

Openclassrooms

# Catégorisation automatique de questions sur Stack Overflow

Comparaison des approches supervisées et non supervisées

Cedric Soares  
30/08/2021

## Table des matières

Rappel du contexte .....	2
Besoin identifié.....	2
Solution préconisée.....	2
Livrables.....	3
Récupération des données.....	3
Analyse des données pour filtrage .....	3
Activité de la plateforme et nombre de vues des posts.....	3
Distribution des variables Score, AnswerCount, CommentCount et FavoriteCount .....	6
Filtrage des données .....	6
Pré-traitement des données .....	6
Nettoyage du texte.....	7
Tokenisation .....	7
Filtrage à l'aide d'un modèle de POS (Part of Speech tagging) .....	7
Racinisation des tokens .....	7
Vectorisation du corpus .....	8
Modélisation.....	8
Approche supervisée.....	8
Dédoublonnage des tags .....	9
Réduction des dimensions des prédicteurs.....	9
Partitionnement des données .....	9
Vectorisation des tags - labels.....	9
Entraînement des modèles .....	10
Approche non supervisée.....	11
Principes .....	11
Entraînement du modèle .....	11
Comparaison des approches .....	13
Développement de l'API et mise en production .....	13
Conclusions.....	14

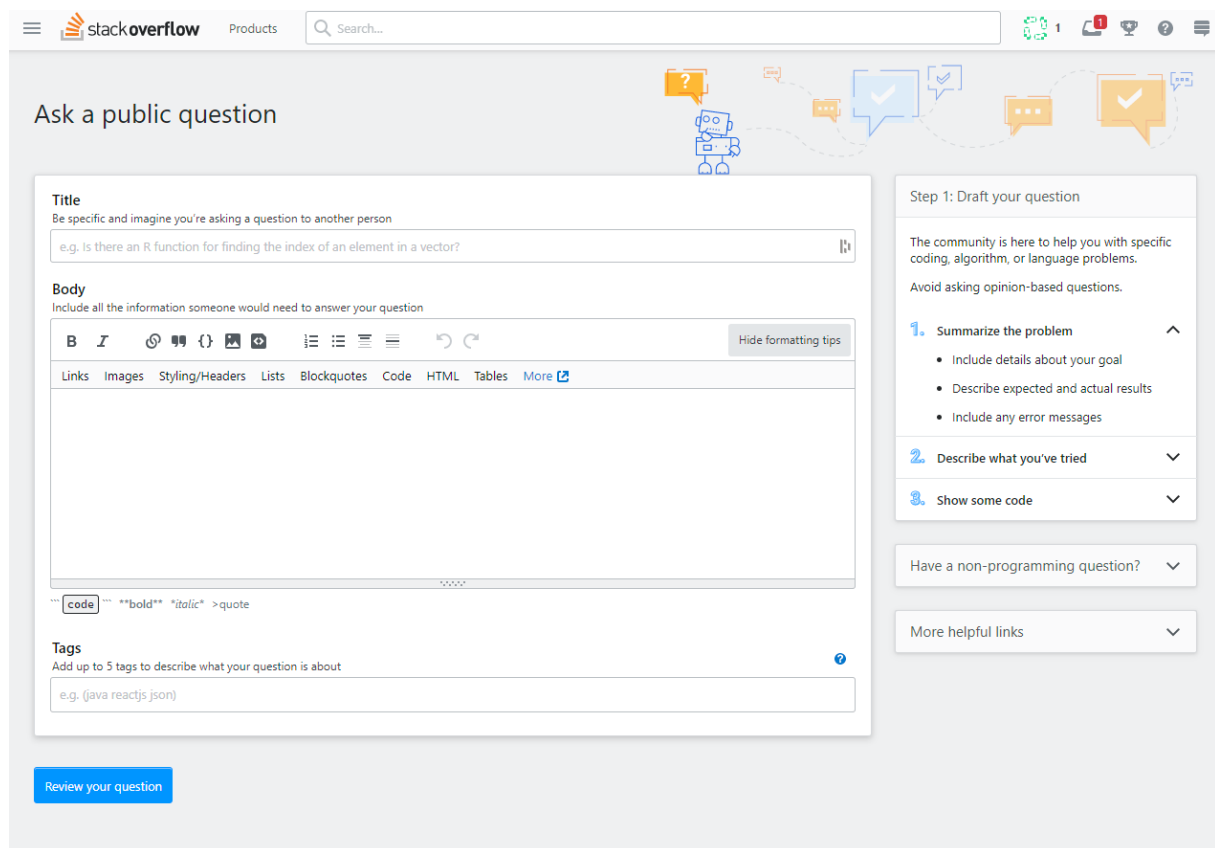
## Rappel du contexte

### Besoin identifié

[Stack Overflow](#) est un site communautaire de questions-réponses. Le support adresse une cible de profils techniques du secteur informatique (développeurs, devops, datascientists...). Le site, lancé en 2008, revendique une audience de 100 millions de visiteurs par mois et en moyenne 4 questions posées toutes les minutes (source : [Stack Overflow advertising](#)).

Un utilisateur souhaitant poser une question passe par un formulaire où il doit renseigner :

- Un titre
- Un corps de texte
- Entre un et cinq tags servant à catégoriser la question



Formulaire soumission de question – source : [Stack Overflow – Août 2021](#)

Le choix des tags peut s'avérer délicat pour un non initié à la plateforme. Afin d'y remédier, nous souhaitons proposer un outil de suggestion de tags à destination des utilisateurs.

### Solution préconisée

L'outil se basera sur le corps de texte fourni afin de suggérer un ensemble de tags. Il sera propulsé par d'un modèle de machine learning. Deux approches, supervisée et non supervisée, seront étudiées au cours des travaux. La solution sera mise à disposition sous forme d'API mise en production.

## Livrables

L'ensemble des livrables qui seront remis à la fin du projet comportent :

- Un notebook Jupyter d'exploration des données
- Un notebook Jupyter d'entraînement des modèles
- Le code de l'API historisé à l'aide de l'outil de versioning Git.
- Un support de présentation
- Un rapport de synthèse (présent document)

## Récupération des données

L'outil de [stackexchange explorer](#), mis à disposition par Stack Overflow, a été utilisé afin de récupérer les données nécessaires à l'entraînement des modèles. L'interface permet d'utiliser des requêtes SQL afin de cibler des publications à exporter. La table des « posts » représente à elle seule 16,6 Go (source : [archive.org](#) – juin 2021). Travailler sur l'ensemble de la base nécessiterait un ensemble de ressources trop important. Par conséquent, pour nos travaux, nous souhaitons limiter les données aux posts les plus récents et ayant suscité des réactions de la part de la communauté. Voici les critères que nous avons retenus :

- Des posts publiés entre le 1<sup>er</sup> janvier 2018 et le 20 juillet 2021 (date de début des travaux)
- Ayant un Score non nul
- Ayant un nombre de vues non nul
- Ayant un nombre de réponses non nul
- Ayant un nombre de commentaires non nul
- Ayant un nombre de mise en favoris non nul

L'outil limitant le nombre de posts retournés par requête à 50 000, nous avons réalisé une requête par mois sur la période. Voici un exemple de requête mensuelle réalisée :

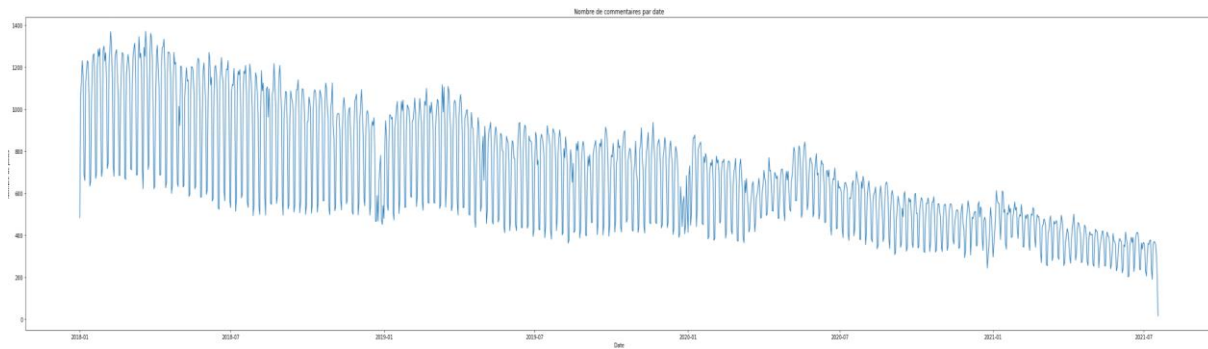
```
SELECT TOP(50000) Id, CreationDate, Score, ViewCount, AnswerCount, CommentCount,
FavoriteCount, Title, Body, Tags
FROM Posts
WHERE CreationDate BETWEEN CONVERT(datetime, '2020-01-01') AND CONVERT(datetime, '2020-
02-01')
AND Score IS NOT NULL
AND ViewCount IS NOT NULL
AND AnswerCount IS NOT NULL
AND CommentCount IS NOT NULL
AND FavoriteCount IS NOT NULL ORDER BY CreationDate
```

L'ensemble des requêtes SQL a permis de récupérer 902 285 posts sur la période du 1<sup>er</sup> janvier 2018 au 20 juillet 2021. Celle-ci n'a pas vocation à filtrer les données, mais à s'assurer que toutes les variables qui vont permettre de le faire sont présentes.

## Analyse des données pour filtrage

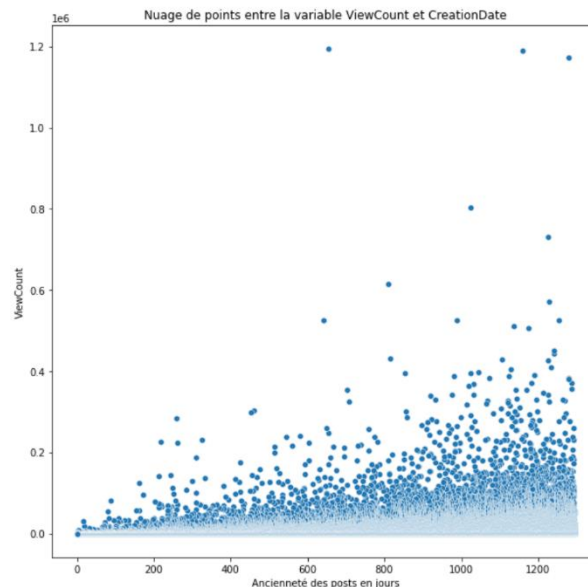
### Activité de la plateforme et nombre de vues des posts

En réalisant un diagramme en ligne du nombre de posts dans le temps, nous constatons une baisse d'activité sur la période.



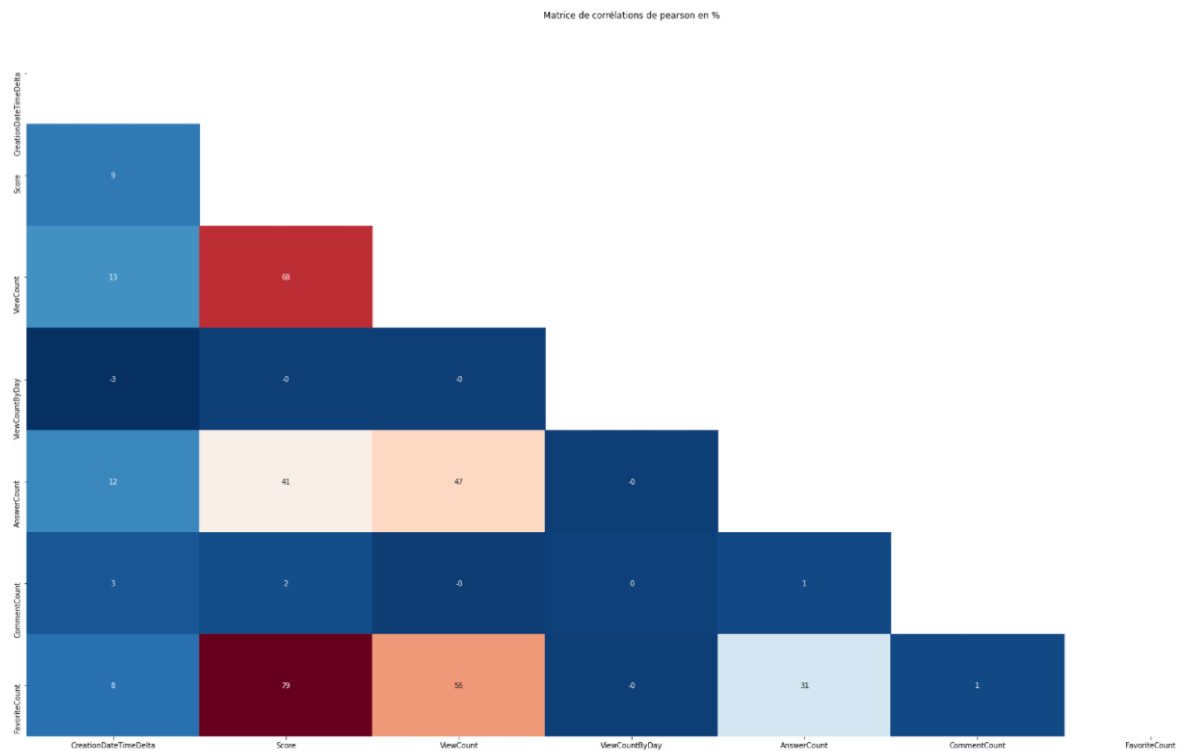
*Diagramme en ligne de la variable `CreationDateByDay` sur la période – Source notebook Jupyter juillet 2021*

Nous pouvons en déduire une proportion de posts décroissante dans le temps. Donc, filtrer les posts par date de création pénaliserait les publications les plus récentes et les thématiques qui y sont associées. En plus d'une baisse d'activité, nous souhaitons savoir si le nombre de vues est corrélée à l'ancienneté des posts. Pour se faire, nous réalisons un nuage de point avec les variables `ViewCount` (initialement récupérée par les requêtes SQL) et `CreationDateTimeDelta` (construite à partir de la variable `CreationDate`).

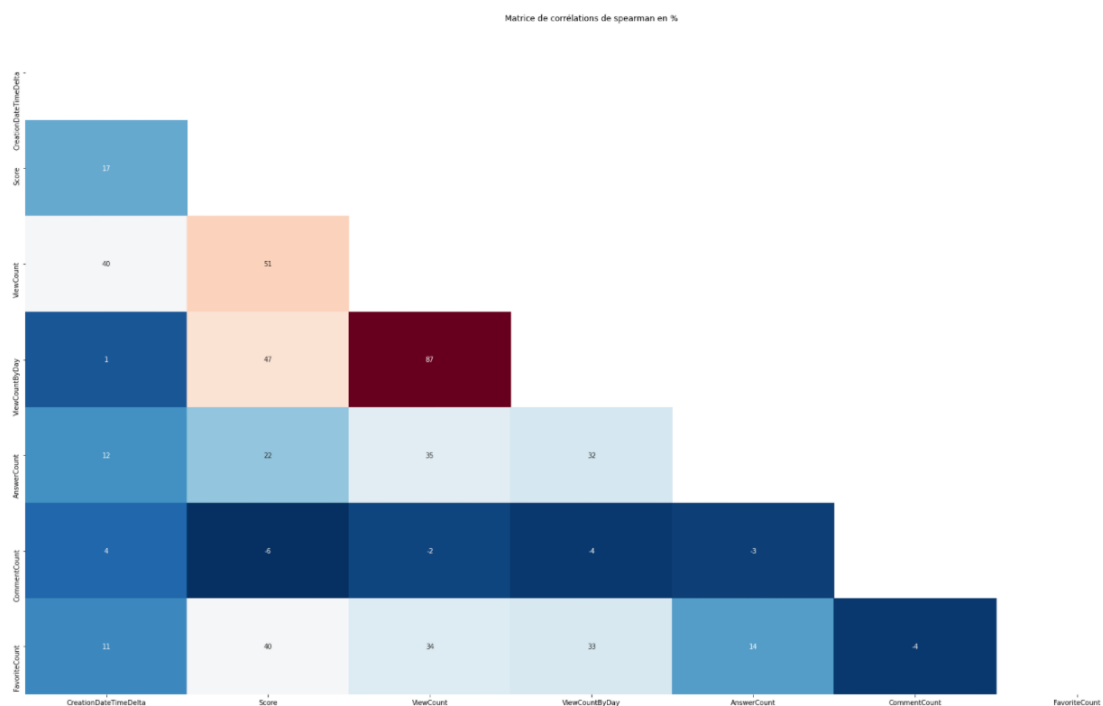


*Nuage de points entre les variables `ViewCount` et `CreationDateTimeDelta` – Source notebook Jupyter juillet 2021*

Le diagramme permet de constater une corrélation entre ces deux variables. Afin d'atténuer ces deux phénomènes nous remplaçons le nombre de vues absolu (`ViewCount`) par le nombre de vues par jour (`ViewCountByDay`). Nous réalisons les matrices de corrélation de Pearson et Spearman afin de vérifier les corrélations de ces deux variables avec les autres variables d'activité.



Matrice de corrélation de Pearson – Source notebook Jupyter juillet 2021



Matrice de corrélation de Spearman – Source notebook Jupyter juillet 2021

Les visualisations permettent effectivement de constater des corrélations plus faibles avec la variable ViewCountByDay. Nous décidons donc d'utiliser cette dernière.

## Distribution des variables Score, AnswerCount, CommentCount et FavoriteCount

Afin de fixer un seuil de filtrage pour chacune de ces variables nous observons leur distribution. Ci-dessous le tableau récapitulatif .

	Moyenne	Ecart type	Minimum	25% (premier quartile)	50% (médiane)	75% (troisième quartile)	Maximum
Score	2,471	8,917	-24,000	0,000	1,000	3,000	2567,000
AnswerCount	1,414	1,330	0,000	1,000	1,000	2,000	91,000
CommentCount	2,322	3,039	0,000	0,000	1,000	3,000	67,000
FavoriteCount	1,161	2,276	0,000	1,000	1,000	1,000	910,000

Globalement nous constatons des distributions centrées autour de 1, concentrées sur des valeurs basses, avec un écart type faible.

## Filtrage des données

In fine nous arrêtons les paramètres de filtrage suivant :

- Un score supérieur à 0
- Un nombre de réponses supérieur à 0
- Un nombre de commentaires supérieur à 0
- Un nombre de mise en favoris supérieur à 0
- Un nombre de vues par jours supérieur à 5

Voici le code Python correspondant à ces critères :

```
filtered_data = data[(data['Score'] > 0) &
                    (data['AnswerCount'] > 0) &
                    (data['CommentCount'] > 0) &
                    (data['FavoriteCount'] > 0) &
                    (data['ViewCountByDay'] > 5)]
filtered_data.sort_values(['Score', 'ViewCount'], ascending=[False, False], inplace=True)
```

Ce filtrage réduit drastiquement le nombre de posts. Le nombre passe de 902 285 posts à 46 504 posts. Soit 5.15% de posts restants après filtrage. Pour la suite des travaux nous n'utiliserons plus les variables qui ont servi à filtrer les données et ne travaillerons uniquement sur les titres, corps de texte et tags des posts.

## Pré-traitement des données

Les entraînements de modèles de machine learning ne peuvent être réalisés sur les textes brutes. D'une part ils contiennent des éléments qui n'ont aucune valeur sémantique (balises html, caractères non alphabétiques, mots génériques ...) et un nombre potentiel de déclinaisons élevé de certains termes. D'autre part, les modèles de machines learning nécessitent des prédicteurs et des cibles de nature numérique. Voici la liste des étapes de traitement que nous appliquons pour y remédier ainsi que les librairies/packages associées :

Traitement	Librairies associées
Suppression des balise HTML	BeautifulSoup, html5lib
Nettoyage du texte	re
Tokenisation	NLTK
Filtrage à l'aide d'un modèle de POS (Part Of Speech tagging)	NLTK
Racinisation des tokens	NLTK
Filtrage des valeurs vides	Pandas
Vectorisation du corpus	Scikit-learn (approche supervisée), Gensim (approche non supervisée)

Hormis la suppression des HMTL et le filtrage des valeurs vides, assez explicites, nous détaillons les autres traitements.

### Nettoyage du texte

Ce traitement passe les textes en minuscule, ne conserve que les caractères alphabétiques et ne garde que les termes de plus de trois lettres. Le filtrage sur la taille des termes permet de retirer ceux qui sont générique liés au code (if, for ...). Il reste toutefois perfectible puisqu'il élimine les occurrences de certains langages (C, C++, C#, R... ).

### Tokenisation

La tokenisation permet de transformer les textes passés en entrée en liste de termes distincts (token). Pendant le traitement les termes génériques (stop words) ne sont pas conservés dans la liste des tokens. NLTK propose des listes de stop words génériques. Nous attirons l'attention sur le fait que le corpus utilisé pour les travaux est de nature spécifique. En effet les posts contiennent souvent du code, des messages d'erreur de compilateurs / interpréteurs ou des logs alors qu'aucune liste de stop words mis à disposition ne porte sur le thématique. Le risque est de laisser passer des termes génériques vis-à-vis du contexte étudié. Sans allouer un important temps de travail il s'avère difficile de construire une liste de stop words spécifique exhaustive.

### Filtrage à l'aide d'un modèle de POS (Part of Speech tagging)

Lors de pré-traitement des posts nous cherchons avant tout à identifier à termes liés à des technologies utilisées. Ces dernières sont généralement des noms. Afin de mettre en œuvre le filtrage nous utilisons un modèle de POS. Ce dernier se base sur des chaînes de Markov . Leur principe est d'identifier la probabilité la plus forte de la fonction grammaticale d'un terme par rapport :

- A la fonction grammaticale du terme précédent
- A la probabilité la plus forte de l'association entre le terme et une fonction grammaticale particulière

Ce type de modèle est entraîné de manière supervisée. NLTK propose un modèle pré-entraîné. Nous attirons une nouvelle fois l'attention sur le fait que le corpus utilisé pour les travaux est de nature spécifique. Aucun corpus mis à disposition par NLTK ne couvre le domaine étudié. Le risque est que certaines technologies, dont la dénomination ne découle pas d'un nom commun, ne soient pas retenues par le filtre.

### Racinisation des tokens

Afin d'éviter d'utiliser de multiples déclinaisons d'un même terme (conjugaisons, féminin, pluriel ...). Les deux méthodes les couramment utilisées sont le stemming et la lemmatisation. Elles consistent toutes deux à dédupliquer les déclinaisons de termes en ne conservant que leur racine. Le stemming procède par troncature simple. De manière générale cette méthode peut poser un problème car



plusieurs mots termes sémantiquement différents peuvent avoir la même racine (stem). La lemmatisation consiste à identifier la racine sémantique (lemme) d'un terme référencé dans un corpus ou un dictionnaire d'une langue. Cette méthode évite, a priori, l'écueil de diminuer le nombre de termes ayant un stem identique mais des lemmes différents. Nous retenons donc la lemmatisation. Par contre une fois encore nous sommes confrontés à la généralité des outils mis à disposition par NLTK. A titre d'exemple, "keras" (Framework de deep learning) devient "kera" une fois lemmatisé.

## Vectorisation du corpus

Afin de pouvoir entraîner les modèles nous avons besoin de transformer les listes de tokens lemmatisés en vecteurs. Deux méthodes sont couramment utilisées :

- Bag of words : Chaque liste de token (appelée document) est transformée en un vecteur indiquant la fréquence brute de chaque terme du corpus dans la liste.
- TF-IDF : Dans cette méthode, la fréquence brute d'un token est remplacée par un indicateur composé de sa fréquence d'apparition du token dans le document et la fréquence inverse du nombre de documents où le token apparaît. Tel que  $TF - IDF = TF \times \log \frac{|D|}{|\{d_j : t_i \in d_j\}|}$  avec  $|D|$  : le nombre total de documents dans le corpus et  $|\{d_j : t_i \in d_j\}|$  : le nombre de documents où le token apparaît. Cette méthode permet de minorer les tokens présents dans un nombre élevé de documents et de normaliser la taille des documents.

Les documents contenant des termes très génériques et de longueurs très variables, nous avons décidé d'utiliser une vectorisation par TF-IDF pour la suite des travaux. Afin de ne garder que les tokens les plus représentatifs, nous avons entraîné les modèles de vectorisation sur des dictionnaire de correspondance (vocabulary) ne comportant que des termes dont la fréquence est supérieure à 1000 occurrences dans le corpus.

## Modélisation

Nous avons testé deux approches dans le cadre de l'étude. La première supervisée, basée sur des tags associés au corpus. La seconde, non supervisée, basée uniquement sur les éléments des posts à savoir le corps du texte et ou le titre. Le but est de définir la meilleure approche et le meilleur modèle basés sur différents critères :

- Performance des modèles
- Temps de calcul
- Pré-traitements supplémentaires avant entraînement du modèle
- Maintenabilité de la solution

Concernant les performances nous avons étudié celles liées à des indicateurs ainsi que des tests empiriques sur des différents documents. Les indicateurs étant différents selon le type d'approche, nous ne pouvons pas comparer des modèles de différentes approches entre eux de ce point de vue.

## Approche supervisée

Cette approche a consisté à entraîner différents modèles supervisés avec des couples de textes vectorisés / tags associés. Chaque document pouvant être associé à un à plusieurs tags, nous avons donc à faire à un problème de classification multi-classes et multi-labels.

Plusieurs étapes de pré-traitement supplémentaires ont été nécessaires avant de pouvoir entraîner les modèles. En voici le récapitulatif :

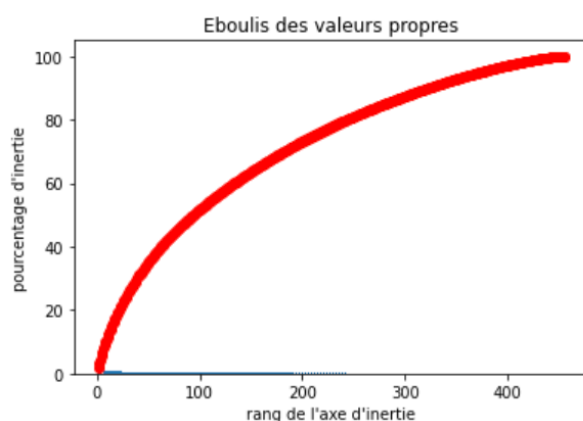
Traitement	Librairies associées
Dédoublonnage des tags - labels	aucune
Réduction des dimensions des prédictors	Scikit-learn
Partitionnement de données	Scikit-learn
Vectorisation des tags - labels	Scikit-learn

### Dédoublonnage des tags

Le dédoublonnage des tags – label s’est avéré nécessaire après le pré-traitement des documents. En effet, il n’est pas rare que les utilisateurs spécifient plusieurs versions d’une même technologie dans leurs tags. Après traitement préalable, notamment avec la suppression des caractères numérique, les tags-label peuvent contenir des doublons pour un même document.

### Réduction des dimensions des prédictors

La vectorisation a transformé les documents en un vecteur de 456 composantes. Afin de casser les corrélations entre celles-ci et optimiser le temps d’entraînement des modèles en réduisant les dimensions nous avons réalisé une ACP (Analyse en Composantes Principales). A défaut d’un coude significatif dans la visualisation l’éboulis des composantes principales, nous avons sélectionné un hyper paramètre consistant à retenir un nombre de vecteurs permettant de conserver 85% d’inertie. L’ACP a conservé 281 composantes principales.



*Eboulis des valeurs propres de l'ACP – Source notebook jupyter août 2021*

### Partitionnement des données

Lors du partitionnement des données nous avons conservé 80% des données pour le jeu d’entraînement et 20% pour le jeu de test.

### Vectorisation des tags - labels

A l’image des documents nous avons besoin de vectoriser les tags – labels afin d’entraîner les modèles. Scikit-learn fournit un module permettant de prendre en charge l’opération. Afin de restreindre la taille du vecteur en sortie du modèle et minimiser les combinaisons de labels possibles, nous n’avons conservé que les 200 tags les plus représentés. Nous avons filtré en conséquence les documents non liés à ces tags.

## Entrainement des modèles

Nous avons testé quatre modèles supervisés :

- KNN
- SVM linéaire
- Random forest
- Gradient boosting

Le design du KNN et de la Random Forest supportent nativement les problèmes de classification multi-classes. Pour la SVM et le Gradient Boosting nous avons eu recours à l'algorithme One vs Rest. Son principe est d'entraîner un modèle par classe afin de déterminer si l'observation / document appartient à la classe sur laquelle il a été entraîné ou non.

Afin de comparer les modèles nous avons évalué quatre indicateurs :

- Précision : capacité à prédire correctement les tags associés à chaque document
- Recall : capacité à retourner pour chaque tags tous les documents qui y sont associés
- F1 Score : moyenne harmonique de ces deux indicateurs
- Temps d'entraînement des modèles

Etant face à une classification multi-classes nous avons utilisées les version micro de ces indicateurs. Concrètement, cette version des indicateurs est calculée par classe puis agrégés là où les indicateurs macro sont calculés globalement. Lors de l'évaluation nous nous sommes principalement intéressés à la précision et au temps d'entraînement. Dans le cadre d'une API de suggestion de tags, la capacité d'un modèle à retourner tous les documents associés à chaque terme n'est pas essentielle. Le recall et le F1 score ont été toutefois été observés pour affiner l'analyse. Voici les résultats des différents modèles.

	<b>Précision</b>	<b>Recall</b>	<b>F1 score</b>	<b>Temps d'entraînement</b>
KNN	0.687998	0.281438	0.399467	38.1 secondes
SVM Linéaire	0.798641	0.342438	0.479345	22.5 secondes
Random Forest	0.868575	0.146893	0.251289	2 minutes 59 secondes
Gradient Boosting	0.505483	0.307369	0.382283	14 heures 49 minutes et 39 secondes

Si la Random Forest offre la meilleur précision (7 points supérieure à la SVM), le modèle a un ordre de grandeur de temps d'entraînement significativement plus grand (8 fois plus long que la SVM). Si ce critère n'est pas pénalisant à l'échelle des données utilisée pour les travaux, il pourrait le devenir à une échelle plus importante. Pour cette raison nous avons sélectionné la SVM pour cette approche.

Afin de filtrer les tags suggérés nous avons également ajouté un filtre à la fonction de prédiction. Celui-ci consiste à parcourir le document ayant servi à la prédiction et de ne garder que les tags prédits y figurant.

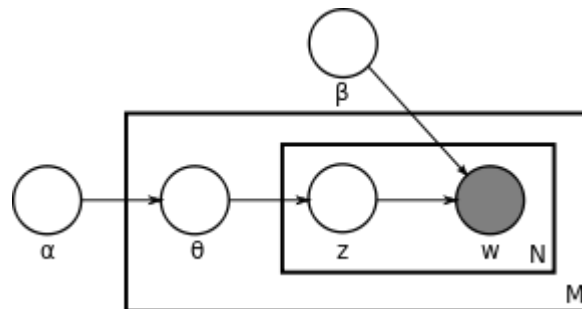
A noter que pour les besoins de l'approche supervisée, nous devons d'entraîner et de maintenir quatre modèles : un vectoriseur de documents, une ACP, un vectoriseur de tags et le modèle supervisé. Il a également été nécessaire de nettoyer les tags et supprimer les documents ne correspondant pas au top 200 tags.

## Approche non supervisée

### Principes

Contrairement à l'approche supervisée, les tags renseignés par les utilisateurs n'ont pas été utilisés. Nous avons utilisé un modèle LDA (Latent Dirichlet Allocation). Il s'agit d'un modèle génératif probabiliste. L'algorithme modélise une approche proche d'un clustering. Il permet de regrouper les documents d'un ensemble par  $k$  topics (thèmes) et d'associer chaque mot  $w$  de chaque document à un des topics. Chaque document est alors composé d'un mélange  $\theta$  d'un petit nombre de topics.

Lors de son initialisation, chaque mot de chaque document est aléatoirement associé à un topic. L'apprentissage consiste à optimiser la probabilité qu'un topic  $t$  génère le mot  $w$  dans un document  $d$ . Voici la représentation graphique du modèle.



Représentation graphique du modèle LDA – source [Wikipédia](#) août 2021

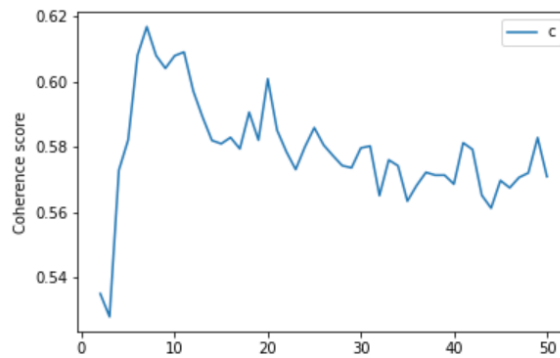
Avec :

- $\alpha$  : L'ensemble de tous les topics
- $\beta$  : L'ensemble des mots de tous les documents
- $M$  : L'ensemble des variables liée à un document
- $\theta$  : La distribution d'un topic pour un document
- $N$  : L'ensemble des variables liées à un mot
- $z$  : La distribution d'un topic pour un mot
- $w$  : Un mot

L'utilisation de la LDA n'a pas nécessité de traitement additionnel en amont. Pour les travaux nous avons utilisé la version multicores proposée par la librairie Gensim. Le module permet de paralléliser les calculs et ainsi d'accélérer l'entraînement.

### Entraînement du modèle

Le modèle a nécessité de définir le nombre de topics que nous avons souhaité générer. Pour se faire nous avons observé le score de cohérence pour un ensemble de modèles LDA entraînés avec de 2 à 50 topics en hyperparamètre. L'indicateur mesure le degré de similitude sémantique des mots les plus représentés pour chaque topic. Il détermine ainsi si chaque topic est cohérent d'un point de vue sémantique ou s'il est un amalgame aléatoire de mots. Plus le score est élevé et plus les topics sont cohérents. Voici diagramme d'évolution du score de cohérence par nombre de topics.



Evolution du score de cohérence par nombre de topics – Source notebook jupyter août 2021

L'entraînement de l'ensemble des modèles a pris 24 minutes et 45 seconds soit environs 30 secondes par modèle. Nous constatons que la cohérence maximale (0.6168) est atteinte avec 7 topics. Voici la répartition de ces topics à l'issu de l'entraînement.

Numéro du topic	Top 10 mots associés	Nombre de document associés	Pourcentage de document associés
1	docker, client, http, access, server, password, service, request, connection, error	7186	15,66%
2	component, button, react, class, import, state, item, const, style, page	6361	13,86%
3	java, android, spring, version, class, boot, system, google, support, device	5260	11,46%
4	project, node, module, package, studio, version, error, json, file, core	6526	14,22%
5	python, file, line, import, model, command, site, install, error, print	6649	14,49%
6	Image, flutter, view, list, color, widget, text, child, context, screen	4462	9,72%
7	value, function, column, array, type, name, number, form, return, date	9458	20,60%

Et voici le diagramme de la distribution spatiale des topics.

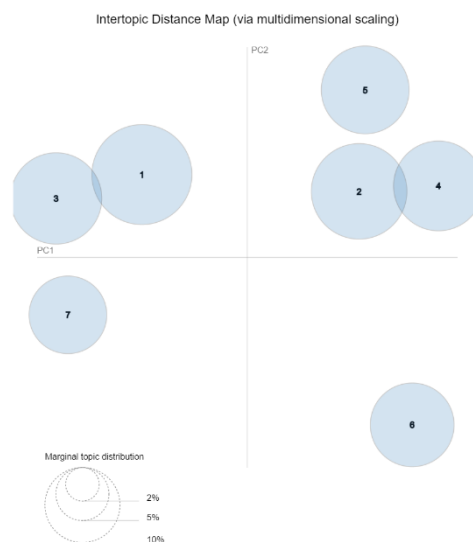


Diagramme de distribution des topics – Source notebook Jupyter août 2021

En observant le tableau et le diagramme nous voyons que les documents sont répartis de manière assez égale hormis le topic 6 (sous-représenté à 9,72%) et le topic 7 (sur-représenté à 20,60%). Hormis le topic 7, des noms de technologies font partie des dix mots clés les plus représentés. Nous constatons cependant un nombre important de mots clés génériques associés à chaque topic comme par exemple *error*, *number* ou *class*. Enfin, les topics 1 et 3 ainsi que 2 et 4 ont de petites zones de recoupement.

La fonction de prédiction des tags que nous avons construite retourne les 20 mots les plus représentés du topic le plus représenté du texte soumis au modèle. A l'image de l'approche supervisée nous avons filtré les résultats en ne gardant que les mots clés présents dans le texte soumis.

A noter que l'approche non supervisée ne nécessite que de maintenir, en plus du TF-IDF le modèle LDA ainsi qu'un vocabulary (dictionnaire de correspondance) lié au corpus.

### Comparaison des approches

Si nous n'avons pas pu pas comparer les performances des approches supervisées et non supervisées sur des indicateurs communs, nous avons comparé les prédictions des deux modèles retenus aux tags initiaux renseignés par des utilisateurs sur dix documents. Voici les résultats.

Tags originaux	Prédiction supervisée (SVM)	Prédiction non supervisée (LDA)
Javascript, ecmaScript	javascript, code	code
xcode, macos, command, line, terminal	- (aucun tag retourné)	line, command, error, path
java, performance, benchmarking, bytecode	java	java, version, system, void, string
Info, plist, xcode	code	error, build, code
java (4 déclinaisons), javac	array, code, java	java, version, class, system, void, string
javascript	react, github	project, react
android, intellij, idea, kotlin, corda	github	error
angular (4 déclinaisons)	Juery, core, chrome, html, router, bootstrap, node, typescript, angular, google, eslint, github, json	project, node, package, version, error, json, core, config, build, index
javascript, type, conversion	javascript, code	Result, code

Sur un nombre aussi restreint de tests empiriques il serait hasardeux de tirer des conclusions quant au choix d'un modèle. Nous notons néanmoins, qu'à une exception près pour chacun d'eux, que les deux modèles ont retourné des tags cohérents avec ceux initialement renseignés par les utilisateurs.

Au-delà des prédictions, il s'avère que l'approche non supervisée est soumise à moins de contraintes que l'approche supervisée. En effet l'approche supervisée nécessite un travail plus important sur le dimensionnement des données et leur pré-traitement avant entraînement. De plus cette approche nécessite de maintenir plus de ressources (modèle de PCA, modèle de vectorisation des tags, SVM) que l'approche non supervisée (modèle LDA et vocabulary). Pour ces raisons nous privilégions l'approche non supervisée. Nous intégrons toutefois les deux approches dans l'API à titre d'illustration.

### Développement de l'API et mise en production

L'API a été développée à l'aide du framework FastAPI et mise en production sur Heroku. FastAPI simplifie le traitement des requêtes et intègre nativement les spécifications OpenAPI. Voici l'arbre des fichiers de l'API.

```
projet 5 - api
├─ Procfile
├─ app
│   ├── __init__.py
│   ├── main.py
│   └── utils.py
├─ models
│   ├── dictionary.pkl
│   ├── lda_model.pkl
│   ├── mlb_model.pkl
│   ├── pca_model.pkl
│   ├── svm_model.pkl
│   ├── tfidf_model.pkl
│   └── vocabulary.pkl
├─ nltk.txt
├─ requirements.txt
└─ runtime.txt
```

Certains fichiers sont spécifiques à la plateforme Heroku utilisée pour la mise en production :

- Procfile : commande de lancement du serveur http de l'API
- nltk.txt : liste les corpus utilisés par NLTK. Heroku utilise la liste pour télécharger ces derniers
- requirements.txt : liste des librairies python utilisées par l'API. Le fichier n'est pas spécifique à Heroku mais est utilisé par la plateforme pour télécharger les librairies.
- runtime.txt : indique la version de python utilisée par l'API

Par ailleurs le dossier app contient les fichiers python permettant de faire fonctionner l'API :

- main.py : Définit les routes et la fonction utilisée pour le traitement des requêtes http
- utils.py : Classes et fonctions auxiliaires permettant de réaliser la prédiction des tags à l'aide des modèles .

Enfin le dossier models contient les modèles et éléments nécessaires à les entraîner. En production l'API utilise un serveur http uvicorn.

## Conclusions

Si les deux approches semblent donner des résultats satisfaisants nous souhaitons, une nouvelle fois, attirer l'attention sur plusieurs faits.

Afin de tenir le chiffrage de 70 heures, le filtrage des documents à retenir a été réalisé selon une analyse très rapide. Le pré-traitement a été réalisé à partir de modules de librairies qui ont été entraînés sur des corpus qui n'ont pas la spécificité de celui étudié. De plus les modèles utilisés ne permettent pas de tenir compte du contexte des documents. Ceux-ci ne différencient pas les textes

écrits par les utilisateurs de copiés-collés de messages d'erreur d'interpréteurs/compilateurs et logs d'erreurs d'API. Enfin très peu de tests empiriques ont été réalisés sur les prédictions.

Pour y remédier nous proposons deux approches. Soit accorder un temps beaucoup plus important au pré-traitement des documents. Soit utiliser une approche basée sur du transfert learning avec des transformers type Bert ou GPT-3.

Les modèles étudiés s'avèrent toutefois être une base de travail intéressante et l'API peut servir de POC pour un MVP.

Les livrables sont disponibles en ligne :

- [Repository](#) de travaux de pré-traitement des documents et entraînement des modèles
- [Repository](#) de l'API
- [Documentation](#) de l'API
- [Endpoint](#) de l'API