Implémentation et Analyse de Différents Types de Tokenizers pour les Modèles de Langage

Elkadiri Anas

Encadré par : LAMGHARI NIDAL

12 juin 2025

Table des matières

1 Introduction

La tokenisation est une étape essentielle du prétraitement dans les modèles de langage modernes (LLMs). Elle consiste à transformer un texte brut en une séquence d'unités appelées *tokens*, facilitant l'apprentissage et l'analyse automatique du langage naturel. Ce rapport présente la conception, l'implémentation et l'analyse de différents types de tokenizers : caractère par caractère, Byte-Pair Encoding (BPE), et basé sur les expressions régulières (RegexTokenizer). Un accent particulier est mis sur le workflow pratique, la structure logicielle, et la comparaison expérimentale de ces méthodes.

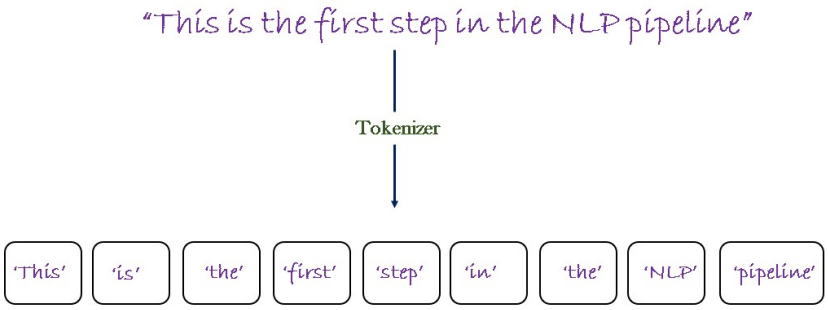

FIGURE 1 – Principe général de la tokenisation

Contexte et Objectif Global

La **tokenisation** constitue la **première étape fondamentale** du pipeline de traitement dans les modèles de langage (LLM) modernes. Elle intervient avant l'apprentissage, la génération ou l'inférence, et conditionne la représentation du texte tout au long du modèle.

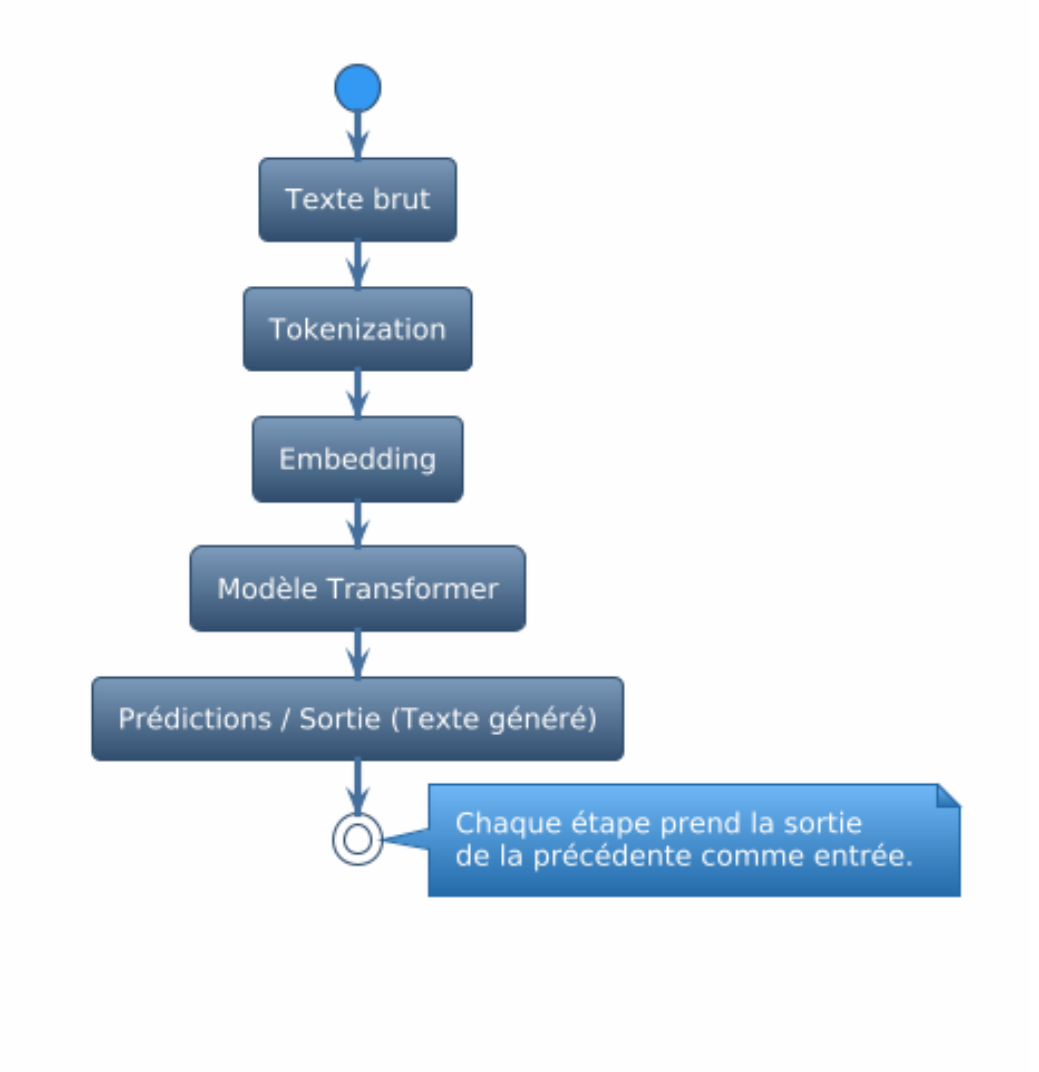

FIGURE 2 – Position de la tokenisation dans la chaîne de traitement d'un LLM

Ce rapport correspond au **premier jalon** d'un projet dont l'ambition est de **développer un LLM complet** (prétraitement, architecture, entraînement, génération...) *from scratch*. Les prochaines étapes incluront l'implémentation des modules d'embedding, d'architecture Transformer, et du processus d'entraînement.

2 Architecture Générale et Diagramme de Classes

Le projet est structuré en modules Python distincts pour chaque tokenizer, facilitant l'extension et la maintenance. Les principales classes sont :

- `BaseTokenizer` : classe abstraite définissant l'interface commune (`encode`, `decode`).
- `BasicTokenizer` : découpe le texte caractère par caractère.
- `BPETokenizer` : applique l'algorithme Byte-Pair Encoding.
- `RegexTokenizer` : segmente le texte selon un motif regex (pattern GPT-4 par défaut).

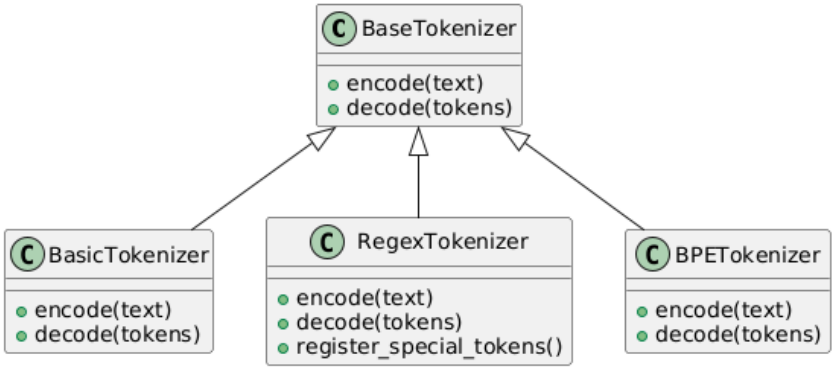

FIGURE 3 – Diagramme de classes des tokenizers

Explication : Toutes les classes de tokenizers héritent de `BaseTokenizer` et implémentent leurs propres versions des méthodes d'encodage et de décodage. Cela garantit l'interopérabilité et la cohérence d'utilisation dans le projet.

3 Workflow du Training et Utilisation

3.1 Workflow d'Entraînement du Tokenizer BPE

L'entraînement d'un tokenizer BPE suit ces étapes :

1. **Chargement du corpus** : lecture d'un ou plusieurs fichiers texte.
2. **Initialisation** : chaque caractère (byte 0-255) fait partie du vocabulaire de départ.
3. **Boucle de fusion** :
   (a) Comptage des fréquences des paires de tokens adjacents.
   (b) Fusion de la paire la plus fréquente pour créer un nouveau token.
   (c) Répétition jusqu'à la taille de vocabulaire visée.
4. **Export du vocabulaire** : sauvegarde des règles de fusion et du vocabulaire.
5. **Utilisation du tokenizer** : encodage de nouveaux textes selon ces règles.

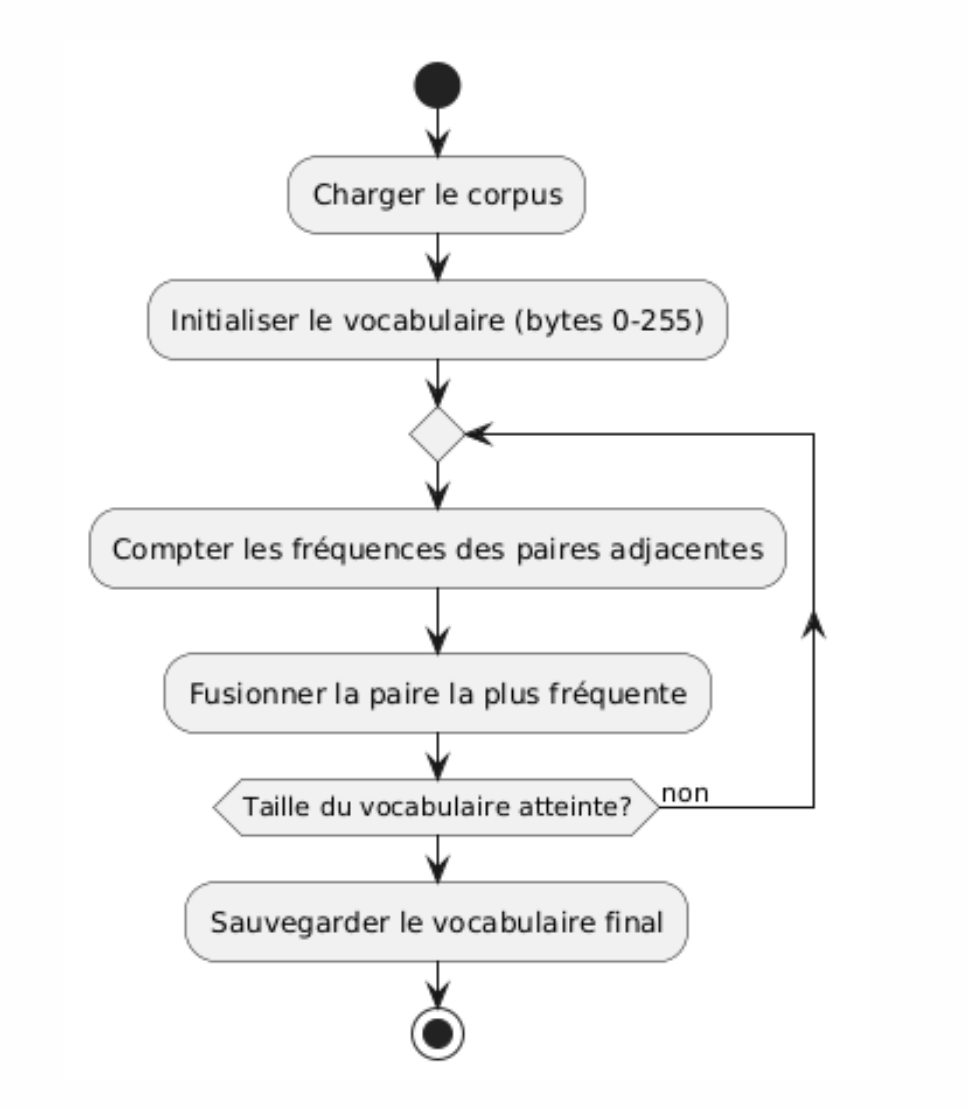

FIGURE 4 – Diagramme d'activité : entraînement BPE

3.2 Workflow d'Utilisation d'un Tokenizer

1. Instancier le tokenizer adapté (`BasicTokenizer`, `BPETokenizer`, `RegexTokenizer`).
2. Appeler `encode(text)` pour obtenir la séquence de tokens.
3. (Facultatif) Appeler `decode(tokens)` pour reconstruire le texte.

4 Description des Tokenizers Implémentés

4.1 Tokenizer Caractère par Caractère (`BasicTokenizer`)

Principe : Segmentation du texte en bytes UTF-8, chaque caractère devient un token.
Avantages : Simplicité, universalité (toute langue, tout symbole).
Limites : Séquences longues et inefficaces, pas de compression sémantique.

4.2 Tokenizer Byte-Pair Encoding (BPE)

Principe : Fusion itérative des paires de tokens les plus fréquentes dans le corpus, pour construire un vocabulaire compact et adapté aux données.

Avantages : Bonne compression, adaptation au corpus.

Limites : Nécessite une phase de training, dépend du corpus utilisé.

4.3 Tokenizer Basé sur les Expressions Régulières (`RegexTokenizer`)

Principe : Utilisation d'un motif regex (pattern GPT-4 par défaut) pour segmenter le texte en unités linguistiques (mots, nombres, ponctuation, etc.), avec gestion avancée des tokens spéciaux.

Avantages : Segmentation linguistiquement pertinente, gestion des tokens spéciaux.

Limites : Dépend de la qualité du motif regex, complexité d'implémentation.

5 Comparaison des Résultats

5.1 Critères d'Évaluation

— Taille moyenne des séquences produites.
— Qualité de la segmentation.
— Capacité de compression.
— Flexibilité d'intégration dans un LLM.

5.2 Tableau Comparatif

Critère	**Caractère**	**BPE**	**RegexTokenizer**
Segmentation Linguistique	Faible	Moyenne	Élevée
Compression	Faible	Élevée	Élevée
Complexité	Très Faible	Moyenne	Élevée
Support des Tokens Spéciaux	Non	Non	Oui
Dépendance Corpus	Non	Oui	Oui (pattern & corpus)

6 Visualisation expérimentale comparative

Pour compléter l'analyse qualitative, nous avons mesuré la performance (temps d'encodage/décodage) et l'efficacité (nombre de tokens produits) de chaque tokenizer sur un texte de grande taille.

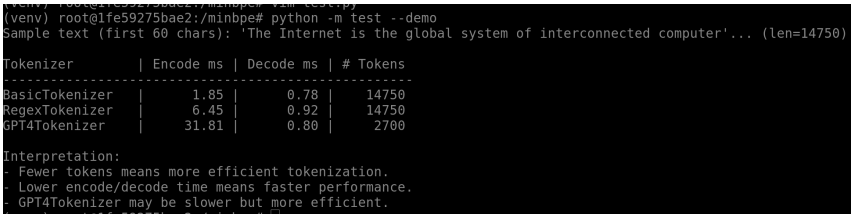

FIGURE 5 – Comparaison expérimentale des tokenizers : vitesse et efficacité

Interprétation : Un nombre de tokens plus faible implique une meilleure compression. Un temps d'encodage/décodage plus bas indique une meilleure rapidité. Le tokenizer BPE atteint généralement le meilleur compromis entre compression et performance, tandis que RegexTokenizer maximise la pertinence linguistique au prix d'une complexité accrue.

7 Déploiement et Virtualisation avec Docker

Pour garantir la portabilité, la reproductibilité et l'isolation de l'environnement, l'application de démonstration a été virtualisée à l'aide de Docker. Grâce à cette approche, tout utilisateur peut lancer l'application web Streamlit sans se soucier de la configuration manuelle des dépendances.

7.1 Architecture du Déploiement

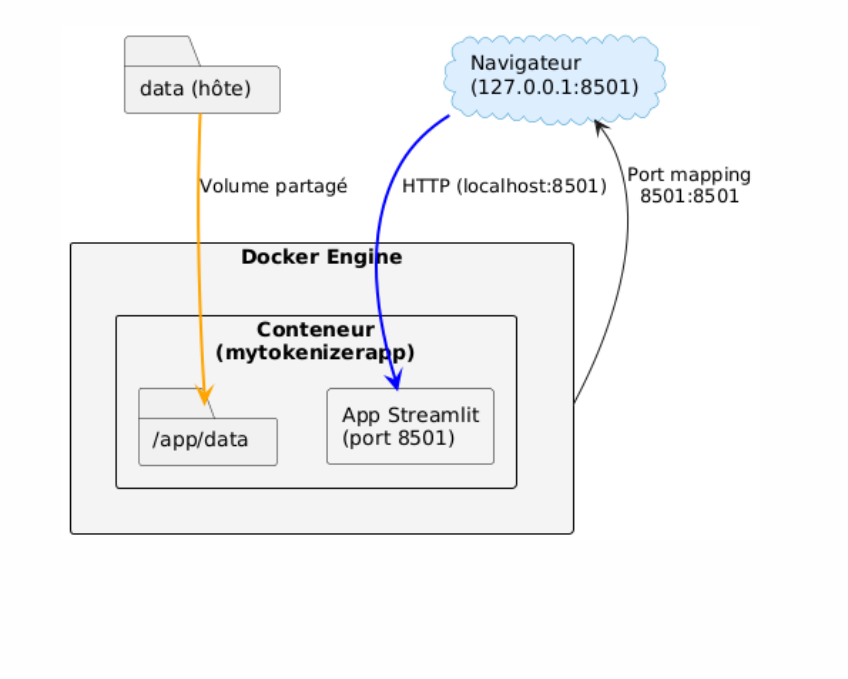

FIGURE 6 – Schéma de virtualisation et d'accès à l'application via Docker

7.2 Utilisation de Docker

L'image Docker inclut :

— Le code source des tokenizers.
— L'application Streamlit pour la démonstration interactive.
— Les dépendances Python nécessaires.

Exemple de commande pour lancer l'application :

```
docker run -p 8501:8501 -v $(pwd)/data:/app/data mytokenizerapp
```

— `-p 8501:8501` : mappe le port de l'application Streamlit (dans le conteneur) vers le port local pour y accéder via `http://localhost:8501`.
— `-v` $(pwd)/data : /app/data : monteledossierlocal`data`dansleconteneurpourpartagerdesfichiers.

L'application web Streamlit offre une interface simple permettant de tester et comparer visuellement les différents algorithmes de tokenisation développés.

8 Capture d'écran de l'application web

La figure ci-dessous présente l'interface principale de l'application Streamlit développée dans le cadre de ce projet.

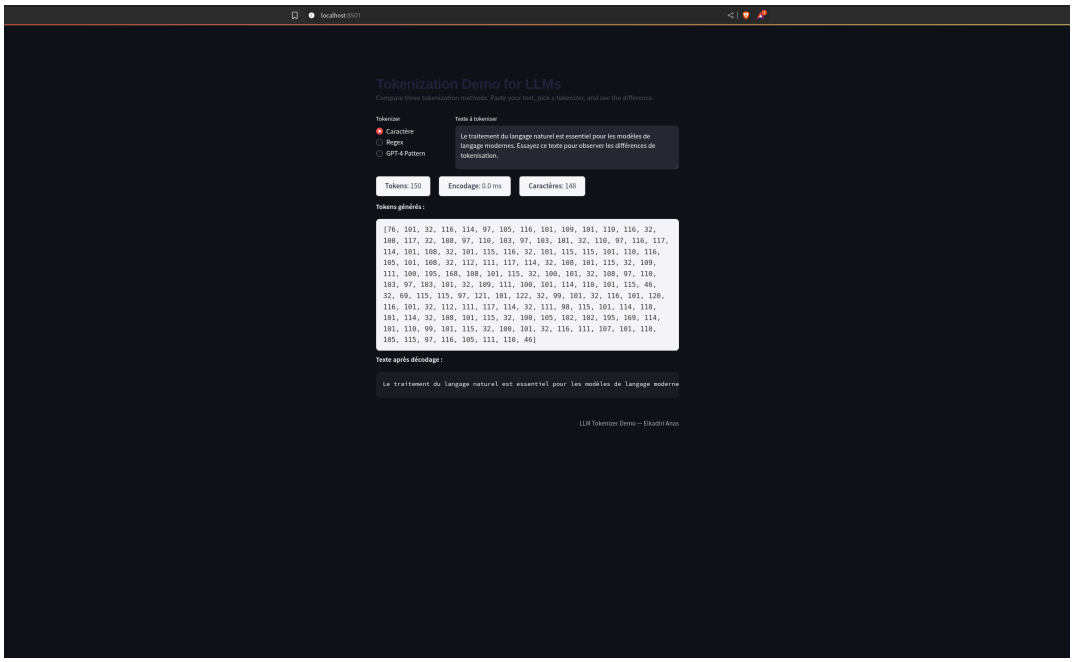

FIGURE 7 – Interface de démonstration des tokenizers pour LLMs.

Cette page web minimaliste permet à l'utilisateur de :

— Saisir ou coller un texte dans une zone dédiée ;

— Sélectionner un algorithme de tokenisation (caractère, regex ou GPT-4 pattern) via un bouton radio ;

— Visualiser instantanément le nombre de tokens générés, le temps d'encodage, la longueur du texte, la liste des tokens produits, ainsi que le texte reconstruit après décodage.

L'interface met l'accent sur la simplicité et la clarté, offrant ainsi une expérimentation rapide et visuelle des différentes méthodes de tokenisation implémentées dans le projet.

9 Perspectives et Suite du Projet

La présente étude sur les tokenizers n'est que la première étape du développement d'un modèle de langage complet. Les prochaines phases du projet porteront sur :

— L'implémentation des modules d'embedding et de préparation des séquences de tokens pour le modèle.

— La conception et la programmation d'une architecture Transformer simplifiée.

— L'entraînement du modèle sur un corpus adapté, suivi de l'évaluation et de la génération de texte.

— L'intégration et la comparaison des différents tokenizers au sein du pipeline complet.

L'objectif final est la **réalisation d'un LLM fonctionnel** construit étape par étape, dont la première pierre est posée dans ce rapport.

10 Conclusion

Ce projet a permis d'implémenter et de comparer plusieurs approches de tokenisation pour les modèles de langage. Le tokenizer caractère par caractère se distingue par sa simplicité, le BPE par son efficacité de compression, tandis que le RegexTokenizer offre la segmentation la plus fine et adaptée au langage naturel, au prix d'une complexité accrue. L'expérimentation met en lumière l'importance du choix du tokenizer pour la performance et l'efficacité des LLMs modernes.