

# Projet 5

## Catégorisez automatiquement des questions

---

Parcours Ingénieur Machine Learning

Soutenance le 3 Septembre 2021



QITOUT Kenza

**Mentor** : Maïeul Lombard

**Evaluateur** : Denis Lecoeuche

Projet complet disponible sur : [https://github.com/KenzaQ/P5\\_OC\\_Parcours\\_IML](https://github.com/KenzaQ/P5_OC_Parcours_IML)

API disponible sur : <https://app-suggestion-tags-p5-oc.herokuapp.com/>

## Table des matières

Introduction.....	1
I. Exportation des données et récupération du corpus des textes .....	1
II. Prétraitement des messages .....	2
a) Création du corpus des messages .....	2
b) Nettoyage des messages .....	3
III. Prétraitement des Tags de Stack Overflow .....	4
IV. Approche non supervisée.....	5
a) Méthode utilisée .....	5
b) Résultats obtenus.....	5
V. Approche supervisée.....	8
Conclusion : Sélection du modèle final et mise en production.....	10

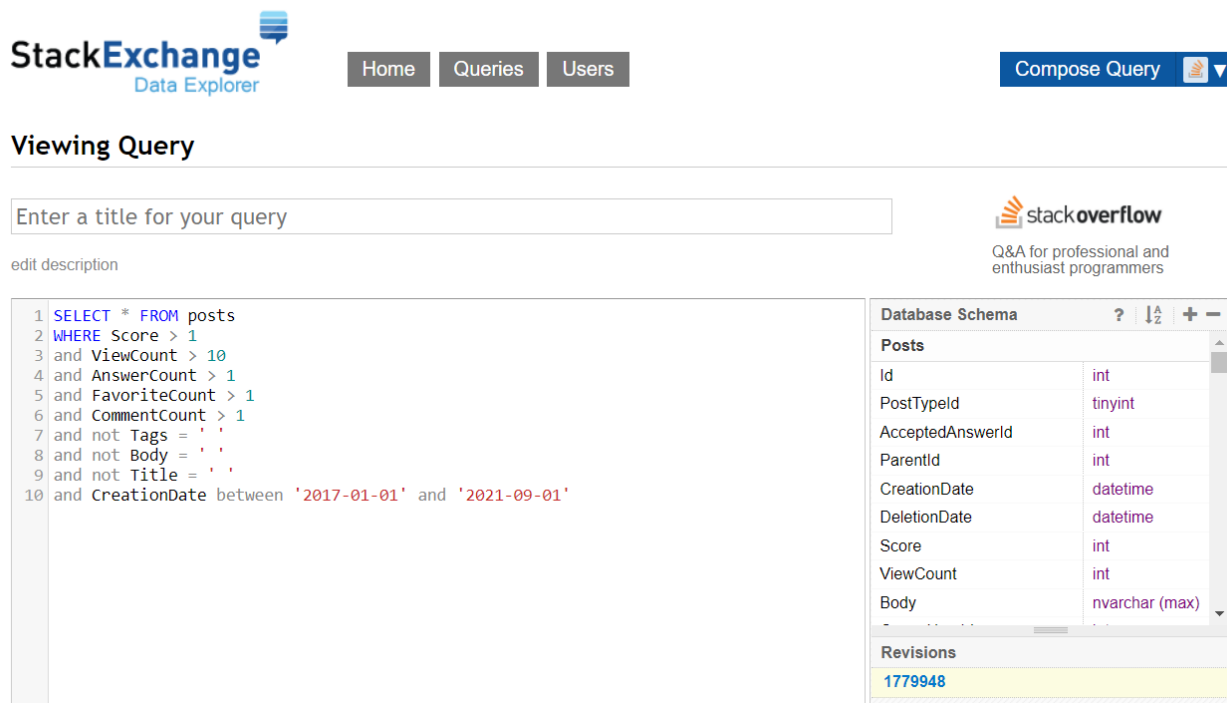
## Introduction

Stack Overflow est un site célèbre de questions-réponses liées au développement informatique. Un utilisateur peut poser une question sur ce site et à l'aide de plusieurs tags, faciliter l'accès aux réponses des autres utilisateurs. L'objectif de ce projet est de développer un système de suggestion de tags pour le site en assignant automatiquement plusieurs tags pertinents à une question. En effet, il est difficile pour des utilisateurs débutants de trouver des tags pertinents à leur question d'où l'étude réalisée lors de ce projet pour faciliter l'utilisation du site des nouveaux utilisateurs en leur proposant des tags associés à un message.

### I. Exportation des données et récupération du corpus des textes

Les données sont disponibles via l'outil d'export de données de Stack Overflow sur le site <https://data.stackexchange.com/stackoverflow/query/new>. Ce site recense de nombreuses informations sur les messages des utilisateurs telles que l'identifiant du message, la date de création/de suppression, le nombre de vues, le message et son titre, la date de dernière modification, le nombre de réponses ...

Pour récupérer les données, il est nécessaire de réaliser une requête SQL sur ce site. J'ai exporté les messages populaires (*Score* > 1) avec un nombre de vues supérieur à 10 (*ViewCount*), un nombre de réponses (*AnswerCount*), de favoris (*FavoriteCount*) et de commentaires (*CommentCount*) supérieur à 1. J'ai également sélectionné les messages les plus récents (depuis le 01/01/2017) et exclusivement les messages possédant des tags (and not *Tags* = ' '), un corps de message (and not *Body* = ' ') et un titre (and not *Title* = ' ').



The screenshot shows the StackExchange Data Explorer interface. At the top, there's a navigation bar with 'Home', 'Queries', and 'Users' buttons, and a 'Compose Query' button. Below this, the 'Viewing Query' section is active. It features a text input field for the query title, an 'edit description' link, and the Stack Overflow logo with the tagline 'Q&A for professional and enthusiast programmers'. The main area displays a SQL query in a text editor:

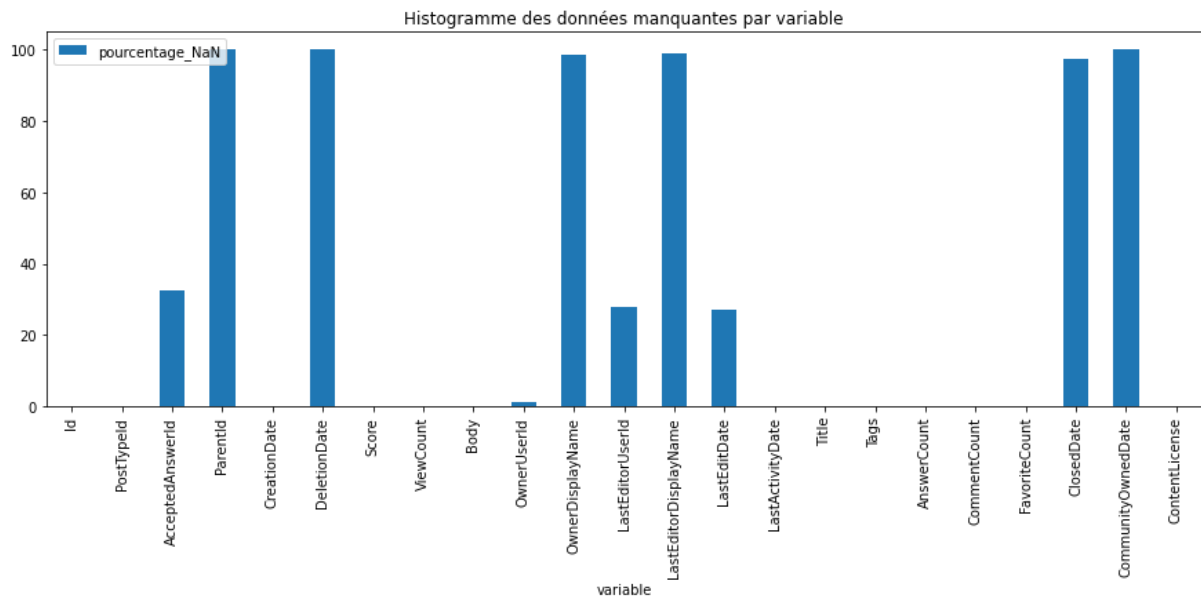
```
1 SELECT * FROM posts
2 WHERE Score > 1
3 and ViewCount > 10
4 and AnswerCount > 1
5 and FavoriteCount > 1
6 and CommentCount > 1
7 and not Tags = ' '
8 and not Body = ' '
9 and not Title = ' '
10 and CreationDate between '2017-01-01' and '2021-09-01'
```

To the right of the query editor, the 'Database Schema' section is visible, showing the structure of the 'Posts' table:

Column Name	Data Type
Id	int
PostTypeId	tinyint
AcceptedAnswerId	int
ParentId	int
CreationDate	datetime
DeletionDate	datetime
Score	int
ViewCount	int
Body	nvarchar (max)

Below the schema, the 'Revisions' section shows a single revision with the ID 1779948.

Le DataFrame qu'on obtient suite à cette requête contient 50 000 lignes et 23 colonnes. L'exploration du DataFrame des données brutes *data\_text* montre un DataFrame composé de 341 964 données manquantes réparties de la manière suivante :



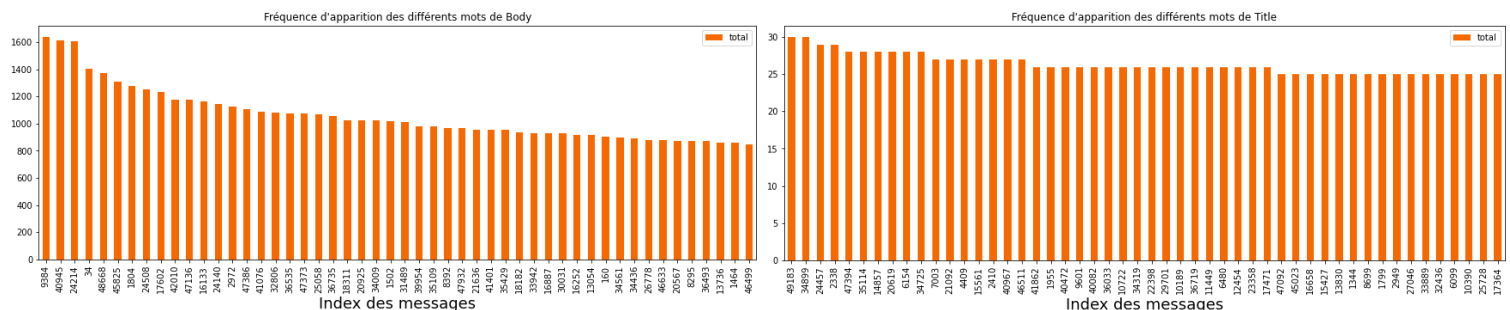
J'ai vérifié que les conditions de la requête SQL étaient bien respectées en créant un nouveau DataFrame *data* ne contenant pas de données manquantes pour les variables '*Body*', '*Title*' et '*Tags*' qui sont les variables utilisées dans l'analyse.

## II. Prétraitement des messages

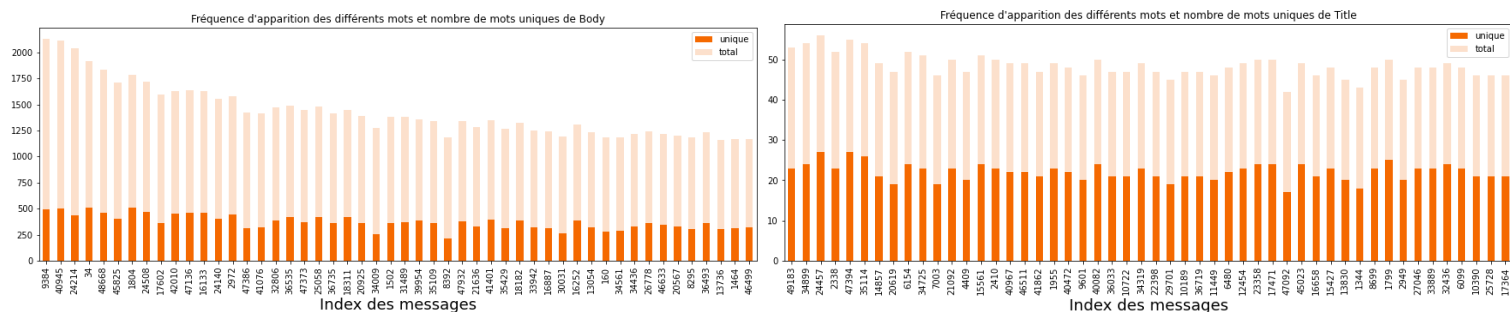
Contrairement aux titres, le corps des messages présentent des balises HTML. Pour les supprimer, j'ai utilisé la librairie **BeautifulSoup** (fonction *suppression\_balises\_html*) sur la variable '*Body*' pour créer une nouvelle variable '*Body\_new*'.

### a) Création du corpus des messages

Le nettoyage des messages et de leurs titres a ensuite été réalisé de la même manière entre les variables '*Body\_new*' et '*Title*'. J'ai créé un dictionnaire *db* de tous les messages et un dictionnaire *db* de tous les titres. J'ai réalisé une analyse des fréquences de mots contenus dans tous les messages et dans tous les titres en récupérant la fréquence d'apparitions des différents mots dans tous les messages. Pour cela, j'ai tokenisé tous les messages normalisés (suppression de la ponctuation et mots en minuscule) avec la fonction **split()** qui va décomposer les messages en tableaux de mots :



Il est possible d'observer quel message (indiqué par son index) présente le plus de mots. Comme attendu, il est possible d'observer que les textes contenus dans le titre sont plus courts que ceux dans les corps des messages. J'ai également étudié le nombre de mots uniques présents dans chaque message avec la fonction *freq\_stats\_corpora1* :



Il est possible d'observer que les titres sont composés d'une proportion plus importante de mots uniques que les corps des messages, qui doivent contenir plus de mots qui se répètent.

## b) Nettoyage des messages

Pour utiliser la meilleure méthode de nettoyage, j'ai comparé différents traitements : Sans traitement, En supprimant les stopwords, **Lemmatisation** et **Stemming**.

Avant cela, j'ai d'abord créé la liste des stopwords qui ne sont pas utiles pour l'étude des messages. J'ai créé une liste des 145 mots les plus fréquents du corpus obtenus avec **nltk.Counter()** et j'ai ajouté la liste par défaut des stopwords en anglais présents dans la librairie **NLTK**.

J'ai ensuite créé des nouvelles variables '*Body\_noms*' et '*Title\_noms*' correspondant aux variables '*Body\_new*' et '*Title*' en ne conservant que les noms et les noms propres grâce au package **Spacy** et la fonction **nlp = spacy.load('en\_core\_web\_sm')**, en supprimant la ponctuation et en transformant tous les mots en minuscule (fonction *conservation\_noms*).

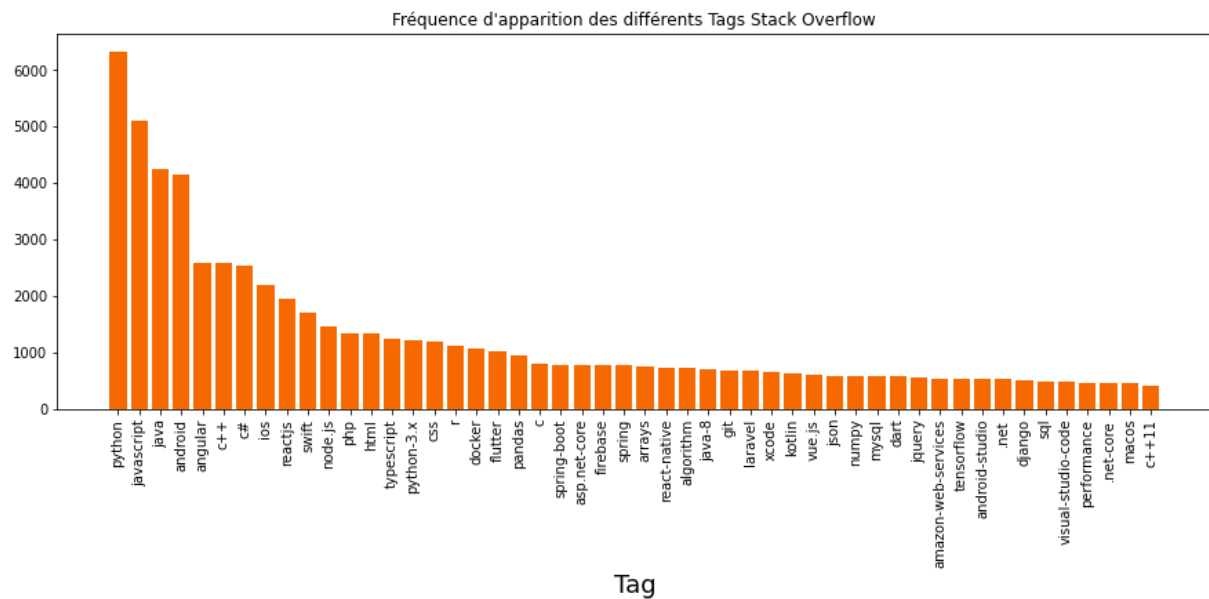
A partir des variables '*Body\_noms*' et '*Title\_noms*', j'ai créé 4 nouvelles variables correspondant à un prétraitement différent :

Nouvelles variables	'Body_nettoye_sans' 'Body_nettoye_sans'	'Body_nettoye_stopwords' 'Body_nettoye_stopwords'	'Body_nettoye_lem' 'Body_nettoye_lem'	'Body_nettoye_stem' 'Body_nettoye_stem'
Fonction	<i>nettoyage_sans_traitement</i>	<i>nettoyage_stopwords</i>	<i>nettoyage_lemmatisation</i>	<i>nettoyage_stemming</i>
Traitement	<ul style="list-style-type: none"> <li>Suppression des nombres seuls (qui ne sont pas dans un mot)</li> <li>Suppression des mots à 1 lettre</li> <li>Suppression des espaces</li> </ul>	<ul style="list-style-type: none"> <li>Tokenisation des chaînes de caractères avec <b>split()</b></li> <li>Suppression des stopwords</li> <li>Suppression des nombres seuls (qui ne sont pas dans un mot)</li> <li>Suppression des mots à 1 lettre</li> <li>Suppression des espaces</li> </ul>	<ul style="list-style-type: none"> <li>Tokenisation des chaînes de caractères avec <b>split()</b></li> <li>Suppression des stopwords</li> <li><b>Lemmatisation</b> avec <b>WordNetLemmatizer()</b></li> <li>Suppression des nombres seuls (qui ne sont pas dans un mot)</li> <li>Suppression des mots à 1 lettre</li> <li>Suppression des espaces</li> </ul>	<ul style="list-style-type: none"> <li>Tokenisation des chaînes de caractères avec <b>split()</b></li> <li>Suppression des stopwords</li> <li><b>Stemming</b> avec <b>EnglishStemmer()</b></li> <li>Suppression des nombres seuls (qui ne sont pas dans un mot)</li> <li>Suppression des mots à 1 lettre</li> <li>Suppression des espaces</li> </ul>

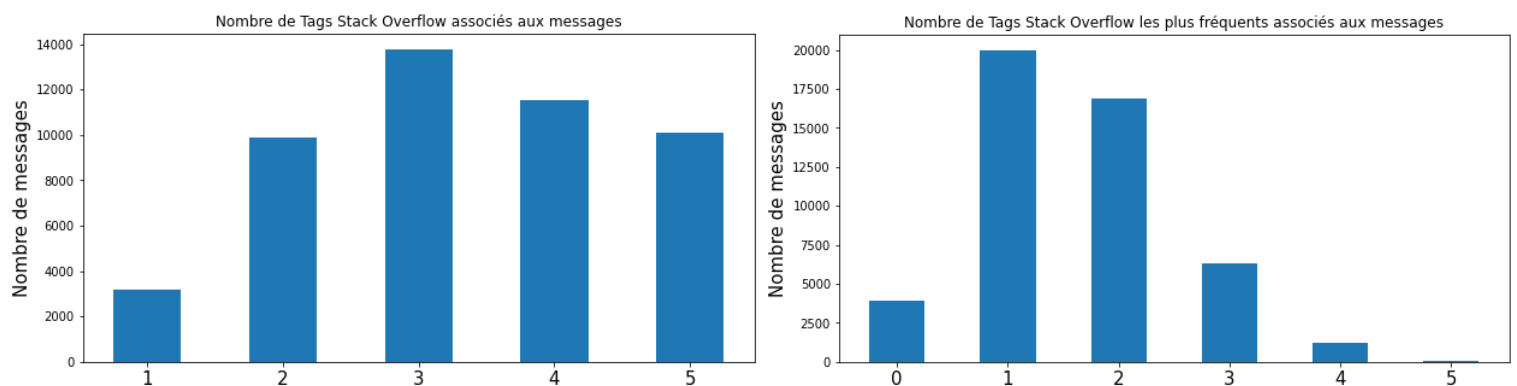
J'ai supprimé les lignes contenant des données manquantes pour les nouvelles variables créées (165 lignes pour les variables liées à 'Body\_new' et 1418 lignes pour les variables liées à 'Title'). J'obtiens un DataFrame de 48 447 lignes.

### III. Prétraitement des Tags de Stack Overflow

Chaque message est associé à un ou plusieurs Tags utilisés pour retrouver facilement une question et recevoir des réponses adaptées des autres utilisateurs du site Stack Overflow. Les Tags sont présentés de la façon suivante : `<python><dask><python-xarray>`. J'ai d'abord nettoyé les Tags en supprimant les symboles `<` et `>` avec le module **re** dans une nouvelle variable 'Tags\_new'. Ce jeu de données est composé de 12 466 Tags différents dont 5 439 n'apparaissant qu'une seule fois :



Pour éviter de compliquer les résultats pour les utilisateurs débutants en leur fournissant des Tags trop précis, je n'ai conservé que les Tags les plus fréquents. Après avoir compté la fréquence d'apparitions de chaque Tag avec **nlk.FreqDist()**, j'ai conservé dans une liste *liste\_Tags\_selectionne* le Top 100 des Tags les plus fréquents dans le jeu de données dans une nouvelle variable 'Tags\_nettoyées', ce qui représente 48.64 % des Tags utilisés. Le nombre de Tags associé à chaque message a été réduit avec la présence de 3 930 messages sans aucun Tag et 19 966 avec un seul Tag :



## IV. Approche non supervisée

### a) Méthode utilisée

Pour cette partie de modélisation, on se place dans une approche non supervisée où l'objectif est de créer nos propres tags que l'on pourra par la suite utiliser dans l'API.

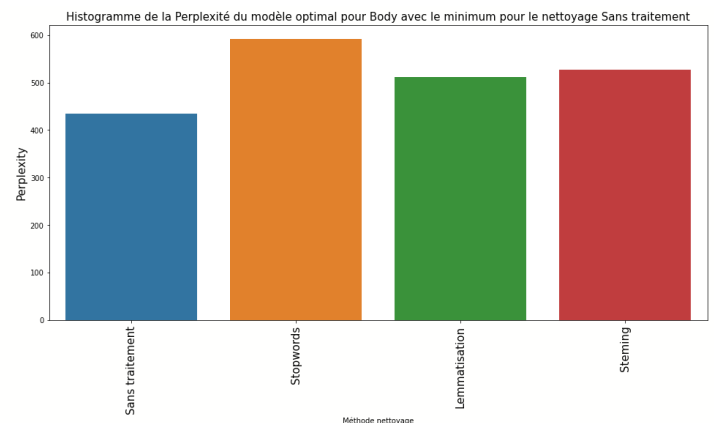
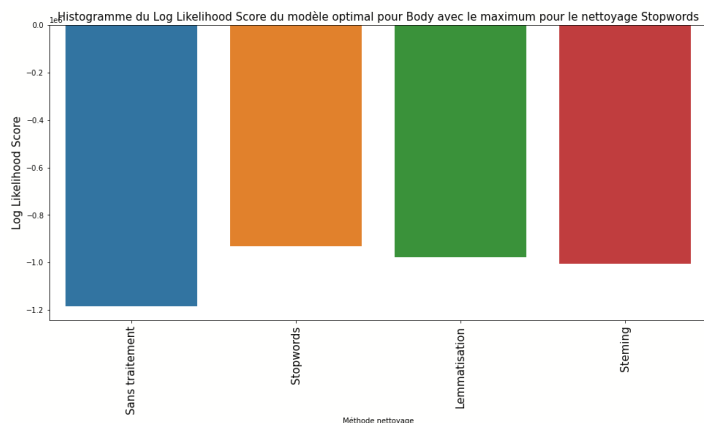
J'ai transformé les messages nettoyés et normalisés selon les 4 traitements décrits précédemment en nouvelles features avec la méthode **LDA (LatentDirichletAllocation())**. Pour cela, j'ai récupéré la fréquence d'apparitions de chaque mot dans une matrice avec **CountVectorizer** et appliqué ce comptage sur la liste de messages avec **tf\_vectorizer.fit\_transform**. J'ai réalisé une recherche des hyperparamètres optimaux de la LDA avec une validation croisée via **GridSearchCV** :

```
search_params = {'n_components': [25, 50, 75, 100], 'learning_decay': [0.5, 0.8], 'max_iter': [5], 'learning_offset': [50], 'learning_method': ['online'], 'n_jobs': [-1], 'max_iter': [5], 'random_state': [0]}
```

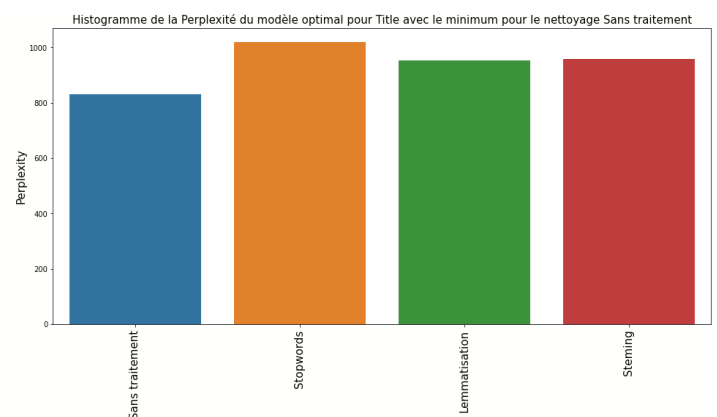
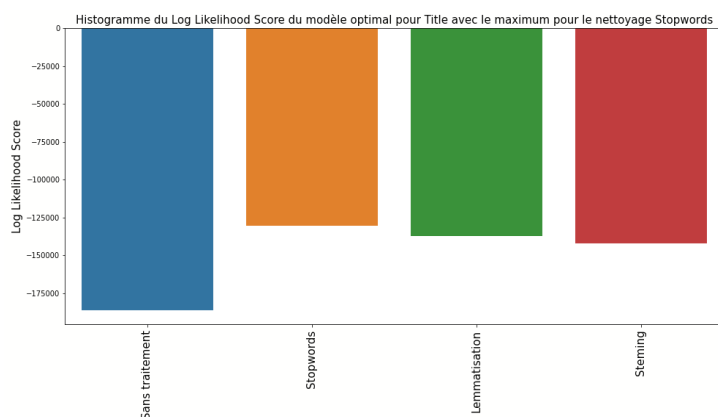
J'ai appliqué la méthode **LDA** sur le comptage de mots et affiché les 10 mots les plus représentatifs des sujets modélisés pour le nombre optimal de Topics trouvé.

### b) Résultats obtenus

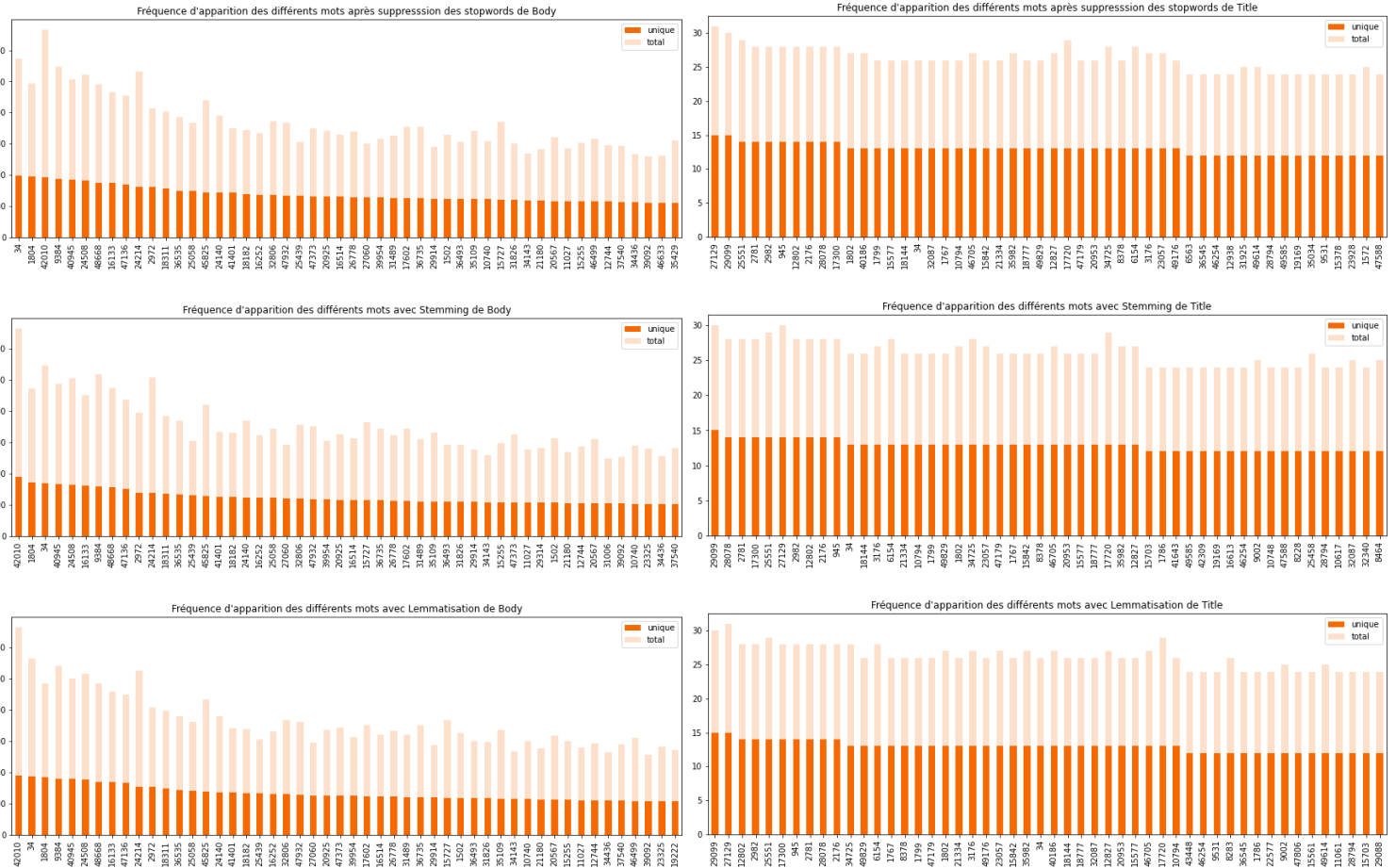
Pour les variables liées à '*Body\_nettoyee*', on obtient de meilleurs résultats avec le nettoyage Stopwords qui a un **Log Likelihood Score** de -9.308.10<sup>5</sup> et une **Perplexity** de 591.47 (même si la **Perplexity** est plus faible pour le nettoyage Sans Traitement) :



L'approche par la LDA des variables liées à '*Title\_nettoyee*' donne des résultats similaires avec un **Log Likelihood Score** de -130638.152 et une **Perplexity** de 1019.09 pour le nettoyage Stopwords :



L'objectif de ce projet est de générer des Tags. En utilisant le **Stemming** pour nettoyer les messages, on récupère uniquement la racine des mots, soit des mots coupés non pertinents et non utilisables pour les utilisateurs via une API. On peut donc éviter d'utiliser le nettoyage **Stemming**. De plus, l'analyse des fréquences d'apparitions des mots avec **Stemming**, avec **Lemmatisation** ne montre pas de différence dans la proportion des mots uniques sur le total :



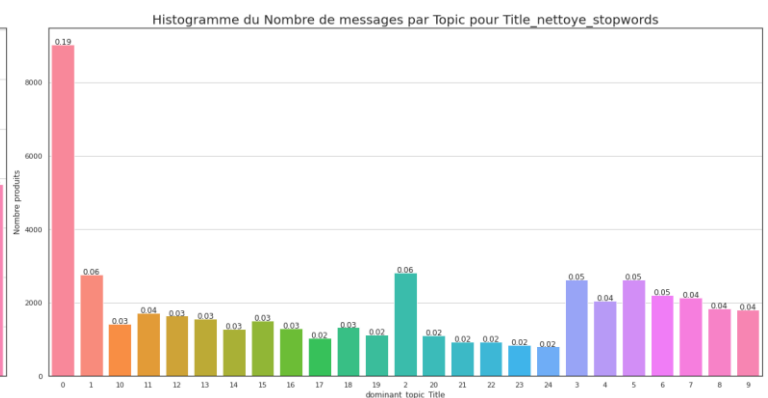
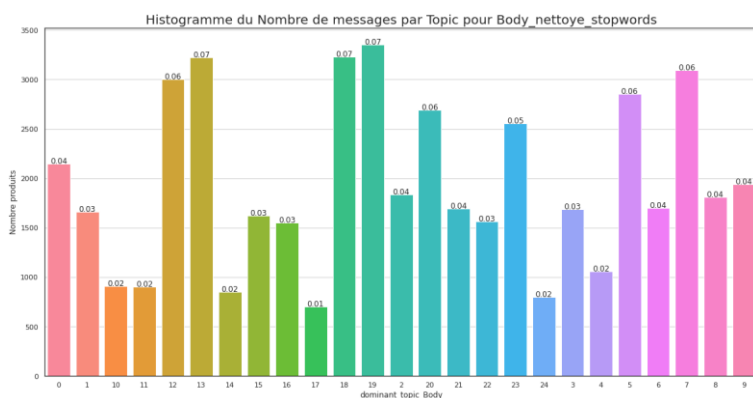
Il est donc plus intéressant d'utiliser le nettoyage Stopwords et de sélectionner la méthode **LDA** appliquée sur les variables *'Body\_nettoye\_stopwords'* et *'Title\_nettoye\_stopwords'* qui donne 25 Topics :

	'Body_nettoye_stopwords'	'Title_nettoye_stopwords'
Topic 0	java values column methods classes dataframe interface names vector kotlin fragment pandas editor dataset bash distribution activity fragments interfaces features	css columns variables pointer interface methods pipeline point style frame host svg safari menu const char conversion solution gcc comparison
Topic 1	component result form field components item fields parent group child network entity forms engine ip children internet level case property	material keras vue store email warning s3 router implementation vscode crash struct provider devices feature backend cpu pointers amazon airflow
Topic 2	number size memory numbers video points frame title seconds mb times gb users limit amount intellig max count ram minutes	bar images client components child kubernetec video azure integer route website plot release db tab auto part practice oracle services
Topic 3	ios background color bar layout search top xml cell icon bottom apple environment macos grid buttons iphone bug text button	property typescript parameter parameters position null recyclerview length angular2 login control mvc jpa axios events connect signature hibernate rust phone
Topic 4	name changes lines sql document box control tutorial click node nodes line packages plot flutter series card procedure days documents	script url model argument selenium strings apps branch bit limit anaconda config machine webview hooks schema differences bytes env postman
Topic 5	test answer property tests case lot properties stream difference solutions answers approach questions branch cases unit integer check stackoverflow posts	service input web nodejs screen system algorithm icon pip body target ionic execution points font iphone issues pages tree upload
Topic 6	thanks text button program advance thread process algorithm threads cpu pdf lot article idea ms report part print game answer	user button key row lambda tensorflow arguments space scroll rails pdf widget apk animation bundle maps unity domain max init
Topic 7	android google studio device window php email sdk gradle apps option devices message developer console build tools emulator graph log	objects query size loop date constructor arrays password member range activity condition structure scala declaration cmake projects alternative foreach ruby
Topic 8	exception loop spring module location npm dependency dependencies modules boot execution release install support promise buffer eclipse jar loops configuration	form field output numbers sum characters process typeerror ui fields regex lists action testing modal postgresql constraint dom groups bean



Topic 9	view model custom date plugin navigation height attribute position header filter ideas views animation options models width training loss label	kotlin stream items rows gradle classes task controller repository extension matrix link root dictionary numpy port maven dates generator options
Topic 10	state content react context statement url operation console section design scope callback statements argument profile worker dom contents debugger urls	js event rest github csv mongodb post jquery node cors cache jwt design golang xml print https center driver purpose
Topic 11	json response xcode controller mode framework parameter target stack headers pattern root trace swift pod pods sheet beta simulator default	database spark authentication content mode format entity load pattern room kafka session production firestore fragment architecture ssh ajax debug identity
Topic 12	reference compiler task types behavior implementation case return pointer syntax nodejs runtime definition template gcc tasks argument assembly situation standard	command sql issue npm message device browser mac errors notification syntax installation compile cli os facebook exit manager sign desktop
Topic 13	command line windows path source system machine environment errors warning linux mac ubuntu terminal os package commands directory installation packages	aws boot connection exception tests click names location support end download start eclipse pytorch go versions instances cluster scope effect
Topic 14	string link build store address extension projects cache day import ui word regex null modal month download bundle part instruction	swiftui expression layout macos plugin packages collection keyboard words ef settings streams tables camera dialog box definition graphql equivalent records
Topic 15	script docker container key event repository git keys typescript block events django scripts containers jenkins handler cli dockerfile entry commit	page element django path environment source permission dependency requests address context document base fetch timeout postgres account headers injection shell
Topic 16	image message images firebase items cloud space messages base notification style layer bytes screenshot widget picture material collection angular notifications	xcode map state import validation search results intellij native graph int es6 word users log simulator auth idea area dplyr
Topic 17	input instance call package objects action body validation trouble instances calls camera copy keyboard inputs direction doc pipe subscription country	php response order group mysql item status network vector http div httpclient level tasks top commit gitlab failure calls websocket
Topic 18	python variable functions variables order default constructor parameters values strings lambda case characters module expression matrix words certificate character member	library cloud test instance storage opencv folder jenkins compiler bash sdk modules lines engine initialization timestamp props terminal ip job
Topic 19	api service request core web requests users authentication docs access aspNet rest services token password permission login endpoint account client	memory template container vuejs linux unit thread program ubuntu jest emulator token play matplotlib assembly refresh integration bottom createreactapp startup
Topic 20	table database query row columns rows index db column job tables operator records results result spark mysql queries s3 record	reference operator performance types result powershell configuration excel runtime security behavior times case usage count side vba copy cell annotation
Topic 21	folder client connection directory github side config setup configuration access status message repo resource microsoft settings root dev bootstrap wrong	index window header changes bootstrap properties statement character ssl operation promise certificate problem flask sequence language display dropdown callback behaviour
Topic 22	screen browser chrome page step site tab route backend laravel port url web sum home returns website load routes frontend	line functions default html navigation console keys attribute filter tag resource option title block tags series bug arrow buttons attributes
Topic 23	array page element html elements css js tag template arrays javascript pages website structure vue tree menu tags jquery div	view chrome package background framework color reactjs dependencies grid messages push forms xamarin detection notifications microsoft scripts reactnative cells pod
Topic 24	library documentation point format map javascript libraries csv pipeline language notebook examples jupyter activity distance facebook password stage book account	elements directory parent height jupyter notebook webpack width apache stack conda importerror proxy tools upgrade returns symfony permissions enum label

A partir de ce modèle que j'ai renommé *best\_lda\_model*, j'ai récupéré les probabilités de chaque Topic associées à chaque message et considéré le Topic dominant comme celui ayant le plus grand poids pour chaque message. J'ai joint ce DataFrame des poids des Topic *df\_document\_topic\_body* et *df\_document\_topic\_title* au DataFrame *data* :



J'ai créé la liste *liste\_globale* du Top 20 des mots des 25 Topics et créé les variables '*Liste\_Features\_Topic\_dominant\_Body*' et '*Liste\_Features\_Topic\_dominant\_Title*' correspondant à la liste du Top 20 des mots du Topic dominant pour chaque message. Avec ces listes, j'ai réalisé 2

matching entre les mots contenus dans les variables 'Body\_nettoye\_stopwords' et 'Title\_nettoye\_stopwords' avec :

- soit les mots contenus dans la liste des mots de tous les Topics *liste\_globale* : 'match\_Topic\_global'
- soit les mots contenus dans la liste des mots du Topic dominant : 'match\_Topic\_dominant\_Body' et 'match\_Topic\_dominant\_Body'

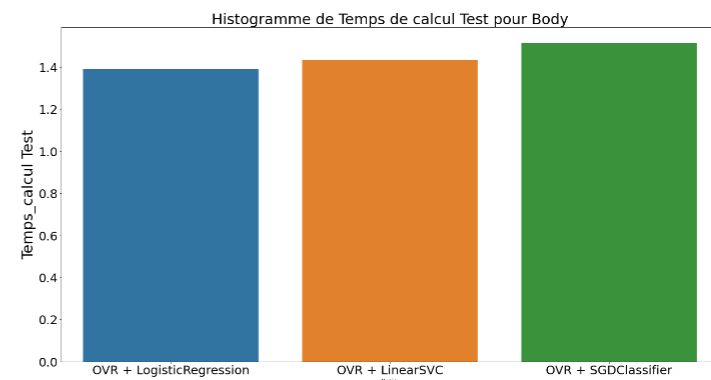
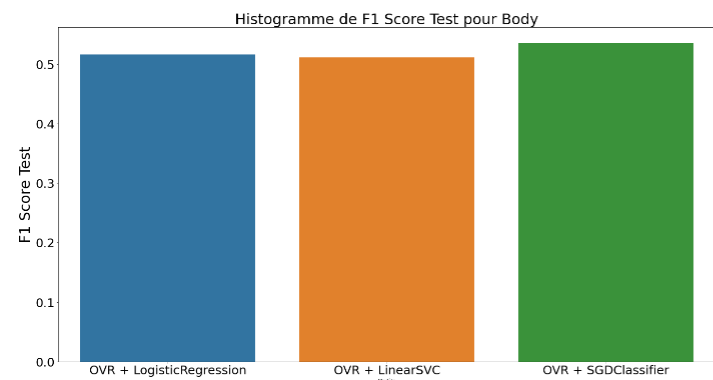
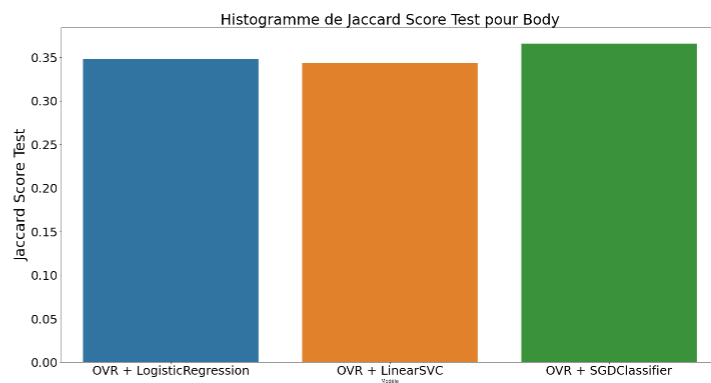
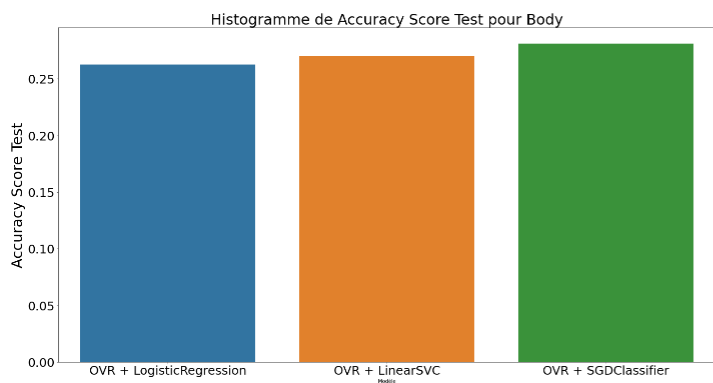
Ces 2 matchings de mots correspondent aux nouvelles features qu'on pourra transmettre à l'utilisateur avec l'API. Ainsi, le nombre de Tags suggérés par cette approche est compris entre 1 et 5, de la même manière que le nombre de Tags Stack Overflow donnés par les utilisateurs.

## V. Approche supervisée

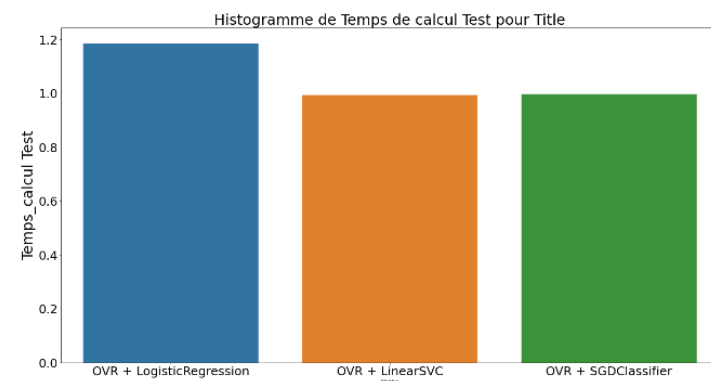
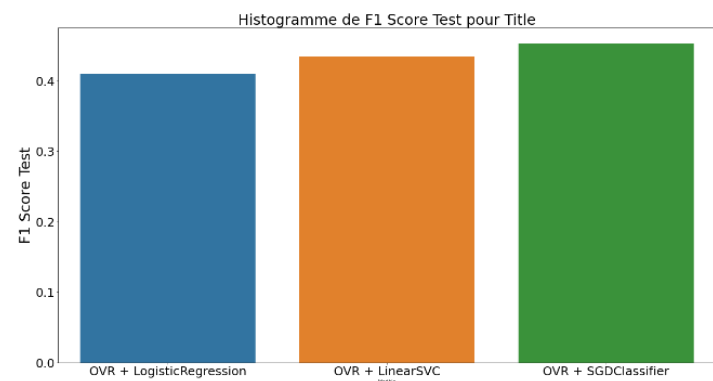
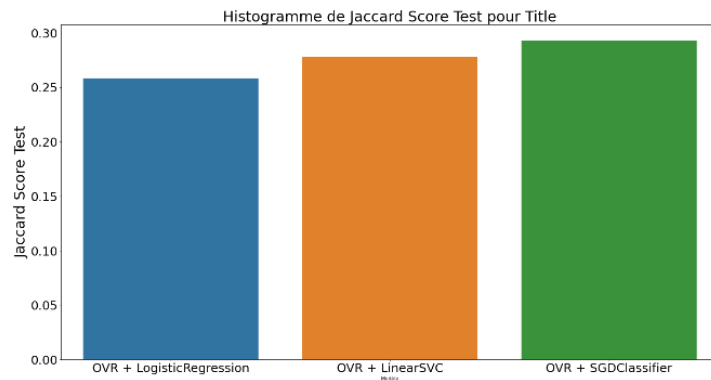
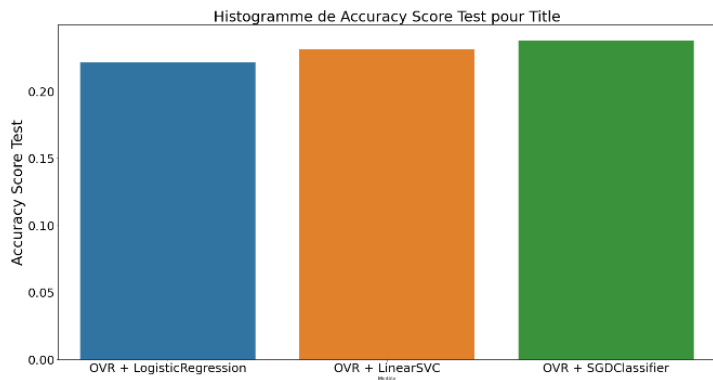
L'objectif de l'approche supervisée est de prédire les Tags Stack Overflow à partir des messages contenus dans les variables 'Body\_nettoye\_stopwords' et 'Title\_nettoye\_stopwords'. Il s'agit donc de variables qualitatives et il est possible de transformer ces variables pour traiter la prédiction des Tags à partir des messages comme un problème de classification multi-classe. Pour cela, j'ai extrait des features utilisables pour les modèles à partir des messages que j'ai vectorisés par un **tf-idf** avec le module **TfidfVectorizer** appliqué aux données par **tf\_idf\_vec.fit\_transform()**. Cette transformation permet de pondérer l'importance d'un mot dans chaque message par rapport à l'ensemble du corpus de messages. Les variables à prédire correspondent au **dummies** des 100 Tags Stack Overflow les plus fréquents. J'obtiens un DataFrame en entrée de 48 447 lignes et 75 779 colonnes et un DataFrame à prédire de 48 447 lignes et 100 colonnes. J'ai séparé les DataFrames en jeux de données d'entraînement **X\_train** et **y\_train** (70% du jeu de données) et en jeux de données test **X\_test** et **y\_test** (30% du jeu de données) avec la fonction **train\_test\_split** de **scikit-learn**. On se place dans un cas de classification multi-classes avec plus de 2 Tags différents. J'ai utilisé l'approche one-versus-rest **OneVsRestClassifier** qui consiste à créer K classifieurs binaires qui séparent chaque classe k de l'union des autres classes. J'ai appliqué cette méthode sur les modèles suivants avec une recherche des hyperparamètres optimaux pour chaque modèle via une validation croisée avec **GridSearchCV** :

- La régression logistique **LogisticRegression(n\_jobs=-1, solver='saga')** avec {'estimator\_\_C': [10, 1, 0.01, 0.001], 'estimator\_\_penalty': ['l1', 'l2']}
- La SCV linéaire **LinearSVC(dual= False)** avec {'estimator\_\_C': [100, 10, 1, 0.01, 0.001], 'estimator\_\_penalty': ['l1', 'l2']}
- Le classifieur de descente de gradient stochastique **SGDClassifier(n\_jobs=-1)** avec {'estimator\_\_alpha': [10, 1, 0.01, 0.001, 1e-04, 1e-05, 1e-06], 'estimator\_\_penalty': ['l1', 'l2']}

J'ai comparé les 3 modèles sur plusieurs metrics : l'**Accuracy Score** (**accuracy\_score**), le **Jaccard Score** (**jaccard\_score**), le **F1 Score** (**f1\_score**) et le temps d'exécution (**timeit.default\_timer()**) obtenus sur le jeu de données test :



Les résultats obtenus pour la variable '*Body\_nettoye\_stopwords*' montrent que le meilleur modèle est le **SGDClassifier** avec des scores d'accuracy (0.28), de Jaccard (0.37) et F1 (0.54) légèrement plus élevés par rapport aux autres modèles (avec un temps de calcul de 1.5 secondes).



On obtient les mêmes conclusions pour la variable '*Title\_nettoye\_stopwords*' avec des scores d'accuracy (0.24), de Jaccard (0.29) et F1 (0.44) plus élevés pour le modèle **SGDClassifier** (avec un temps de calcul de 0.99 seconde).

## Conclusion : Sélection du modèle final et mise en production

Après toutes les analyses réalisées pour trouver l'approche la plus optimale pour proposer les Tags d'un message, l'approche non supervisée sur le traitement **Stopwords** (avec le matching des mots de tous les Topics et du Topic dominant) et l'approche supervisée **SGDClassifier** ont été retenues. De cette manière, l'utilisateur aura le choix entre l'approche non supervisée lui présentant des Tags précis (*Tags dominants*) à partir du Topic dominant et des Tags plus larges (*Tags globaux*) à partir de mots de tous les Topics et l'approche supervisée qui prédit des Tags à partir des Tags Stack Overflow.

L'approche supervisée sélectionnée semble être plus performante en prédisant en moyenne 62.3% à partir de *Body* et 48.09% à partir de *Title* des Tags Stack Overflow malgré une très grande variance. Une autre approche possible aurait été d'utiliser un dictionnaire de Tags reliant certains mots présents dans les messages avec les Tags Stack Overflow sans utiliser de modèle de Machine Learning. Il est également possible de passer par du word embeddings et du Deep Learning pour traiter les données textuelles. Cela permettrait de mieux représenter le sens des mots dans un espace multidimensionnel en rapprochant les mots similaires ou qui font partie du même champ lexical.

Pour tester les modèles sélectionnés, j'ai développé une API déployée sur **Heroku** avec **Flask** et **Python** de suggestion de tag à partir d'un message rentré par l'utilisateur :

### Formulaire de Tags

The image shows a web form titled "Message". It contains two input fields: a text box for the title with the placeholder "Titre" and a larger text area for the message body with the placeholder "Message". Below these fields is an "Envoyer" button.

Ce formulaire demande à l'utilisateur de rentrer son message contenant un titre et le corps du texte et nettoie les textes selon la méthode *Stopwords*. Pour cela, les modèles des approches sélectionnées, les listes des stopwords et des mots des Topics et le DataFrame nettoyé ont été téléchargés et placés dans le dossier **static** de l'API. Le dossier **templates** contient les fichiers .html, le fichier **views.py** les chemins de l'API et le fichier **resultat.py** le code final utilisé pour proposer les Tags à l'utilisateur à partir du message (*Body*) et de son titre (*Title*).