3, on peut observer que certains n-grammes apparaissant sous les deux méthodes correspondent à des mots qui, dans l'usage courant de la langue, sont communément associés à la classe de polarité dans laquelle ils deviennent les plus représentatifs. C'est le cas du unigramme *bad* ou *sorry* dans la classe de polarité négative, qui apparaît parmi les principaux termes pour les deux méthodes de vectorisation. Néanmoins, il est évident que, puisqu'il s'agit d'approches purement statistiques, des mots qui n'ont pas été retirés lors du prétraitement mais qui sont très fréquents en anglais—tels que *it*—peuvent également

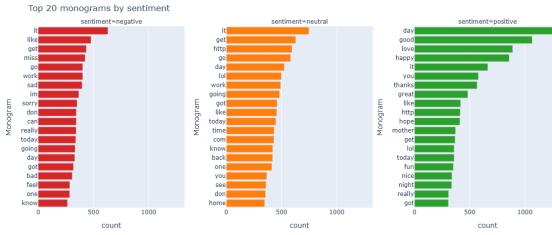


(a) Monograms

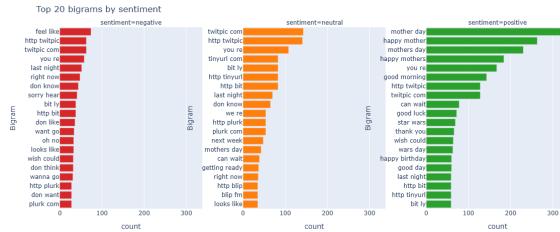


(b) Bigrams

FIGURE 2 – Top 20 n-gramas avec BoW.



(a) Monograms



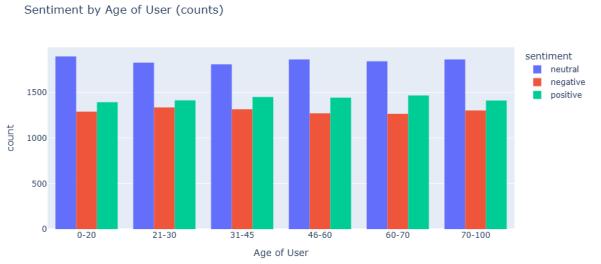
(b) Bigrams

FIGURE 3 – Top 20 n-gramas avec TF-IDF.

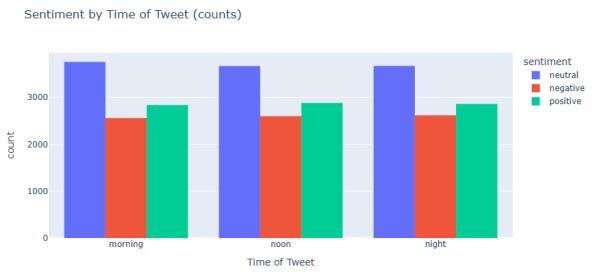
apparaître. De même, certains bigrammes qu'un locuteur pourrait classifier comme neutres, mais qui deviennent fréquents en raison du contexte de collecte des données (par exemple, *feels like*), apparaissent parmi les n-grammes les plus représentatifs à travers plusieurs classes. Cela anticipe l'un des effets abordés dans des étapes ultérieures : l'entraînement de modèles utilisant des vectorisations basées sur la fréquence, plutôt que des embeddings qui incorporent une information sémantique dans la représentation vectorielle creuse des documents.

En plus de la colonne de texte traité, qui constitue le focus principal de ce travail, il est intéressant de déterminer si certaines des autres colonnes du jeu de données présentent

une relation avec la classification de sentiment. À cette fin, les distributions de classes sont analysées en fonction de l'âge de l'utilisateur et du moment de la journée où le tweet est publié, comme le montre la Fig. 4 Cependant, la distribution des classes pour ces variables catégorielles reste très similaire à travers leurs domaines respectifs dans les deux cas ; par conséquent, elles ne sont pas considérées comme pertinentes pour l'entraînement du modèle de classification de sentiment.



(a) Sentiments by User



(b) Sentiments by type of tweet

FIGURE 4 – Distribution comparative des sentiments par catégorie, stratifiée par utilisateur et type de tweet

Dans le cas du pays dans lequel le tweet est publié, une distribution des pays avec le plus grand nombre de publications est analysée à la Fig. 5. On observe que les comptes de tweets ne varient pas de manière fortement représentative entre eux et, bien que les distributions de classes soient légèrement différentes selon les pays, cette caractéristique n'est pas incluse dans les étapes ultérieures. Une raison est que le modèle peut "mémoriser" des motifs spécifiques aux pays qui ne se vérifient pas en dehors du jeu de données ou qui changent au cours du temps. De plus, compte tenu de la forme de la distribution par pays, de nombreux pays n'ont qu'un petit nombre d'exemples, ce qui génère des caractéristiques rares et augmente la probabilité de surapprentissage. Pour ces raisons, cette colonne n'est finalement pas incluse dans l'analyse.

Enfin, en tant que dernière analyse du jeu de données, il est proposé d'examiner si la longueur du texte prétraité a une relation avec la classe, et donc s'il pourrait être utile de l'inclure comme variable durant l'entraînement des modèles de classification. Cependant, comme le montre la Fig. 6, la distribution du nombre de mots par message pour chaque classe est comparable (ou effectivement équivalente). Pour

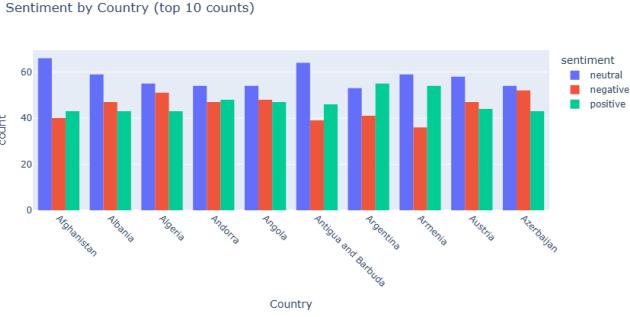


FIGURE 5 – Distribution des sentiments dans les 10 principaux pays.



FIGURE 6 – Distribution du nombre de mots par sentiment

cette raison, cette variable n'est pas non plus considérée comme pertinente pour l'objectif des modèles construits à l'étape suivante.

### B. Entraînement de modèles classiques

Après l'analyse et la description du jeu de données présentées dans la section précédente, il est proposé d'implémenter des modèles traditionnels d'apprentissage automatique permettant de classer des messages textuels reçus et traités dans chacune des trois catégories de polarité (sentiment). Dans ce travail, le concept de modèles classiques d'apprentissage automatique renvoie à des approches supervisées qui ne relèvent pas du deep learning et ne s'appuient pas sur des Transformers ou des embeddings ; à la place, ils opèrent sur une représentation explicite du texte obtenue à partir de l'effet de l'un des vectoriseurs mentionnés précédemment. Cela permet de suivre le workflow : représentation du document, entraînement du classifieur, évaluation [7].

Dans ce cadre, et compte tenu de la variabilité associée à chacun des vectoriseurs décrits précédemment, il est proposé que, pour chaque représentation de texte, un ensemble de classificateurs soit entraîné en utilisant des algorithmes classiques d'apprentissage automatique : Multinomial Naive Bayes, Régression Logistique, Support Vector Machine et Random Forest. Cependant, étant donné la variabilité des hyperparamètres associés à chaque classifieur et au vectoriseur lui-même, une recherche d'hyperparamètres est réalisée sur un pipeline

composé du vectoriseur et du modèle sélectionné, et évaluée via une validation croisée stratifiée. L'approche compare la performance obtenue pour chaque configuration candidate d'hyperparamètres à travers les folds de validation croisée, et sélectionne la configuration qui maximise une métrique choisie. Enfin, le pipeline le plus performant est réentraîné sur l'ensemble d'entraînement complet en utilisant le réglage d'hyperparamètres sélectionné.

Afin de décrire adéquatement le fonctionnement de l'algorithme d'entraînement employé, il est d'abord proposé de détailler l'influence de chacun des hyperparamètres considérés pour chaque élément du pipeline, tant dans le cas des vecteuriseurs que dans le cas des modèles entraînés.

Il est opportun de commencer par les hyperparamètres associés au vecteuriseur au sein du pipeline. Le premier d'entre eux est `ngram_range`, qui est un paramètre défini comme un tableau de tuples indiquant la taille des n-grammes pris en compte dans la construction de la représentation vectorielle. Ces n-grammes correspondent à des mots dans le cas de BoW, à des n-grammes de mots dans le cas du TF-IDF au niveau des mots, et à des n-grammes de caractères dans le cas du TF-IDF au niveau des caractères. Le paramètre `min_df`, à son tour, contient un ensemble de seuils minimaux à évaluer pour le critère de document-frequency, c'est-à-dire le nombre minimum de documents dans lesquels un n-gramme doit apparaître afin d'être considéré. Ceci est utile pour retirer des termes très rares de la représentation. Le paramètre `max_df` spécifie la fréquence documentaire maximale qu'un n-gramme peut avoir pour être inclus dans la vectorisation, ce qui est utile pour retirer des n-grammes trop fréquents et fournissant donc une information discriminante limitée. Enfin, `max_features` contient un ensemble de valeurs candidates, où chaque valeur correspond à une limite sur le nombre de caractéristiques testée durant la phase de recherche d'hyperparamètres afin de contraindre la taille du vocabulaire, principalement en fonction de la fréquence et du coût computationnel. Cette limitation est particulièrement importante pour adapter l'entraînement des réseaux neuronaux précédés d'une vectorisation au niveau des caractères dans la Question Q2, comme cela sera détaillé plus tard.

D'autre part, les hyperparamètres associés à chaque modèle sont spécifiques à chaque algorithme, selon sa nature et sa formulation mathématique, et ils sont décrits ci-dessous :

- **Multinomial Naive Bayes**  $\alpha$  est un paramètre de lissage additif dont l'objectif principal est d'éviter des probabilités nulles dans le cas où une caractéristique n'apparaît pas dans les données d'entraînement. Rapelons que Naive Bayes est formulé comme un produit de probabilités conditionnelles sous une hypothèse d'indépendance ; dans ce contexte, le lissage fonctionne en augmentant artificiellement les comptes observés de la valeur  $\alpha$  [8].

- **Régression Logistique**  $C$  est un paramètre qui représente l'inverse de la force de régularisation : des valeurs plus petites impliquent une régularisation plus forte et, par conséquent, un risque plus faible de surapprentis-

sage, tandis que des valeurs plus grandes impliquent une régularisation plus faible et un modèle avec davantage de liberté dans ses paramètres. L'option `penalty` définit le type de régularisation appliquée par le modèle : *L1* pénalise la somme des valeurs absolues des coefficients, tandis que *L2* pénalise la somme des carrés des coefficients, ce qui tend à réduire les poids plus en douceur et n'encourage pas explicitement des solutions creuses où certains coefficients deviennent exactement nuls. Le paramètre `solver` se réfère à l'algorithme d'optimisation utilisé durant l'entraînement, et `max_iter` indique le nombre maximal d'itérations autorisées pour que le solveur converge [9].

— **Support vector machine** Le paramètre `C` agit comme l'inverse de la force de régularisation. L'hyperparamètre `estimator_loss` est utilisé pour définir la fonction de loss du SVM linéaire. Un paramètre `max_iter` est également inclus afin de définir le nombre maximal d'itérations autorisées pour la convergence. De plus, le SVM est encapsulé dans `CalibratedClassifierCV`, qui convertit la sortie du Support Vector Machine en probabilités de classe, rendant les prédictions plus interprétables et, par conséquent, plus adaptées à la procédure d'ajustement d'hyperparamètres du pipeline [11].

— **Random forest** `n_estimators` spécifie le nombre d'arbres (estimateurs) entraînés dans la forêt. Le paramètre `max_depth` définit la profondeur maximale de chaque arbre ; lorsqu'il est fixé à `None`, l'arbre est autorisé à croître jusqu'à ce que ses feuilles satisfassent au critère d'arrêt défini par `min_samples_leaf`. En revanche, lorsqu'une valeur finie est utilisée, le nombre de niveaux de l'arbre est restreint, réduisant le nombre de noeuds et agissant comme un mécanisme de régularisation pouvant diminuer le risque de surapprentissage. Le paramètre `min_samples_split` définit le nombre minimum d'échantillons requis pour scinder un noeud interne ; il peut également être utilisé pour contrôler le surapprentissage et peut donc avoir un effet de régularisation. Enfin, `class_weight` est utilisé pour définir des poids de classe durant l'entraînement : `None` utilise des poids égaux, tandis que `balanced` ajuste les poids en fonction des fréquences des classes dans les données d'entraînement afin de contrebalancer un déséquilibre de classes [12].

Avec l'ensemble de paramètres définis et un pipeline spécifique composé d'un vectoriseur et d'un modèle d'apprentissage supervisé, une grille de paramètres est construite. À partir de cette grille, les combinaisons candidates sont énumérées et la performance de chaque combinaison est estimée via une validation croisée en 5 folds, en agrégant les métriques à travers les folds afin d'obtenir une représentation plus robuste de l'estimation du modèle. Une fois que l'algorithme de recherche sur grille identifie une configuration optimale, le pipeline sélectionné est réentraîné sur l'ensemble d'entraînement complet. Il convient de souligner que cette conception

inclut également la possibilité d'activer ou non le parallélisme à chaque exécution de l'algorithme d'entraînement, selon les exigences spécifiques de chaque exécution du processus d'entraînement.

Après avoir conclu l'analyse de la méthode, l'entraînement de tous les pipelines composés des vectoriseurs et des modèles référencés est proposé en utilisant l'algorithme de recherche d'hyperparamètres, et les résultats de ce processus sont discutés dans la section suivante.

### III. Q1-CLASSIFICATION AVEC DES MODÈLES CLASSIQUES ET ANALYSE DES PERFORMANCES

Un total de 12 configurations expérimentales ont été entraînées (3 schémas de vectorisation  $\times$  4 modèles). L'objectif de cette étape est de comparer les performances de ces combinaisons sur l'ensemble de test et de définir une baseline classique solide, qui servira ensuite de point de référence face aux méthodes basées sur des embeddings et aux modèles de type BERT.

Les métriques rapportées sont l'Accuracy, le Macro-Recall et le Macro-F1. Les métriques macro sont particulièrement pertinentes car elles évaluent la performance de manière plus équilibrée entre les classes, réduisant le risque qu'une classe dominante pilote l'évaluation globale.

#### A. Temps d'entraînement

Chaque expérience entraînée avec `GridSearchCV` produit un fichier CSV contenant le détail complet de toutes les combinaisons évaluées, y compris les hyperparamètres du vectoriseur et du classifieur. Ces fichiers CSV permettent une analyse systématique des changements de configuration et une sélection informée du pipeline final le plus performant.

Dans ces fichiers, le champ `mean_fit_time` représente le temps moyen d'entraînement par fold pour chaque configuration évaluée. En utilisant cette valeur, il est possible d'estimer le temps de calcul global associé à la recherche d'hyperparamètres.

1) *Estimation du temps total (calcul séquentiel)*: L'estimation séquentielle du temps total est calculée comme la somme du temps moyen d'entraînement par fold pour chaque configuration candidate, multipliée par le nombre de folds de validation croisée :

$$T_{\text{total}} \approx \sum_{i=1}^{n_{\text{candidates}}} (\text{mean\_fit\_time}_i \times n_{\text{splits}}), \quad n_{\text{splits}} = 5. \quad (4)$$

Ici,  $n_{\text{candidates}}$  correspond au nombre de lignes dans le CSV, c'est-à-dire le nombre de combinaisons d'hyperparamètres évaluées, étant le total accumulé :

- $n_{\text{candidates}} = 968$
- $T_{\text{total}} \approx 287,978.76$  s

2) *Temps réel (wall-clock) avec parallélisation*: La valeur ci-dessus correspond à un temps de calcul séquentiel théorique (c'est-à-dire une estimation mono-thread). Cependant, puisque la parallélisation a été activée avec `n_jobs=-1`, la charge de travail est distribuée sur plusieurs coeurs CPU. Par conséquent,

un temps wall-clock approximatif peut être estimé comme suit :

$$T_{\text{wall}} \approx \frac{T_{\text{total}}}{n_{\text{jobs}}}.$$
 (5)

En supposant 8 cœurs effectifs, l'estimation wall-clock devient :

$$T_{\text{wall}} \approx \frac{287,978.76}{8} \approx 35,997.35 \text{ s} \approx 10.00 \text{ h.}$$

### B. Performance sur l'ensemble de test

Les performances sur test sont résumées par les résultats présentés aux Figs. 7–9, qui rapportent respectivement le Macro-F1, le Macro-Recall et l'Accuracy. Ces graphiques montrent, pour chaque classifieur, la performance obtenue sous chaque méthode de vectorisation, permettant d'identifier immédiatement des tendances robustes.

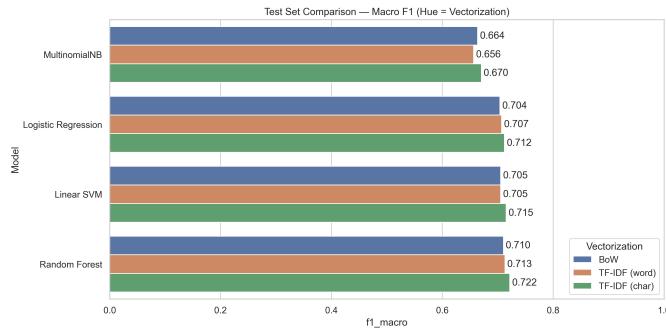


FIGURE 7 – Comparaison sur test — Macro-F1 (teinte = vectorisation).

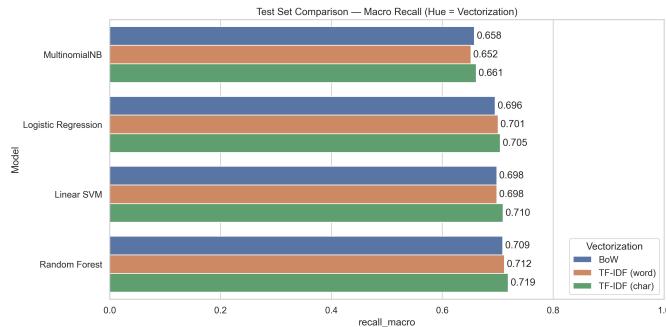


FIGURE 8 – Comparaison sur test — Macro-Recall (teinte = vectorisation).

*1) Impact de la méthode de vectorisation:* Globalement, les figures indiquent que le TF-IDF au niveau des caractères (TF-IDF char) tend à produire les meilleures valeurs sur les trois métriques. Ce comportement est cohérent avec la nature du jeu de données (tweets courts et informels), où la modélisation au niveau des caractères capture des motifs sub-lexicaux importants tels que les fautes d'orthographe, les variations morphologiques, les allongements (par exemple, “soooo good”), les hashtags, les abréviations, et des indices de préfixe/suffixe.

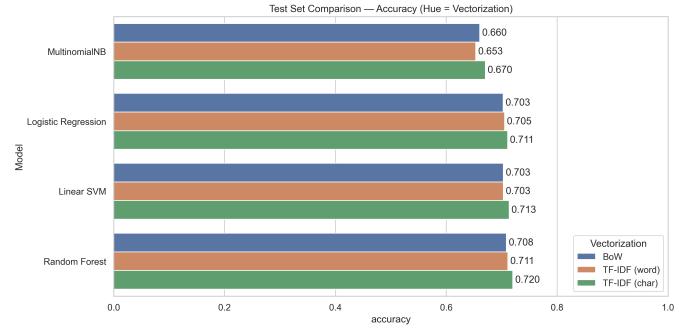


FIGURE 9 – Comparaison sur test — Accuracy (teinte = vectorisation).

En revanche, BoW présente souvent la performance la plus faible car il repose sur des comptages bruts sans pondérer les termes selon leur pertinence globale. Le TF-IDF au niveau des mots améliore BoW, mais reste moins robuste au bruit orthographique et à l'usage d'une langue non standard comparé aux n-grammes de caractères.

*2) Comparaison des modèles et compromis:* Parmi les classifiers, Linear SVM et la Régression Logistique se distinguent par leur cohérence et leur stabilité, en particulier lorsqu'ils sont combinés à TF-IDF (notamment TF-IDF char). Cela est attendu car les représentations BoW/TF-IDF produisent des matrices de très grande dimension et creuses, un régime où les modèles linéaires sont typiquement à la fois efficaces et computationnellement efficientes.

Bien que Random Forest obtienne des résultats très compétitifs (et puisse même atteindre le meilleur score), plusieurs inconvénients sont importants dans des contextes textuels de grande dimension :

- **Scalabilité et coût computationnel :** entraîner de nombreux arbres dans de grands espaces TF-IDF augmente substantiellement le temps et l'usage mémoire, en particulier sous GridSearchCV.
- **Risque de surapprentissage avec des caractéristiques très spécifiques :** TF-IDF char introduit de nombreux n-grammes très particuliers ; un modèle non linéaire tel que Random Forest peut capturer des motifs accidentels d'entraînement et réduire la robustesse hors domaine.
- **Interprétabilité pratique plus faible :** bien qu'un arbre unique soit interprétable, une forêt avec de nombreux estimateurs est plus difficile à justifier de manière transparente. En revanche, les modèles linéaires permettent une analyse plus directe des signaux discriminants.

Par conséquent, même si Random Forest produit un petit gain absolu, les modèles linéaires restent un choix fort en raison de la stabilité, du coût, et de la clarté méthodologique.

*3) Analyse de la matrice de confusion (TF-IDF char):* Afin d'analyser le comportement des meilleurs modèles linéaires sur l'ensemble de test, nous examinons les matrices de confusion pour TF-IDF au niveau des caractères avec Linear SVM et Régression Logistique. Cette analyse révèle non seulement combien de prédictions sont correctes, mais aussi quels types

d'erreurs sont les plus fréquents et quelles classes sont le plus souvent confondues dans ce cadre à trois classes (négatif, neutre, positif).

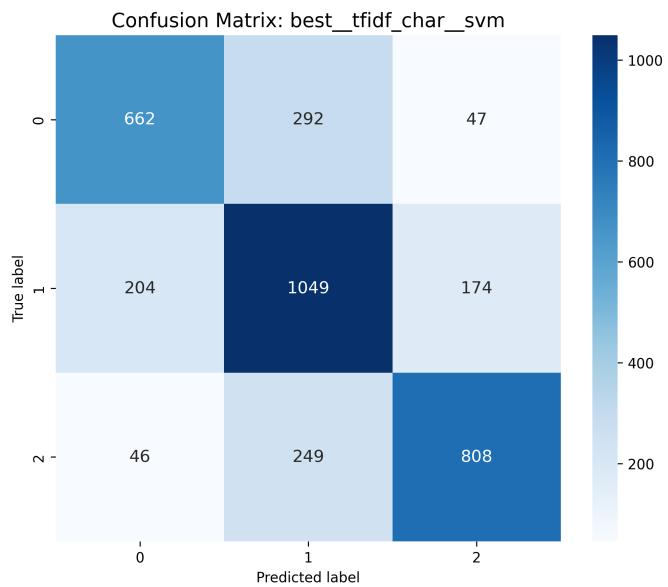


FIGURE 10 – Matrice de confusion de best\_tfidf\_char\_svm.

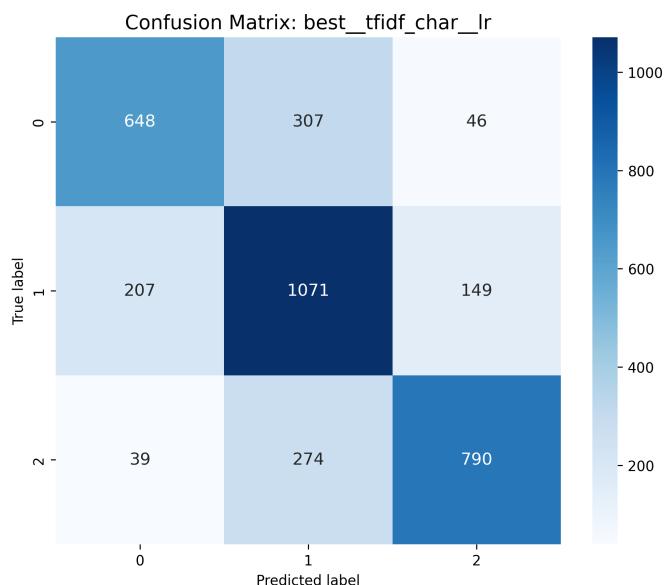


FIGURE 11 – Matrice de confusion de best\_tfidf\_char\_lr.

a) *Motif général : la classe “neutre” est la principale source d’ambiguïté.*: Dans les deux matrices, la plupart des erreurs se concentrent autour de la classe neutre, ce qui est attendu en analyse de sentiment de tweets courts. De nombreux messages contiennent de faibles indices émotionnels, de l’ironie, des abréviations, ou des fragments dépendants du

contexte, ce qui rend difficile la séparation entre neutre et des textes faiblement positifs ou négatifs.

Concrètement, les erreurs les plus fréquentes correspondent à des exemples polarisés étant “absorbés” dans la classe neutre :

- Négatif → Neutre : SVM = 292, Régression Logistique = 307
- Positif → Neutre : SVM = 249, Régression Logistique = 274

Ce motif suggère que, lorsque la polarité n'est pas fortement marquée, le modèle tend à assigner neutre comme option “sûre”. En pratique, la frontière la plus difficile n'est pas Négatif vs. Positif, mais plutôt Neutre vs. (Négatif/Positif).

b) *Bon signe : faible confusion directe entre “négatif” et “positif”.*: Un résultat pertinent est que la confusion directe entre les extrêmes de polarité (Négatif ↔ Positif) est relativement faible comparée aux confusions vers neutre :

- Négatif → Positif : SVM = 47, Régression Logistique = 46
- Positif → Négatif : SVM = 46, Régression Logistique = 39

Cela indique que lorsque des marqueurs linguistiques forts existent, les modèles inversent rarement la polarité du sentiment. Ainsi, les signaux de sentiment clairs sont capturés correctement ; le principal défi réside dans les cas modérés ou ambigus où l’émotion est faible ou dépendante du contexte.

c) *Differences subtiles entre SVM et Régression Logistique.*: Bien que les deux modèles atteignent des performances globales très similaires, les matrices suggèrent un comportement légèrement différent vis-à-vis de la classe neutre :

- La Régression Logistique prédit “neutre” plus souvent que SVM (attraction plus forte vers neutre).
- Cela se reflète par des confusions Négatif → Neutre et Positif → Neutre légèrement plus élevées pour LR.

En termes de compromis :

- SVM tend à maintenir les classes polarisées légèrement mieux séparées (un petit avantage pour récupérer les positifs et les négatifs).
- LR tend à favoriser la classe neutre, ce qui peut augmenter le nombre d'exemples polarisés classés comme neutres.

Cela est cohérent avec la nature de la tâche : neutre est une classe large et moins bien définie sémantiquement, de sorte que de petits déplacements de frontières de décision peuvent produire des changements perceptibles dans les motifs de confusion.

#### C. Top-20 et Bottom-20 n-grammes de caractères par classe (LR vs. SVM)

1) *Classe positive*: Pour la classe Positive, les n-grammes de caractères les plus influents sont hautement cohérents entre LR et SVM. Les Top-20 n-grammes sont montrés à la Fig. 12 et la Fig. 13. Des fragments typiques d’approbation tels que *love/lov, nice, good/goo, fun, best, amaz, awes, yum, cool*, et des variantes de *thank* apparaissent de manière proéminente, confirmant le bénéfice de la modélisation au niveau des

Top 20 Character n-grams Supporting the 'Positive' Class

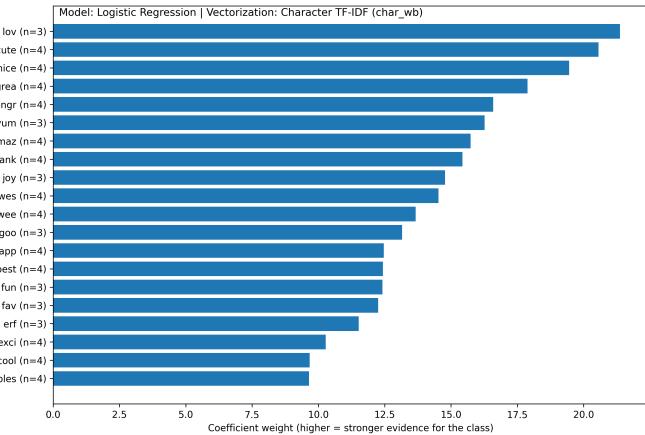


FIGURE 12 – Classe positive : Top-20 n-grammes de caractères (Régression Logistique) sous TF-IDF char (char\_wb).

Top 20 Character n-grams Penalizing the 'Positive' Class

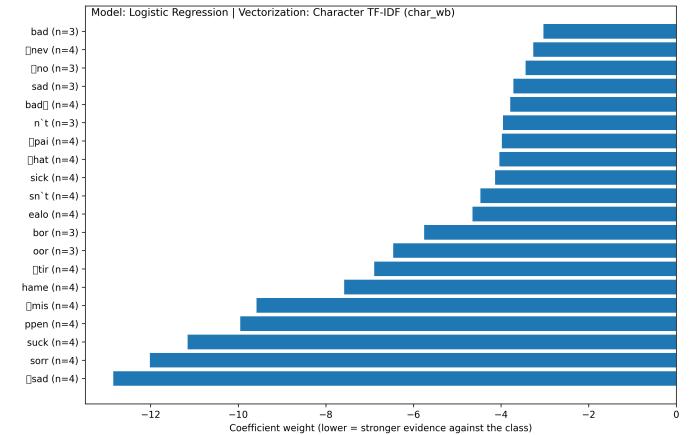


FIGURE 14 – Classe positive : Bottom-20 n-grammes de caractères (Régression Logistique) sous TF-IDF char (char\_wb).

Top 20 Character n-grams Supporting the 'Positive' Class

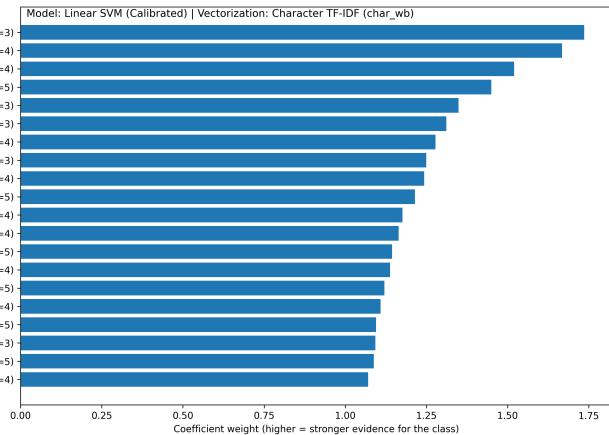


FIGURE 13 – Classe positive : Top-20 n-grammes de caractères (Linear SVM) sous TF-IDF char (char\_wb).

caractères : même des mots incomplets ou abrégés conservent de forts indices de sentiment.

En observant les Bottom-20 (Fig. 14 et Fig. 15), les pénalités les plus fortes pour la classe Positive sont des marqueurs clairs de négativité tels que *sad*, *hate/hat*, *bad*, *suck*, et des négations (par exemple, *no*, *n't*). En pratique, prédire "Positive" requiert non seulement des preuves positives mais aussi l'absence d'indices négatifs forts.

2) *Classe négative*: Pour la classe Négative, les Top-20 n-grammes (Fig. 16 et Fig. 17) reflètent l'inconfort et la plainte (*sad*, *hate/hat*, *bad*, *hurt*, *wors*, *sick*, *poor*, et des fragments intenses tels que *uck*). Leur présence à la fois dans LR et SVM indique que la classe négative est bien définie même lorsque les mots sont abrégés ou incomplets.

Inversement, les Bottom-20 (Fig. 18 et Fig. 19) montrent que de forts indices positifs tels que *lov/love*, *thank/than*, *hope*, ou *good* pénalisent la classe Négative, démontrant que les modèles capturent à la fois des preuves négatives et de fortes

Top 20 Character n-grams Penalizing the 'Positive' Class

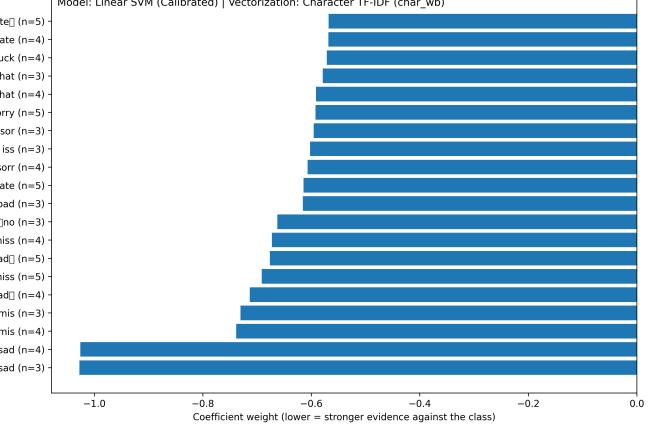


FIGURE 15 – Classe positive : Bottom-20 n-grammes de caractères (Linear SVM) sous TF-IDF char (char\_wb).

Top 20 Character n-grams Supporting the 'Negative' Class

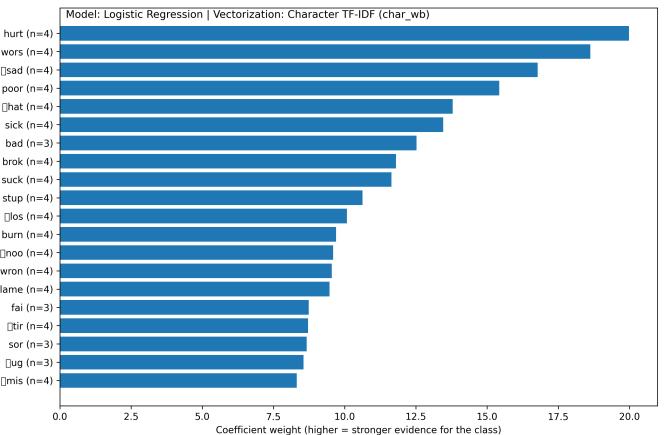


FIGURE 16 – Classe négative : Top-20 n-grammes de caractères (Régression Logistique) sous TF-IDF char (char\_wb).

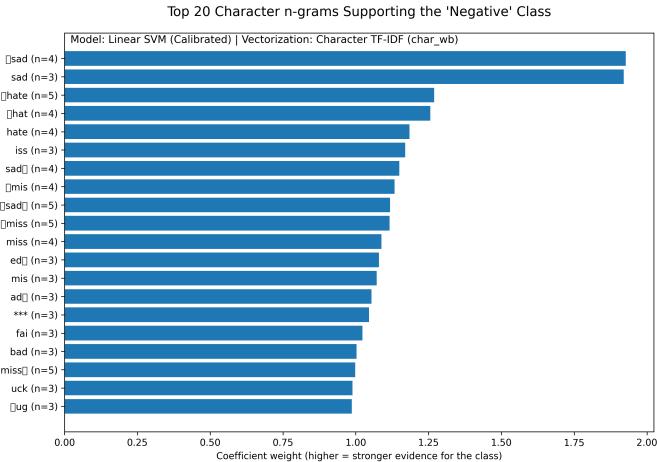


FIGURE 17 – Classe négative : Top-20 n-grammes de caractères (Linear SVM) sous TF-IDF char (char\_wb).

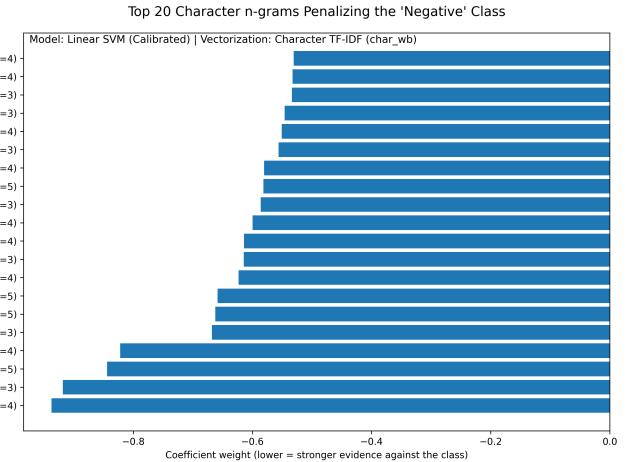


FIGURE 19 – Classe négative : Bottom-20 n-grammes de caractères (Linear SVM) sous TF-IDF char (char\_wb).

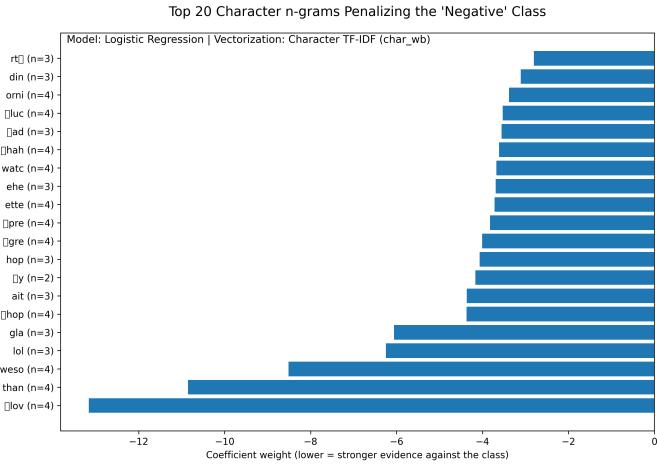


FIGURE 18 – Classe négative : Bottom-20 n-grammes de caractères (Régression Logistique) sous TF-IDF char (char\_wb).

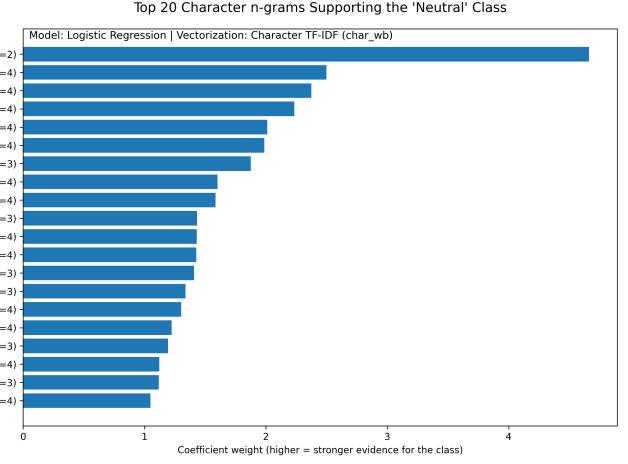


FIGURE 20 – Classe neutre : Top-20 n-grammes de caractères (Régression Logistique) sous TF-IDF char (char\_wb).

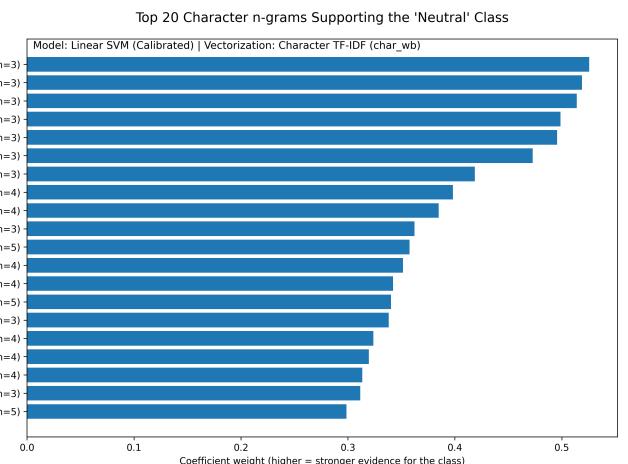


FIGURE 21 – Classe neutre : Top-20 n-grammes de caractères (Linear SVM) sous TF-IDF char (char\_wb).

contre-preuves orientant vers Positive.

3) *Classe neutre*: Neutre est la classe la plus intéressante à interpréter. Les Top-20 n-grammes (Fig. 20 et Fig. 21) tendent à être plus structurels que sentimentaux (par exemple, de la ponctuation telle que des points d'interrogation et des connecteurs discursifs), suggérant que le modèle exploite des motifs typiques de déclarations informatives ou de questions.

Dans les Bottom-20 (Fig. 22 et Fig. 23), la classe Neutre est pénalisée lorsque des signaux de polarité clairs apparaissent, qu'ils soient positifs (*love/lov, good, fun, nice*) ou négatifs (*sad, hate, etc.*). En d'autres termes, Neutre est largement défini par l'absence d'indices de sentiment forts, ce qui aide à expliquer pourquoi il est souvent la classe la plus difficile.

#### D. Synthèse et considérations pratiques

Après comparaison de toutes les configurations entraînées, les résultats les plus robustes et les plus cohérents sont obtenus avec TF-IDF au niveau des caractères (TF-IDF char) et des

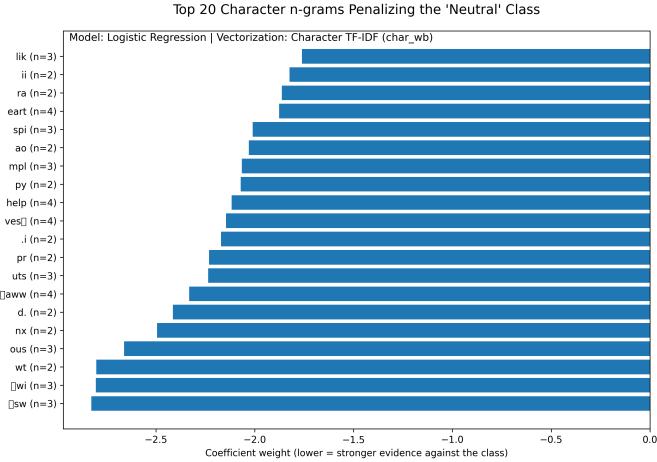


FIGURE 22 – Classe neutre : Bottom-20 n-grammes de caractères (Régression Logistique) sous TF-IDF char (char\_wb).

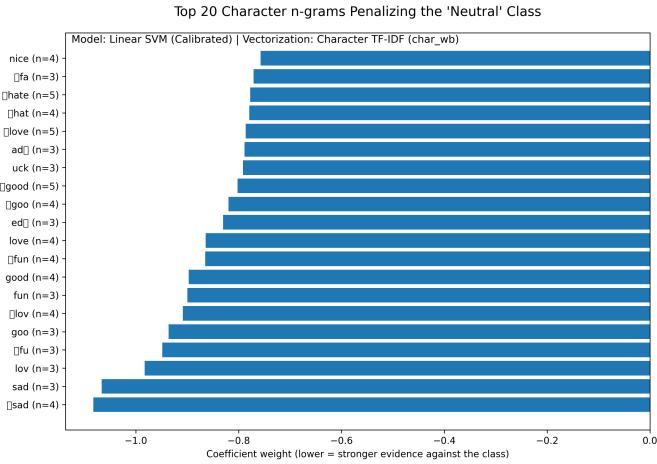


FIGURE 23 – Classe neutre : Bottom-20 n-grammes de caractères (Linear SVM) sous TF-IDF char (char\_wb).

modèles linéaires. Les deux meilleures approches (en termes d'équilibre entre Accuracy, Macro-Recall et Macro-F1) sont :

- Linear SVM + TF-IDF char
- Régression Logistique + TF-IDF char

Bien que Random Forest puisse atteindre un score légèrement plus élevé (voir Fig. 7), l'écart de métriques n'est pas suffisamment grand pour justifier ses inconvénients dans ce contexte (coût computationnel plus élevé, risque de surapprentissage plus élevé, et interprétabilité plus faible). Pour ces raisons, nous sélectionnons TF-IDF char + Linear SVM comme baseline classique principale, avec TF-IDF char + Régression Logistique comme alternative très proche.

#### IV. Q2-ARCHITECTURES MLP POUR LA CLASSIFICATION DE SENTIMENT DE TEXTES COURTS

Dans cette section, il est proposé d'employer des architectures de perceptron multicouche (MLP) pour entraîner des classificateurs qui identifient efficacement la polarité dans des textes courts en anglais. Afin de définir la structure et le

processus d'entraînement de l'ensemble des MLP mis en œuvre, il est proposé de discuter d'abord du pipeline concret qui permet d'obtenir une classification finale.

Le premier élément du pipeline est le vectoriseur, qui peut être de l'un des trois types considérés précédemment. Afin de définir les hyperparamètres de chaque vectoriseur, le réglage d'hyperparamètres associé au vectoriseur est sélectionné à partir de la grille de paramètres du pipeline le plus performant qui inclut ce vectoriseur spécifique.

Dans le cas des vectoriseurs sélectionnés pour le TF-IDF au niveau des mots et BoW, le nombre de caractéristiques résultantes est de 6689. Cependant, pour le TF-IDF au niveau des caractères, comme attendu compte tenu de la définition de ses n-grammes, le nombre de caractéristiques dépasse 100,000 lorsqu'aucune limite explicite n'est imposée. Cela rend l'entraînement d'un MLP dans ce pipeline significativement plus coûteux en termes de GPU et de RAM, et cela exigerait également de définir des structures de réseau différentes de celles utilisées avec les autres vectoriseurs. Pour cette raison, la configuration d'hyperparamètres la plus performante pour le vectoriseur TF-IDF au niveau des caractères est conservée, mais son nombre de caractéristiques est restreint à 10,000 afin que sa dimensionnalité soit comparable à celle des autres vectoriseurs. Cela permet d'implémenter des structures MLP équivalentes à travers les pipelines, ne différant que par la dimension d'entrée de la première couche tout en conservant le même nombre de couches cachées, d'une manière qui reste cohérente.

L'élément suivant du pipeline est le MLP utilisé pour la classification ; toutefois, plusieurs composants sont nécessaires pour son entraînement et pour définir sa structure. Le premier d'entre eux est un objet qui convertit la matrice creuse produite par le vectoriseur en une matrice au format Compressed Sparse Row (CSR), permettant un accès efficace ligne par ligne, à savoir SparseBowDataset [13].

Cette matrice CSR est ensuite utilisée pour construire un DataLoader, un itérateur PyTorch qui, à chaque époque d'entraînement, parcourt l'ensemble d'entraînement et fournit un batch d'échantillons par itération. Avant chaque époque, les échantillons sont mélangés afin de réduire le risque de biais dû à des effets d'ordre. Dans cette configuration, collate agit comme la fonction qui convertit les matrices creuses des batches en matrices denses puis en tenseurs PyTorch, renvoyant un dictionnaire de la forme {"x" : X\_batch, "label" : y\_batch}, qui est directement utilisable pour le forward pass et pour le calcul de la loss [14].

Après avoir défini ces composants, la discussion peut être complétée concernant la structure de réseau de neurones proposée à mettre en œuvre. Afin d'obtenir un modèle qui satisfasse à la fois les exigences d'entrée et la tâche de classification en sortie, un ensemble de quatre réseaux modérément profonds est proposé (3 ou 4 couches cachées), tout en maintenant une tête de sortie commune pour la classification finale à trois classes. La décision de limiter la profondeur est motivée par le fait que les représentations d'entrée (BoW/TF-IDF/char ou TF-IDF) condensent déjà une grande partie de

l'information discriminante au niveau des caractéristiques ; par conséquent, augmenter excessivement le nombre de couches peut accroître la variance du modèle et favoriser le surapprentissage sans gain proportionnel en généralisation.

De cette manière, les différences entre les structures de réseau proposées se concentrent sur deux éléments principaux dans la recherche d'une performance améliorée. Le premier est la modification des couches selon une architecture en forme d'entonnoir, et le second est l'ajustement de la stratégie de régularisation en faisant varier le dropout. Dans cette optique, il est proposé d'analyser plus en détail chacune des structures proposées et implémentées pour chaque type de vectorisation. De cette manière, les différences entre les structures de réseau proposées se concentrent sur deux éléments principaux dans la recherche d'une performance améliorée du réseau. Le premier est la modification des couches selon une architecture en forme d'entonnoir, et le second est l'ajustement de la stratégie de régularisation en faisant varier le dropout. Dans cette optique, il est proposé d'analyser plus en détail chacune des structures proposées et implémentées pour chaque type de vectorisation. Dans chaque cas, l'entrée du réseau est organisée sous forme d'une matrice dont la taille dépend de la taille de batch, définie de manière cohérente à travers tous les réseaux de neurones comme étant 128, et de la dimension d'entrée, qui correspond au nombre de caractéristiques produites par la méthode de vectorisation sélectionnée. Ainsi, pour les modèles basés sur des mots cette dimension est 6689, tandis que pour le modèle basé sur des caractères elle est 10,000.

#### — MLP\_1024\_512\_256\_drop0\_3

- Couche d'entrée (reshape).
- Couche 1 (Linear INPUT\_DIM → 1024) : transforme la représentation vectorielle en une représentation dense de 1024 unités.
- Activation 1 : ReLU.
- Régularisation 1 : Dropout ( $p = 0.3$ ).
- Couche 2 (Linear 1024 → 512) : compresse la représentation à 512 unités.
- Activation 2 : ReLU.
- Régularisation 2 : Dropout ( $p = 0.3$ ).
- Couche 3 (Linear 512 → 256) : réduit la représentation de 512 à 256 unités.
- Activation 3 : ReLU.
- Régularisation 3 : Dropout ( $p = 0.3$ ).
- Couche de sortie (Linear 256 → NUM\_CLASSES) : projette la représentation finale vers 3 classes, correspondant aux catégories de sentiment/polarité des messages courts et, par conséquent, à la décision de classe.

#### — MLP\_2048\_1024\_512\_drop0\_2\_gelu

- Couche d'entrée (reshape).
- Couche 1 (Linear INPUT\_DIM → 2048) : projette l'entrée vectorisée dans un espace de dimension 2048, augmentant la capacité à capturer des combinaisons de caractéristiques par rapport à la structure précédente.
- Activation 1 : GELU.

- Régularisation 1 : Dropout ( $p = 0.3$ ).
- Couche 2 (Linear 2048 → 1024) : compression intermédiaire à 1024 unités.
- Activation 2 : GELU.
- Régularisation 2 : Dropout ( $p = 0.2$ ).
- Couche 3 (Linear 1024 → 512) : compression supplémentaire vers un espace de dimension 512.
- Activation 3 : GELU.
- Régularisation 3 : Dropout ( $p = 0.2$ ).
- Couche de sortie (Linear 512 → NUM\_CLASSES) : produit les logits finaux.
- MLP\_1536\_768\_384\_192\_drop0\_25\_SiLU
- Couche d'entrée (reshape).
- Couche 1 (Linear INPUT\_DIM → 1536) : projection initiale à 1536 unités, fournissant une capacité élevée sans atteindre l'expansion maximale de l'architecture précédente.
- Activation 1 : SiLU.
- Régularisation 1 : Dropout ( $p = 0.25$ ).
- Couche 2 (Linear 1536 → 768) : réduction à 768 unités, consolidant les motifs appris.
- Activation 2 : SiLU.
- Régularisation 2 : Dropout ( $p = 0.25$ ).
- Couche 3 (Linear 768 → 384) : réduction à 384 unités.
- Activation 3 : SiLU.
- Régularisation 3 : Dropout ( $p = 0.25$ ).
- Couche 4 (Linear 384 → 192) : réduction finale à 192 unités, imposant une représentation compacte avant la couche de décision.
- Activation 4 : SiLU.
- Régularisation 4 : Dropout ( $p = 0.25$ ).
- Couche de sortie (Linear 192 → NUM\_CLASSES) : produit les logits.
- MLP\_4096\_2048\_1024\_drop0\_1\_ReLU
- Couche d'entrée (reshape).
- Couche 1 (Linear INPUT\_DIM → 4096) : projection à haute capacité vers 4096 unités, cherchant à maximiser l'espace latent afin de combiner un plus grand nombre de caractéristiques.
- Activation 1 : ReLU.
- Régularisation 1 : Dropout ( $p = 0.1$ ).
- Couche 2 (Linear 4096 → 2048) : compression à 2048 unités tout en maintenant une capacité élevée.
- Activation 2 : ReLU.
- Régularisation 2 : Dropout ( $p = 0.1$ ).
- Couche 3 (Linear 2048 → 1024) : compression à 1024 unités comme représentation préalable à la sortie.
- Activation 3 : ReLU.
- Régularisation 3 : Dropout ( $p = 0.1$ ).
- Couche de sortie (Linear 1024 → NUM\_CLASSES) : produit les logits finaux pour la classification.

Pour l'entraînement de tous les MLP proposés, le critère d'entropie croisée est employé. L'entraînement est effectué

pendant 50 époques, avec un critère d'arrêt rapide de  $1 \times 10^{-4}$  basé sur la variation de la loss d'entraînement entre époques consécutives. De plus, un taux d'apprentissage  $lr$  est défini afin de contrôler la taille du pas des mises à jour de paramètres durant l'optimisation, réalisée au moyen de l'optimiseur Adam, indépendamment de l'architecture du réseau considérée. De cette manière, chaque réseau est entraîné séparément en minimisant la fonction de loss et en mettant à jour les poids via la méthode de descente de gradient stochastique adaptative Adam, permettant de comparer leurs performances dans des conditions d'entraînement équivalentes.

L'entraînement est réalisé avec une boucle basée sur les époques, comme proposé dans le développement du cours. À chaque itération, le `train_loader` (le DataLoader pour l'ensemble d'entraînement) fournit le dictionnaire d'entrée contenant les tenseurs et leurs labels, qui sont transférés sur le dispositif de calcul, en l'occurrence le GPU. Pour chaque batch, l'optimiseur est réinitialisé et la sortie du modèle est calculée, suivie de la valeur de loss correspondante en utilisant la loss d'entropie croisée, telle que définie précédemment. Ensuite, la rétropropagation est exécutée et les paramètres du modèle sont mis à jour via une étape d'optimiseur Adam. Cette procédure est appliquée jusqu'à 50 époques, sous réserve du critère d'arrêt anticipé, et le modèle entraîné est sauvegardé à la fin soit comme un objet en mémoire soit comme un fichier `.pt`.

Compte tenu des choix de conception et d'entraînement décrits ci-dessus, ces modèles sont obtenus pour chacune des représentations vectorisées, et leurs résultats sont analysés plus tard. Toutefois, le point discuté précédemment reste pertinent : les méthodes de vectorisation basées sur la fréquence ne fournissent qu'une représentation dérivée d'une analyse fréquentielle du document, et elles ne prennent pas en compte des aspects tels que la position des tokens au sein du texte ou des considérations sémantiques. Cela motive la réflexion selon laquelle il peut être opportun d'implémenter une représentation vectorielle plus dense qui intègre cette information conceptuelle via la tokenisation des mots. Ainsi, l'utilisation d'embeddings BERT est considérée, et par conséquent l'évaluation de structures MLP au sein d'un pipeline dans lequel la première étape n'est pas une vectorisation basée sur la fréquence, mais plutôt une vectorisation basée sur des embeddings utilisant BERT. Les causes et les conséquences de cette implémentation sont discutées dans les sections spécifiques consacrées aux embeddings BERT, et en particulier en termes des implications qu'elle introduit par rapport à une approche basée sur du texte brut.

Changer l'approche de vectorisation implique que les éléments utilisés pour entraîner les réseaux de neurones décrits précédemment, ainsi que la structure du réseau elle-même, doivent être modifiés. Pour cette raison, il est proposé de décrire d'abord le changement dans la structure du DataLoader, qui, contrairement au cas des vecteurs basés sur la fréquence, ne fournit plus les caractéristiques finales mais produit à la place du texte tokenisé, puisque les caractéristiques (embeddings) sont générées lorsque les tenseurs passent à

travers BERT. Dans cette configuration, la fonction `collate` n'est pas appliquée parce que le tokenizer de BERT renvoie déjà des tenseurs de taille fixe, compatibles. Ainsi, le DataLoader fournit "`input_ids`" + "`attention_mask`" + "`label`", et l'embedding est produit en faisant passer ces tenseurs à travers BERT [15].

Ce DataLoader est converti en embeddings via une fonction d'extraction d'embeddings, en réglant BERT en mode évaluation et en exécutant le forward pass. Après cela, un pooling est appliqué afin d'obtenir un seul vecteur par texte, et les batches résultants sont concaténés. La sortie finale est donc une matrice dense, qui est standardisée en utilisant un `StandardScaler` qui transforme chaque caractéristique de sorte que, sur l'ensemble d'entraînement, la moyenne soit nulle et l'écart-type soit un. Cela garantit que les dimensions de l'embedding ont une échelle comparable, ce qui améliore la stabilité de l'entraînement et empêche certaines dimensions d'avoir un effet dominant en raison d'une variance plus élevée [16].

La sortie de l'étape de mise à l'échelle est convertie en un DataLoader pour le MLP comparable à celui utilisé lors de l'entraînement des méthodes basées sur un vectoriseur. Ce DataLoader construit des batches d'embeddings denses et leurs labels correspondants sous la forme `{"x" : X_batch, "label" : y_batch}`, qui sont ensuite utilisés pour l'entraînement du modèle et pour son évaluation de performance ultérieure.

Compte tenu du processus de vectorisation basé sur des embeddings et du rôle des DataLoaders à ce stade avec BERT et le MLP, il est approprié de discuter des modifications générales dans la structure des modèles entraînés pour les pipelines qui utilisent des embeddings BERT. Le premier effet est que la dimension d'entrée change considérablement : elle passe de plus de 6000 caractéristiques d'entrée à une taille d'entrée beaucoup plus petite et fixe, qui dans le cas de base est 768. BERT produit une matrice de sortie dense contenant des activations à valeurs réelles dans presque toutes les dimensions, qui est stockée comme un vecteur dense ; à l'inverse, les vectorisations basées sur la fréquence produisent des représentations creuses avec de nombreux zéros. Cette différence implique que la conception du MLP doit contrôler la capacité (largeur et profondeur) en relation avec l'espace d'entrée : pour des entrées creuses, de grande dimension, la régularisation joue un rôle fondamental afin d'éviter des effets de mémorisation causés par un espace d'entrée très élevé en dimension et principalement vide. Dans le cas de BERT, l'entrée dense implique que la majeure partie de la complexité de représentation est prise en charge par le forward pass de BERT qui produit les embeddings, de sorte que la cartographie ultérieure de cet espace vectoriel informatif vers les classes cibles devient plus simple.

De plus, lors de l'utilisation d'embeddings BERT, les transformations appliquées durant le forward pass encodent déjà des informations syntaxiques et sémantiques dans le vecteur selon les caractéristiques du modèle BERT utilisé, tandis que TF-IDF correspond à une représentation compa-

rativement superficielle qui reflète principalement une analyse fréquentielle. Par conséquent, pour les pipelines dans lesquels BERT fournit l'entrée du MLP, la représentation est déjà hautement informative et la complexité requise du MLP diminue considérablement ; dans certains cas, même une tête linéaire est suffisante pour atteindre la performance attendue [17]. Compte tenu de ces conditions, les quatre architectures proposées pour les pipelines à embeddings BERT sont moins complexes, avec moins de couches et des largeurs plus faibles que celles utilisées dans les pipelines précédents, et elles sont les suivantes.

### 1) LinearHead\_baseline (une seule couche linéaire)

- **Couche d'entrée (reshape).**
- **Couche finale** (Linear 768 → NUM\_CLASSES) : projette directement l'embedding de dimension 768 vers les 3 classes (sentiment/polarité), sans couches cachées ni activations intermédiaires.
- **Sortie** : correspond à la décision finale de classe.

### 2) MLP\_256\_64\_drop0\_2

- **Couche d'entrée (reshape).**
- **Couche 1** (Linear 768 → 256) : transforme la représentation d'entrée en 256 unités.
- **Activation 1** : ReLU.
- **Régularisation 1** : Dropout ( $p = 0.2$ ).
- **Couche 2** (Linear 256 → 64) : réduit/comprime la représentation à 64 unités.
- **Activation 2** : ReLU.
- **Régularisation 2** : Dropout ( $p = 0.2$ ).
- **Couche de sortie** (Linear 64 → NUM\_CLASSES) : projette la représentation finale vers 3 classes et définit la prédiction de polarité.

### 3) MLP\_128\_32\_drop0\_2

- **Couche d'entrée (reshape).**
- **Couche 1** (Linear 768 → 128) : compresse la représentation vectorielle à 128 unités.
- **Activation 1** : ReLU.
- **Régularisation 1** : Dropout ( $p = 0.2$ ).
- **Couche 2** (Linear 128 → 32) : réduit la représentation à 32 unités afin d'extraire des motifs plus compacts.
- **Activation 2** : ReLU.
- **Régularisation 2** : Dropout ( $p = 0.2$ ).
- **Couche de sortie** (Linear 32 → NUM\_CLASSES) : projette la sortie vers les 3 classes de sentiment et correspond à la décision finale.

### 4) MLP\_512\_128\_drop0\_3

- **Couche d'entrée (reshape).**
- **Couche 1** (Linear 768 → 512) : étend/transforme la représentation dans un espace dense de 512 unités afin de capturer des relations plus complexes.
- **Activation 1** : ReLU.
- **Régularisation 1** : Dropout ( $p = 0.3$ ).

— **Couche 2** (Linear 512 → 128) : compresse la représentation à 128 unités tout en préservant l'information la plus pertinente.

— **Activation 2** : ReLU.

— **Régularisation 2** : Dropout ( $p = 0.3$ ).

— **Couche de sortie** (Linear 128 → NUM\_CLASSES) : projette la représentation finale vers les 3 classes (sentiment/polarité) et définit la décision de classe.

Pour l'entraînement, la loss d'entropie croisée est définie comme critère d'optimisation. De plus, un maximum de 50 époques d'entraînement est proposé, avec un critère d'arrêt rapide de  $1 \times 10^{-4}$ , et un taux d'apprentissage de  $1 \times 10^{-4}$  est utilisé. L'optimiseur Adam est défini et appliqué à tous les réseaux entraînés. Il est également important de souligner que l'algorithme d'entraînement utilisé pour ces MLP est le même que celui employé pour entraîner les autres modèles, ce qui permet de comparer leurs performances dans des conditions d'entraînement cohérentes. Les résultats de tous les pipelines décrits dans cette section sont analysés dans la section suivante.

## V. Q3-ANALYSE COMPARATIVE DES PERFORMANCES DES MLP ET IMPLICATIONS POUR LA SÉLECTION DE LA BASELINE

Cette section fournit une analyse interprétative des résultats des perceptrons multicouches (MLP) introduits en Q2. Bien que Q1 se concentre sur des baselines linéaires classiques, il est utile d'examiner d'abord si l'introduction de non-linéarité via des MLP apporte une amélioration significative par rapport aux pipelines classiques, et quelles familles de représentations en bénéficient le plus. La discussion s'appuie sur le Macro-F1, le Macro-Recall et l'Accuracy sur l'ensemble de test, complétés par une inspection des métriques sur l'ensemble d'entraînement afin de caractériser d'éventuels écarts train-test.

### A. Analyse préliminaire des performances des MLP (Q2)

Nous considérons d'abord les résultats sur l'ensemble de test pour le Macro-F1, le Macro-Recall et l'Accuracy, présentés respectivement à la Fig. 24, la Fig. 25 et la Fig. 26.

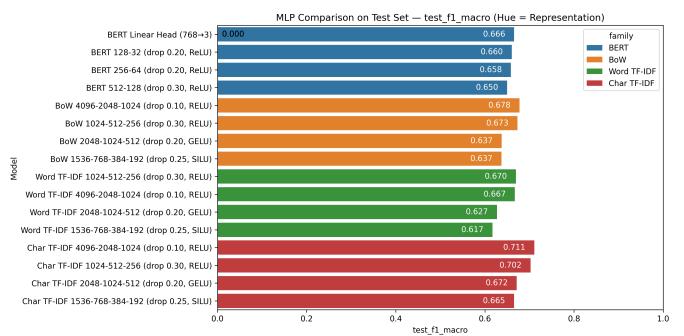


FIGURE 24 – Comparaison des performances MLP sur test — Macro-F1 (teinte = famille de représentation).

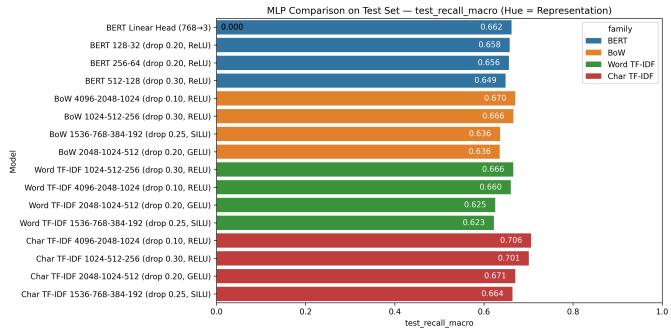


FIGURE 25 – Comparaison des performances MLP sur test — Macro-Recall (teinte = famille de représentation).

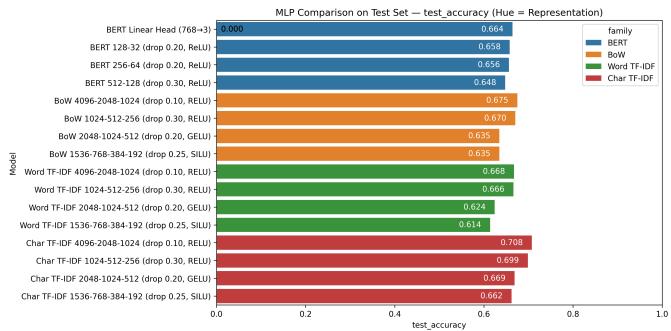


FIGURE 26 – Comparaison des performances MLP sur test — Accuracy (teinte = famille de représentation).

À travers ces trois graphiques, un motif cohérent émerge : les configurations basées sur **Char TF-IDF** dominent globalement les performances des MLP. En particulier, le MLP utilisant Char TF-IDF (4096–2048–1024, dropout 0.10, ReLU) atteint les valeurs les plus élevées sur les trois métriques (Macro-F1 = 0.711, Macro-Recall = 0.706, Accuracy = 0.708). La variante Char TF-IDF (1024–512–256, dropout 0.30, ReLU) reste très proche (0.702 / 0.701 / 0.699). Ce comportement suggère que, pour les tweets, les signaux au niveau des caractères (abréviations, variantes orthographiques, motifs fréquents de suffixes/préfixes, et écriture informelle) demeurent hautement informatifs même lorsque le classifieur est non linéaire.

En revanche, les familles **BoW** et **Word TF-IDF** occupent un second niveau, avec des meilleurs cas autour de 0.67–0.68. Enfin, les variantes **basées sur BERT** (têtes MLP au-dessus d'embeddings) sont compétitives mais restent plus limitées dans ce cadre (environ 0.65–0.67), ce qui est cohérent avec l'utilisation d'embeddings contextuels sans fine-tuning complet de l'encodeur de bout en bout.

#### B. Comparaison entre familles de représentations MLP (qualité prédictive)

En considérant les trois métriques de test rapportées à la Fig. 24–26, la comparaison au niveau des familles est stable : **Char TF-IDF** est la représentation qui produit les MLP les plus efficaces et ce, de manière cohérente.

1) **Char TF-IDF (meilleure famille en qualité prédictive)**: Globalement, les MLP utilisant **Char TF-IDF** obtiennent les meilleures performances. Le meilleur modèle est Char TF-IDF 4096–2048–1024 (dropout 0.10, ReLU), avec environ Macro-F1 ≈ 0.711, Macro-Recall ≈ 0.706, et Accuracy ≈ 0.708. En outre, cette famille présente la meilleure performance moyenne (environ Macro-F1 ≈ 0.688).

Ce comportement est cohérent avec le domaine des tweets : la modélisation au niveau des caractères est typiquement plus robuste aux fautes d'orthographe, aux abréviations, aux mots allongés (par exemple, “soooo”), aux hashtags et aux variations d'écriture. Même si un mot est incomplet ou non standard, le modèle peut tout de même capturer des motifs sous-lexicaux informatifs et préserver un signal de sentiment stable.

2) **BoW et Word TF-IDF (performance intermédiaire, BoW légèrement supérieur)**: Les MLP basés sur **BoW** et **Word TF-IDF** sous-performent par rapport à Char TF-IDF. Dans cette comparaison, la meilleure configuration BoW atteint environ Macro-F1 ≈ 0.678, tandis que la meilleure configuration Word TF-IDF atteint environ Macro-F1 ≈ 0.670.

Une interprétation simple est que, dans des textes très courts, détecter la présence/l'absence de termes saillants (BoW) peut être aussi efficace que d'appliquer une pondération plus fine fondée sur la rareté au niveau des mots (Word TF-IDF). De plus, les représentations au niveau des mots sont moins robustes aux abréviations, au bruit orthographique et à l'écriture informelle, que Char TF-IDF gère plus naturellement grâce à sa granularité.

3) **BERT (compétitif, mais non supérieur à Char TF-IDF sous ce régime)**: Au sein des modèles **basés sur BERT**, le meilleur résultat est atteint par l'option la plus simple : BERT + tête linéaire (768→3), avec environ Macro-F1 ≈ 0.666. Lorsque des couches MLP supplémentaires et du dropout sont ajoutés au-dessus des embeddings, la performance sur test ne s'améliore pas et diminue typiquement légèrement. Une interprétation cohérente est que l'espace d'embeddings BERT fournit déjà une séparabilité raisonnable des classes avec un classifieur linéaire, et qu'augmenter la capacité de la tête ajoute des paramètres qui ne se traduisent pas par une meilleure généralisation sous le régime expérimental actuel.

Afin de mieux caractériser le comportement de généralisation, il est utile de contraster ces résultats de test avec les métriques sur l'ensemble d'entraînement pour les mêmes architectures, présentées à la Fig. 27, la Fig. 28 et la Fig. 29.

Ces graphiques d'entraînement mettent en évidence un phénomène important : les MLP sur **BoW / Word TF-IDF / Char TF-IDF** atteignent souvent des scores d'entraînement extrêmement élevés (dans plusieurs cas approchant 1.0), même pour des architectures relativement petites. Cela indique qu'avec des représentations creuses et de grande dimension, un MLP peut ajuster l'ensemble d'entraînement très efficacement (c'est-à-dire mémoriser des combinaisons fréquentes de n-grammes). Cependant, cette facilité d'ajustement ne garantit pas une amélioration proportionnelle sur l'ensemble de test. Autrement dit, la forte performance sur test de Char TF-IDF

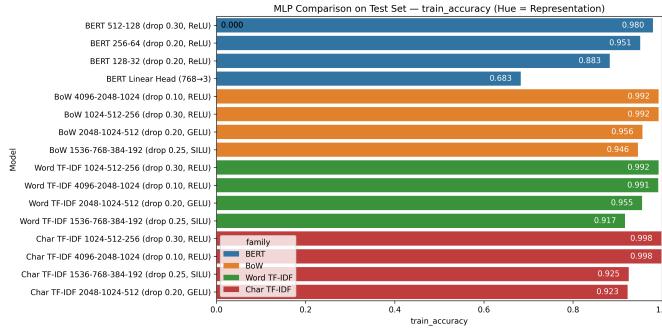


FIGURE 27 – Comparaison des performances MLP sur l’entraînement — Accuracy (teinte = famille de représentation).

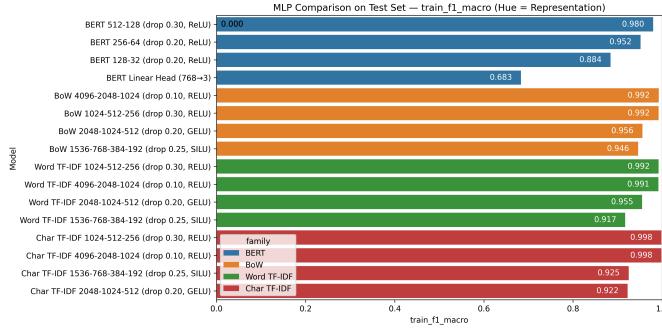


FIGURE 28 – Comparaison des performances MLP sur l’entraînement — Macro-F1 (teinte = famille de représentation).

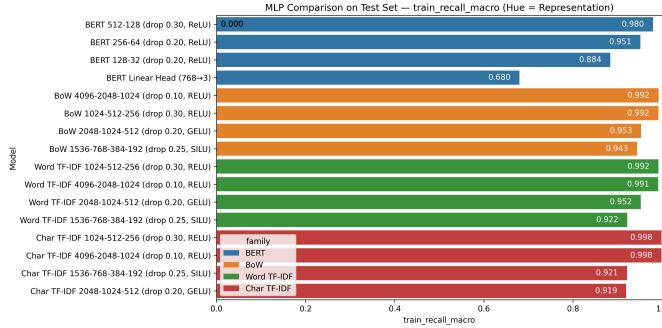


FIGURE 29 – Comparaison des performances MLP sur l’entraînement — Macro-Recall (teinte = famille de représentation).

+ MLP coexiste avec un écart train–test clair, ce qui motive l’interprétation du résultat comme un compromis : forte capacité d’ajustement mais risque structurel de surapprentissage.

Cela aide également à expliquer pourquoi, au sein de la famille Char TF-IDF, l’augmentation de la profondeur ne produit pas systématiquement des gains proportionnels en performance sur test. Une fois que les métriques d’entraînement approchent un plafond, les améliorations dépendent principalement de la généralisation plutôt que d’une capacité supplémentaire.

Enfin, si l’objectif est d’identifier le meilleur MLP unique en Q2 en termes de performance prédictive sur l’ensemble

de test, la référence est le MLP avec **Char TF-IDF (4096-2048-1024, dropout 0.10, ReLU)**, qui surpassé la meilleure variante basée sur BERT (BERT + tête linéaire). Cependant, l’intégration de la perspective train–test suggère une limitation pratique : les MLP avec des entrées TF-IDF (y compris Char TF-IDF) montrent un ajustement d’entraînement très fort, ce qui rend conseillé de les traiter comme des modèles qui requièrent un contrôle explicite de la généralisation (par exemple, régularisation, early stopping, et/ou contraintes de capacité effective) si la robustesse hors échantillon est un objectif principal. À l’inverse, la configuration **BERT + tête linéaire** se comporte comme une alternative plus conservatrice : sa performance sur test est plus faible, mais sa simplicité réduit les incitations à augmenter la complexité de la tête, ce qui est précisément là où la dégradation de performance est observée dans le cadre actuel.

### C. Considération comparative

Le Tableau I fournit une comparaison consolidée entre les pipelines *classiques* évalués en Q0–Q1 et les modèles *MLP* évalués en Q2, en utilisant les mêmes métriques sur l’ensemble de test (Accuracy, Macro-Recall, et Macro-F1) et en classant les modèles par Macro-F1. Une première observation est que plusieurs approches classiques restent hautement compétitives et, dans ce cadre expérimental, surpassent même les contreparties MLP non linéaires. En particulier, le groupe le plus performant est dominé par des pipelines classiques basés sur TF-IDF (en particulier au niveau des caractères) combinés à des classificateurs solides tels que Random Forest, Linear SVM et Logistic Regression, qui occupent les premiers rangs du tableau.

Dans le même temps, la meilleure configuration MLP (Char TF-IDF avec une architecture modérément profonde) atteint un Macro-F1 proche des meilleures baselines classiques, indiquant que des modèles non linéaires peuvent exploiter des représentations robustes au niveau des caractères pour s’approcher des performances des méthodes classiques les plus fortes. Cependant, le tableau suggère également qu’augmenter uniquement la capacité du modèle ne garantit pas un gain systématique par rapport à des pipelines classiques bien régularisés dans des régimes creux de grande dimension, où les modèles linéaires sont particulièrement efficaces.

Il est important d’interpréter cette comparaison comme un instantané intermédiaire plutôt que comme une conclusion finale concernant les approches neuronales. La branche basée sur Transformer n’a pas encore été pleinement exploitée : les expériences BERT actuelles reposent sur des embeddings fixes avec des têtes peu profondes, et l’étape de fine-tuning avec LoRA (prévue comme prochaine étape) devrait mieux adapter la représentation au jeu de données cible. Il est donc raisonnable d’anticiper qu’un Transformer ajusté (ou une variante MLP/Transformer avec des représentations adaptées) puisse réduire l’écart restant et rivaliser plus directement avec les baselines classiques les plus fortes (notamment le groupe de tête rapporté dans le Tableau I). En ce sens, les résultats classiques établissent un point de référence rigoureux

et exigeant, tandis que les résultats de fine-tuning à venir détermineront si des représentations apprises peuvent dépasser de manière cohérente cette baseline sous le même protocole d'évaluation.

## VI. Q4 : ANALYSE COMPARATIVE AVEC DES GRANDS MODÈLES DE LANGAGE (LLMs)

Cette analyse se concentre sur la comparaison des performances de l'architecture optimisée basée sur BERT avec un grand modèle de langage (LLM) moderne. L'objectif fondamental est d'évaluer si un modèle génératif généraliste, opérant uniquement en mode inférence, est capable de rivaliser avec ou de surpasser l'efficacité d'un encodeur spécialisé ayant subi un ajustement fin.

### A. Méthodologie : Inférence générative

Pour cette comparaison, gemma-3-4b-it a été sélectionné, un modèle ajusté par instruction de la famille Gemma de Google avec environ 4 milliards de paramètres. Contrairement à la baseline BERT considérée à ce stade—qui utilise des embeddings BERT fixes avec un classifieur linéaire simple (LinearHead(768→3))—le LLM est évalué via du Few-Shot Prompting au moyen de l'API Google GenAI, sans aucune mise à jour de paramètres.

Le dispositif expérimental a été défini comme suit :

- **Ingénierie de prompt** : Le modèle a été explicitement instruit d'agir comme un "expert en analyse de sentiment".
- **Contexte few-shot** : Pour guider le raisonnement du modèle sans mise à jour des poids, trois exemples étiquetés (un pour chaque classe : Positive, Negative, Neutral) ont été fournis dans la fenêtre de contexte du prompt avant la requête cible.
- **Contraintes de sortie** : La génération a été restreinte à produire des étiquettes d'un seul mot mappées à nos classes numériques (0, 1, 2).
- **Sous-ensemble d'évaluation** : En raison des limites de débit de l'API et des contraintes de latence, l'évaluation a été réalisée sur un sous-ensemble représentatif de 1,000 échantillons de l'ensemble de test.

### B. Évaluation des performances

Nous comparons les capacités génératives de Gemma au modèle BERT Linear Head, qui a obtenu la meilleure performance parmi les configurations BERT testées dans la section précédente.

1) **Résultats LLM (Gemma-3-4b-it)**: La Fig. 30 présente le rapport de classification pour le modèle génératif. Gemma a atteint une **Accuracy de 0.632** et un **Macro F1 de 0.630**.

La matrice de confusion révèle une faiblesse spécifique : une tendance à mal classifier les tweets polarisés (Positif/Négatif) comme Neutres. Ce comportement est souvent attribué à l'alignement de sécurité (RLHF) des modèles ajustés par instruction, qui les biaise vers des réponses neutres ou peu engageantes face à l'ambiguïté.

**Model Performance Report: Gemini gemma-3-4b-it**

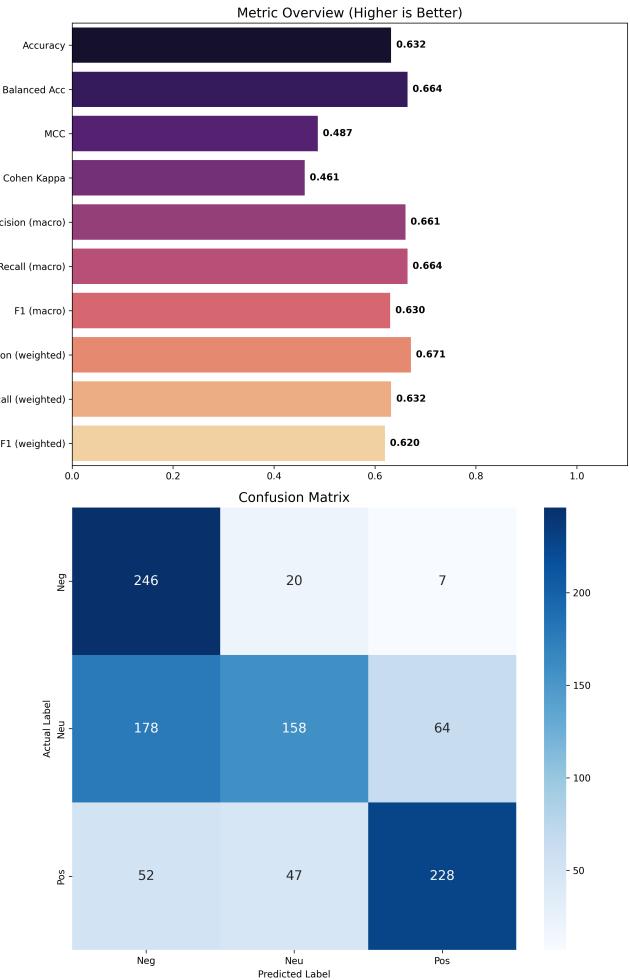


FIGURE 30 – Rapport de performance pour Gemma-3-4b-it (inférence few-shot). Le modèle montre une performance équilibrée mais modérée, avec une confusion notable entre les sentiments polarisés et la classe neutre.

2) **Résultats basés sur BERT**: La Fig. 31 affiche les performances du BERT Linear Head ajusté. Ce modèle a atteint une **Accuracy de 0.664** et un **Macro F1 de 0.666**.

Le modèle BERT démontre une diagonale plus marquée dans la matrice de confusion, indiquant un pouvoir discriminant supérieur pour cette distribution de données spécifique par rapport au LLM en few-shot.

### C. Discussion : Spécialisation vs. Généralisation

La comparaison entre l'encodeur spécialisé (BERT) et le décodeur généraliste (Gemma) offre des enseignements précieux, résumés dans le Tableau II.

1) **Écart de performance** : Le modèle BERT ajusté surpassé le LLM d'environ **3.6% en Macro F1**. Cela confirme que, pour ce jeu de données, un modèle plus petit adapté au domaine spécifique est plus efficace

TABLE I – Comparaison sur l’ensemble de test entre les 12 pipelines classiques (Q0–Q1) et les 16 modèles MLP (Q2). Les modèles sont classés par Macro-F1 (décroissant). Le meilleur global, le meilleur classique et le meilleur MLP sont mis en évidence en gras.

| Rang     | Approche  | Représentation | Modèle                                  | Acc.         | Macro-R      | Macro-F1     |
|----------|-----------|----------------|---|--------------|--------------|--------------|
| 1        | Classique | TF-IDF (char)  | <b>Random Forest</b>                    | <b>0.720</b> | <b>0.719</b> | <b>0.722</b> |
| 2        | Classique | TF-IDF (char)  | Linear SVM                              | 0.713        | 0.710        | 0.715        |
| 3        | Classique | TF-IDF (word)  | Random Forest                           | 0.711        | 0.712        | 0.713        |
| 4        | Classique | TF-IDF (char)  | Logistic Regression                     | 0.711        | 0.705        | 0.712        |
| <b>5</b> | MLP       | Char TF-IDF    | <b>4096-2048-1024 (drop 0.10, ReLU)</b> | <b>0.708</b> | <b>0.706</b> | <b>0.711</b> |
| 6        | Classique | BoW            | Random Forest                           | 0.708        | 0.709        | 0.710        |
| 7        | Classique | TF-IDF (word)  | Logistic Regression                     | 0.705        | 0.701        | 0.707        |
| 8        | Classique | BoW            | Linear SVM                              | 0.703        | 0.698        | 0.705        |
| 9        | Classique | TF-IDF (word)  | Linear SVM                              | 0.703        | 0.698        | 0.705        |
| 10       | Classique | BoW            | Logistic Regression                     | 0.703        | 0.696        | 0.704        |
| 11       | MLP       | Char TF-IDF    | 1024-512-256 (drop 0.30, ReLU)          | 0.699        | 0.701        | 0.702        |
| 12       | MLP       | BoW            | 4096-2048-1024 (drop 0.10, ReLU)        | 0.675        | 0.670        | 0.678        |
| 13       | MLP       | BoW            | 1024-512-256 (drop 0.30, ReLU)          | 0.670        | 0.666        | 0.673        |
| 14       | MLP       | Char TF-IDF    | 2048-1024-512 (drop 0.20, GELU)         | 0.669        | 0.671        | 0.672        |
| 15       | Classique | TF-IDF (char)  | MultinomialNB                           | 0.670        | 0.661        | 0.670        |
| 16       | MLP       | Word TF-IDF    | 1024-512-256 (drop 0.30, ReLU)          | 0.666        | 0.666        | 0.670        |
| 17       | MLP       | Word TF-IDF    | 4096-2048-1024 (drop 0.10, ReLU)        | 0.668        | 0.660        | 0.667        |
| 18       | MLP       | BERT           | Linear Head (768→3)                     | 0.664        | 0.662        | 0.666        |
| 19       | MLP       | Char TF-IDF    | 1536-768-384-192 (drop 0.25, SiLU)      | 0.662        | 0.664        | 0.665        |
| 20       | Classique | BoW            | MultinomialNB                           | 0.660        | 0.658        | 0.664        |
| 21       | MLP       | BERT           | 128-32 (drop 0.20, ReLU)                | 0.658        | 0.658        | 0.660        |
| 22       | MLP       | BERT           | 256-64 (drop 0.20, ReLU)                | 0.656        | 0.656        | 0.658        |
| 23       | Classique | TF-IDF (word)  | MultinomialNB                           | 0.653        | 0.652        | 0.656        |
| 24       | MLP       | BERT           | 512-128 (drop 0.30, ReLU)               | 0.648        | 0.649        | 0.650        |
| 25       | MLP       | BoW            | 2048-1024-512 (drop 0.20, GELU)         | 0.635        | 0.636        | 0.637        |
| 26       | MLP       | BoW            | 1536-768-384-192 (drop 0.25, SiLU)      | 0.635        | 0.636        | 0.637        |
| 27       | MLP       | Word TF-IDF    | 2048-1024-512 (drop 0.20, GELU)         | 0.624        | 0.625        | 0.627        |
| 28       | MLP       | Word TF-IDF    | 1536-768-384-192 (drop 0.25, SiLU)      | 0.614        | 0.623        | 0.617        |

TABLE II – Comparaison directe : encodeur ajusté vs. LLM few-shot

| Modèle                 | Paramètres  | Méthode            | Macro F1     |
|------------------------|-------------|--------------------|--------------|
| <b>BERT (Linéaire)</b> | <b>110M</b> | <b>Fine-Tuning</b> | <b>0.666</b> |
| Gemma-3-4b-it          | 4B          | Few-Shot           | 0.630        |

qu’un modèle massif s’appuyant sur des connaissances générales (few-shot).

- 2) **Efficacité des ressources :** L’écart d’efficacité est substantiel. Le modèle BERT (110M paramètres) peut être déployé pour l’inférence avec une faible latence sur du matériel standard. En revanche, le LLM (4B paramètres) requiert une VRAM significative ou une dépendance à une API.

## VII. Q5-EMBEDDINGS BERT VS. REPRÉSENTATIONS BRUTES

Bien que la différence impliquée par la représentation du texte brut au moyen de l’une des méthodes classiques de vectorisation basées sur la fréquence et l’utilisation d’embeddings BERT ait déjà été abordée dans la section sur l’entraînement du MLP, il est pertinent de discuter plus en détail la différence entre ces deux formes de représentation numérique des éléments textuels : les approches basées sur des comptages d’occurrences en fréquence et les représentations basées sur les transformateurs.

Le premier élément qui ressort parmi ces différences est que BERT est un encodeur de transformeur pré-entraîné avec

l’objectif de produire des représentations contextuelles bidirectionnelles. Cela signifie que les représentations de tokens résultantes sont encodées en utilisant à la fois le contexte gauche et le contexte droit, c’est-à-dire les mots qui apparaissent avant et après chaque token dans la séquence tokenisée [18].

BERT produit des représentations contextualisées au niveau des tokens qui, via une opération de pooling, donnent un embedding dense de taille fixe par texte. Cet embedding correspond à un vecteur de dimension 768, qui devient la dimension d’entrée du classifieur. Cette différence significative de dimensionnalité entre l’utilisation d’un modèle de vectorisation basé sur la fréquence et l’utilisation d’embeddings BERT est le premier facteur différenciant qui motive l’utilisation de ces derniers plutôt que l’approche traditionnelle.

Pour mettre en évidence l’effet que ce changement de dimensionnalité a sur l’entraînement du modèle classifieur, il est proposé d’isoler le MLP du pipeline et d’analyser le nombre de paramètres du réseau en fonction de son entrée, et de comparer quantitativement la différence entre la taille d’un réseau qui reçoit en entrée une représentation de vectoriseur basée sur la fréquence, comme le cas au niveau des caractères utilisé lors de l’entraînement, et un réseau qui utilise des embeddings BERT. Pour cet exercice, l’équation de comptage de paramètres Eq. (6) pour une couche entièrement connectée est utilisée ; elle consiste en une matrice de poids et un terme de biais, et est déduite des informations fournies par PyTorch [19].

### Model Performance Report: bert\_mlp\_baseline

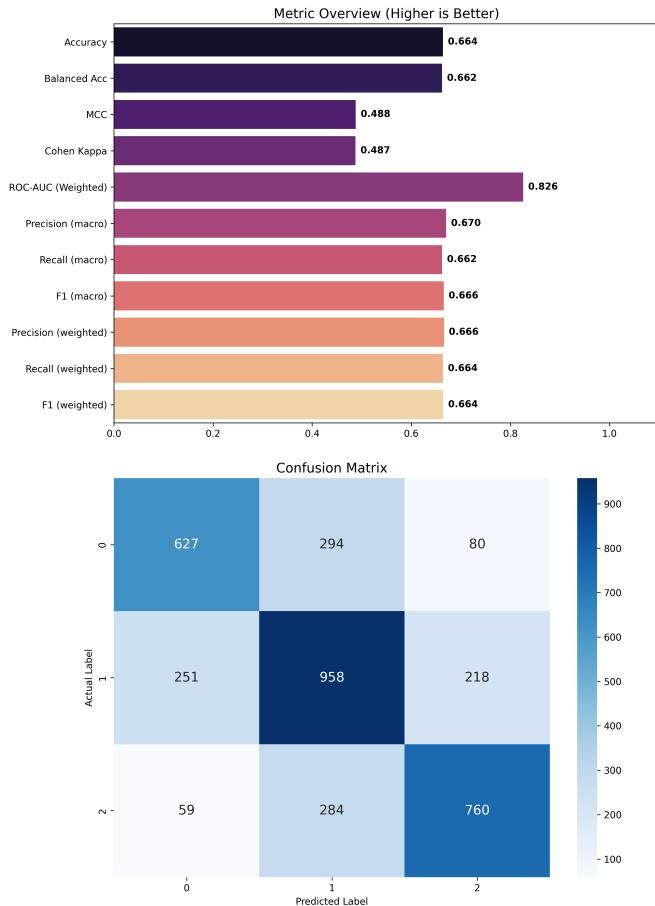


FIGURE 31 – Rapport de performance pour la baseline BERT Linear Head.

$$\text{params} = (\text{infeatures} \cdot \text{outfeatures}) + \text{outfeatures} \quad (6)$$

En appliquant cette équation à un réseau de neurones avec la structure `INPUT_DIM` → 1024 → 512 → 256 → 3, et en utilisant Eq. (7), un total de 10,897,923 paramètres est obtenu pour le MLP, alors qu’avec la même structure mais avec une couche d’entrée adaptée à un embedding BERT, comme montré dans Eq. (8), 1,444,355 paramètres sont obtenus, une réduction de  $\approx 86.75\%$ . Cette réduction est hautement pertinente lorsqu’on vise la généralisation et la stabilité de l’entraînement, car elle diminue le risque de surapprentissage : la relation entre les données d’entraînement et les paramètres rend moins probable la mémorisation du bruit ou de dynamiques parasites. De plus, on peut obtenir un modèle moins sensible aux stratégies de régularisation et, en se concentrant uniquement sur l’entraînement du MLP, le coût computationnel de l’entraînement est réduit, un effet qui devient évident dans l’entraînement réalisé pendant la phase de test détaillée en Q2 et Q3.

$$\begin{aligned}
 P_1 (10000 \rightarrow 1024) &= 10000 \cdot 1024 + 1024 = 10,241,024, \\
 P_2 (1024 \rightarrow 512) &= 1024 \cdot 512 + 512 = 524,800, \\
 P_3 (512 \rightarrow 256) &= 512 \cdot 256 + 256 = 131,328, \\
 P_4 (256 \rightarrow 3) &= 256 \cdot 3 + 3 = 771, \\
 P_{\text{total}} &= P_1 + P_2 + P_3 + P_4 = 10,897,923. \tag{7}
 \end{aligned}$$

$$\begin{aligned}
 P_1 (768 \rightarrow 1024) &= 768 \cdot 1024 + 1024 = 787,456, \\
 P_2 (1024 \rightarrow 512) &= 1024 \cdot 512 + 512 = 524,800, \\
 P_3 (512 \rightarrow 256) &= 512 \cdot 256 + 256 = 131,328, \tag{8} \\
 P_4 (256 \rightarrow 3) &= 256 \cdot 3 + 3 = 771, \\
 P_{\text{total}} &= P_1 + P_2 + P_3 + P_4 = 1,444,355.
 \end{aligned}$$

La sortie des modèles de vectorisation basés sur la fréquence est généralement une matrice creuse, ce qui implique que l’information est effectivement concentrée sur seulement quelques indices actifs. Cela est directement lié non seulement à la parcimonie déjà mentionnée, mais aussi à la présence implicite de valeurs nulles dans la matrice CSR : même si elles ne sont pas explicitement stockées en mémoire, elles exposent le modèle à une sensibilité considérable à la largeur du réseau et aux méthodes de régularisation, nécessitant un contrôle strict pour éviter le surentraînement. Cela complique clairement non seulement l’obtention de performances optimales, mais aussi la définition de la structure du réseau, puisque de subtiles variations de la largeur des couches peuvent avoir un effet substantiel sur le risque de surapprentissage. En revanche, lorsqu’on utilise des embeddings BERT, la matrice est dense et de faible dimension. Cela réduit typiquement la taille du classifieur en aval nécessaire, et rend donc la sélection d’une structure de MLP et le choix des paramètres de régularisation plus faciles et moins critiques. La simplicité dans la sélection de la structure du réseau de neurones augmente encore lorsqu’on considère le troisième facteur clé proposé pour différencier les avantages des embeddings par rapport aux méthodes classiques, à savoir la sensibilité de chaque méthode au contexte.

La sensibilité au contexte est peut-être la différence la plus perceptible et la plus marquée entre les deux méthodes, et elle favorise la mise en œuvre de pipelines avec des embeddings BERT plutôt qu’avec des vectoriseurs basés sur des comptages de fréquence. Cela s’explique par le fait que BERT est solidement pré-entraîné sur des corpus linguistiques beaucoup plus vastes et, en outre, il est spécifiquement conçu pour intégrer le contexte dans la définition vectorielle des mots d’un document, grâce à la structure et au fonctionnement détaillés dans la Section Q4. En revanche, les stratégies classiques de vectorisation ne peuvent capturer qu’une information contextuelle superficielle obtenue par des comptages de fréquence ou, au mieux, par l’analyse dérivée de cet espace de fréquence, ce qui limite leur représentation d’aspects sémantiques fondamentaux pouvant être décisifs lors de la classification d’un court message dans une catégorie de sentiment/polarité. En

outre, le fait qu'un embedding BERT soit si robuste, condensé et riche en information permet l'exploration de modèles plus simples pour la classification de messages. Autrement dit, dans BERT la complexité est principalement concentrée dans le pré-entraînement du modèle utilisé pour obtenir les embeddings, et le MLP est isolé de cette complexité et n'a besoin que de mapper et de classifier des vecteurs hautement informatifs. En revanche, lorsque des méthodes de vectorisation traditionnelles sont utilisées, une partie de la complexité couverte par le pré-entraînement de BERT doit être prise en charge par le réseau de neurones lui-même, ce qui rend possible, dans le cas BERT, l'utilisation de réseaux avec moins de couches, voire d'une seule tête de classification linéaire, tout en obtenant des résultats comparables à des modèles MLP plus complexes entraînés avec des techniques de vectorisation traditionnelles.

Bien que les bénéfices de l'utilisation de BERT comme extracteur d'embeddings deviennent évidents, il est essentiel de souligner qu'utiliser BERT comme extracteur d'embeddings figé n'est pas la même chose que l'utiliser adapté à la tâche, c'est-à-dire appliquer un fine-tuning qui permet aux représentations internes de s'aligner avec l'objectif de classification et avec la distribution des données d'entraînement. Cela est d'une importance critique, par exemple, dans l'analyse ou les tâches de classification de textes courts comme celle considérée ici, où l'adoption d'une stratégie de fine-tuning telle que LoRA, décrite en Q6, permet d'obtenir de meilleures performances de classifieur que l'utilisation de BERT uniquement pour une extraction d'embeddings figée. C'est grâce à ces stratégies de fine-tuning que le principal inconvénient de BERT en tant qu'extracteur d'embeddings figé peut être atténué par rapport aux vecteurs classiques, à savoir que sa représentation est externe au domaine des données d'entraînement. En d'autres termes, elle est sensible au contexte par la propre définition de BERT, mais pas à des paramètres tels que la fréquence de présence des tokens dans l'ensemble d'entraînement et d'autres propriétés qui apparaissent parce que l'entraînement du transformeur est déjà fixé et, dans l'extraction d'embeddings figée, aucun apprentissage n'a lieu.

Il est finalement conclu que BERT est plus attractif que les méthodes classiques principalement en raison de sa capacité à représenter la sémantique et le contexte, ce que les méthodes classiques ne possèdent pas. Cela est donc associé à une entrée plus dense qui requiert moins de complexité dans le MLP de classification. De plus, la possibilité de mettre en œuvre des stratégies de fine-tuning qui contrebalancent l'effet agnostique au domaine du pré-entraînement de l'extracteur d'embeddings, et qui permettent au domaine du jeu de données d'entraînement de perméer la représentation d'embedding du document, peut améliorer substantiellement les performances du modèle.

### VIII. Q6-ARCHITECTURE DE BERT ET CADRE THÉORIQUE

Les Bidirectional Encoder Representations from Transformers (BERT) représentent un changement de paradigme dans l'apprentissage des représentations du langage. Contrairement

aux modèles précédents qui reposaient sur un traitement unidirectionnel (par exemple, OpenAI GPT) ou sur une concaténation superficielle de modèles indépendants gauche-à-droite et droite-à-gauche (par exemple, ELMo), BERT est conçu pour pré-entraîner des représentations bidirectionnelles profondes à partir de texte non annoté en conditionnant conjointement sur les contextes gauche et droit dans toutes les couches [20].

#### A. Architecture du modèle

L'architecture de BERT est un encodeur Transformer bidirectionnel multi-couches basé sur l'implémentation originale décrite par Vaswani et al. [21]. Le modèle est composé d'une pile de  $L$  couches identiques (blocs Transformer).

Devlin et al. définissent deux tailles de modèle principales afin d'évaluer l'effet de la capacité du modèle [20] :

- **BERT<sub>BASE</sub>** : Défini avec  $L = 12$  couches, une taille cachée de  $H = 768$ , et  $A = 12$  têtes d'auto-attention, totalisant environ 110 millions de paramètres. Cette taille a été choisie pour être comparable à OpenAI GPT pour une évaluation équitable.
- **BERT<sub>LARGE</sub>** : Un réseau significativement plus profond et plus large avec  $L = 24$ ,  $H = 1024$ , et  $A = 16$ , résultant en environ 340 millions de paramètres.

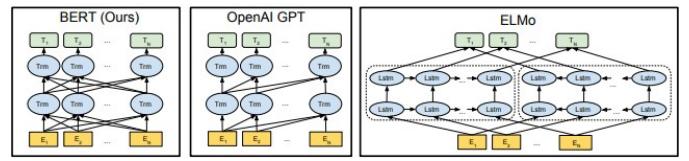


FIGURE 32 – Différences dans les architectures de modèles de pré-entraînement. BERT utilise un Transformer bidirectionnel, ce qui le distingue d'OpenAI GPT et d'ELMo [20].

Au sein de chaque couche, l'architecture emploie deux sous-couches principales : un mécanisme d'Auto-Attention Multi-Têtes et un Réseau Feed-Forward positionnel. De manière cruciale, une connexion résiduelle est employée autour de chacune de ces deux sous-couches, suivie d'une normalisation de couche.

1) *Auto-Attention multi-têtes*: Ce mécanisme permet au modèle d'assister conjointement à des informations provenant de différents sous-espaces de représentation à différentes positions. La fonction d'attention est calculée sur un ensemble de requêtes ( $Q$ ), de clés ( $K$ ), et de valeurs ( $V$ ) regroupées en matrices. La sortie est une attention à produit scalaire mis à l'échelle :

$$\text{Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V, \quad (9)$$

où  $d_k$  est la dimension des vecteurs clés, servant de facteur d'échelle pour prévenir la disparition des gradients dans la fonction softmax.

2) *Réseau Feed-Forward et activation GELU*: Après la sous-couche d'attention, la sortie est traitée par un Réseau Feed-Forward (FFN) entièrement connecté appliquée à chaque

position séparément et de manière identique. Celui-ci consiste en deux transformations linéaires avec une fonction d'activation non linéaire. Une caractéristique distinctive de BERT, comparée au Transformer original qui utilisait ReLU, est l'adoption de la Gaussian Error Linear Unit (GELU). Le FFN est formulé comme suit :

$$\text{FFN}(x) = \text{GELU}(xW_1 + b_1)W_2 + b_2, \quad (10)$$

où  $W_1$  projette l'état caché de  $H$  vers une dimension intermédiaire (3072 pour BERT<sub>BASE</sub>), et  $W_2$  le reprojette vers  $H$ . L'activation GELU, définie comme  $x\Phi(x)$  (où  $\Phi(x)$  est la fonction de répartition gaussienne standard), fournit une non-linéarité plus lisse qui aide l'optimisation dans les architectures profondes.

*3) Connexions résiduelles et normalisation:* Pour permettre l'entraînement de réseaux profonds, la sortie de chaque sous-couche est stabilisée en utilisant des connexions résiduelles et la Normalisation de Couche (LayerNorm) :

$$\text{Output} = \text{LayerNorm}(x + \text{Sublayer}(x)). \quad (11)$$

Contrairement à la Normalisation de Lot, la Normalisation de Couche calcule des statistiques à travers la dimension des caractéristiques pour chaque token indépendamment, la rendant invariante à la taille du lot et à la longueur de la séquence.

### B. Représentation d'entrée

Pour gérer une grande variété de tâches en aval, BERT nécessite une représentation d'entrée flexible capable de traiter à la fois des phrases uniques et des paires de phrases.

*1) Tokenisation WordPiece:* BERT utilise un vocabulaire WordPiece de 30,000 tokens. Cette stratégie de tokenisation atténue le problème hors-vocabulaire (OOV) en décomposant les mots rares en unités sous-mots (par exemple, "playing" → play + #ing).

*2) Somme des embeddings:* La représentation d'entrée pour un token donné est construite en sommant trois embeddings distincts :

$$\mathbf{E} = \mathbf{E}_{token} + \mathbf{E}_{segment} + \mathbf{E}_{position}. \quad (12)$$

- **Embeddings de tokens** : Représentations vectorielles des tokens WordPiece.
- **Embeddings de segments** : Vecteurs appris indiquant si un token appartient à la phrase A ou à la phrase B.
- **Embeddings de position** : Vecteurs appris injectant une information positionnelle absolue, nécessaire en raison de l'invariance par permutation de l'auto-attention.

Comme illustré à la Fig. 33, chaque token d'entrée est mappé vers un embedding qui combine trois composantes : les embeddings de tokens (identité lexicale), les embeddings de segments (appartenance à une phrase/segment), et les embeddings de position (ordre des tokens). En outre, chaque séquence commence avec le token spécial de classification [CLS]. L'état caché final associé à [CLS] ( $C \in \mathbb{R}^H$ ) est couramment utilisé comme une représentation agrégée de

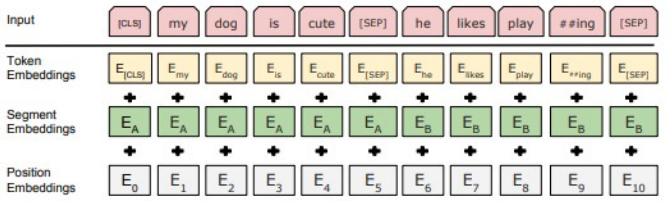


FIGURE 33 – Représentation d'entrée de BERT [20].

l'ensemble de la séquence pour la classification en aval. Un token séparateur [SEP] est utilisé pour délimiter (ou séparer) des séquences lorsque nécessaire.

Chaque séquence commence avec un token spécial de classification ([CLS]). L'état caché final correspondant à ce token ( $C \in \mathbb{R}^H$ ) est utilisé comme la représentation agrégée de la séquence pour les tâches de classification. Un token séparateur ([SEP]) est utilisé pour distinguer entre les séquences.

### C. Objectifs de pré-entraînement

BERT est pré-entraîné sur deux tâches non supervisées en utilisant le BooksCorpus (800M mots) et Wikipédia anglais (2,500M mots).

*1) Modèle de langage masqué (MLM):* Pour atteindre une bidirectionnalité profonde sans permettre aux tokens de "se voir eux-mêmes" dans des contextes multi-couches, BERT masque 15% des tokens d'entrée aléatoirement. L'objectif est de prédire l'identifiant de vocabulaire original du mot masqué en se basant uniquement sur son contexte. Pour atténuer le décalage entre le pré-entraînement et le fine-tuning (où le token [MASK] n'apparaît pas), le générateur de données d'entraînement emploie la stratégie suivante pour les tokens sélectionnés :

- 80% du temps : Remplacer par le token [MASK].
- 10% du temps : Remplacer par un token aléatoire.
- 10% du temps : Conserver le token original inchangé.

*2) Prédiction de la phrase suivante (NSP):* De nombreuses tâches en aval (par exemple, QA et NLI) requièrent de comprendre la relation entre deux phrases. Pour modéliser cela, BERT est entraîné sur une tâche de classification binaire afin de prédire si une phrase  $B$  est la véritable phrase suivante qui suit la phrase  $A$  (IsNext) ou une phrase aléatoire du corpus (NotNext).

### D. Stratégies d'application

Comme démontré dans les études d'ablation par Devlin et al., BERT peut être appliqué aux tâches en aval de deux manières : le *fine-tuning*, où tous les paramètres sont mis à jour de bout en bout, et l'*approche basée sur des caractéristiques*, où des caractéristiques fixes sont extraites de couches spécifiques.

Les résultats expérimentaux montrés à la Fig. 34 indiquent que la concaténation des représentations de tokens provenant des quatre couches cachées supérieures donne des performances compétitives avec le fine-tuning (96.1 F1 vs 96.4 F1). Dans notre cadre théorique, cela justifie l'utilisation d'embeddings extraits comme entrées robustes pour des classificateurs en aval.

| System   | Dev F1 | Test F1     |
|--|--------|-------------|
| ELMo (Peters et al., 2018a)                    | 95.7   | 92.2        |
| CVT (Clark et al., 2018)                       | -      | 92.6        |
| CSE (Akbik et al., 2018)                       | -      | <b>93.1</b> |
| Fine-tuning approach                           |        |             |
| BERT <sub>LARGE</sub>                          | 96.6   | 92.8        |
| BERT <sub>BASE</sub>                           | 96.4   | 92.4        |
| Feature-based approach (BERT <sub>BASE</sub> ) |        |             |
| Embeddings                                     | 91.0   | -           |
| Second-to-Last Hidden                          | 95.6   | -           |
| Last Hidden                                    | 94.9   | -           |
| Weighted Sum Last Four Hidden                  | 95.9   | -           |
| Concat Last Four Hidden                        | 96.1   | -           |
| Weighted Sum All 12 Layers                     | 95.5   | -           |

FIGURE 34 – Résultats de l’approche basée sur des caractéristiques sur CoNLL-2003 NER.

## IX. Q7-AJUSTEMENT FIN AVEC LORA POUR LA CLASSIFICATION DE SENTIMENT

LoRA (*Low-Rank Adaptation*) est une méthode d’*Ajustement fin efficace en paramètres* (PEFT) conçue pour adapter de grands modèles pré-entraînés (par exemple, des Transformers) sans mettre à jour tous les paramètres. Au lieu d’effectuer un ajustement fin complet du backbone, LoRA fige les poids d’origine et injecte de petites matrices de faible rang entraînables dans des couches sélectionnées (typiquement des projections linéaires au sein de l’attention). Cela réduit substantiellement le nombre de paramètres entraînables et le coût d’entraînement/mémoire, tout en préservant la capacité du modèle à se spécialiser sur la tâche cible.

### A. Aperçu de l’implémentation (pipeline notebook)

L’implémentation suit un flux de travail standard de classification basé sur Transformer : (1) préparation du jeu de données et découpage *train/validation/test*; (2) tokenisation avec le tokenizer du modèle de base; (3) chargement d’un modèle de classification de séquence pré-entraîné (AutoModelForSequenceClassification); (4) injection d’adaptateurs LoRA via `get_peft_model`; (5) entraînement avec Trainer et TrainingArguments; et (6) évaluation finale sur l’ensemble de test, incluant des métriques et des rapports diagnostiques (voir Fig. 35).

### B. Configuration LoRA (justification)

La configuration LoRA est définie comme suit :

- `task_type=TaskType.SEQ_CLS` configure PEFT pour la classification de séquence (sortie logits), plutôt que pour des gabarits orientés génération.
- `target_modules=["query", "value"]` injecte LoRA dans des projections d’attention très influentes (Q/V), un choix courant qui équilibre la capacité d’adaptation et le coût.

Model Performance Report: BERT + LoRA (SEQ\_CLS)

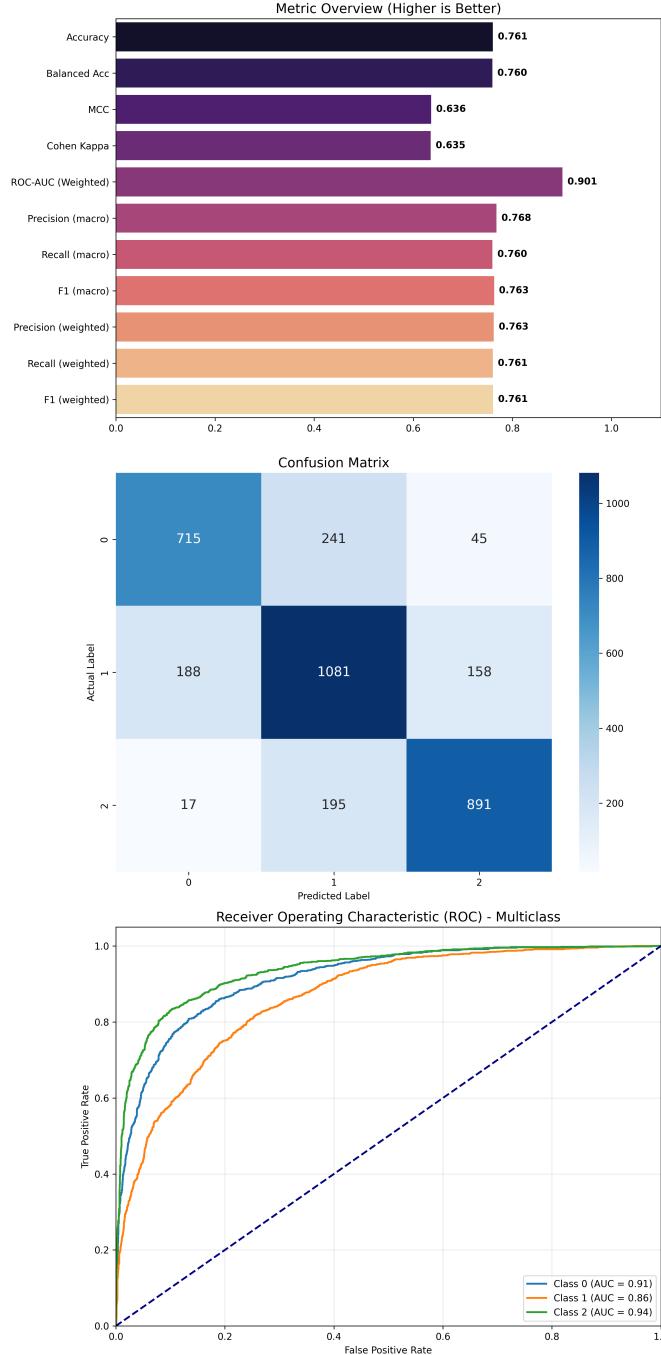


FIGURE 35 – Rapport d’évaluation complet pour BERT + LoRA (SEQ\_CLS) sur l’ensemble de test (résumé des métriques, matrice de confusion et analyse ROC).

- `r=8` fixe le rang de l’adaptateur (capacité). Un rang modéré est typiquement suffisant pour la classification de sentiment tout en limitant les paramètres et le risque de surapprentissage.
- `lora_alpha=16` met à l’échelle la contribution LoRA (souvent proportionnelle à  $\alpha/r$ ), soutenant des mises à

jour stables.

- **lora\_dropout=0.1** régularise le chemin LoRA, réduisant le surapprentissage sur des jeux de données non massifs.

#### C. Résultats sur l'ensemble de test : BERT + LoRA (SEQ\_CLS)

Le modèle **BERT + LoRA (SEQ\_CLS)** atteint :

- Accuracy = 0.761
- Macro-Recall = 0.760
- Macro-F1 = 0.763

De plus, il rapporte un ROC-AUC pondéré = 0.901, avec des valeurs d'AUC par classe de 0.91 (classe 0), 0.86 (classe 1) et 0.94 (classe 2), indiquant une forte séparabilité globale et une discrimination particulièrement robuste pour la classe 2 (voir Fig. 35).

#### D. Comparaison directe avec la baseline d'embeddings BERT (LinearHead 768→3)

Comparé à la baseline précédente (*Approach = MLP ; Representation = BERT ; Model = LinearHead(768→3)*), qui rapporte Acc = 0.664, Macro-R = 0.662 et Macro-F1 = 0.666, LoRA améliore de manière cohérente toutes les métriques principales :

- **Accuracy** : 0.761 vs. 0.664 ⇒ +0.097 (environ +14.6% relatif)
- **Macro-Recall** : 0.760 vs. 0.662 ⇒ +0.098 (environ +14.8% relatif)
- **Macro-F1** : 0.763 vs. 0.666 ⇒ +0.097 (environ +14.6% relatif)

Ce gain est cohérent avec l'objectif de LoRA : au lieu d'entraîner uniquement une tête peu profonde sur des embeddings figés, l'encodeur est légèrement adapté via des mises à jour de faible rang dans des modules d'attention clés, améliorant la représentation interne pour la classification de sentiment. D'un point de vue analyse d'erreurs, la matrice de confusion de la Fig. 35 reste fortement diagonale (forte classification correcte à travers les classes), tandis que les erreurs principales se concentrent vers la classe 1, suggérant que la classe 1 se comporte comme une région intermédiaire plus ambiguë.

#### E. Positionnement du projet

Avec Macro-F1 = 0.763, BERT + LoRA (SEQ\_CLS) dépasse à la fois la baseline BERT + LinearHead et les meilleurs pipelines classiques/MLP rapportés plus tôt (cf. Table I). Par conséquent, BERT + LoRA (SEQ\_CLS) se positionne actuellement comme le modèle le plus performant de ce projet, fournissant la meilleure performance sur test avec des améliorations claires des métriques macro, qui sont les plus informatives en présence de déséquilibre de classes.

## X. CONCLUSION

Cette section vise à établir un ensemble de conclusions concernant la mise en œuvre de méthodes de classification de sentiment pour des textes courts en anglais. Tout d'abord, il est important de souligner l'analyse réalisée lors de la

phase préliminaire du jeu de données et de ses principales caractéristiques. À partir de cette analyse, on observe que, compte tenu des informations d'entraînement disponibles, la variable principale d'intérêt pour l'entraînement de ce type de classifieur est sans aucun doute le texte traité après suppression des stopwords, principalement parce que la distribution des classes par rapport aux autres variables incluses dans le jeu de données reste soit constante, comme dans le cas des tranches d'âge, soit pourrait impliquer un risque accru de mémoriser des motifs incorrects, puisque les éléments de chaque classe au sein de la catégorie non considérée ne sont pas représentatifs de la compilation globale, comme dans le cas de la variable pays.

En ce qui concerne l'entraînement des modèles classiques d'apprentissage automatique, les résultats de classification atteignent de hautes performances, principalement grâce à la prise en compte de multiples pipelines de modèles combinant un vectoriseur avec le classifieur et les variations correspondantes de leurs grilles d'hyperparamètres. Cela conduit à un total de 956 modèles entraînés, parmi lesquels les configurations les plus performantes sont sélectionnées. De plus, à travers les trois types de représentations, les meilleurs résultats sont obtenus pour SVM, la Régression Logistique et la Forêt Aléatoire, avec de légers avantages en termes de F1 et d'accuracy pour la Forêt Aléatoire dans la plupart des cas. Néanmoins, des modèles tels que la Régression Logistique et le SVM sont plus intelligibles et interprétables ; par conséquent, et en considérant que les différences de performance entre la Forêt Aléatoire et la Régression Logistique/SVM ne sont pas suffisamment significatives pour indiquer que la Forêt Aléatoire est la meilleure option de classification pour des modèles NLP, il est important de noter que, malgré ses métriques légèrement meilleures, la Forêt Aléatoire implique un coût d'entraînement substantiellement plus élevé en temps de calcul et en utilisation de ressources.

Bien que, dans le cas de l'entraînement de modèles classiques, la pertinence de la méthode de vectorisation et son choix soient abordés et référencés, l'algorithme d'entraînement avec recherche sur grille permet d'atténuer cet effet lors de la sélection d'un pipeline, puisque la recherche sur les paramètres du vectoriseur est incluse dans la procédure de recherche sur grille. Cependant, dans le cas des MLP, le plus petit nombre de candidats, l'influence de chaque type de vectorisation sur la structure du réseau, et l'incorporation d'embeddings BERT figés avant l'entraînement du MLP amènent la discussion à considérer non seulement le meilleur modèle, mais plutôt le pipeline le plus performant, incluant à la fois la vectorisation et la classification.

Lors de l'analyse des résultats sur test des MLP entraînés avec des entrées TF-IDF au niveau des caractères, l'écart entre les performances d'entraînement et de test indique que le modèle possède une forte capacité d'ajustement et, par conséquent, un très haut risque de surentrainement ; des changements dans l'architecture du réseau, principalement en termes de largeur des couches et de méthodes de régularisation, compte tenu de la matrice creuse compressée qui entre

dans le MLP, ne produisent pas toujours des améliorations proportionnelles sur l'ensemble de test parce que le risque de surentraînement augmente. En revanche, l'utilisation d'embeddings BERT figés, bien qu'elle montre des performances attendues avec moins de paramètres que les modèles TF-IDF au niveau des caractères sur les données de test, bénéficie de la nature dense de la matrice d'entrée et de la grande différence de complexité du réseau par rapport aux données d'entraînement. Cela fait apparaître les structures plus simples mises en œuvre avec ce type de vectorisation comme une stratégie conservatrice d'un intérêt significatif, comme dans le cas d'une seule couche de décision pour classifier les vecteurs d'information produits par BERT, tel que le modèle baseline. En effet, dans les modèles qui emploient ce type de vectorisation, l'augmentation de la profondeur et de la complexité globale du réseau peut être associée à une dégradation des performances sur les données de test, précisément parce que, compte tenu des caractéristiques de BERT et de la forte densité d'information qu'il fournit, la structure du réseau et le nombre de paramètres requis pour la classification peuvent rester simples.

Après avoir achevé l'entraînement des modèles MLP, les modèles classiques d'apprentissage automatique, conjointement avec la vectorisation TF-IDF au niveau des caractères, conservent de meilleures métriques de performance que les modèles MLP, y compris ceux qui utilisent des embeddings BERT pour la vectorisation, pour deux raisons principales. La première est la rigueur déjà mentionnée de la recherche d'hyperparamètres dans les modèles classiques d'apprentissage automatique. La seconde est que, pour le modèle MLP avec embeddings BERT, il reste une marge d'amélioration, en particulier pour la structure plus conservatrice qui montre de meilleures performances et un risque plus faible de surentraînement ; cette amélioration est rendue possible par la possibilité de réaliser un fine-tuning avec LoRA, qui permet au processus de vectorisation de s'adapter au domaine spécifique du jeu de données d'entraînement. Les résultats obtenus montrent que, parmi tous les modèles employés, le MLP à embeddings BERT avec fine-tuning LoRA atteint de loin les meilleures performances en termes de F1 et d'accuracy, et constitue donc une structure hautement recommandée pour développer des modèles de classification de textes courts basés sur la polarité ou le sentiment.

## RÉFÉRENCES

- [1] T. Finn and A. Downie, "How can sentiment analysis be used to improve customer experience?," *IBM Think*, accessed Jan. 18, 2026. [Online]. Available : <https://www.ibm.com/think/insights/how-can-sentiment-analysis-be-used-to-improve-customer-experience>
- [2] B. Pang and L. Lee, "Opinion mining and sentiment analysis," *Foundations and Trends in Information Retrieval*, vol. 2, no. 1–2, pp. 1–135, 2008.
- [3] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA : Low-Rank Adaptation of Large Language Models," *arXiv preprint arXiv:2106.09685*, Jun. 2021, doi : 10.48550/arXiv.2106.09685.
- [4] The NLTK Project, "Sample usage for corpus," *NLTK Documentation*, accessed Jan. 18, 2026. [Online]. Available : <https://www.nltk.org/howto/corpus.html>
- [5] scikit-learn developers, "Feature extraction," *scikit-learn Documentation*, accessed Jan. 18, 2026. [Online]. Available : [https://scikit-learn.org/stable/modules/feature\\_extraction.html](https://scikit-learn.org/stable/modules/feature_extraction.html)
- [6] scikit-learn developers, "sklearn.feature\_extraction.text.TfidfVectorizer," *scikit-learn Documentation*, accessed Jan. 18, 2026. [Online]. Available : [https://scikit-learn.org/stable/modules/generated/sklearn.feature\\_extraction.text.TfidfVectorizer.html](https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html)
- [7] F. Sebastiani, "Machine Learning in Automated Text Categorization," *ACM Computing Surveys*, vol. 34, no. 1, pp. 1–47, Mar. 2002, doi : 10.1145/505282.505283. [Online]. Available : <https://nmlis.isti.cnr.it/sebastiani/Publications/ACMCS02.pdf>
- [8] scikit-learn developers, "sklearn.naive\_bayes.MultinomialNB," *scikit-learn Documentation*. [Online]. Available : [https://scikit-learn.org/stable/modules/generated/sklearn.naive\\_bayes.MultinomialNB.html](https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html). Accessed : Jan. 18, 2026.
- [9] scikit-learn developers, "sklearn.linear\_model.LogisticRegression," *scikit-learn Documentation*. [Online]. Available : [https://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.LogisticRegression.html](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html). Accessed : Jan. 18, 2026.
- [10] scikit-learn developers, "sklearn.svm.LinearSVC," *scikit-learn Documentation*, accessed Jan. 18, 2026. [Online]. Available : <https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html>
- [11] scikit-learn developers, "sklearn.calibration.CalibratedClassifierCV," *scikit-learn Documentation*, accessed Jan. 18, 2026. [Online]. Available : <https://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifierCV.html>
- [12] scikit-learn developers, "sklearn.ensemble.RandomForestClassifier," *scikit-learn Documentation*, accessed Jan. 18, 2026. [Online]. Available : <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>
- [13] SciPy developers, "scipy.sparse.csr\_matrix," *SciPy Documentation*. [Online]. Available : [https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr\\_matrix.html](https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html). Accessed : Jan. 18, 2026.
- [14] PyTorch contributors, "Data Loading and Processing Tutorial," *PyTorch Tutorials*. [Online]. Available : [https://docs.pytorch.org/tutorials/beginner/data\\_loading\\_tutorial.html](https://docs.pytorch.org/tutorials/beginner/data_loading_tutorial.html). Accessed : Jan. 18, 2026.
- [15] Hugging Face, "BERT," *Transformers Documentation*. [Online]. Available : [https://huggingface.co/docs/transformers/en/model\\_doc/bert](https://huggingface.co/docs/transformers/en/model_doc/bert). Accessed : Jan. 18, 2026.
- [16] scikit-learn developers, "sklearn.preprocessing.StandardScaler," *scikit-learn Documentation*, accessed Jan. 18, 2026. [Online]. Available : <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>
- [17] Sutriawan, Supriadi Rustad, Guruh Fajar Shidik, and Pujiono, "Performance Evaluation of Text Embedding Models for Ambiguity Classification in Indonesian News Corpus : A Comparative Study of TF-IDF, Word2Vec, FastText, BERT, and GPT," *Ingénierie des Systèmes d'Information*, vol. 30, no. 6, pp. 1469–1482, June 2025, doi : 10.18280/isi.300606. [Online]. Available : <https://www.ieta.org/journals/isi/paper/10.18280/isi.300606>
- [18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding," *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics : Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, Jun. 2019, doi : 10.18653/v1/N19-1423. [Online]. Available : <https://aclanthology.org/N19-1423/>
- [19] PyTorch contributors, "torch.nn.Linear," *PyTorch Documentation*, accessed Jan. 18, 2026. [Online]. Available : <https://docs.pytorch.org/docs/stable/generated/torch.nn.Linear.html>
- [20] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics : Human Language Technologies*, vol. 1, Minneapolis, Minnesota, Jun. 2019, pp. 4171–4186.
- [21] A. Vaswani et al., "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, Long Beach, CA, Dec. 2017.