



# Master d'informatique

Parcours Science de Données et Système Complexe

---

## Traitement Automatique de Langage

---

JAIDANE Chaïma  
OBERHAUSER Clément

Année universitaire 2023-2024

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Classification automatique des articles de presse</b>	<b>2</b>
2.1	Pré-traitement des données . . . . .	2
2.1.1	Nettoyage et préparation des données . . . . .	2
2.1.2	Rééquilibrer les données . . . . .	2
2.2	Entraînement des modèles de classification . . . . .	3
2.2.1	Classification . . . . .	3
2.2.2	BILSTM . . . . .	3
2.2.3	Transformers . . . . .	4
2.3	Analyse des résultats . . . . .	4
<b>3</b>	<b>Indexation des articles dans Solr et création de l'interface de recherche</b>	<b>5</b>
3.1	Préparation du core . . . . .	5
3.2	Préparation des données . . . . .	5
3.3	Indexation . . . . .	5
3.4	Paramétrage . . . . .	5
3.5	Mise en forme . . . . .	6
<b>4</b>	<b>Répartition du travail</b>	<b>6</b>

# 1 Introduction

Le traitement automatique des langages (TAL) est un outil permettant d'analyser et de comprendre des langues naturelles par le biais de l'informatique. Dans ce contexte, le projet que nous présentons vise à mettre en œuvre des méthodes de TAL pour la réalisation d'un système de recherche d'informations dans une collecte d'articles de presse de sources francophones d'Afrique.

Ce projet se divise en deux étapes indépendantes. Dans la première étape, notre objectif est d'entraîner un outil de classificateur automatique des articles en fonction de leur catégorie, en nous basant sur **le texte** et **le titre** de chaque article. Nous mettrons en œuvre différentes techniques de pré-traitement et d'apprentissage automatique, en comparant notamment divers algorithmes de classification.

Dans la deuxième étape du projet, nous indexerons le corpus **train.tsv** dans une instance du serveur de recherche **Solr**. Nous développerons ensuite une interface de recherche en texte intégral, permettant également de filtrer les articles selon leur catégorie. Pour cela, nous exploiterons la fonction de recherche par facette de **Solr**, en s'assurant d'utiliser une configuration linguistique française.

Dans ce rapport, nous détaillerons la méthodologie, les expériences réalisées ainsi que les résultats obtenus lors de ces deux étapes. Nous fournirons également une répartition du travail au sein du groupe.

## 2 Classification automatique des articles de presse

### 2.1 Pré-traitement des données

Pour commencer, nous avons importé nos jeux de données d'entraînements et de tests à partir des fichiers **train.tsv** et **test.tsv** à l'aide de la fonction **read.csv** de la bibliothèque Pandas en précisant le séparateur, une tabulation. En utilisant les méthodes prédéfinies par Python telles que **info()**, **head()** et **describe()**, nous avons pu visualiser et comprendre la structure de nos données. Chacun de ces jeux de données est composé de quatre colonnes mais dans le cadre de ce projet, seules les colonnes **headline**, **text** et **category** nous intéressent. La colonne **headline** stocke le titre de l'article, **text** le contenu de l'article et **category** la catégorie à laquelle appartient l'article. Ainsi, nous avons réduit notre jeu de données à ces trois caractéristiques (features). De plus, grâce aux méthodes **isna()** et **sum()**, nous avons constaté qu'aucune valeur manquante n'était présente dans nos données.

#### 2.1.1 Nettoyage et préparation des données

Pour nettoyer et préparer les textes des articles, nous avons utilisé plusieurs techniques de pré-traitement, notamment **la tokénisation** afin de diviser les phrases en mots. Ensuite, nous avons appliqué **la lemmatization** pour réduire les mots à leur forme de base. Cette étape nous a permis de normaliser le texte, ce qui nous facilitera le processus de classification. Enfin, nous avons supprimé **les mots vides** et **la ponctuation** car ils ne contiennent pas d'informations utiles pour la classification des articles.

Cette étape de pré-traitement est essentielle car elle permet d'obtenir des données propres et normalisées, ce qui aura un impact significatif sur les performances de nos modèles de classification. En standardisant le texte, nous améliorons la qualité de nos données et facilitons le travail de nos algorithmes d'apprentissage automatique.

#### 2.1.2 Rééquilibrer les données

Afin de compenser le déséquilibre entre les différentes catégories d'article, nous avons utilisé la technique de sur-échantillonnage aléatoire (**RandomOverSampler**). Cette méthode consiste à augmenter le nombre d'instances de la classe minoritaire en les répliquant aléatoirement jusqu'à ce que le jeu de données soit équilibré. En utilisant cette technique, nous nous assurons que chaque catégorie est représentée de manière équitable dans notre jeu de données. Une fois le sur-échantillonnage effectué, nous avons un nouveau DataFrame composé de données équilibrées. Ce pré-traitement est crucial pour garantir que nos modèles de classification puissent apprendre efficacement à partir de nos données et produire des prédictions précises.

## 2.2 Entraînement des modèles de classification

### 2.2.1 Classification

Notre processus transforme les textes en représentation numérique à l'aide de la méthode TF-IDF (Term Frequency-Inverse Document Frequency), ce qui permet de mesurer l'importance des mots dans les documents par rapport à l'ensemble. Cette étape est crucial pour la classification car elle transforme le texte en un vecteur de caractéristique. Cette étape est réalisé pour la colonne **text** et **headline**.

Afin de corriger les problèmes potentiels de poids, nous allons normaliser nos données pour qu'ils puissent avoir le même impact lors de l'apprentissage. Enfin, nous avons utilisé **ColumnTransformer** pour appliquer ces transformations dans nos données.

Tous les pré-traitement et transformations ont été intégrés dans un pipeline de traitement qui inclut également l'apprentissage du modèle. Nous avons choisi une régression logistique pour la classification en raison de sa capacité à modéliser des probabilités de classes binaires ou multiples à travers une fonction logistique. Le pipeline assure une séquence cohérente d'étapes de traitement et d'apprentissage, facilitant l'exécution et l'optimisation.

Après l'entraînement, le modèle est évalué sur l'ensemble de validation pour tester sa performance. Les métriques telles que la précision peuvent être calculées pour voir comment le modèle performe sur des données qu'il n'a pas vues pendant l'entraînement.

L'apprentissage par pipeline montre des performances robustes avec une précision globale de 0.86. Le modèle excelle particulièrement dans les catégories de "Sports" et "Technology" avec des précisions respectives de 0.93 et 0.88. Ce qui indique une forte capacité à identifier correctement les instances de ces catégories. Toutefois, il y a une légère sous-performance dans la catégorie "Business", où la précision est de 0.80.

Pour renforcer l'évaluation, nous avons implémenté une validation croisée stratifiée, qui assure que chaque pli (5 dans notre cas) de l'ensemble de données contient une représentation proportionnelle de chaque classe de catégorie. Cela permet une évaluation plus robuste et moins biaisée du modèle. En outre, nous avons comparé la régression logistique à d'autres classificateurs.

L'apprentissage par validation croisée a montré une légère amélioration en terme de précision globale, atteignant 0.87. Les performances sont généralement équilibrées à travers les différentes catégories, avec une précision particulièrement élevée pour "Sports" et "Technology" (respectivement 0.95 et 0.92 ). La validation croisée aide à démontrer la stabilité du modèle, car elle assure que chaque instance de l'ensemble de données a été utilisé à la fois pour l'entraînement et pour la validation, minimisant ainsi le risque de biais de sélection.

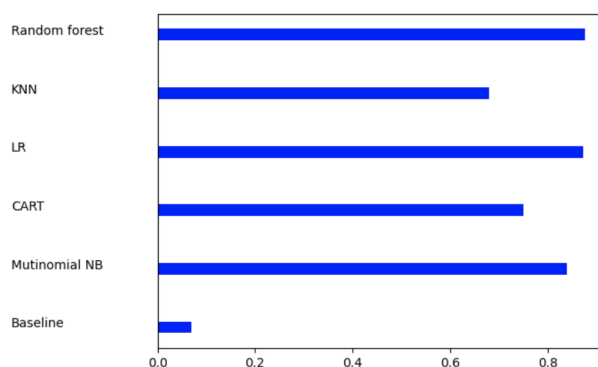


FIGURE 1 – Précision des différents modèle

Au vue du tableau de la figure 1, les meilleures résultats proviennent de l'algorithme de Random Forest et de la Régression logistique (LR).

### 2.2.2 BiLSTM

Le modèle BiLSTM (Bidirectional Long Short-Term Memory) est un modèle capable de capturer des informations à long terme grâce à leur architecture en traitant les informations dans deux directions -vers

l'avant et vers l'arrière. Ce qui est particulièrement utile pour comprendre le contexte entourant chaque mot dans les phrases.

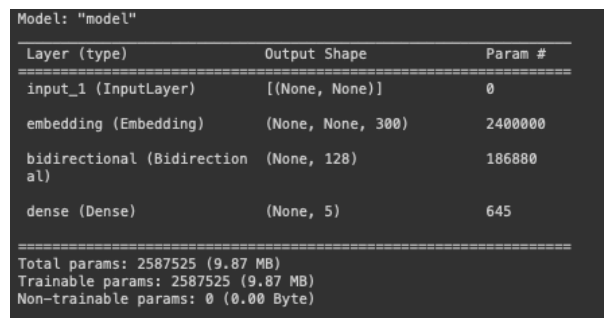
Dans le contexte de l'utilisation du modèle BiLSTM, nous avons effectué un pré-traitement supplémentaire des données qui consiste à indexer les valeurs textuelles de la colonne **category** en valeurs numériques. Cette étape est essentielle pour faciliter le processus de classification, car elle permet à notre modèle de mieux interpréter les données. En outre, nous avons combiné les colonnes **headline** et **text** pour simplifier la vectorisation ultérieure. Pour commencer, après le pré-traitement et préparation, nous avons divisé nos données en un ensemble d'entraînement et un ensemble de validation.

Ensuite, avant de pouvoir entraîner notre modèle, nous avons convertis nos données textuelles en vecteurs numériques à l'aide d'une couche de vectorisation textuelle. Cette technique nous permet de traiter les mots comme des caractéristiques directes dans le modèle. Nous avons également intégré une couche d'embedding pré-entraîné (issu du TP). Les embeddings enrichissent la représentation des mots en utilisant des vecteurs issus de grands corpus de texte, ce qui améliore significativement la capacité du modèle à comprendre et à traiter les sémantiques entre les mots.

Pendant l'entraînement, les poids du modèle sont ajustés itérativement afin de minimiser une fonction de perte. Nous avons utilisé l'entropie croisée comme fonction de perte, qui est bien adaptée pour les tâches de classification multi-classes. L'optimiseur **RMSprop** a été choisi pour minimiser la perte pendant l'apprentissage, grâce à sa capacité à ajuster le taux d'apprentissage de manière adaptative pour chaque poids, ce qui induit une meilleure convergence.

L'évaluation de notre modèle BiLSTM a été réalisée à travers une série d'entraînement utilisant **la validation croisée** pour s'assurer de la généralisabilité et de la robustesse du modèle sur divers sous-ensembles de données. Chaque pli de validation croisée a révélé des insights sur la capacité du modèle à traiter et classifier efficacement les textes selon les catégories définies. En analysant les métriques de perte et de précision pour chaque pli, ainsi que la moyenne de ces métriques sur tous les plis, nous avons pu évaluer de manière approfondie les performances du modèle.

Les résultats ont montré une précision moyenne d'environ 77.79%, avec un écart-type de 2.47%, indiquant une certaine variabilité dans les performances selon les plis, mais une tendance générale vers des résultats satisfaisants.



Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, None)]	0
embedding (Embedding)	(None, None, 300)	2400000
bidirectional (Bidirectional)	(None, 128)	186880
dense (Dense)	(None, 5)	645

=====  
Total params: 2587525 (9.87 MB)  
Trainable params: 2587525 (9.87 MB)  
Non-trainable params: 0 (0.00 Byte)

FIGURE 2 – L'architecture du modèle BiLSTM

### 2.2.3 Transformers

Les transformers n'ont pas pu être entraînés car le code produisait des erreurs et refusait de fonctionner. Nous suggérons donc que le modèle nécessite une machine beaucoup plus puissante que nos machines personnelles.

## 2.3 Analyse des résultats

Nous avons évalué notre jeu de test en utilisant le modèle de régression logistique. Les résultats indiquent une précision de 91

Test Accuracy: 91.14				
Test F1-Score: 90.94				
	precision	recall	f1-score	support
business	0.89	0.93	0.91	75
health	0.95	0.96	0.95	74
politics	0.87	0.97	0.92	71
sports	0.88	1.00	0.94	58
technology	1.00	0.71	0.83	72
accuracy			0.91	350
macro avg	0.92	0.91	0.91	350
weighted avg	0.92	0.91	0.91	350

FIGURE 3 – Modèle LR avec jeu de test

### 3 Indexation des articles dans Solr et création de l'interface de recherche

Pour commencer, nous avons récupéré le fichier `train.tsv` issu du sujet. Pour indexer les données, il faut réaliser les étapes suivantes.

#### 3.1 Préparation du core

Pour commencer, il faut créer le core pour pouvoir réaliser l'ensemble du projet Solr. Pour cela, il faut simplement exécuter la commande

```
./bin/solr create -c article-presse
```

Sur le panneau de configuration, nous pouvons apercevoir que dans l'onglet de sélection une nouvelle ligne apparaît nommé "article-presse", qui nous confirme la création du core.

#### 3.2 Préparation des données

Pour pouvoir indexer les données, il faut commencer par ajouter les balises `<add>` pour chaque "article". Pour ce faire, nous avons créé un fichier python qui permet de générer le fichier xml pour chaque quatuor de colonne : **category**, **headline**, **text** et **url**. Ce fichier python génère un nouveau dossier nommé **temp\_file** pour permettre de stocker tous les fichiers à indexer. Les fichiers xml servent ensuite à indexer tous les documents. Tous ces documents se trouvent dans les dossiers `data` sur le Gitlab du projet.

#### 3.3 Indexation

Ensuite, la commande

```
for x in $(ls) ; do curl -F "fileupload=@$x"
"http://localhost:8983/solr/article-projet/update" ; done
```

à l'intérieur du dossier permet d'indexer l'ensemble des fichiers xml générés. Dans le panel de configuration, onglet "Overview" on peut voir qu'il y a 1440 documents.

Pour s'en rendre compte, on peut suivre le lien suivant dans le navigateur (avec le nom `article-projet` spécifiquement pour notre projet) :

```
http://localhost:8983/solr/article-projet/select?q=description%3Acheval
```

Le retour de cette fonction en json permet de prouver que l'indexation fonctionne correctement.

#### 3.4 Paramétrage

Pour que l'on puisse avoir une recherche avec les variantes en français, il faut ajouter quelques paramètres aux fichiers de configuration. À commencer par le fichier `solrconfig.xml`, qui permet de spécifier les paramètres de recherche. En l'occurrence, ceux qui permettent d'afficher le texte recherché en surbrillance, le champs par défaut ou encore les facettes.

Avant ceci, il faut correctement régler la spécification des champs dans le fichier managed-schema généré automatiquement. En effet, il faut que les champs soient bien nommés en français et en indiquant la recherche exacte ou inexacte pour les champs nécessaires (notamment exacte pour les catégories et inexactes pour le reste).

### 3.5 Mise en forme

Une fois que l'ensemble de l'application a été mise en place, nous nous sommes penché sur le visuel de l'application pour le rendre plus proche d'une application mise en production.

Pour cela, nous avons commencé par ajouter les fichiers du dossier "velocity" pour se servir de template pour les modifier par la suite. Nous avons donc modifié les fichiers suivants :

- head.vm : est le fichier principal qui permet surtout les modifications du css (layout est réellement le fichier principal html mais le premier modifié est head.vm).
- hit.vm : pour permettre de modifier l'affichage de chaque item correspondant à la recherche.
- facets.vm : a pour utilité de modifier l'affichage des facettes à gauche de la page.

Les autres fichiers n'ont pas été modifiés et sont identiques aux fichiers de base du template.

## 4 Répartition du travail

La répartition du travail a été organisée de manière logique : Chaïma s'est occupée de la partie 1 et Clément de la partie 2. Cependant, lorsque nous rencontrons des difficultés, nous communiquons entre nous pour nous entraider. Sans oublier de partager le code pour que chacun puisse comprendre la partie de l'autre.

*Tous les fichiers sources se trouvent sur le Gitlab du projet.*

## Références

- [1] <https://huggingface.co/models?language=fr&sort=trending&search=camembert>
- [2] Tensorflow<https://www.tensorflow.org/?hl=fr>