

Rapport Projet 5

Catégorisez automatiquement des questions

Marie-France LAROCHE-BARTHET

Février 2022

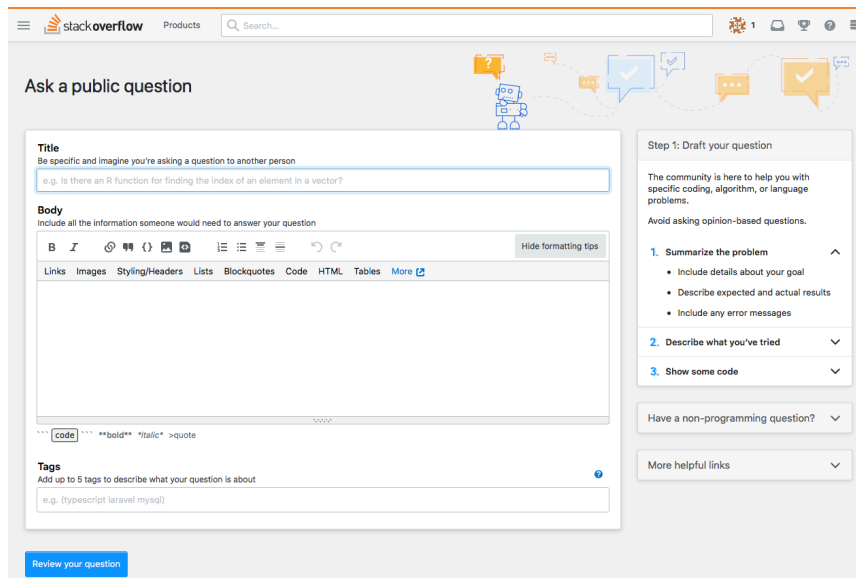
Table des matières

Introduction	3
1. Préparation des données	3
1.1. Récupération des données	3
1.2. Analyse exploratoire des données	4
a. Evolution du nombre de questions entre 2019 et 2021	4
b. Corrélation entre les variables numériques	4
1.3. Filtrage des données	6
2. Pré-traitements des données textes bruts	6
2.1. Suppression des balises HTML	6
2.2. Tokenisation du texte	6
2.3. Filtrage sur les noms communs	7
2.4. Normalisation des mots	7
2.5. Filtrage des données	8
2.6. Liste des mots à garder	8
3. Derniers traitements avant modélisation	8
3.1. Split des données	8
3.2. Vectorisation des données	8
4. Modélisations effectuées	8
4.1. Approche non supervisée : Latent Dirichlet Allocation (LDA)	9
a. Choix du nombre de sujets	9
b. Répartition des topics	9
c. Prédiction des tags	10
4.2. Approche supervisée	10
a. Pré-traitement	10
b. Entrainement des modèles	11
c. Optimisation du SVM	12
d. Prédiction des tags	12
4.3. Comparaison des approches non supervisée et supervisée	12
a. Comparaison qualitative	12
b. Essai de comparaison quantitative	13
5. Mise en production et API	14
Conclusion	14

Introduction

Créé en 2008, [Stack Overflow](#), est un site communautaire d'entraide via des questions-réponses, s'adressant à un public du secteur informatique. Ce site est rapidement devenu une référence en la matière. Les questions portent sur des problèmes de programmation rencontrés par des utilisateurs et d'autres utilisateurs peuvent répondre à leurs questions. Via un système de votes, les utilisateurs peuvent se rendre compte des meilleures questions et des meilleures réponses données sur le site.

Après s'être enregistré sur le site, l'utilisateur peut poser une question via l'interface suivante :



Il doit renseigner un titre, un corps de texte (body) et 1 à 5 tags pour catégoriser sa question. Concernant ce dernier point, les utilisateurs expérimentés n'auront aucun mal à associer des tags à leur question. Par contre, pour des utilisateurs novices, il serait judicieux de suggérer quelques tags.

L'objectif de ce projet est donc de proposer un système de suggestion de tags :

- un corps de texte est fourni par l'utilisateur ;
- celui-ci, après différents traitements, entre dans un modèle de machine learning ;
- le modèle fournira en sortie, après traitements, une liste de tags.

Deux types d'approche, non-supervisée et supervisée, seront envisagés dans le projet et comparés.

1. Préparation des données

1.1. Récupération des données

Stack Overflow propose un outil d'export de données, [StackExchange Explorer](#), qui recense un grand nombre de données authentiques de la plateforme d'entraide. Nous pouvons récupérer les données via un système de requêtes SQL. Nous avons limité le nombre de données pour des raisons de performance de machine.

Nous avons donc récupéré :

- les posts de type question (PostTypeId = 1) avec leur id, titre, corps de texte et tags ;
- sur les 3 dernières années (2019, 2020 et 2021) ;
- avec un score strictement supérieur à 0 ;
- avec un nombre de vues strictement supérieur à 0 ;
- avec un nombre de réponses strictement supérieur à 0 ;

- avec un nombre de commentaires strictement supérieur à 0 ;
- avec un nombre de favoris strictement supérieur à 0.

Par ailleurs, nous ne pouvions récupérer les données que par « paquet » de 50000 lignes, du fait des limitations de l'outil. Nous avons donc fait des requêtes par tranche de 6 mois. Voici un exemple de requête SQL :

```
SELECT Id, Title, Body, Tags, CreationDate, Score, ViewCount, AnswerCount,
CommentCount, FavoriteCount
From Posts
WHERE PostTypeId = 1
AND CreationDate BETWEEN '2020-01-01T00:00:00.00' AND '2020-06-30T23:59:59.999'
AND Score >= 1
AND ViewCount >=1
AND AnswerCount >=1
AND CommentCount >=1
AND FavoriteCount >=1
ORDER BY CreationDate
```

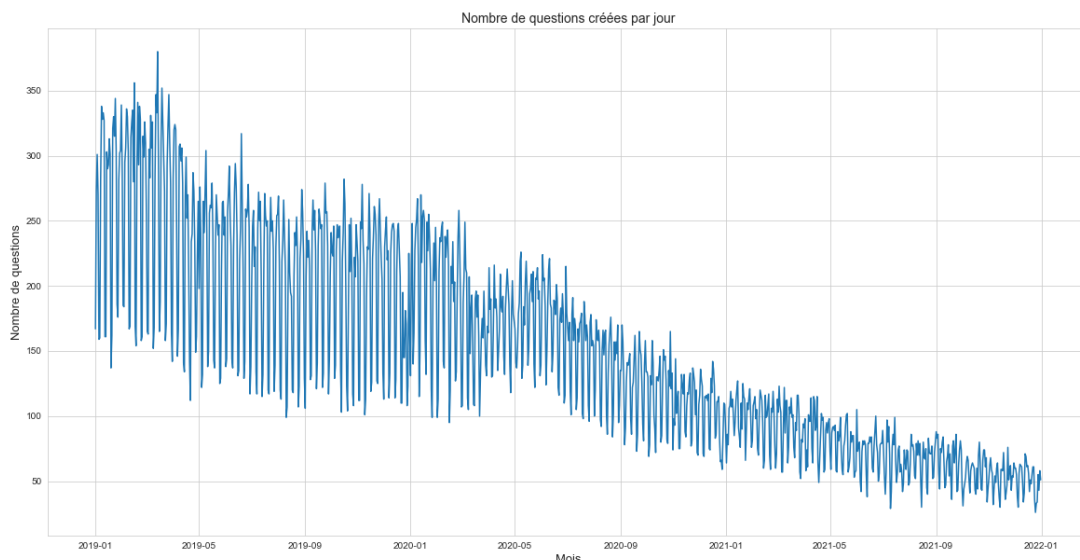
Nous avons ainsi récupéré 164 535 lignes.

1.2. Analyse exploratoire des données

En vue de savoir comment filtrer nos données (nous souhaitons travailler avec environ 50000 lignes pour raison de performance), nous analysons rapidement les données.

a. Evolution du nombre de questions entre 2019 et 2021

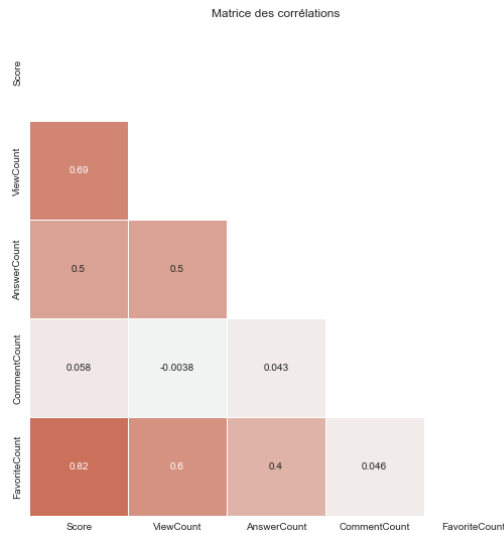
Nous observons une diminution du nombre de questions créées par jour au cours du temps (pandémie ? autres outils d'aide utilisés ?), les anciennes questions sont donc favorisées. Un filtrage par CreationDate ne semble donc pas pertinent.



EVOLUTION DU NOMBRE DE QUESTIONS ENTRE 2019 ET 2021

b. Corrélation entre les variables numériques

A l'aide d'une matrice des corrélations de Pearson, nous regardons les corrélations deux à deux entre les variables numériques (Score, ViewCount, AnswerCount, CommentCount, FavoriteCount).

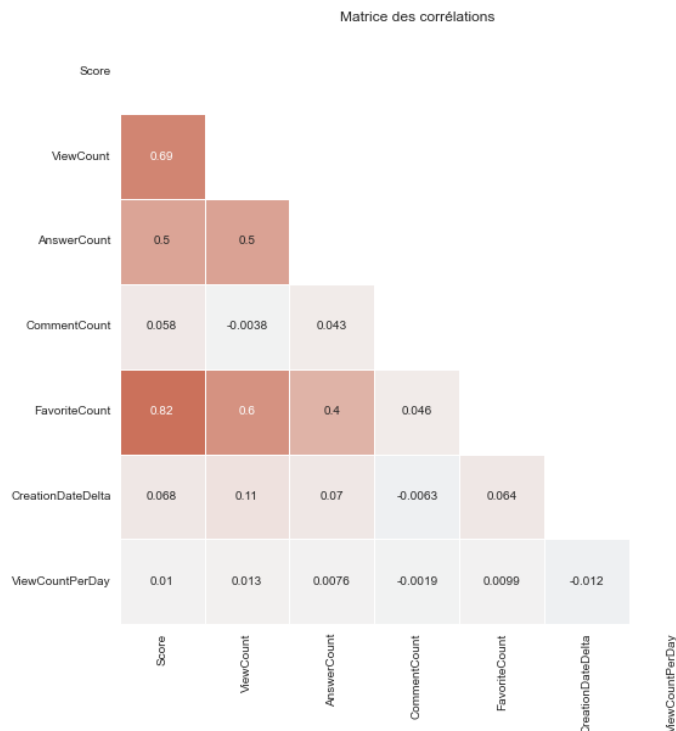


MATRICE DES CORRÉLATIONS DE PEARSON

Globalement les variables sont corrélées entre elles sauf CommentCount. Cependant cette variable n'est pas forcément très pertinente pour y appliquer un filtre. Les questions les plus anciennes sont favorisées en terme de ViewCount, et d'après la matrice des corrélations, cela influence sur le Score et un peu sur AnswerCount et FavoriteCount. Par ailleurs, nous avons également vu que les posts anciens étaient plus nombreux. Nous allons donc avoir un biais si l'on choisit une des variables actuelles pour filtrer les données.

Nous avons donc créé une nouvelle variable ViewCountPerDay qui est le nombre de vues par jour, défini par le rapport entre ViewCount et CreationDateDelta (que l'on doit créer aussi) qui est le temps écoulé entre la CreationDate d'une question et la CreationDate de la toute dernière question créée dans le dataset.

La matrice des corrélations de Pearson devient donc avec ces nouvelles variables :



MATRICE DES CORRÉLATIONS DE PEARSON AVEC DEUX NOUVELLES VARIABLES

Ainsi, nous voyons que ViewCountPerDay n'a quasiment aucun lien avec les autres variables. Nous pouvons donc filtrer les données sur cette variable.

1.3. Filtrage des données

Nous souhaitons avoir environ 50000 samples et d'après la distribution de la variable ViewCountPerDay :

mean	std	min	25 %	50 %	75 %	max
11.268226	1319.365444	0.019084	0.374957	1.161924	3.597449	522800

nous ne gardons que les questions qui ont été vues au moins 3 fois par jour, soit ViewCountPerDay ≥ 3 . Nous avons ainsi récupéré 47032 lignes, soit 28.6% du dataset initial.

Pour la suite, nous ne gardons que les colonnes Title (titre), Body (corps de texte) et Tags (tags).

2. Pré-traitements des données textes bruts

Nous devons traiter des données textuelles, nous allons donc « nettoyer » le texte de toutes les informations qui ne nous intéressent pas et qui ne nous aideront pas à catégoriser les questions. Nous ne garderons que des mots pertinents. Par la suite, pour les entraînements de modèles de machine learning, il nous faudra transformer ces mots pertinents en vecteurs donc en objets de nature numérique.

Après avoir fusionné pour chaque question son titre avec son corps de texte (obtenant ainsi un seul « texte »), nous effectuons dans un premier temps les étapes suivantes :

- supprimer les balises html (*librairie utilisée : BeautifulSoup*) ;
- tokenisation du texte (*librairie utilisée : NLTK*) ;
- ne garder que les noms communs (*librairie utilisée : NLTK*) ;
- normaliser les mots : lemmatisation ou stemmatisation (*librairie utilisée : NLTK*).

Nous appliquons également ces étapes sur les tags.

2.1. Suppression des balises HTML

BeautifulSoup permet d'enlever les balises HTML contenues dans le texte.

2.2. Tokenisation du texte

Cette étape permet de récupérer sous forme d'une liste de mots (ou tokens) le texte initial. Nous avons effectué les nettoyages suivants :

- récupération uniquement des caractères alphabétiques ;
- mise en minuscule du texte ;
- récupération des mots (ou tokens) sous forme de liste ;
- filtrage de cette liste pour ne conserver que les mots d'au moins 3 lettres : cela nous permet de nous débarrasser de mots récurrents inutiles ou de variables symbolisées par des lettres (comme a, b, c, x..). Cependant, cette solution n'est pas optimale car nous éliminons des termes liés à des langages de programmation comme C et R ;
- filtrage à nouveau pour supprimer les stopwords de la langue anglaise : il s'agit de termes génériques couramment employés. Idéalement, l'utilisation d'un dictionnaire de stopwords spécifiques à nos textes aurait été pertinente. Pour des raisons de timing, cela n'a pas été fait.

2.3. Filtrage sur les noms communs

Les process et technologies utilisés sont en général identifiés sous forme de nom commun. Nous utilisons la technique de POS Tagging (Parts of Speech Tagging) dont le principe est d'identifier le type d'un mot d'une partie d'un texte, basé sur sa définition et le contexte. Vu la spécificité des textes ici, les limites de cette technique sont que certaines dénominations de technologies pourraient ne pas être prises en compte.

2.4. Normalisation des mots

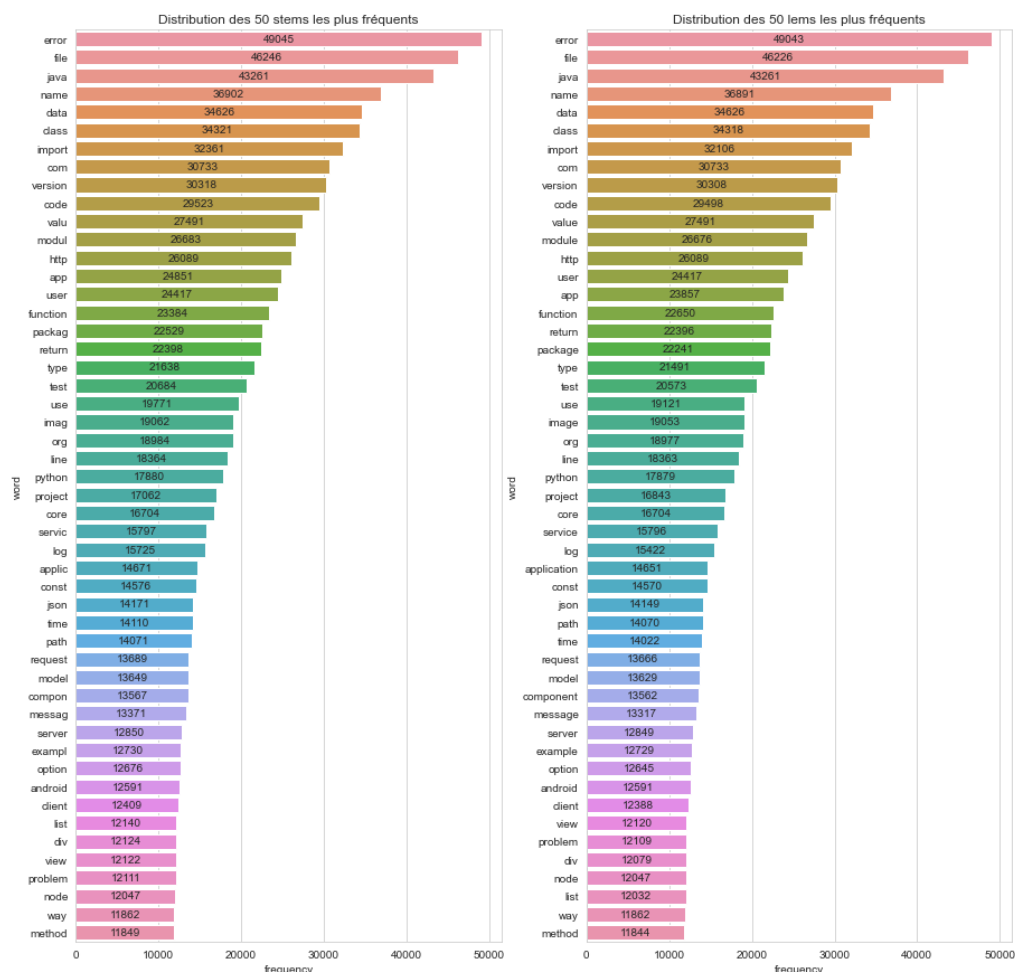
Nous allons normaliser les mots pour éviter les forme multiples. Deux techniques sont possibles :

- le Stemming (racinisation) : conserve la racine du mot en supprimant les préfixes et suffixes. Cette méthode est plus rapide, par contre des mots peuvent avoir une même racine mais être différents d'un point de vue sémantique ;
- la Lemmatisation : représente les mots sous leur forme canonique, nous récupérons ainsi des lemmes, ou lems, (nom commun au singulier, verbe à l'infinitif...). Nous obtenons plus de mots proches d'un point de vue « racine » mais pas forcément d'un point de vue sémantique.

La limite de NLTK est que certains termes au pluriel vont être mis au singulier, ce qui va modifier le nom de certaines technologies (« ios » écrit sous forme « io » ou « kera » en « kera »).

Nous avons appliqué ces deux méthodes sur les textes et tags. Pour les tags, nous nous sommes retrouvés avec des lignes sans tags. Nous avons donc supprimé ces lignes du dataset (0.5% des données).

Nous avons comparé les deux méthodes. Sur les 50 premiers mots les plus présents de tout le corpus, nous observons les distributions suivantes :



DISTRIBUTION DES 50 PREMIERS STEMS ET LEMS

Les 16 premiers mots sont les mêmes, après le classement change mais nous retrouvons les mêmes mots.

D'une manière générale, par la lemmatisation nous obtenons 10% de termes en plus. Pour des raisons d'une plus grande richesse sémantique, nous allons garder les termes générés par la lemmatisation.

2.5. Filtrage des données

En vue de l'approche supervisée lors de la modélisation et pour des raisons de performance (vecteurs en sortie de modèle de taille pas trop grande), nous limitons à 200 le nombre de tags différents. Nous conservons ainsi les tags ayant une occurrence d'au moins environ 150.

Nous filtrons ainsi les données : les tags issus de la lemmatisation (« lems tags ») doivent être dans la liste de 200 premiers tags. Par cette méthode, nous nous retrouvons avec des lignes sans tags. Nous supprimons ces lignes (soit 4.9% du dataset).

2.6. Liste des mots à garder

Toujours en vue de la modélisation, nous ne pouvons pas garder tous les mots présents dans le corpus. Nous gardons donc uniquement la liste des 514 lems les plus fréquents (soit ceux avec une occurrence d'au moins 1400). Nous stockons cette liste dans un fichier .joblib afin de pouvoir l'utiliser ultérieurement sans tout relancer des pré-traitements.

3. Derniers traitements avant modélisation

3.1. Split des données

Nous splittons le jeu de données : 80% des données pour le jeu d'entraînement et 20% pour le jeu de test. Le jeu de test nous servira à comparer les approches.

3.2. Vectorisation des données

Nous utilisons la méthode tf-IDF. Cette méthode permet de pondérer la fréquence d'un mot dans un document selon sa fréquence dans les autres documents du corpus. La fréquence d'un mot d'un document (tf) est multipliée par la fréquence inverse de document (idf). Cette dernière consiste à calculer le logarithme (en base 10) de l'inverse de la proportion de documents du corpus qui contiennent le terme. Ainsi, la fréquence de mots présents dans de nombreux documents est diminuée.

Cette méthode est adaptée ici car nous avons beaucoup de termes génériques et des documents de taille variable. La vectorisation s'effectue sur la liste des 514 lems les plus fréquents avec TfidfVectorizer de la librairie de scikit-learn.

4. Modélisations effectuées

Nous allons dans cette partie tester des méthodes de taggage des textes. Nous effectuons tout d'abord une approche non supervisée, basés sur le texte en lui-même, sans utiliser les tags déjà existants. Ensuite, nous effectuerons une approche supervisée des méthodes de taggage, utilisant ces tags déjà existants.

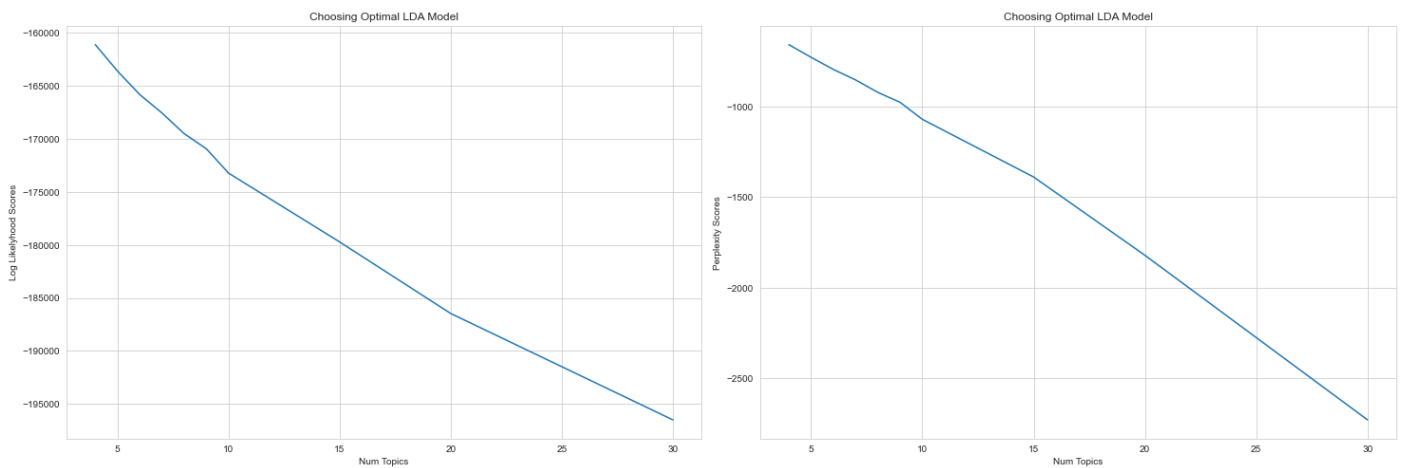
4.1. Approche non supervisée : Latent Dirichlet Allocation (LDA)

Nous utilisons l'algorithme appelé Allocation Latente de Dirichlet (Latent Dirichlet Allocation en anglais) de la librairie scikit-learn. Cet algorithme nous permet de trouver les n sujets (= topics) les plus importants dans nos documents. Un nombre n de topics est fixé et le modèle cherche à apprendre ces topics représentés dans chaque document et les mots associés à ces topics.

a. Choix du nombre de sujets

Pour le choix du nombre de topics, nous utilisons un GridSearchCV (grille de validation), en prenant comme hyperparamètre ce nombre de topics dont les valeurs seront prises dans la liste suivante : 4, 5, 6, 7, 8, 9, 10, 15, 20, 30.

Nous avons utilisé comme scores pour l'évaluation des modèles le maximum de vraisemblance (log-likelihood) et la perplexité (perplexity). Plus le log-likelihood est élevé, meilleur est le modèle et plus la perplexité est basse, meilleure est le modèle. Voici les courbes obtenues (attention, ici nous avons représenté l'opposé de la perplexité donc un score élevé est meilleur) :



EVOLUTION DES SCORES DE LOG LIKE HOOD ET PERPLEXITÉ SELON LE NOMBRE DE TOPICS

D'après les courbes ci-dessus, le nombre optimal de topics est de 4. Nous avons donc retenu ce nombre. La durée du GridSearchCV a été d'environ 25 min.

Remarque : ce modèle n'est pas parfait et après différentes recherches, il se trouve que le modèle implémenté dans scikit-learn n'est pas optimal. La librairie Gensim propose une meilleure implémentation de LDA, en utilisant notamment le score de cohérence. Par manque de temps, cette librairie n'a pas pu être testée.

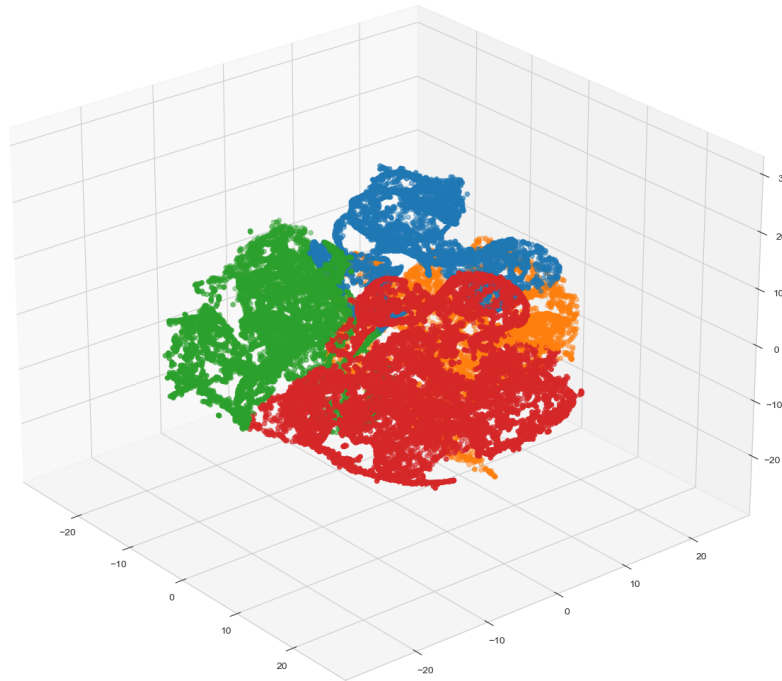
b. Répartition des topics

Voici les informations sur les 4 topics retenus :

Topic n°	Top 10 des mots associés	Nombre de textes associés	Pourcentage de textes associés
1	java docker flutter android com version io app spring container	6139	17.3%
2	file python package version line error command project import code	7562	21.2%
3	error request module http json response client service app function	9538	26.8%
4	value data class color button view component item function list	12359	34.7%

Les topics ne sont pas trop déséquilibrés (pas de topics avec moins de 10% ou un topic très majoritaire). Nous notons la présence néanmoins de termes génériques comme « error », « function » présents dans plusieurs topics.

Voici la représentation en 3D du jeu de données d'entraînement par t-SNE en assignant à chaque point son topic.



REPRÉSENTATION EN 3D DU JEU DE DONNÉES TRAIN PAR T-SNE

Les topics sont globalement bien séparés, nous observons néanmoins quelques recouvrements.

c. Prédiction des tags

Nous avons construit une fonction de prédiction des tags à appliquer sur le jeu de test, en vue de comparer ultérieurement cette prédiction à celle faite par une approche supervisée. Nous avons appliqué cette fonction en utilisant deux dictionnaires différents :

- un avec les 20 mots les plus représentatifs de chaque topic ;
- l'autre avec les 50 mots les plus représentatifs de chaque topic.

Enfin, dans cette fonction, nous filtrons les tags prédits sur les lemmes présents dans le texte.

Remarque : nous avons également testé la méthode non supervisée de NMF (Negative Matrix Factorization). Nous avons obtenu à peu près les mêmes topics, avec néanmoins des termes très largement prédominants. N'ayant pas de métriques pour ce modèle (pour le choix du nombre de topics par exemple), nous n'avons pas retenu ce modèle.

4.2. Approche supervisée

Dans toute cette partie, nous avons exclusivement utilisé la librairie scikit-learn.

a. Pré-traitement

Suppression des doublons dans les tags

Nous devons encore prétraiter nos données afin de pouvoir utiliser les méthodes supervisées. En effet, lorsque nous avons récupéré nos « lems tags » (c'est-à-dire les tags indiqués par les utilisateurs et

que nous avons traités pour les mettre sous forme de lems), nous avons des doublons au sein d'une même liste de tags pour un texte. Nous devons donc supprimer les doublons.

Réduction de dimension

Nous avons gardé les 514 lems les plus présents, ce qui après vectorisation, donne 514 variables en entrée. Nous allons réduire ce nombre. Dans un premier temps, nous avons utilisé une analyse en composantes principales (PCA). Néanmoins, comme ensuite nous avons mis en place une pipeline contenant vectorisation, réduction de dimension et classification, nous sommes passés à TruncatedSVD (décomposition en valeurs singulières, aussi appelée latent semantic analysis [LSA] dans le cadre d'utilisation avec la vectorisation tf-idf), à cause de la présence de matrice creuse que la PCA ne pouvait traiter telle quelle dans la pipeline.

Dans les deux cas, nous voulions avoir au moins 85% de variance cumulée expliquée. Dans le cadre de PCA, nous avons réduit à 324 variables et avec TruncatedSVD 325 variables, soit dans les deux cas une réduction d'environ 37% du nombre de variables.

Encodage des tags

Nous avons filtré précédemment nos tags sur les 200 « lems tags » les plus fréquents, afin que les données en sortie ne soient pas des vecteurs de trop grande taille (donc limités ici à 200). Nous encodons ces labels sous forme de vecteurs à l'aide de MultiLabelBinarizer.

b. Entrainement des modèles

Mise en place d'une pipeline

Dans une pipeline, nous rassemblons les étapes de vectorisation des données d'entrée, réduction de dimension de ces données et enfin de classification des données. Cette pipeline va ensuite être entraînée en prenant différents classifieurs :

- DummyClassifier pour avoir une base line (noté Dummy par la suite) ;
- KNeighborsClassifier (noté KNN par la suite) ;
- LinearSVC (SVM), couplé à un algorithme de One vs Rest (noté SVM par la suite) ;
- RandomForestClassifier (noté Random Forest par la suite).

Les modèles sont ensuite entraînés avec une validation croisée (cross validation).

Métriques utilisées

Pour comparer les modèles, nous avons utilisé les métriques suivantes :

- accuracy : proportion de prédictions correctes
- précision (micro et macro) : proportion d'items pertinents parmi l'ensemble des items sélectionnés
- recall (micro et macro) : proportion d'items sélectionnés parmi l'ensemble des items pertinents
- f1 score (micro et macro), moyenne harmonique de la précision et du recall

Pour la distinction entre macro et micro, prenons par exemple la précision. La précision macro calcule la précision indépendamment pour chaque classe puis prend la moyenne (toutes les classes sont considérées comme équivalentes). La précision micro agrège les contributions de toutes les classes pour calculer la moyenne. Cette méthode est préférable dans le cas d'une suspicion de classes déséquilibrées. Nous utiliserons de préférence la version micro des métriques. Nous avons également inclus le temps d'entraînement des modèles.

Le tableau suivant rassemble les résultats :

Modèle	Accuracy	Precision micro	Recall micro	f1 micro	Temps d'entraînement
Dummy	0.0	0.013115	0.498786	0.025558	37.1s
KNN	0.100483	0.688424	0.273560	0.391530	3min 58s
SVM	0.147256	0.800425	0.367456	0.503668	3min 48s
Random Forest	0.048851	0.888180	0.115611	0.204583	48min 35s

Le Dummy a le meilleur recall, par contre les autres scores sont très mauvais. La Random Forest a la précision micro la plus élevée, mais les autres scores sont mauvais. Par ailleurs, elle a mis beaucoup plus de temps que KNN ou SVM (48 min vs 4 min !), par conséquent, nous éliminons ce modèle. Le SVM présente des meilleurs scores par rapport au KNN, nous retenons donc ce modèle.

c. Optimisation du SVM

Nous avons ensuite optimisé le modèle de SVM par une recherche de l'hyperparamètre C (paramètre de régularisation), dont la valeur par défaut est 1. Nous avons testé les valeurs 0.001, 0.01, 0.1, 1, 10 et comparé les métriques :

C	Accuracy	Precision micro	Recall micro	f1 micro
0.001	0.003849	0.947473	0.006432	0.012777
0.01	0.063318	0.873285	0.154423	0.262435
0.1	0.122957	0.823625	0.304882	0.445006
1	0.147256	0.800425	0.367456	0.503668
10	0.149812	0.768308	0.393184	0.520163

Pour C = 10, les scores d'accuracy, de recall et de f1 sont les plus élevés, contrairement à la précision. Pour des valeurs plus élevées de C (15, 20, 30,...), l'accuracy diminue et des problèmes de convergence apparaissent (il faut augmenter le nombre d'itérations).

Comme pour l'approche non-supervisée, nous avons conservé deux modèles, ici nous gardons les modèles avec C = 1 et C = 10 pour la comparaison finale.

d. Prédiction des tags

Comme pour l'approche non supervisée, nous avons construit une fonction de prédiction des tags à appliquer sur le jeu de test. Nous avons appliqué cette fonction sur les deux modèles de SVM retenus. Dans cette fonction également, nous filtrons les tags prédits sur les lems présents dans le texte.

4.3. Comparaison des approches non supervisée et supervisée

a. Comparaison qualitative

Nous avons comparé pour des textes donnés du jeu de test les prédictions faites par les modèles non supervisés (LDA avec dictionnaires à 20 mots et 50 mots) et supervisés (SVM C = 1 et SVM C = 10) avec les tags mis par les utilisateurs (« lems tags »).

Tags utilisateurs (lems tags)	Approche non supervisée		Approche supervisée	
	LDA 20 mots	LDA 50 mots	SVM C=1	SVM C=10
github action	project test	project test job way run	server action	server action
reactjs react hook	<i>aucun tag trouvé</i>	context	hook	hook
android viewmodel mvvm	value data class button view return name type	value data class button view return name type page	<i>aucun tag trouvé</i>	type
docker compose codespaces	docker container image	docker container image build test volume	json github docker image	json github docker image
php laravel vue pusher	error http app server	error http app server use com php post route option name config login console	server laravel php image	server laravel php http image

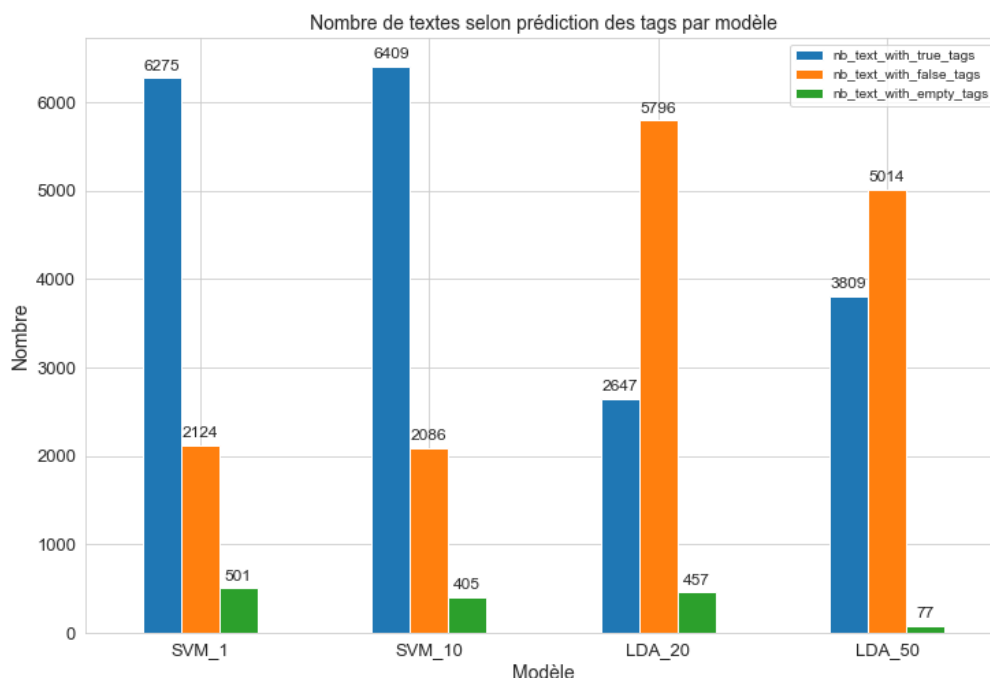
Les tags restent globalement cohérents avec les originaux. Néanmoins, nous ne pouvons conclure de manière générale sur ces quelques tests. Nous avons donc essayé d'utiliser une comparaison « plus quantitative ».

b. Essai de comparaison quantitative

Nous allons quantifier la comparaison entre les deux approches à l'aide d'une fonction qui attribue un score à chaque texte. Si le modèle prédit :

- aucun tag, un score de -1 est attribué (« empty tags ») ;
- aucun tag **de la liste des lems tags**, un score de 0 est attribué (« false tags ») ;
- au moins un tag de la liste des lems tags, un score de 1 est attribué (« true tags »).

Nous comptons ensuite dans le jeu de test, le nombre de textes dans chaque catégorie : textes avec « true tags », « false tags » et « empty tags ». Les résultats suivants ont été obtenus :



RÉPARTITION DU NOMBRE DE TEXTES PAR MODÈLE SELON LE « TYPE » DE TAGS PRÉDITS

Globalement, les modèles supervisés ont plus de textes avec au moins un tag correctement prédit par rapport aux modèles non supervisés.

En passant de C = 1 à C = 10 pour le modèle de SVM, nous augmentons le nombre de textes avec au moins un tag correct (70.5% vs 72%), en diminuant principalement le nombre de textes sans aucun tag

(5.6% vs 4.6%). Par contre, la diminution est faible en ce qui concerne le nombre de textes avec aucun tag prédit dans la liste des lems tags d'origine (23.9% vs 23.4%)

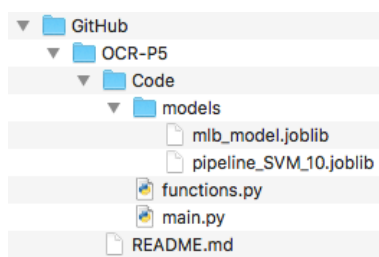
Pour les modèles non supervisés, en augmentant le nombre de mots par topic de 20 à 50, les proportions passent de 28.4% / 66.5% / 5.1% à 40.2% / 59% / 0.8% . Le nombre de textes avec au moins un tag correct est nettement amélioré, en diminuant principalement le nombre de textes sans aucun tag. Le modèle LDA avec 50 mots stockés par topic a le nombre de textes sans aucun tag le plus faible.

La limite de ces comparaisons est que nous avons filtré nos tags prédits sur le contenu du texte (mis sous forme de lems). Or parfois la technologie ou le langage n'est pas forcément indiqué dans le texte, mais l'utilisateur va tagger sa question avec.

Nous retenons néanmoins comme modèle le SVM avec $C = 10$.

5. Mise en production et API

Nous avons stocké dans un répertoire :



- un fichier main.py qui contient le code pour l'API ;
- un fichier functions.py qui contient toutes les fonctions nécessaires à l'exécution du fichier main.py, notamment le traitement du texte et la fonction de prédiction des tags ;
- un répertoire models qui contient les éléments enregistrés au format .joblib nécessaires à l'exécution des fonctions (pipeline du modèle final, encodage des 200 tags).

Ce code est disponible sur [Github](#). L'API a été développée avec [FastApi](#).

Conclusion

Faute de métriques communes, les approches supervisées et non supervisées sont difficilement comparables en terme de performance. Nous avons néanmoins mis en place des comparaisons, notamment vis à vis des tags mis par les utilisateurs.

Même si l'approche non supervisée requiert moins d'étapes (pas de réduction de dimension), elle offre de moins bons résultats : elle prédit beaucoup trop de tags faux. Nous avons donc choisi le modèle de SVM (avec $C = 10$) qui fournit des résultats corrects.

Néanmoins, des pistes d'amélioration sont envisageables, avec un temps alloué au projet plus important :

- utilisation d'une librairie autre que sklearn pour le modèle de LDA (librairie Gensim) pour mieux optimiser le nombre de topics ;
- lors du traitement de texte, utilisation d'un dictionnaire pertinent des mots précis à garder :
 - seuls les mots d'au moins 3 lettres ont été conservés, ce qui a éliminé certains langages de programmation ;
 - ne pas éliminer forcément tous les caractères spéciaux ;
- pour des raisons de performances de machine, nous avons limité certains paramètres :
 - plus de samples (moins de 50000 ici) pourraient être inclus ;
 - plus d'autres variables (514 premiers mots gardés, 200 tags gardés) pourraient être incluses.