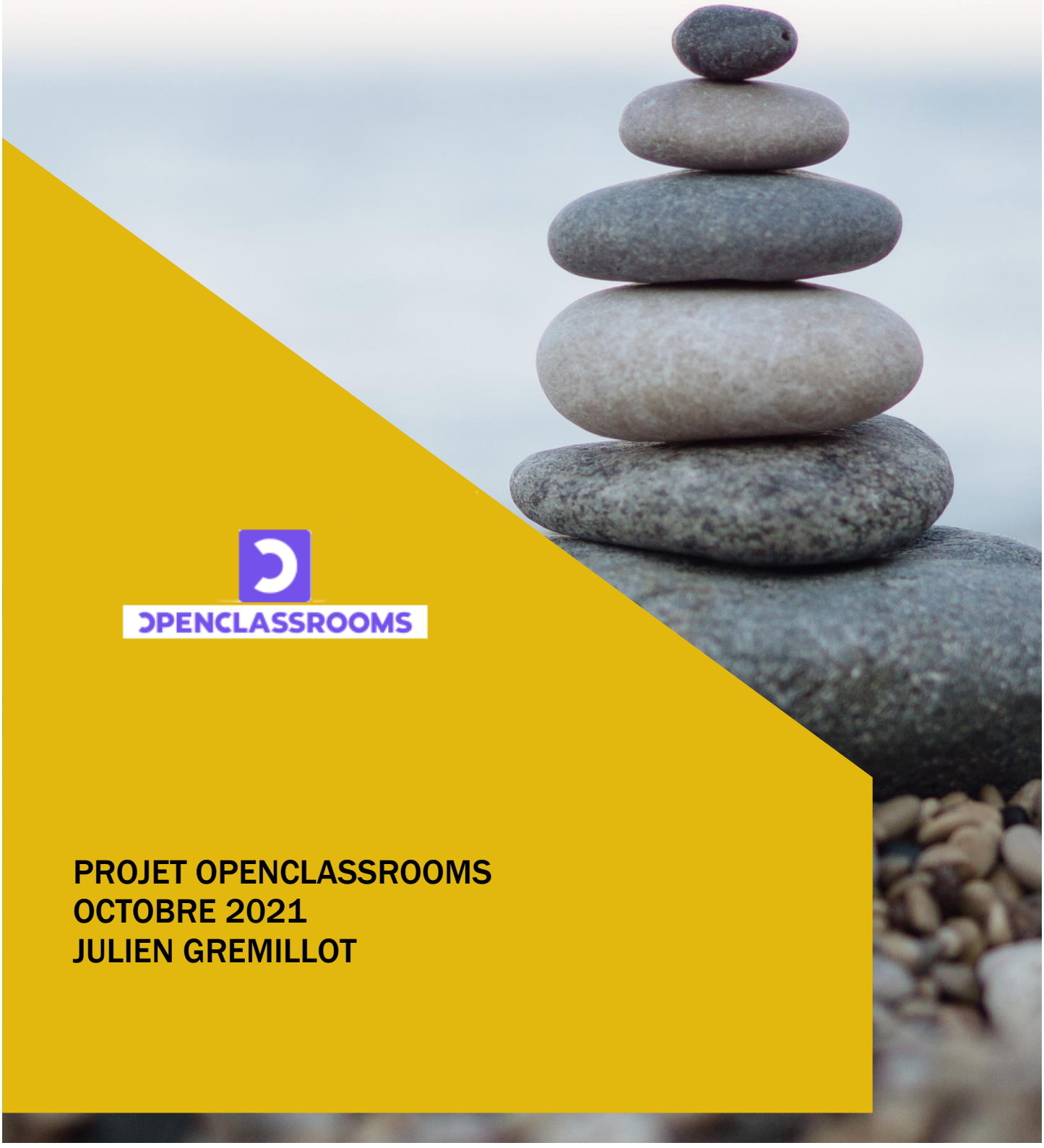
CATEGORISATION DE QUESTIONS



PROJET OPENCLASSROOMS
OCTOBRE 2021
JULIEN GREMILLOT

INTRODUCTION

Le projet

En utilisant les questions du site Stack Overflow et les tags qui leurs sont associées, le but de ce projet est de suggérer des tags appropriés à de nouvelles questions.

Différentes approches ont été utilisées, que nous allons comparer dans ce document.

Table des matières

Exploration des données

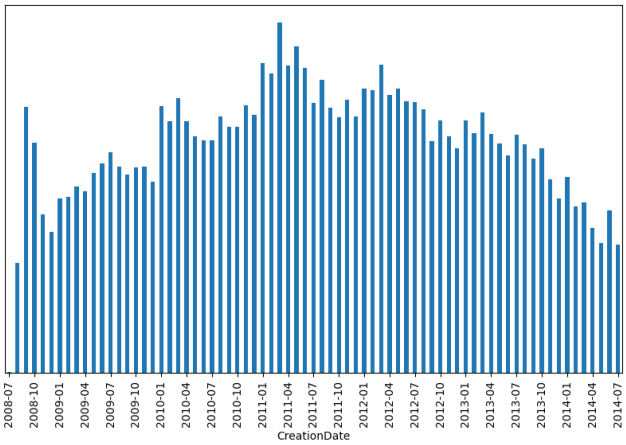
Récupération des questions

Pour commencer, il a été nécessaire de récupérer un grand nombre de questions sur le site de Stack Overflow. Celui-ci met à disposition un outil d'export basé sur des requêtes SQL sur le site Stack Exchange.

J'ai essayé de limiter mon extraction à des questions pertinentes, j'ai donc filtré la base de questions disponibles à celles ayant des tags, ayant reçu des réponses (dont une réponse acceptée), ayant été mise en favoris, avec plus de 1000 vues et un score supérieur à 10.

En itérant sur l'identifiant des questions pour contourner la limitation d'export (fixée à 50.000 questions), j'ai obtenu un ensemble de **188.065 questions**.

Les dates de création de ces questions s'étalent entre 2008 et 2014 de façon relativement uniforme.



Nettoyage des données

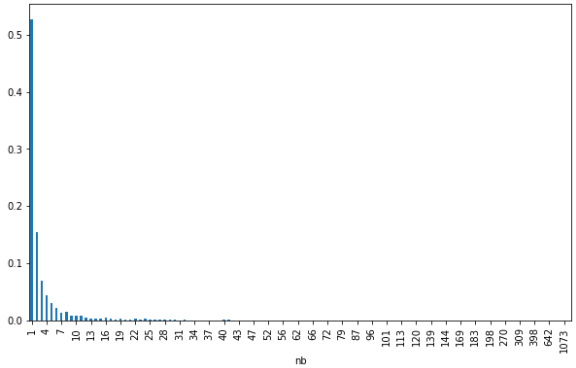
Je n'ai conservé que les colonnes 'Title', 'Body' et 'Tags', puis j'ai converti la chaine de caractères en liste dans la colonne "Tags".

En examinant les tags, j'ai constaté que de nombreux tags comportaient des suffixes indiquant la version ou un sous-module de la catégorie. J'ai donc établi une liste de « préfixes » à retenir et nettoyé l'ensemble des tags, ce qui m'a permis de passer d'un nombre de tags de 4.297 à 4.040 **(-6%)**.

Ce nombre de tags me semblait toutefois encore bien trop élevé pour constituer la cible d'un modèle prédictif.

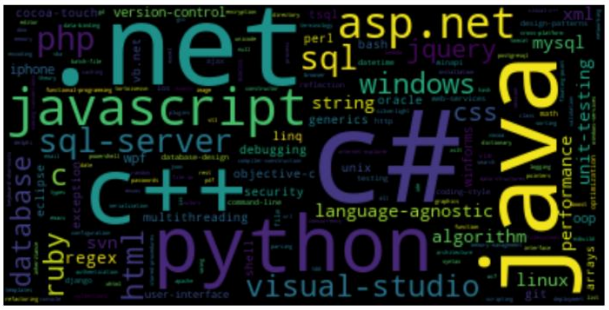
La distribution de ces 4040 tags nous montre :

- 2129 tags présents 1 seule fois (53%)
- 1464 tags entre 1 et 10 fois (36%)
- 361 tags entre 10 et 50 fois (9%)
- 45 tags entre 50 et 100 fois (1%)
- 41 tags plus de 100 fois (1%)

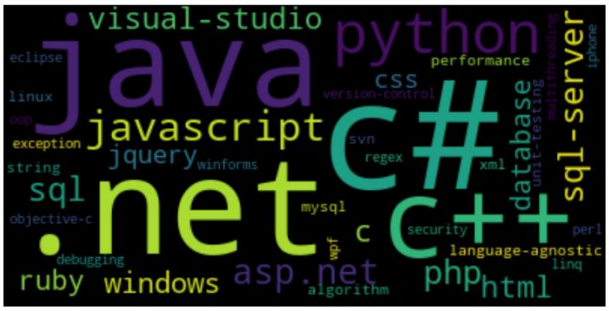


Pour cette raison, j'ai choisi de ne conserver que les **41 tags présents plus de 100 fois** sur notre jeu de données.

Avant simplification :



Après simplification :



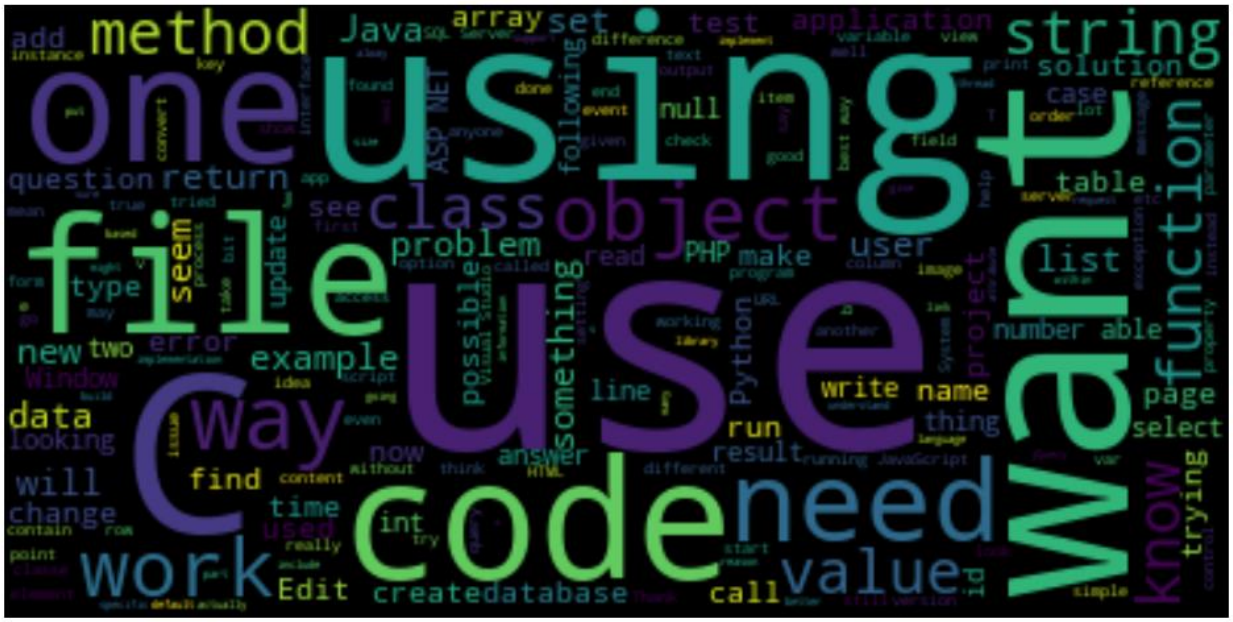
Suppression du HTML et concaténation « Title » & « Body »

Afin d'obtenir un texte cohérent pour entrainer mon modèle, j'ai utilisé la librairie BeautifulSoup afin de supprimer les balises HTML dans le « Body » de la question.

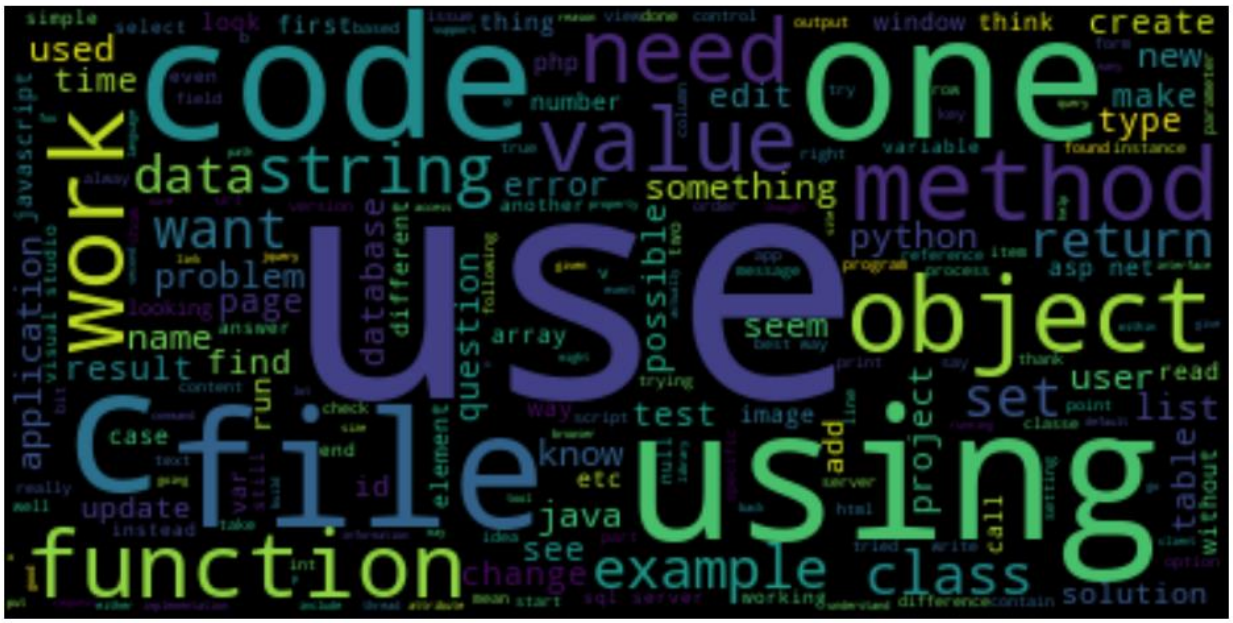
Puis j'ai concaténé les colonnes « Title » et « Body » dans une nouvelle colonne « txt ».

	Title	Body	txt
0	How to convert a Decimal to a Double in C#?	<p>I want to use a <code>Track-Bar</code> to c...	How to convert a Decimal to a Double in C#? I ...
1	Why did the width collapse in the percentage w...	<p>I have an absolutely positioned <code>div</...	Why did the width collapse in the percentage w...
2	How do I calculate someone's age based on a Da...	<p>Given a <code>DateTime</code> representing ...	How do I calculate someone's age based on a Da...
3	Calculate relative time in C#	<p>Given a specific <code>DateTime</code> valu...	Calculate relative time in C# Given a specific...
4	Binary Data in MySQL	<p>How do I store binary data in <a href="http...	Binary Data in MySQL How do I store binary dat...

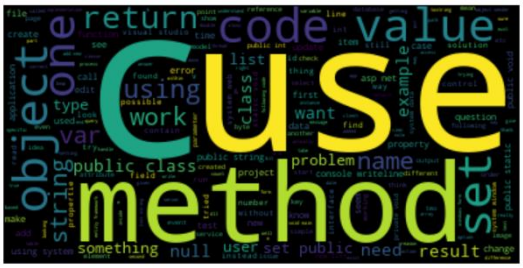
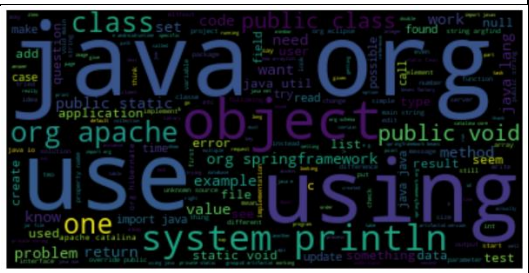
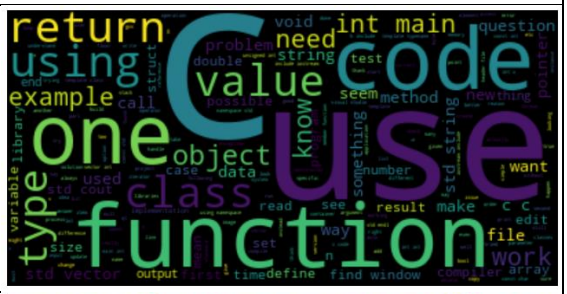
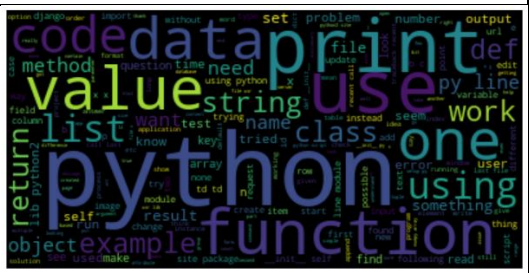
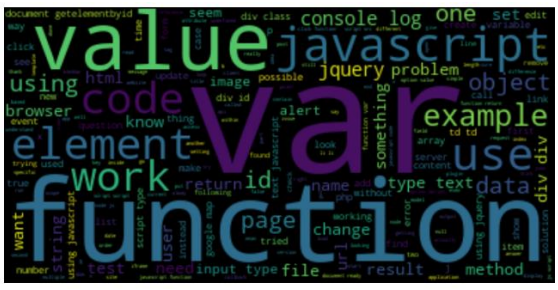
J'observe le contenu du texte :



Puis je fais une suppression de « stop-words » :



Enfin, je regarde le texte des questions pour les premiers tags retenus :

C#		.net	
java		C++	
python		javascript	

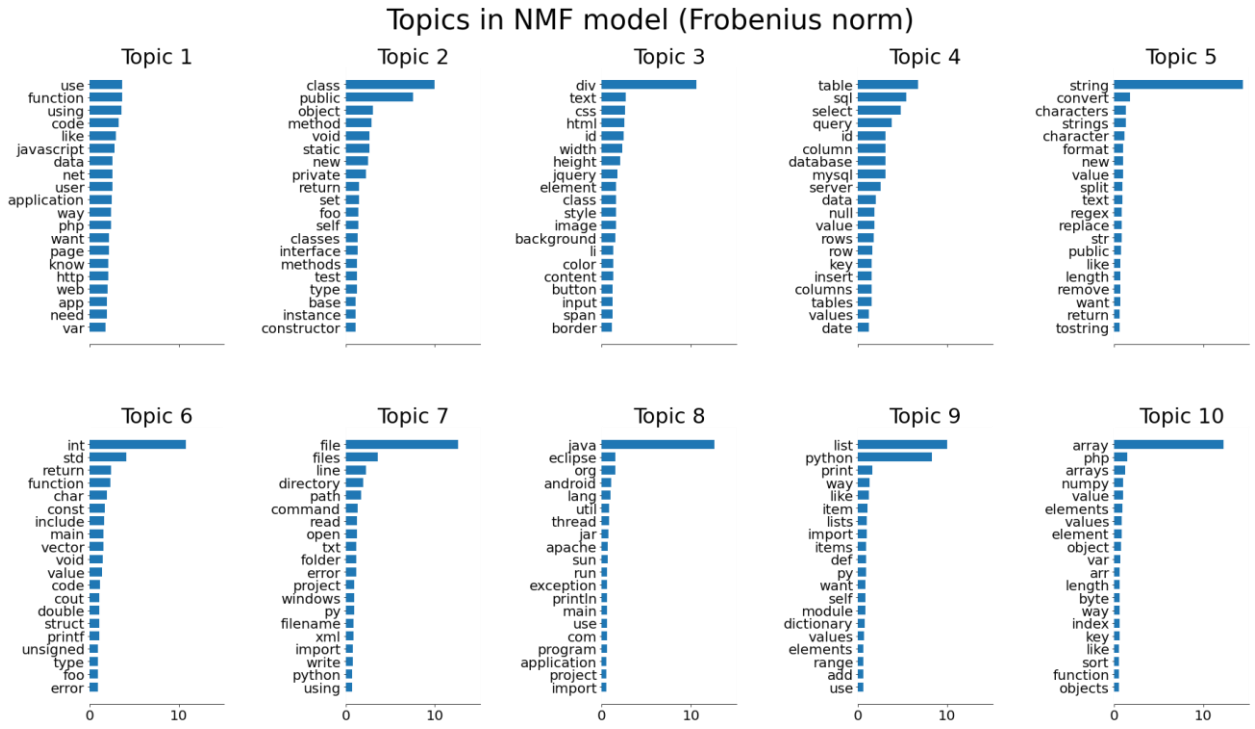
Modélisation

Approche non-supervisée

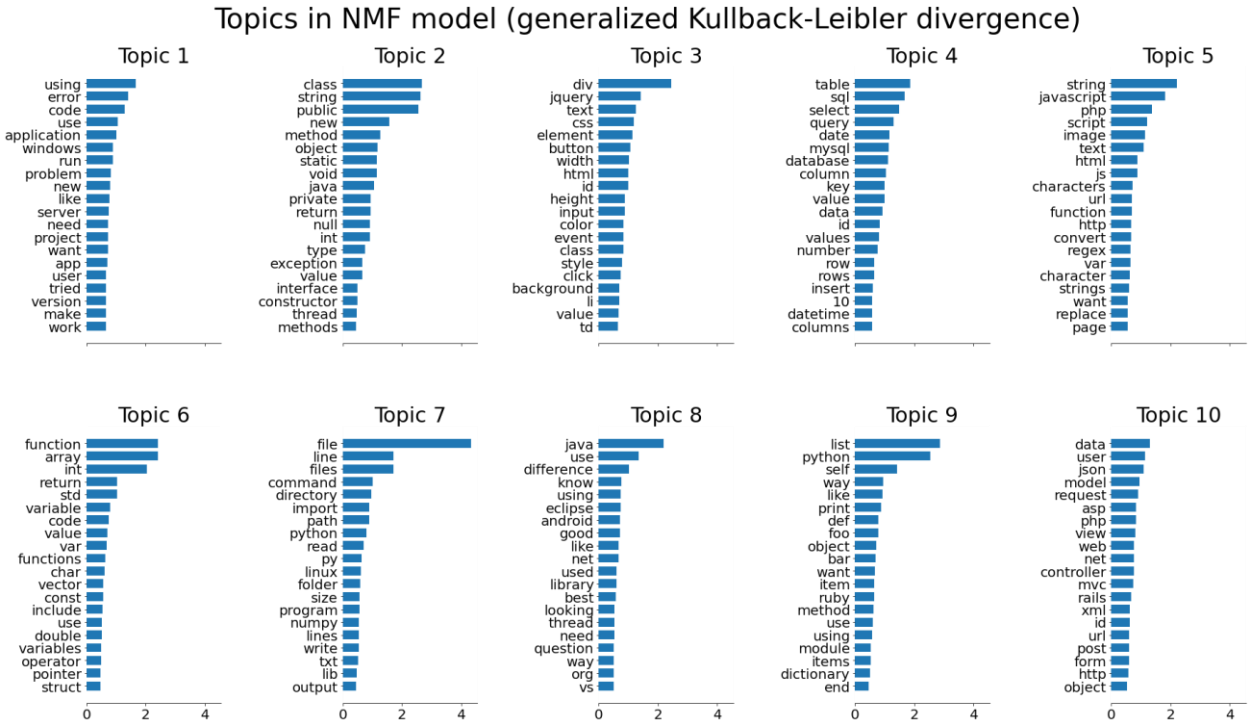
J'ai testé différents types de modèles non-supervisés, pour obtenir des catégories à partir du texte des questions :

- **La méthode NMF** (« Non-negative matrix factorization » ou « factorisation matricielle non négative ») est disponible dans la librairie scikit-learn avec 2 options de fonction de perte. Par défaut, le modèle NMF utilise un critère de moindres carrés (LS ou norme de Frobenius des matrices ou "norme trace"). En ajoutant un paramètre "beta_loss", on peut utiliser la divergence de Kullback-Leibler (KL).

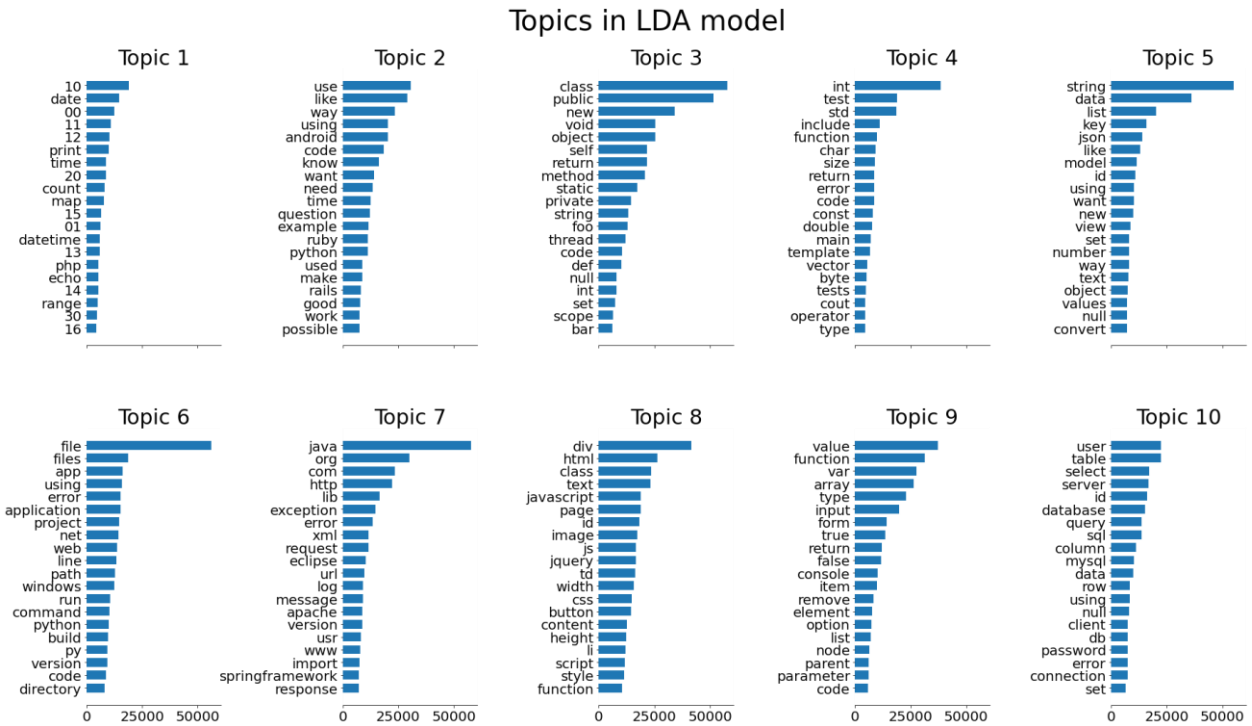
- **La méthode LDA** (« Latent Dirichlet Allocation » ou « Allocation de Dirichlet latente », également disponible dans scikit-learn.



Avec cette première méthode, j'identifie bien les topics 2 (java/objet), 3 (html/css), 4 (sql), 5 (strings), 7 (python/fichiers), 8 (java/android), 9 (python/structure). Il reste 3 topics sur les 10 qui sont plus "flous" à mes yeux. Il s'agit cependant d'un ressenti personnel, car j'ignore peut-être tout d'un domaine identifié par le modèle.



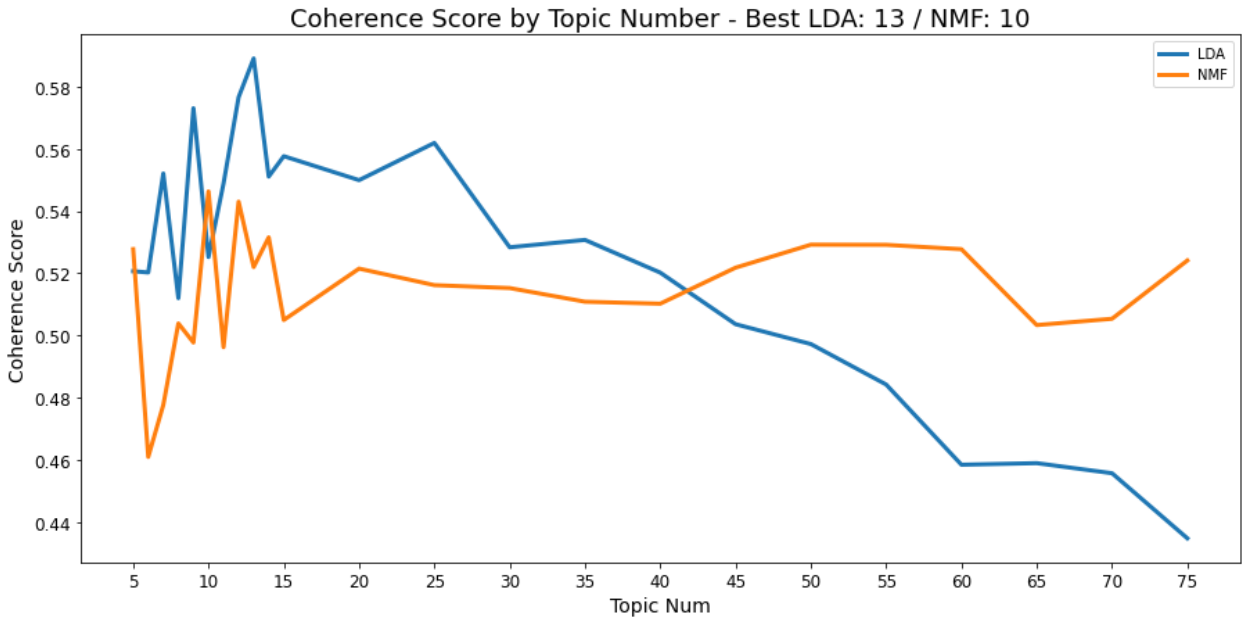
Avec cette deuxième méthode, j'identifie bien les topics 1 (C++/windows), 2, (java/objet), 3 (html/css), 4 (sql), 5 (html/php/js), 7 (python/fichiers), 8 (java/android), 9 (python/structure). Il reste 2 topics sur les 10 qui sont plus "flous" à mes yeux, ce qui semble être mieux que la méthode précédente.



Avec cette troisième méthode, j'identifie les topics 6 (python/windows), 7 (java), 8 (html/js), 9 (js), 10 (sql). Il reste 5 topics sur les 10 qui sont plus "flous" à mes yeux, ce qui semble être moins bien que les deux méthodes précédentes.

Pour obtenir des scores plus fiables que mes premières impressions, j'utilise une librairie : "**Gensim**" qui permet le calcul d'un score de cohérence. En refaisant le calcul des modèles via cette librairie, j'ai obtenu avec le modèle LDA obtient un meilleur score de cohérence que le NMF, ce qui va à l'encontre de mon intuition initiale mais qui confirme que le modèle est capable de trouver des catégories dont je n'ai pas idée.

J'avais choisi le nombre de 10 catégories pour entrainer ces modèles, mais ce choix étant arbitraire et puisque je dispose d'un critère d'évaluation grâce au score de cohérence, j'ai procédé à des tests supplémentaires afin de déterminer le nombre de catégories optimal.



C'est donc bien le modèle « **LDA** » qui obtient le meilleur score de cohérence, et son score maximal est obtenu avec une catégorisation sur **13 « topics ».**

Approche supervisée

J'ai ensuite expérimenté la modélisation supervisée. Une difficulté dans le cadre de la prédiction de catégories, c'est que l'on a des valeurs multiples en sortie du modèle. J'ai donc procédé à la transformation de la liste de tags en matrice binaire à l'aide d'un **MultiLabelBinazer** de la librairie scikit-learn.

Le texte de la question en entrée devait également être transformé. J'ai choisi le « **TdIdfVectorizer** », car il permet de dégager plus d'informations sur le texte qu'un simple « **CountVectorizer** ».

	OneVsRestClassifier LogisticRegression	ClassifierChain LogisticRegression	DecisionTreeClassifier	RandomForestClassifier	KNeighborsClassifier & MLkNN	OneVsRestClassifier SVC
Accuracy	0.451	0.489	0.422	0.422	0.323	**0.514**
Precision	0.655	0.701	0.661	0.661	0.504	**0.835**
Recall	0.612	0.657	0.642	0.642	0.467	**0.644**
F1 Score	0.613	0.658	0.627	0.627	0.467	**0.715**
Jaccard	0.572	0.616	0.574	0.574	0.430	**0.649**

A propos des scores : on est ici dans un contexte de prédiction multilabels, il faut donc tenir compte du nombre de labels prédits et du nombre de labels attendus.

Accuracy : moyenne des quotients « labels justes / total prédits ou attendus »
Precision : moyenne des quotients « labels justes / total attendus »
Recall : moyenne des quotients « labels justes / total prédits »
F1 score : moyenne harmonique de la « precision » et du « recall »
Jaccard : rapport entre l'intersection des labels et l'union des labels - c'est la similarité des deux ensembles.

C'est donc le modèle **OneVsRestClassifier avec un SVC** qui permet d'obtenir les meilleurs scores.

J'ai cherché dans un second temps à optimiser les résultats de ce modèle à l'aide d'un **GridSearchCV** permettant de tester différents paramètres.

```
debut = time.time()
classifier = Pipeline([
    ('vect', TfidfVectorizer()),
    ('clf', OneVsRestClassifier(SVC(kernel='linear')))])
param_grid = {
    'vect__max_df': (0.5, 0.75, 1.0),
    'vect__max_features': (None, 5000, 10000, 50000),
    'vect__ngram_range': ((1, 1), (1, 2), (1, 3)),
    "clf__estimator__C": [1, 2, 4, 8],
    "clf__estimator__kernel": ["poly", "rbf"],
    "clf__estimator__degree": [1, 2, 3, 4]
}
search = GridSearchCV(classifier, param_grid, n_jobs=-1, scoring='f1')
search.fit(X_train_1, y_train_1)
print("Best parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_)
print("Temps écoulé pour ce GridSearchCV : {:.2f}".format(time.time() - debut))
```

Malheureusement, les ressources de mon ordinateur personnel ne m'ont pas permis de terminer le calcul après 48h à 100% de CPU 8 cores.

J'ai finalement testé moins de paramètres sur seulement 10% des données.

```
Best parameter (CV score=nan):
{'clf__estimator__C': 1, 'clf__estimator__degree': 2, 'clf__estimator__kernel': 'linear', 'vect__max_df': 0.5, 'vect__max_fe
atures': None}
Temps écoulé pour ce GridSearchCV : 31224.94
```

En utilisant les paramètres obtenus et en entrainant à nouveau le modèle, je suis arrivé aux résultats suivants :

	OneVsRestClassifier(SVC) optimisé
Accuracy	0.519
Precision	0.741
Recall	0.693
F1 Score	0.695
Jaccard	0.651

Ces scores sont relativement proches des scores précédents, avec un score F1 un peu moins bon et un score Jaccard un peu meilleur.

J'ai donc conservé le modèle précédent et l'ai exporté au format Pickle pour pouvoir l'utiliser dans l'application cible.

Approche semi-supervisée

J'ai également testé une approche semi-supervisée, qui enchaine :
- CountVectorizer
- LatentDirichletAllocation (avec un nombre de « topics » à 13)
- OneVsRestClassifier(SVC(kernel='linear'))

Le « pipeline » ainsi construit permet de déduire des catégories depuis le texte en mode non-supervisé, puis d'y associer le meilleur modèle supervisé pour la prédiction des tags.

Les résultats obtenus se sont révélés très mauvais :

	Pipeline semi-supervisé
Accuracy	0.042
Precision	0.115
Recall	0.049
F1 Score	0.065
Jaccard	0.055

Je n'ai donc pas poursuivi dans cette approche.

Déploiement d'une API

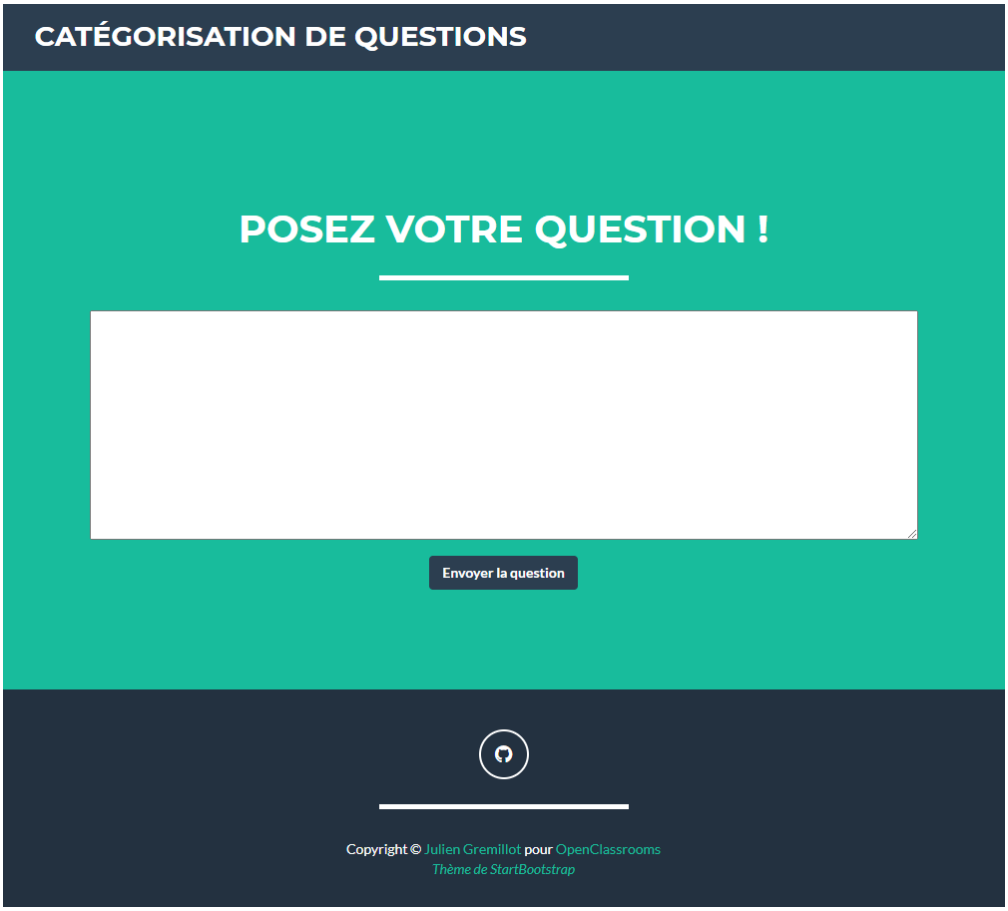
Développement

J'ai réalisé mes développements sous l'IDE « PyCharm » de JetBrains.

Après avoir suivi le cours OpenClassrooms « Concevez un site avec Flask », j'ai repris l'architecture proposée pour le projet lié à ce cours, ainsi que les styles et modèles de templates.

L'application est disponible à l'adresse suivante :

https://categorize-questions.herokuapp.com/



Le code produit reprend les étapes réalisées lors de l'exploration de données, à savoir :

- Suppression du HTML (à l'aide de BeautifulSoup)
- Passage en minuscules
- Tokenisation
- Suppression des « stop-words »

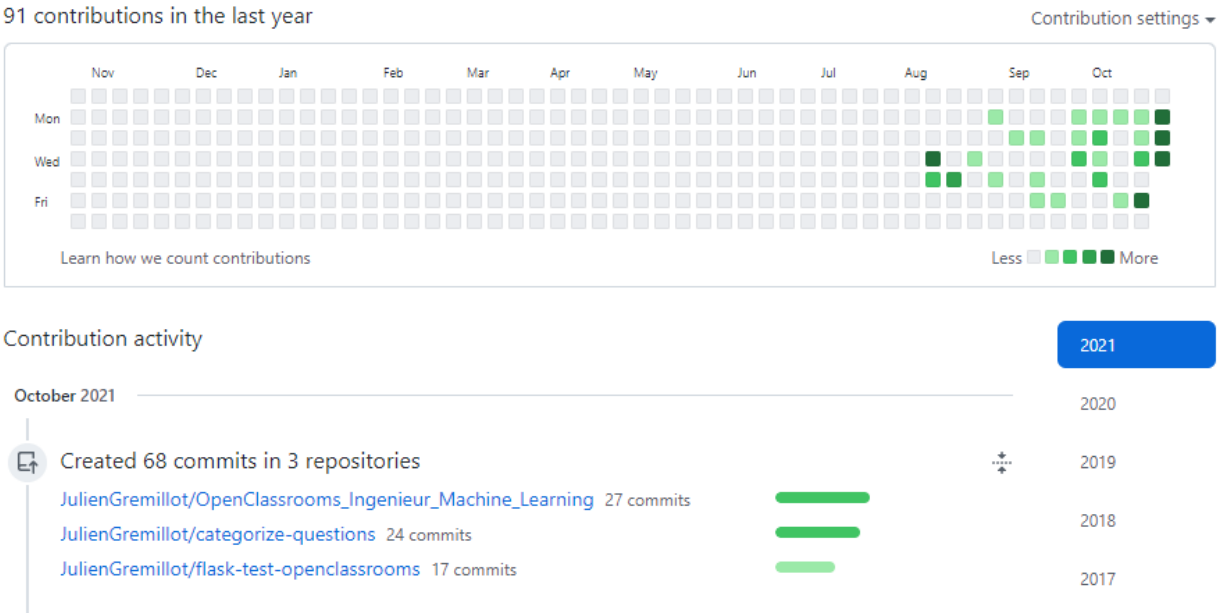
La prédiction est ensuite réalisée à l'aide du modèle exporté depuis les notebooks, et la sortie retransformée via le MultiLabelBinazer (également exporté depuis les notebooks).

J'obtiens donc une liste de tags correspondant à la question (ou un message « Aucune catégorie identifiée » si le modèle a échoué à prédire un tag).


CATÉGORISATION DE QUESTIONS
CATÉGORIES POUR VOTRE QUESTION :
html
python
string
Copyright © Julien Gremillot pour OpenClassrooms
Thème de StartBootstrap

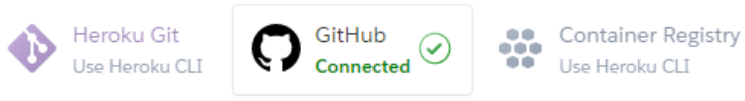
Gestion des versions

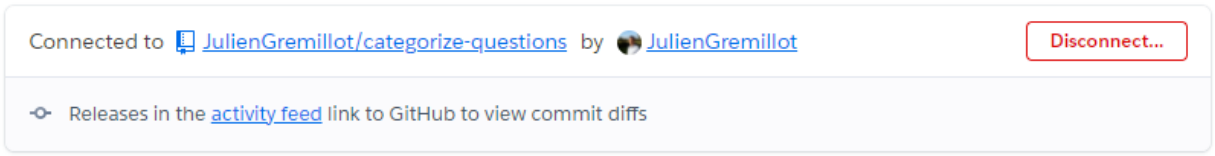
Depuis le début de mon parcours « Ingénieur Machine Learning » sur OpenClassrooms, j'ai pris l'habitude de versionner mes notebooks et autres développements sur Github, dans un repository global pour cette formation : [OpenClassrooms_Ingenieur_Machine_Learning]



Cependant, lorsque j'ai suivi le cours « Concevez un site avec Flask » pour la réalisation de ce projet, je me suis aperçu qu'un repository spécifique était préférable pour le déploiement sur Heroku.

J'ai donc créé un nouveau repository pour l'application à l'adresse suivante :
https://github.com/JulienGremillot/categorize-questions





La connexion avec Heroku a été très facile à mettre en place et les déploiements se font en un simple clic.