Multi-label Text Classification

Catégorisation automatique des questions de StackOverflow

Luciana Zeferino

1. Introduction

Stack Overflow est une plateforme incontournable pour les développeurs, où des millions de questions et réponses sont échangées chaque jour. Cependant, la gestion des tags y représente un défi majeur. Les tags sont essentiels pour organiser et faciliter l'accès aux informations, mais leur attribution manuelle par les utilisateurs peut être imprécise ou incomplète, notamment pour les débutants.

Dans ce contexte, la mission vise à concevoir un système de suggestion automatique de tags. L'objectif est d'améliorer l'expérience utilisateur en proposant des tags pertinents en fonction des questions posées. Cela permettra de simplifier la recherche d'informations et d'accroître la qualité des contenus.

Pour atteindre cet objectif, deux approches complémentaires seront explorées : une approche supervisée et une approche non supervisée. Ce projet s'articule autour de l'extraction de caractéristiques des données textuelles, l'entraînement de modèles adaptés, et l'évaluation des performances des solutions proposées.

1.1. Objectifs du projet

Extraction et Préparation des Données :

- Collecter les données via des requêtes SQL et l'API Stack Exchange, en appliquant des filtres sur la qualité des questions.
- Nettoyer les textes et effectuer un prétraitement des données textuelles, incluant l'extraction de caractéristiques (bag-of-words, embeddings).

Approche Non Supervisée :

- Implémenter un modèle de type LDA pour identifier les topics dominants et proposer des tags associés.
- Visualiser les résultats en 2D pour une meilleure interprétation des topics.

Approche Supervisée :

- Construire un pipeline supervisé avec diverses techniques d'extraction de features (bag-of-words, Word2Vec, BERT, USE).
- Implémenter MLFlow pour suivre les expérimentations, les performances et le stockage des modèles.

Développement et Déploiement d’une API de Prédiction :

- Concevoir une API permettant de suggérer des tags pour des questions en texte libre.
- Utiliser GitHub pour la gestion de version

2. Extraction et Préparation des Données

2.1. Extraction des données

L'extraction des données a été réalisée à l'aide de la requête SQL suivante sur l'outil StackExchange Data Explorer. Cet outil permet d'accéder aux données de Stack Overflow grâce à des requêtes personnalisées. La requête utilisée a permis de filtrer les questions en fonction de critères qualitatifs(comme par exemple le nombre de tags) afin de garantir la pertinence des données collectées.

Voici la requête SQL utilisée :

```
SELECT TOP 50000 Title, Body, Tags, Id, Score, ViewCount, AnswerCount
FROM Posts
WHERE PostTypeId = 1 AND ViewCount IS NOT NULL
AND AnswerCount > 0 AND LEN(Tags) - LEN(REPLACE(Tags, '<','')) >= 5
```

2.2. Nettoyage du text

2.2.1. Nettoyage des Tags

Une fonction dédiée a été utilisée pour traiter les tags associés aux questions :

- Extraction : Les tags encadrés par des crochets `< >` sont extraits.
- Nettoyage : Les caractères non pertinents (caractères spéciaux, chiffres, tirets inutiles) sont supprimés, à l'exception de certains mots-clés techniques comme `c#` ou `node.js`.
- Filtrage : Seuls les mots d'une longueur significative (supérieure à 2 caractères) ou faisant partie d'une liste de mots spéciaux sont conservés.

2.2.2. Nettoyage des Corps et Titres des questions

Une autre fonction a été utilisée pour nettoyer et prétraiter les textes des questions :

- **Extraction de texte brut** : Le contenu HTML est supprimé à l'aide de BeautifulSoup.
- **Conversion** : Tout le texte est converti en minuscule pour uniformiser les données.
- **Suppression des contractions** : Les abréviations courantes comme "don't" ou "you're" sont remplacées par leurs formes complètes.
- **Filtrage des mots** :
    - Suppression des caractères spéciaux et des mots contenant des chiffres.
    - Exclusion des mots courts ou non significatifs (stop words).
    - Suppression des mots spécifiques (e.g., "please", "thank").

    - Suppression des termes correspondant à certaines catégories grammaticales inutiles (e.g., déterminants, nombres).
- **Lemmatisation** : Les mots sont ramenés à leur forme racine pour réduire les variations inutiles (e.g., "running" → "run").

Résultat des ces transformations sur les données:

Avant:

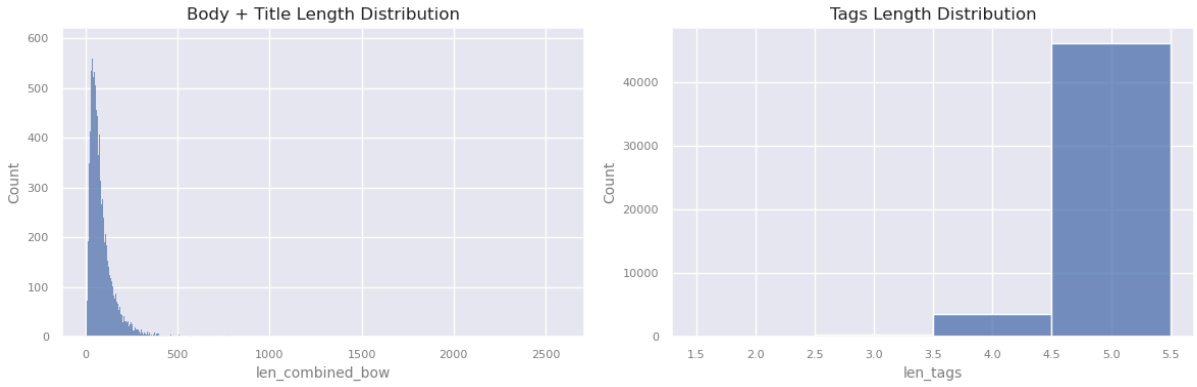
	Title	Body	Tags
0	How to convert Decimal to Double in C#?	I want to assign the decimal variable "trans" to the double variable "this.Opacity". \n<pre class="lang-cs prettyprint-override"><code>decimal trans = trackBar1.Value / 5000;\nthis.Opacity = trans;\n</code></pre>\n When I build the app it gives the following error: \n<blockquote>\n Cannot implicitly convert type decimal to double \n</blockquote>\n	[c#, floating-point, type-conversion, double, decimal]

Après:

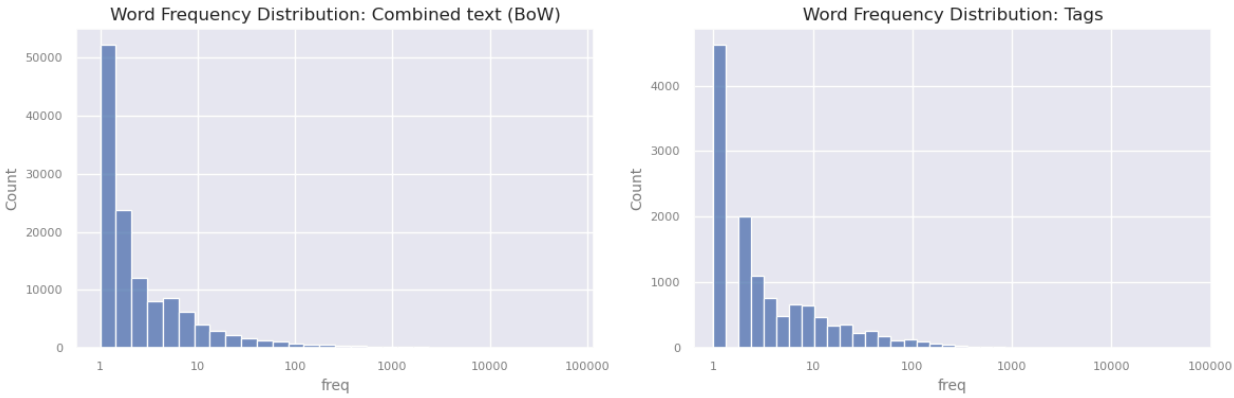
	title_clean_bow	body_clean_bow	Tags
0	[convert, decimal, double, c#]	[want, assign, decimal, variable, trans, double, variable, opacity, decimal, trans, value, opacity, trans, build, app, give, follow, error, implicitly, convert, type, decimal, double]	[c#, floating-point, type-conversion, double, decimal]

3. Analyse Exploratoire

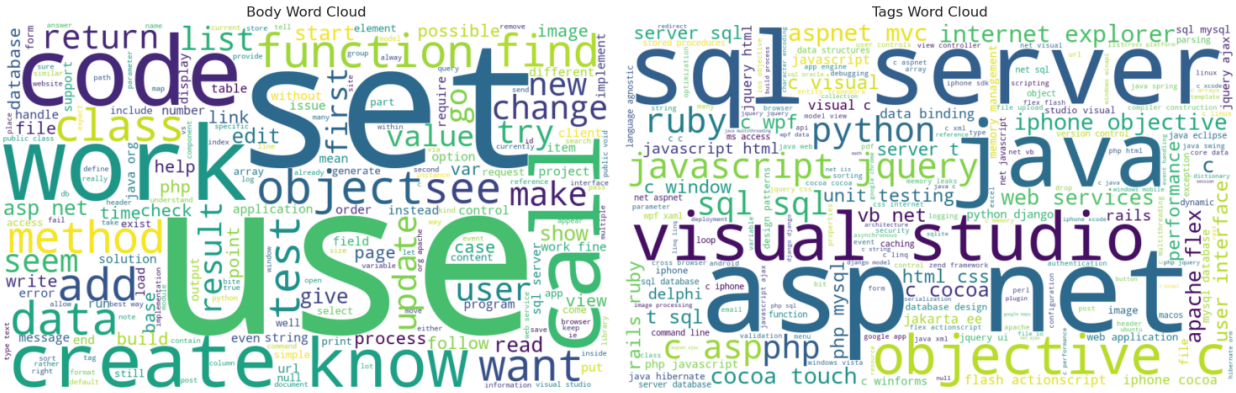
3.1. Distribution du nombre de mots (tokens) par document



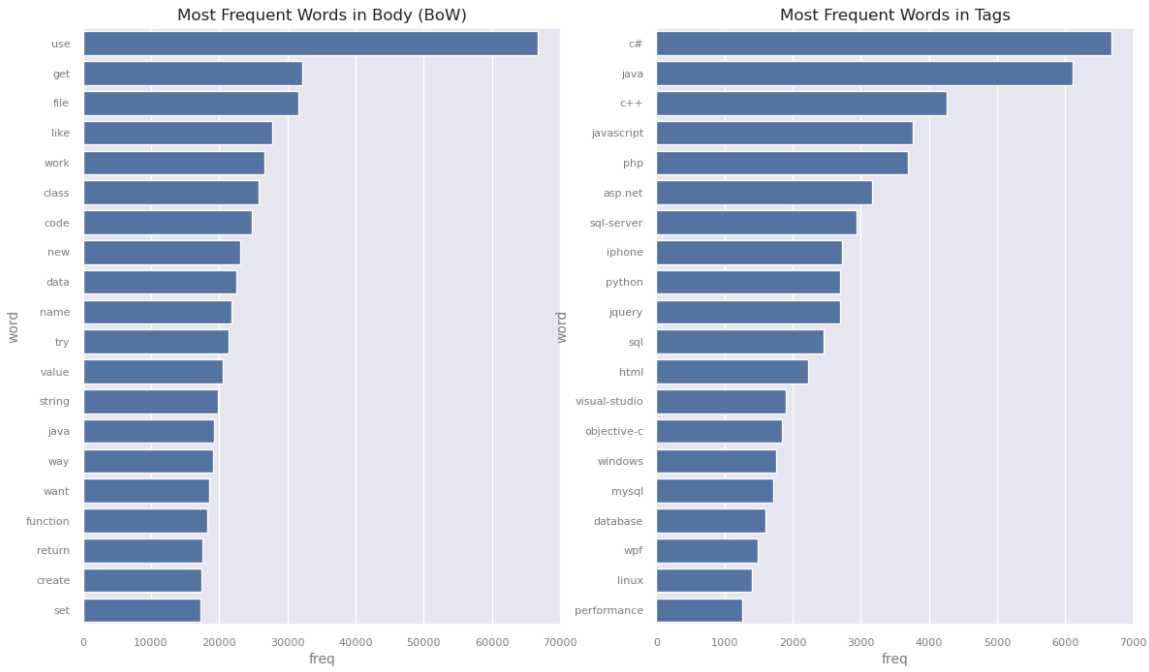
3.2. Distribution de la fréquence des tokens



3.3. Nuage de mots des tokens par leur fréquence



3.4. Top 20 tokens



4. Vectorization

4.1. Vectorization des 'Tags'

Pour préparer les tags en vue de leur utilisation dans des modèles supervisés, la transformation suivante a été réalisée :

MultiLabelBinarizer :
La classe MultiLabelBinarizer de sklearn a été utilisée pour convertir la colonne contenant les tags en une matrice binaire. Chaque colonne de cette matrice représente un tag unique, et chaque ligne indique si ce tag est associé à une question donnée.

Sélection des tags les plus fréquents :
Parmi tous les tags, seuls les 100 plus fréquents ont été retenus à l'aide de la méthode sum() sur les colonnes binaires.

Filtrage des questions pertinentes :
Les lignes correspondant à des questions sans tags parmi les 100 sélectionnés ont été supprimées. Par la suite, les index de X (corps et titres des questions) et de y (tags binarisés) ont été réinitialisés pour garantir la correspondance entre les deux jeux de données.

Après transformation:

	c#	java	c++	javascript	php	asp.net	iphone	jquery	python	sql	...	jpa	svn	web-applications	exception	linq-to-sql	math	testing	database-design	dll	networking
0	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

4.2. Vectorisation du corpus

4.2.1. CountVectrorizer

Le CountVectorizer génère une matrice où chaque ligne représente une question, et chaque colonne indique combien de fois un mot ou un bigramme particulier apparaît dans cette question. Contrairement à TF-IDF, il ne pondère pas les termes en fonction de leur importance dans le corpus global, mais se concentre uniquement sur leur fréquence.

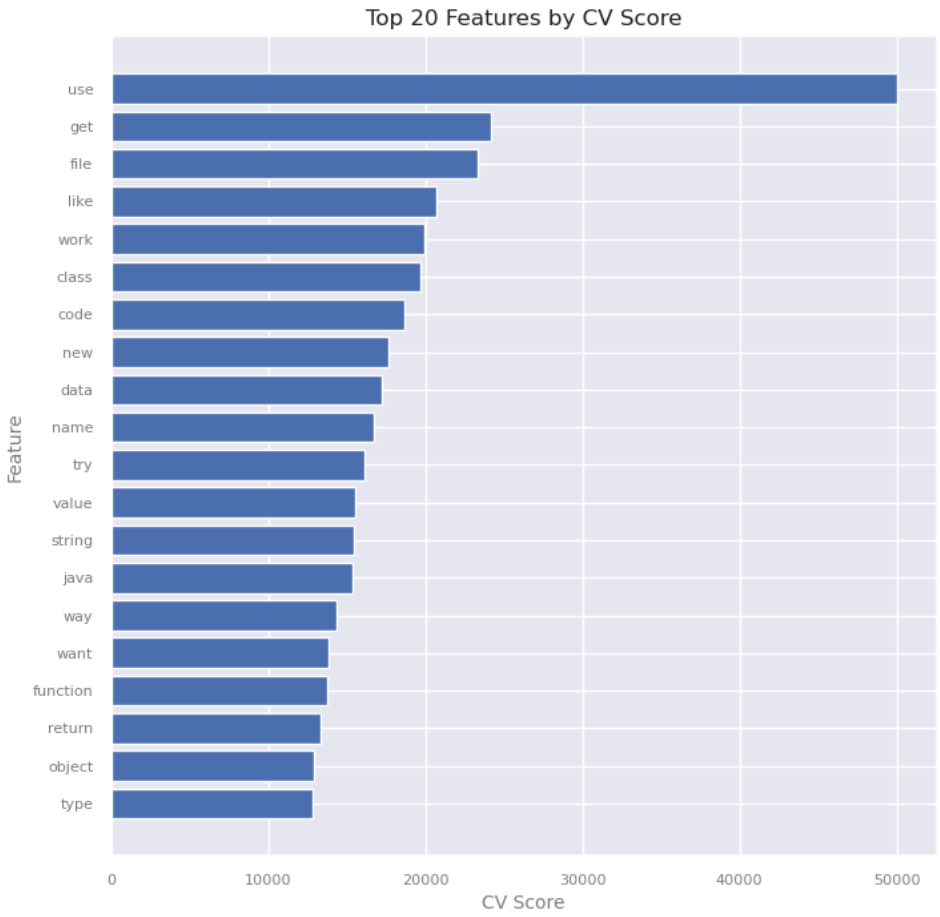
Exemple:

```
doc=["One Cent, Two Cents, Old Cent,
New Cent: All About Money"]
```

	about	all	cent	cents	money	new	old	one	two
doc	1	1	3	1	1	1	1	1	1

Fréquence des mots après la transformation:

Top 20 features par CV scores



4.2.2. TFIDF (Term Frequency Inverse Document Frequency)

Le TF-IDF (Term Frequency-Inverse Document Frequency) est une méthode qui transforme les textes en valeurs numériques, en mettant en avant les mots qui sont importants dans un document tout en pénalisant les mots trop communs dans l'ensemble des documents.

La fréquence des termes (TF) mesure combien de fois un mot apparaît dans un document par rapport au nombre total de mots de ce document.

$$TF = \frac{Number\ of\ times\ a\ word\ "X"\ appears\ in\ a\ Document}{Number\ of\ words\ present\ in\ a\ Document}$$

L'**inverse de la fréquence des documents (IDF)** mesure l'importance d'un mot en tenant compte de sa fréquence dans l'ensemble des documents. Les mots

fréquents dans tous les documents (comme "the", "and") reçoivent un poids faible.

$$IDF = log\left(\frac{Number\ of\ Documents\ present\ in\ a\ Corpus}{Number\ of\ Documents\ where\ word\ "X"\ has\ appeared}\right)$$

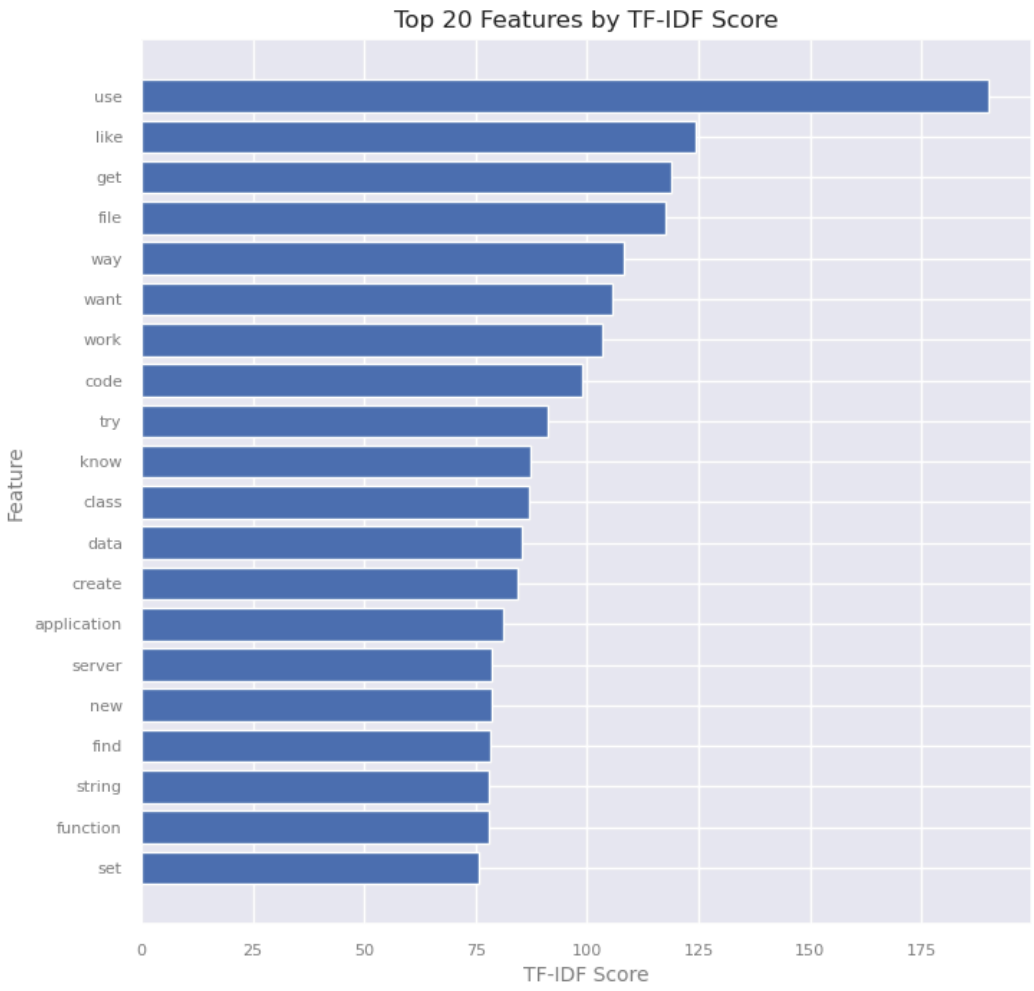
Exemple:

```
Sentence A : The car is driven on the road.
Sentence B: The truck is driven on the highway.
```

Word	TF		IDF	TF*IDF	
	A	B		A	B
The	1/7	1/7	log(2/2) = 0	0	0
Car	1/7	0	log(2/1) = 0.3	0.043	0
Truck	0	1/7	log(2/1) = 0.3	0	0.043
Is	1/7	1/7	log(2/2) = 0	0	0
Driven	1/7	1/7	log(2/2) = 0	0	0
On	1/7	1/7	log(2/2) = 0	0	0
The	1/7	1/7	log(2/2) = 0	0	0
Road	1/7	0	log(2/1) = 0.3	0.043	0
Highway	0	1/7	log(2/1) = 0.3	0	0.043

Fréquence des mots après la transformation:

Top 20 features par TFIDF scores



4.2.3. Word2vec

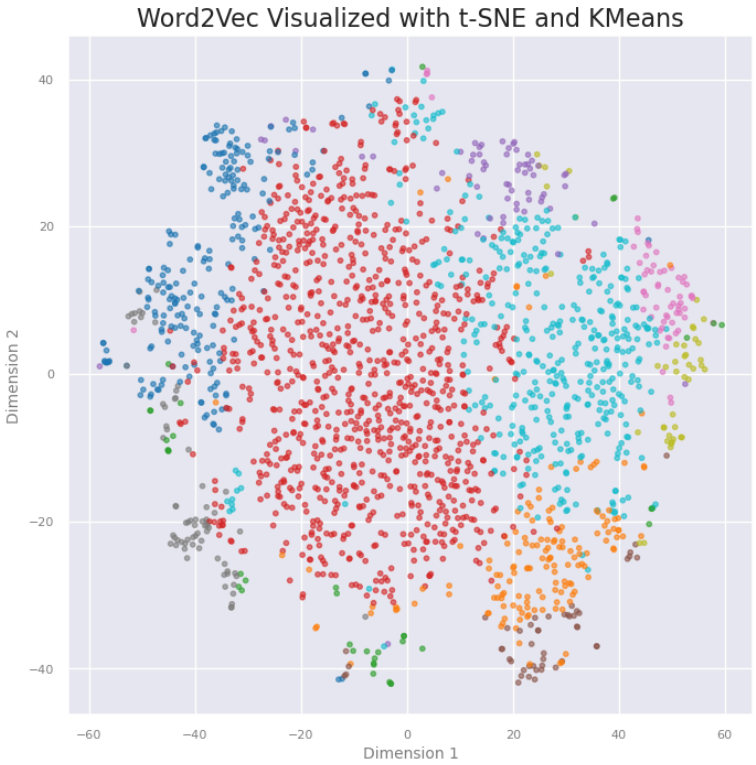
Word2Vec est un modèle d'apprentissage qui repose sur un réseau de neurones **profond**(appelé **word embedding**) qui utilise des techniques spécifiques pour apprendre les représentations des mots à partir des contextes dans lesquels ils apparaissent dans les textes.

Fonctionnement:

1. Tokenisation : Les textes sont d'abord tokenisés en une liste de mots (ou tokens).
2. Apprentissage : Word2Vec utilise un modèle de réseau neuronal (CBOW ou Skip-gram) pour apprendre les représentations des mots. Il entraîne le modèle à prédire soit :
    - Le mot central en fonction du contexte (CBOW), ou
    - Les mots autour en fonction du mot central (Skip-gram).

3. Vecteurs sémantiques : À la fin de l'entraînement, chaque mot est représenté sous forme de vecteurs (X dimensions) qui capturent les similarités sémantiques.

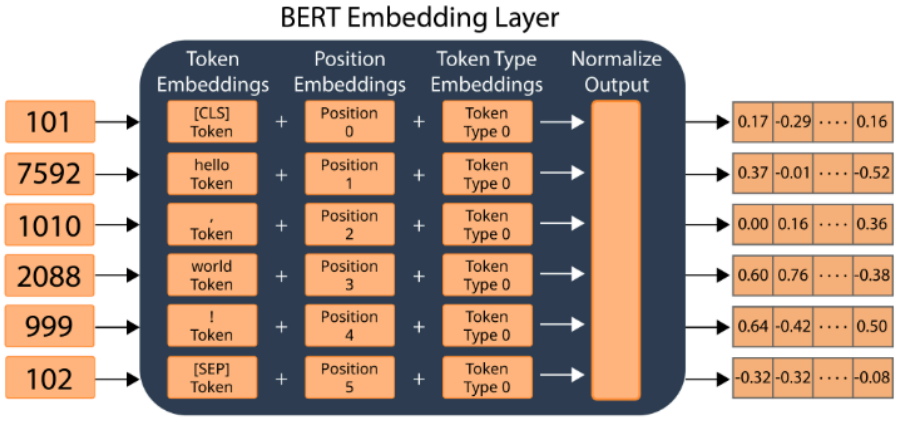
Visualisation de 500 mots avec Kmeans et t-SNE:



4.2.4. BERT

BERT (Bidirectional Encoder Representations from Transformers) est un modèle de machine learning développé par Google pour la compréhension et le traitement du langage naturel (NLP). BERT est pré-entraînée sur de vastes ensembles de données pour comprendre les schémas linguistiques généraux. Basé sur l'architecture Transformer, un réseau de neurones, BERT traite l'ensemble de la phrase en un seul passage, capturant ainsi les relations entre les mots. BERT génère soit :

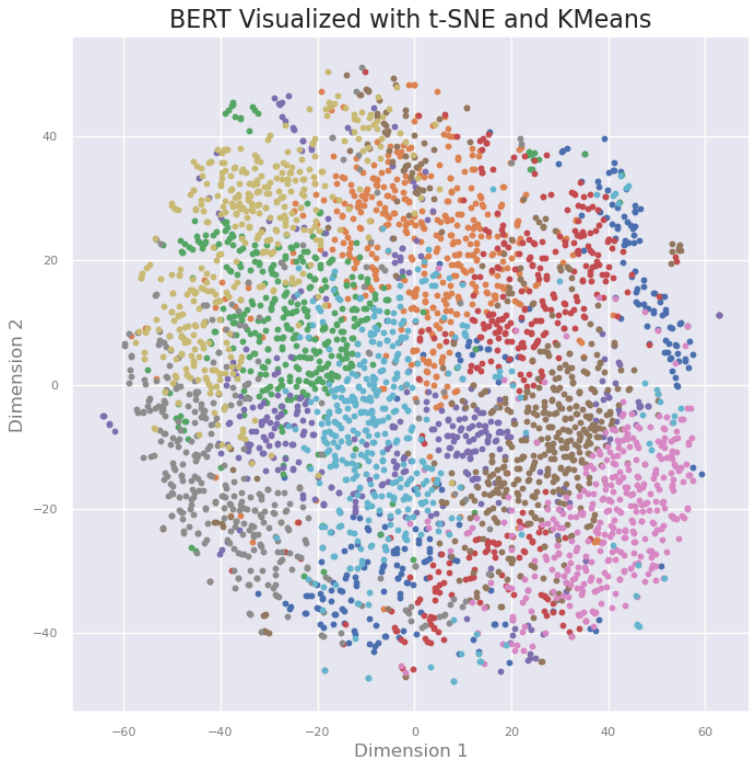
- **Sentence embeddings** : Un vecteur unique résumant toute la phrase,
- **Word embeddings** : Un vecteur pour chaque mot ou token, prenant en compte son contexte.



Fonctionnement :

1. **Découpage de la phrase** : Il divise (ou "tokenize") le texte.
2. **Attribution de sens** : Ensuite, BERT utilise ses connaissances pré-entraînées pour attribuer un sens à ces parties. Par exemple, il sait que "machine learning" fait référence à un concept technologique.
3. **Création d'une représentation unique (Embedding)** : BERT combine les significations de tous les tokens dans le texte pour créer une représentation vectorielle de taille 768 (le token CLS). Ce vecteur capture l'essence de la phrase entière.

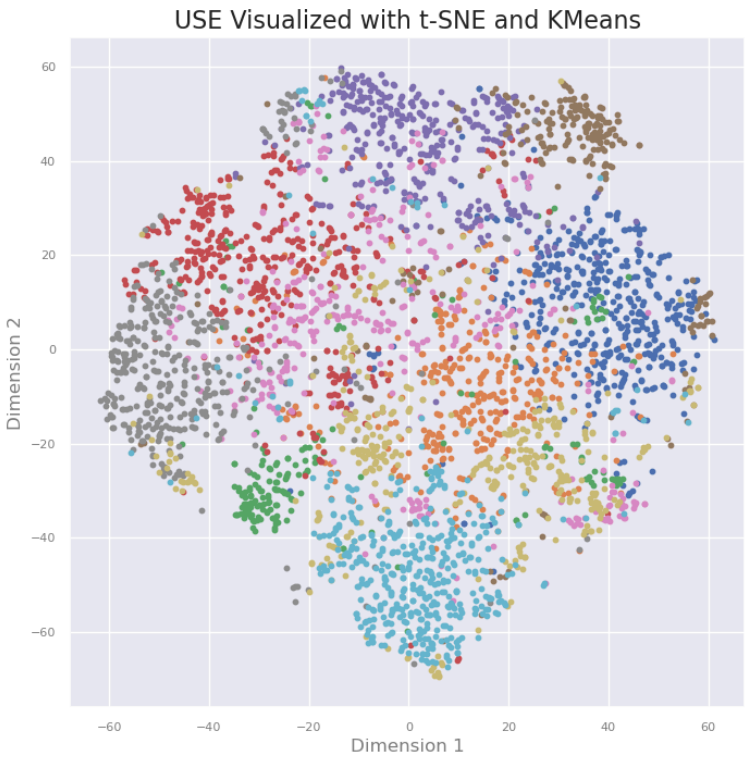
Visualisation des sentences embeddings avec Kmeans et t-SNE:



4.2.5. USE

Universal Sentence Encoder (USE) est un modèle d'encodage développé par Google, conçu pour transformer des textes en vecteurs sémantiques. Pré-entraîné sur des millions de phrases et documents, il est similaire à BERT. Toutefois, BERT se concentre à la fois sur les mots et leur contexte dans la phrase, tandis que USE se focalise sur la capture du sens sémantique des phrases entières ou des documents.

Visualisation des sentences embeddings avec Kmeans et t-SNE:



5. Modélisation non-supervisée

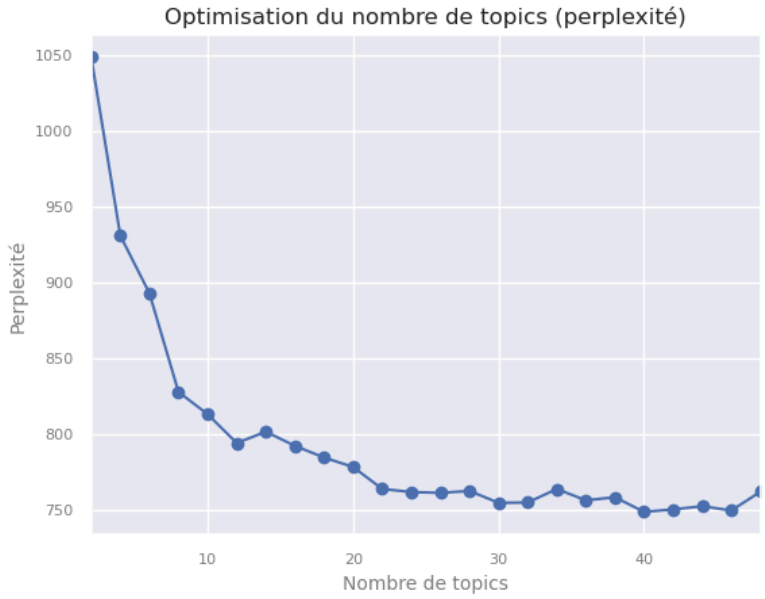
Dans le cadre de ce projet, nous avons été chargés d'évaluer des modèles de classification supervisés et non supervisés pour générer des suggestions de tags. Notre approche a impliqué l'utilisation de la méthode LDA (Latent Dirichlet Allocation) comme modèle non supervisé pour identifier des topics (thèmes principaux) à partir des données textuelles vectorisées avec CountVectorizer.

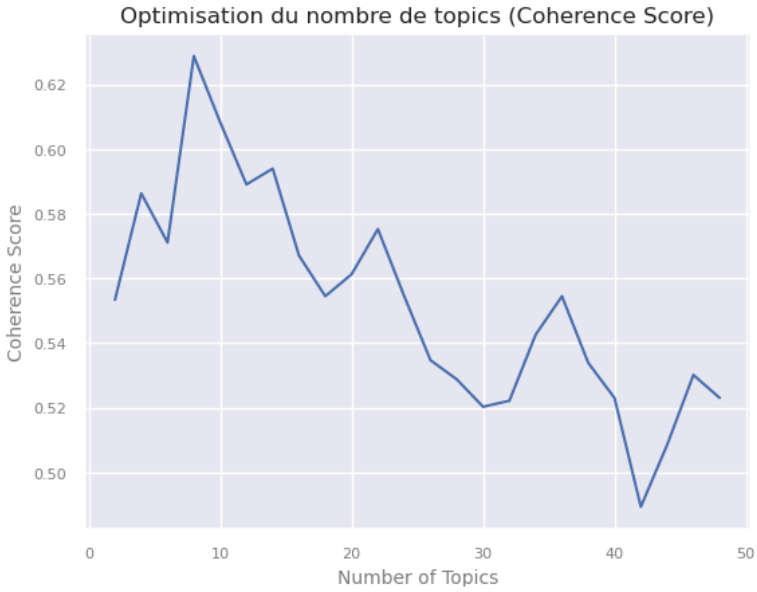
5.1. LDA (Latent Dirichlet Allocation)

LDA (Latent Dirichlet Allocation) est une méthode utilisée pour **analyser un ensemble de textes** et **découvrir les thèmes principaux (appelés "topics")** qui y sont présents, sans qu'on ait besoin de dire à l'avance ce que sont ces thèmes.
En plus, LDA vous donne une liste de mots importants pour chaque topic. Cependant, si les textes sont très mélangés ou si les sujets sont trop proches, cette méthode peut mal trier les documents ou ne pas bien identifier les thèmes.

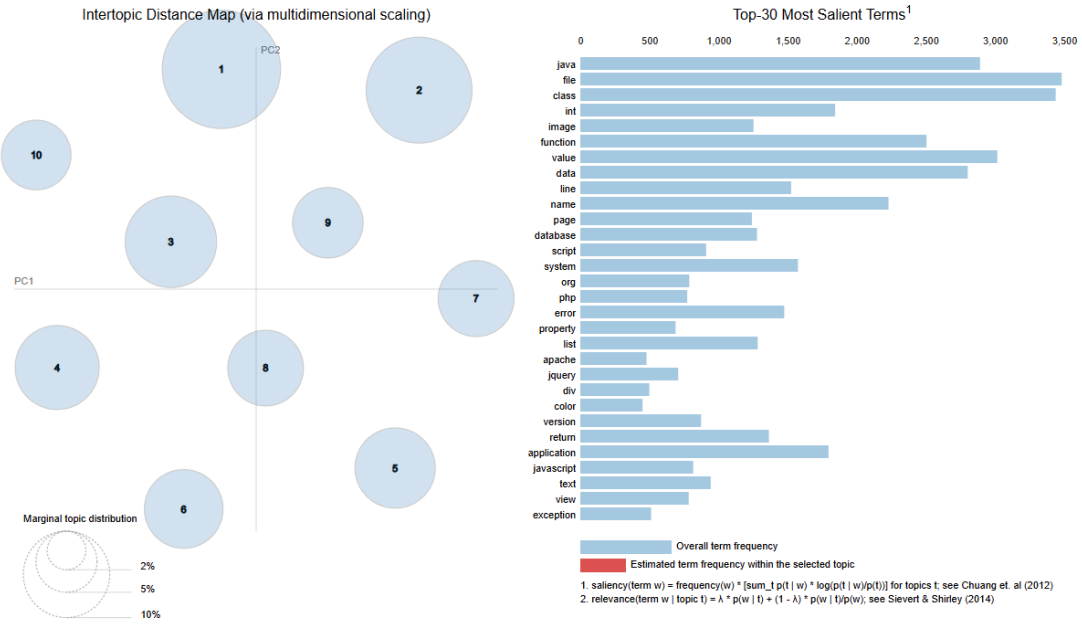
5.2. Choix du nombre de topics (k)

Pour sélectionner le nombre optimal de composants (topics), deux métriques ont été utilisées : le **score de perplexité** et le **coherence score.**


Optimisation du nombre de topics (perplexité)
Perplexité
1050
1000
950
900
850
800
750
10
20
30
40
Nombre de topics


Optimisation du nombre de topics (Coherence Score)
Coherence Score
0.62
0.60
0.58
0.56
0.54
0.52
0.50
0
10
20
30
40
50
Number of Topics

5.3. Visualisation des topics

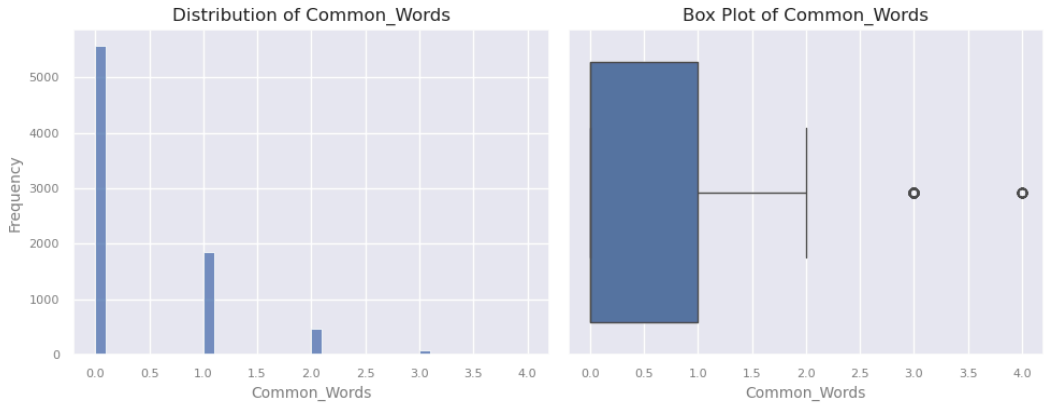


5.4. Résultats

Les résultats de la modélisation non supervisée avec LDA ne sont pas convaincants, comme le montrent :

- Les faibles scores de perplexité et de cohérence: **812.13**
- La validation métier des mots-clés générés par les topics: **0.60**

Une analyse comparative des **principaux mots-clés des topics avec les tags réels** dans y_train a également montré une faible correspondance, comme le montre l'image ci-dessous.



6. Modélisation supervisée avec TF IDF

Pour la tâche de prédiction des tags, une approche supervisée a également été mise en place à l'aide de plusieurs algorithmes de classification, tels que **Logistic Regression**, **SGDClassifier**, et **LinearSVC**.

Dans le cadre d'une classification multi label, où une même question peut avoir plusieurs tags associés, l'approche **OneVsRestClassifier** a été utilisée en combinaison avec ces algorithmes. Cette méthode simplifie le problème complexe (prédire plusieurs tags pour une question) en plusieurs problèmes plus simples (prédire indépendamment la présence ou l'absence d'un seul tag à la fois).

6.1. Fonctionnement de OneVsRestClassifier

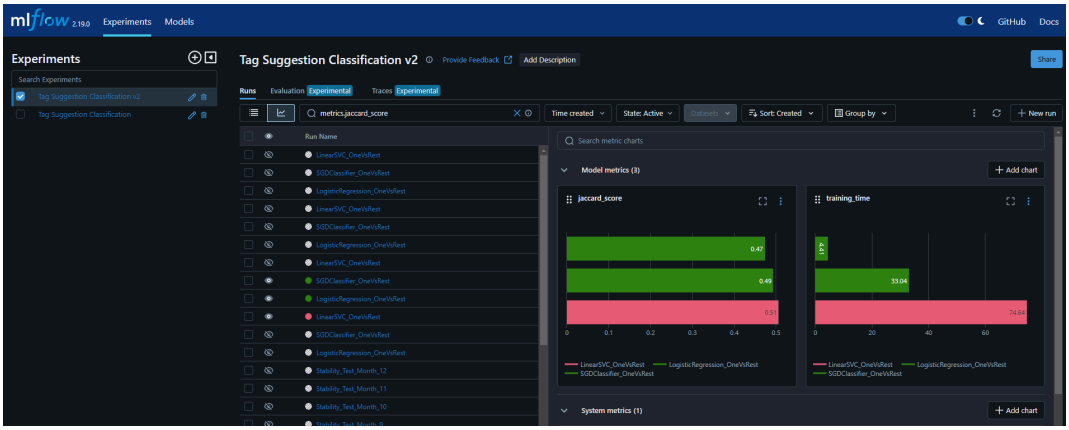
1. **Division des Tags** :
    - Pour ce dataset, nous avons conservé les **100 tags les plus fréquents**. Le **OneVsRestClassifier** crée un modèle indépendant pour chaque tag.
    - Chaque modèle apprend à répondre à une seule question : *"Est-ce que ce tag spécifique s'applique à cette question ?"*
2. **Entraînement des Modèles** :
    - Pour chaque tag, le modèle apprend à distinguer les questions qui possèdent ce tag de celles qui ne l'ont pas.
    - Par exemple, pour le tag **"Python"**, le modèle utilise les questions qui ont ce tag comme classe positive, et les autres comme classe négative.
3. **Prédiction** :
    - Lorsqu'une nouvelle question est soumise au modèle, chaque sous-modèle est évalué indépendamment si son tag doit être attribué ou non.
    - Les résultats des 100 sous-modèles sont ensuite combinés pour obtenir l'ensemble final des tags prédits.

6.2. Optimisation et Validation

Pour améliorer les performances des modèles, un **RandomizedSearchCV** a été utilisé pour explorer les hyperparamètres des algorithmes de classification ainsi que les paramètres de vectorisation (**TF-IDF**).

6.3. Suivi des Expérimentations

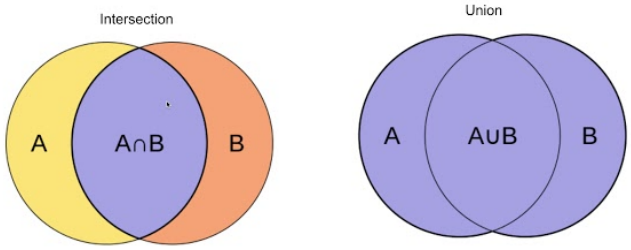
MLFlow a aussi été utilisé pour le suivi des expérimentations comme le temps d'entraînement, et les scores (ex. Jaccard Score) de chaque modèle ont été enregistrés pour chaque essai.



6.4. Jaccard Score

La métrique du **Jaccard Score** a été utilisée comme principal indicateur de performance des modèles.

Le **Jaccard Score** mesure la similitude entre deux ensembles. Dans le contexte de ce projet, il évalue la qualité des tags prédits par un modèle en les comparant aux tags réels. Il est obtenu en divisant le nombre de tags correctement prédits par un modèle (**l'intersection** des tags prédits et des tags réels) par le nombre total de tags uniques présents dans les deux ensembles combinés (**l'union** des tags prédits et des tags réels).



$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

6.5. Résultats

Les meilleurs scores Jaccard ont été obtenus avec le modèle **LinearSV**C, suivi de Logistic Regression.

7. Modélisation supervisée avec Embeddings

Pour la modélisation supervisée utilisant les embeddings comme entrées, seules 5000 lignes de données ont été utilisées. Ce projet sert principalement d'introduction aux techniques de vectorisation avancées et ne vise pas à une mise en production dans le cadre de la création de l'API de suggestion de tags.

Résultats de la classification supervisée obtenus à l'aide de trois méthodes de vectorisation : Word2Vec, BERT et USE.

```
BERT---------------------------------
SGDClassifier()  Jaccard Score: 0.20926223651304038
LogisticRegression(max_iter=200, solver='saga')  Jaccard Score: 0.24870617057112235
LinearSVC(C=1.5, dual=False, max_iter=2000, penalty='l1')  Jaccard Score: 0.2150151414620868

 WORD2VEC---------------------------------
SGDClassifier()  Jaccard Score: 0.1995088304573835
LogisticRegression(max_iter=200, solver='saga')  Jaccard Score: 0.14377456234369415
LinearSVC(C=1.5, dual=False, max_iter=2000, penalty='l1')  Jaccard Score: 0.15605573419078242

 USE---------------------------------
SGDClassifier()  Jaccard Score: 0.349746082784668
LogisticRegression(max_iter=200, solver='saga')  Jaccard Score: 0.38597713469096107
LinearSVC(C=1.5, dual=False, max_iter=2000, penalty='l1')  Jaccard Score: 0.32787092328892975
```

8. Développement et Déploiement d'une API de Prédiction

Cette phase du projet visait à développer une API capable de suggérer des tags pour des questions formulées en texte libre. L'API a été déployée sur le cloud AWS, et une interface utilisateur a été créée et hébergée sur GitHub Pages, accessible via ce lien.
Côté Github, un webhook a été configuré pour envoyer une requête POST à l'API Gateway à chaque mise à jour.

8.1. Architecture AWS

- **API Gateway** : Mise en place d'API Gateway pour gérer les requêtes POST et les rediriger vers le backend.
- **AWS Lambda** : Configuration de Lambda pour traiter les requêtes API et exécuter la logique de prédiction des tags.
- **Amazon EC2** : Déploiement des composants backend sur une instance EC2. Une image **Docker** du projet a été créée localement, puis poussée sur l'instance EC2 pour faciliter le déploiement et la gestion des dépendances.
- **CloudWatch** : Activation de CloudWatch pour les journaux, le débogage et la surveillance des performances d'API Gateway et de la fonction Lambda.

8.2. Intégration et Configuration

Configuration des mappages des paramètres des requêtes dans API Gateway pour assurer une communication correcte entre l'API, Lambda et EC2.

8.3. Tests et Validation

Réalisation de tests approfondis avec Postman pour valider le comportement de l'API et garantir une intégration fluide entre les services AWS.