# Introduction

Stack Overflow est un site de question-réponses liées au développement informatique. Pour poser une question sur ce site, il faut entrer plusieurs tags de manière à retrouver facilement la question par la suite. Pour les utilisateurs expérimentés cela ne pose pas de problème, mais pour les nouveaux utilisateurs c’est souvent plus difficile.

Membre de Stack Overflow aujourd’hui, l’objectif de ce projet est de développer un système de suggestion de tag pour le site. Celui-ci doit prendre la forme d’un algorithme qui assigne automatiquement plusieurs tags à une question.

Pour mener à bien ce projet, et cela servira de plan à ce rapport, les étapes suivantes ont été réalisées :

* Explication de la problématique, de son interprétation et des déductions effectuées quant aux pistes de recherche possibles
* Cleaning effectué, feature engineering et exploration des données
* Exploitation des données
* Les différentes pistes de modélisation effectuées
* Présentation du modèle final sélectionné ainsi que des performances et améliorations effectuées.

# Explication de la problématique, de son interprétation et des déductions effectuées quant aux pistes de recherche possibles

## La problématique

La problématique de ce projet est de créer des recommandations de tags pour toute question nouvelle qui sera posée sur le site Stack Overflow

## Pistes de recherches possibles

Pour répondre à la problématique, il va donc falloir faire les étapes suivantes :

* Récupérer les questions sur le site
* Les nettoyer
* Essayer différentes sortes d’algorithme
  + Non-supervisé
  + Non-supervisé, mais un peu aidé
  + Supervisé

# Cleaning effectué, feature engineering et exploration des données

## Nettoyage 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. Il a donc été décidé d’aller chercher un certain nombre de questions (de manière aléatoire) afin de pouvoir travailler dessus. Le nombre retenu en premier lieu a été de 50000, mais ce nombre a ensuite été réduit à 20000 pour des problèmes d’efficacité de machine.

L’avantage majeur de ces données est qu’elles sont réelles.

Cinq catégories ont été retenues :

* **Title :** C’est le titre de la question
* **Body :** C’est le corps de la question
* **Tags :** Ce sont les tags d’origine, donc ceux postés par l’auteur de la question
* **CreationDate :** Il s’agit de la date de création de la question
* **Score :** Nombre de « up » qu’a reçu la question. Plus ce chiffre est élevé, plus on peut se permettre de penser que la question est de qualité.

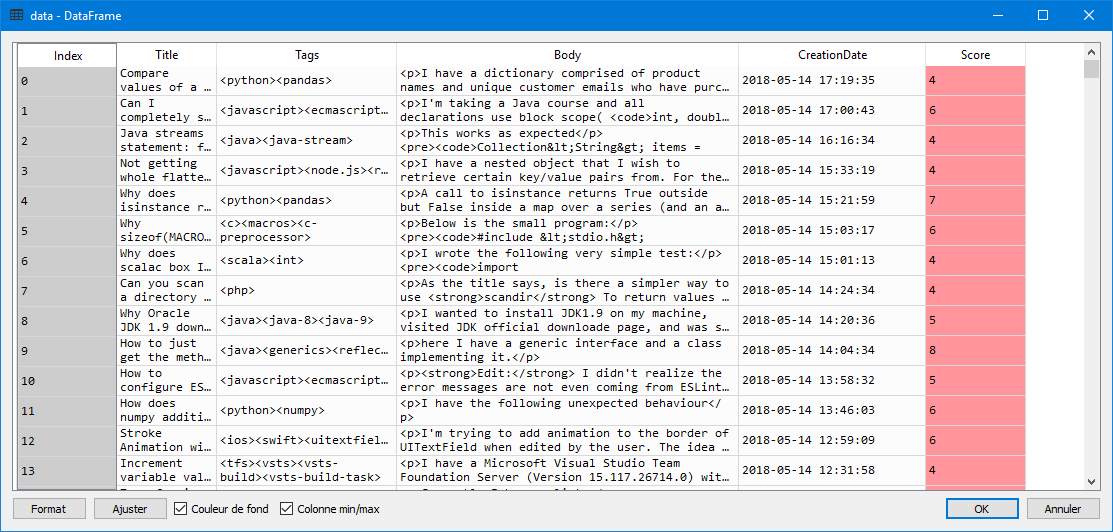


Figure 1 - Extrait de la database qui servira dans ce projet

## Feature engineering et exploration des données

### Données à exploiter

Des 5 données qu’on a récupérées et qui forment notre database, il y en a trois qui sont particulièrement importantes : « Title », « Tags » et « Body ». Laissons « Tags » de coté pour le moment, et intéresserons-nous aux deux autres.

« Title » et « Body » contiennent l’essence même de la question et ce qui va devoir être exploité dans notre projet afin de prédire des Tags qui doivent être cohérent avec ces deux données. Plutôt que de traiter les deux catégories séparément, il a été décidé de les joindre et de créer un bloc unique de texte.

# Fusion du body et du title

data['Body'] = data['Title'] + data['Body']

data['Tags'] = data['Tags'].str.replace("<", "")

data['Tags'] = data['Tags'].str.replace(">", " ")

Figure 2 – Code python : fusion des données « Title » et « Tag »

« Tag » subit en même temps un premier lifting, avec la suppression des caractères « < » et « > ». Cela servira un peu plus tard.

### Exploration des données

Les deux graphiques ci-dessous représentent :

* La distribution des scores (le premier graphique s’arrête pour un score de 10, le deuxième pour 25).
* Les 10 tags les plus fréquent rencontrés dans notre échantillon

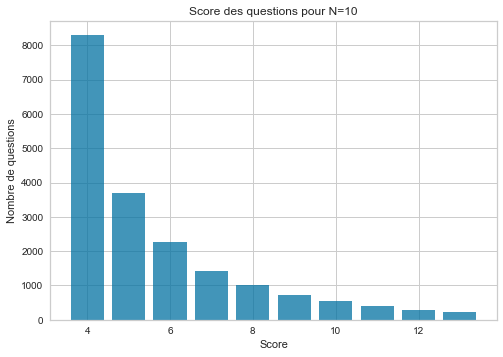


Figure 3 – Graphique : distribution des scores, max = 10

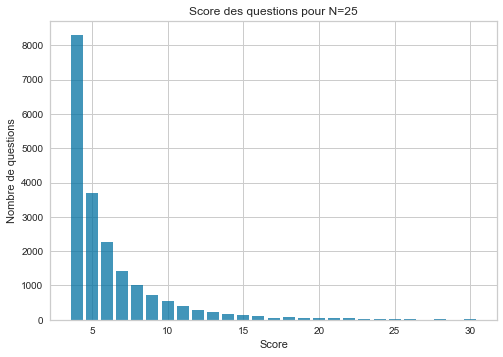


Figure 4 – Graphique : distribution des scores, max = 25

On remarque que les questions ont des notes très basses (volontairement, je n’ai pris que les questions ayant au moins un score de 4). Plus on cherche des questions bien notées, moins on en trouve.

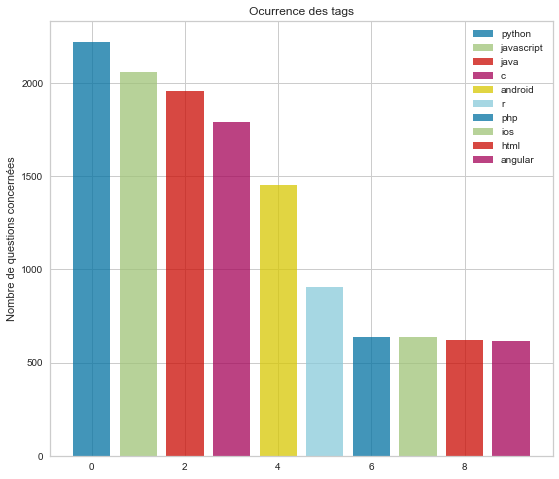


Figure 5 – Graphique : occurrence des tags

Les 10 tags les plus fréquent sont donc les suivants :

('python', 2223), ('javascript', 2060), ('java', 1960), ('c', 1790), ('android', 1455), ('r', 904), ('php', 636), ('ios', 636), ('html', 619), ('angular', 615)

### Balises html

La première chose que l’on remarque dans ces données est le fait qu’elles contiennent un certain nombre de balise html. Une balise html sert à adapter l’affichage d’un texte (ou autre) sur une page web. Dans notre cas, ces balises sont complètement inutiles, il faut donc les supprimer.

# Suppression des balises html et des parties de code

data['Body'] = [suppr\_html\_code(x) for x in data['Body']]

Figure 6 – Code python : Boucle qui permets de supprimer les signes html

Pour nous aider dans cette étape, la bibliothèque de parsage *Beautiful Soup* est utilisée.

Cette bibliothèque écrite en Python par *Leonard Richardson* permet de naviguer au sein de l'arbre créé par le parser, de chercher des éléments dans cet arbre ou les modifier.

Dans notre cas, on cherche à modifier toutes les marques HTML. J’ai donc écrit une fonction qui permet de supprimer ces marques de manière automatisée.

En poussant un peu plus la réflexion, hormis les balises HTML, une autre chose très importante dans les questions n’est pas forcément utile pour notre projet : il s’agit du code en lui-même (qui est encadré par des balises de code HTML justement). Dans la fonction, j’ai donc inséré également une suppression complète des codes.

def suppr\_html\_code(fichier):

# Suppression des balises HTML

soupe = BeautifulSoup(fichier, "lxml")

# Recherche des balises de 'code'

liste = soupe.findAll('code')

# Suppression des données qui sont entre les balises de code

for balise in liste:

balise.decompose()

# Sortie formatée en texte

return soupe.text

Figure 7 – Code python : fonction suppr\_html\_code

A l’issue de cette partie, on obtient du texte brut.

### Exploration du texte brut

On cherche dans un premier temps étudier le texte.

On a utilisé la fonction *word\_tokenize* qui va décomposer le texte en tableaux de mots afin de pouvoir effectuer des opérations dessus.

Il faut également pouvoir supprimer la ponctuation qui ne nous sert à rien.

Enfin, un autre problème est que certains mots ont des majuscules car ils apparaissent en début de phrases, alors que ce sont les mêmes mots que l’on peut retrouver plus tard, mais sans majuscule. On pourra les éliminer avec la fonction « lower ».

#### Première passe de nettoyage : supprimer les stopwords

La première manipulation effectuée dans le traitement du texte est la suppression des stopwords. Ce sont les mots très courants dans la langue étudiée ("et", "à", "le"... en français) qui n'apportent pas de valeur informative pour la compréhension du "sens" d'un document.

La bibliothèque NLTK fournit une liste par défaut des stopwords dans plusieurs langues, notamment l’anglais. Elle a été utilisée telle quelle, mais incrémentée d’autres éléments.

Les mots les plus fréquents du dataset font partie du vocabulaire commun et n'apportent aucune information, on supprima donc les N mots plus fréquents en plus des stepwords

#### Deuxième passe : lemmatisation

Le processus de « lemmatisation » consiste à représenter les mots sous leur forme canonique. Par exemple pour un verbe, ce sera son infinitif. Pour un nom, son masculin singulier. L'idée étant encore une fois de ne conserver que le sens des mots utilisés dans le corpus.

# Comptage du nombre d'occurence

cpt = comptage(data['Body'])

# Liste des stop words anglais

least\_used = set([word for word in cpt if cpt[word] < 100])

stop\_words = set(stopwords.words('english')) | set([word for word, freq, in cpt.most\_common(100)]) | least\_used

Figure 8 – Code python : Comptage d’occurrence des mots, puis création d’une liste de step\_words

def fct\_nltk(text, stop\_words):

# Création de l'objet

lemma = wordnet.WordNetLemmatizer()

# Tokenization et mise en minuscule

words = word\_tokenize(text.lower())

# Suppression des pluriels et de la ponctuation. Boucle pour toutes les lignes

new\_sentence = [lemma.lemmatize(x) for x in words if (not x in stop\_words) and x.isalpha()]

# Sortie

return new\_sentence

Figure 9 – Code python : fonction fct\_nltk

### Texte nettoyé

Les étapes essentielles du prétraitement du texte ont été réalisées :

* Tokenisation
* Suppression des stop-words,
* Lemmatisation.

# Exploitation du texte nettoyé

## Préambule

Avant d’entrer dans l’exploitation en elle-même, il convient de parler de trois techniques qui vont y être utilisées :

### Latent Dirichlet Allocation (LDA)

C’est une méthode non-supervisée générative qui se base sur les hypothèses suivantes :

* Chaque document du corpus est un ensemble de mots sans ordre (bag-of-words) ;
* Chaque document m aborde un certain nombre de thèmes dans différentes proportions qui lui sont propres ***p(θm)***
* Chaque mot possède une distribution associée à chaque thème *p(ϕk)*
* On peut ainsi représenter chaque thème par une probabilité sur chaque mot.
* **Zn** représente le thème du mot **wn**

Cette méthode permet d'expliquer des ensembles d'observations, par le moyen de groupes non observés, eux-mêmes définis par des similarités de données.

lda = LatentDirichletAllocation( n\_components=20,

max\_iter=5,

learning\_method='online',

learning\_offset=50,

random\_state=0)

Figure 10 – Code python : exemple LDA

### Réduction de la dimension

L’algorithme TruncatedSVD permet une décomposition en valeurs singulières. Cela permet ensuite de réduire la taille d’un dataset tout en gardant un pourcentage de variabilité du dataset initial assez important pour ne pas le dénaturer.

Cette technique a été testée dans ce projet mais n’a pas été retenu à la fin. Son apport n’a pas été jugée suffisamment pertinent pour le cas retenu.

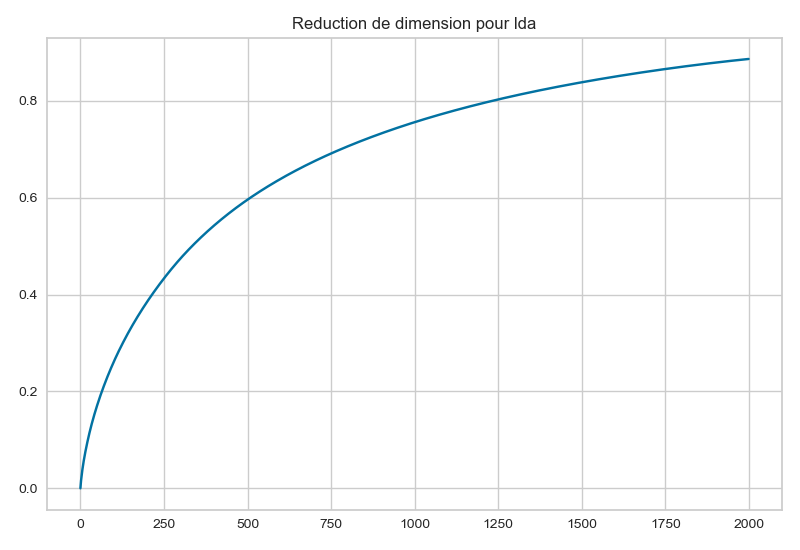


Figure 11 – Graphique : réduction de dimension

Sur cette courbe, on peut voir qu’en gardant 2000 éléments seulement, on gardait une variabilité d’environ 0,9.

### Visualisation des topics créés

## Bag of words

Il faut maintenant étudier comment l’extraction de l'information du texte pour le traitement ultérieur par des modèles de machine learning. La première méthode testée a été le « bag of words »

La manière la plus simple de représenter un document est de le représenter par un ensemble des mots qu'il contient. En pratique, ça peut être par exemple un vecteur de fréquence d'apparition des différents mots utilisés.

Une représentation bag-of-words classique sera donc celle dans laquelle on représente chaque document par un vecteur de la taille du vocabulaire |V| et on utilisera la matrice composée de l’ensemble de ces N documents qui forment le corpus comme entrée de nos algorithmes.

def countV(new\_df):

# Création de la matrice finale

matrixnum\_count, names\_count = creer\_countvectorizer(new\_df['Sentences'])

# Conversion de la matrice finale en dataframe pour facilité d'usage

matrixnum\_count = pd.DataFrame(matrixnum\_count,

columns=names\_count,

index=new\_df.index)

# Run LDA

lda = LatentDirichletAllocation(n\_components=20,

random\_state=0)

# Fit du LDA crée au-dessus

lda.fit(matrixnum\_count)

# Visualisation de la liste des mots, plus nuage de mots

display\_topics(lda, names\_count, 15, 'lda')

# Visualisation de la fréquence d'occurence

visualizer = FreqDistVisualizer(features=names\_count,

n=20,

orient='h',

color='g')

visualizer.fit(matrixnum\_count)

visualizer.poof()

# Tentative de réduction de la dimension

matrixnum\_count\_num = reduction\_dimension(matrixnum\_count, 2000, 'lda')

return matrixnum\_count, names\_count, lda

Figure 12 – Code python : fonction countV

def creer\_countvectorizer(text):

# Création de l'objet

vectorizer = CountVectorizer()

# Fit du texte d'entrée, et mis au format tableau

liste\_mots = vectorizer.fit\_transform(text).toarray()

# On ressort le tableau, et la liste des mots

return liste\_mots, vectorizer.get\_feature\_names()

Figure 13 – Code python : fonction creer\_countvectorizer

## Le tf-idf

On peut pousser la représentation un peu plus loin. Dans le « bag of words », on a seulement utilisé les fréquences d'apparition des différents mots présents dans notre corpus.

À présent, il ne faut pas considérer le poids d'un mot dans un document comme sa fréquence d'apparition uniquement, mais pondérer cette fréquence par un indicateur si ce mot est commun ou rare dans tous les documents.

En l’occurence, la métrique tf-idf (Term-Frequency - Inverse Document Frequency) utilise comme indicateur de similarité l'inverse document frequency qui est l'inverse de la proportion de document qui contient le terme, à l'échelle logarithmique. Il est appelé logiquement « inverse document frequency » (idf).

def tfidf(new\_df, data):

# Création de la matrice finale

matrixnum\_tfidf, names\_tfidf = creer\_tfidfvectorizer(new\_df['Sentences'])

# Conversion de la matrice finale en dataframe pour facilité d'usage

matrixnum\_tfidf = pd.DataFrame(matrixnum\_tfidf,

columns=names\_tfidf,

index=new\_df.index)

# Sépération des datasets

matrixnum\_tfidf\_train, matrixnum\_tfidf\_test = train\_test\_split(matrixnum\_tfidf,

test\_size=0.25,

random\_state=3)

# Run LDA

lda = LatentDirichletAllocation(n\_components=20,

random\_state=0)

# Fit du LDA crée au-dessus

lda.fit(matrixnum\_tfidf\_train)

# Visualisation de la liste des mots, plus nuage de mots

display\_topics(lda, names\_tfidf, 15, 'lda')

# Visualisation de la fréquence d'occurence

visualizer = FreqDistVisualizer(features=names\_tfidf,

n=20,

orient='h',

color='g')

visualizer.fit(matrixnum\_tfidf\_train)

visualizer.poof()

# Tentative de réduction de la dimension

matrixnum\_tfidf\_num = reduction\_dimension(matrixnum\_tfidf, 500, 'lda')

return matrixnum\_tfidf\_train, matrixnum\_tfidf\_test, names\_tfidf, lda

Figure 14 – Code python : fonction tfidf

def creer\_tfidfvectorizer(text):

# Création de l'objet

t\_vectorizer = TfidfVectorizer(min\_df=0.01)

# Fit du texte d'entrée, et mis au format tableau

liste\_mots = t\_vectorizer.fit\_transform(text).toarray()

# On ressort le tableau, et la liste des mots

return liste\_mots, t\_vectorizer.get\_feature\_names()

Figure 15 – Code python : fonction creer\_tfidfvectorizer

## Conclusion

Plus efficace et bien que les deux furent testées, seule la matrice TF-IDF a été gardée dans ce projet

# Les différentes pistes de modélisation effectuées

Trois pistes de modélisation ont été testées, elles suivront le plan de ce chapitre :

* Non-supervisé
* Non-supervisé, mais un peu aidé
* Supervisé

## Non-supervisé

Sans les tags d'origine, on va chercher à déterminer la probabilité d'appartenance d'un mot (et non un tag) à un topic. Autrement dit, on cherche à prédire des tags en se basant uniquement sur le texte, sans avoir connaissance des tags réels.

Le LDA nous permet d’obtenir la probabilité d'appartenance d'un message à un topic. On peut dire que les composants sont "composés" du mot et du nombre de fois qu'il apparait dans le topic.

Ensuite, en multipliant ces deux matrices entre elles, on obtient la matrice qu’on voulait avoir : la matrice de probabilité questions/mots. En prenant, pour chaque message, les 5 mots les plus probables de le représenter, on peut s’en servir comme tags proposés à l’utilisateur.

# Probabilité d'appartenance d'un message à un topic

df\_tp1 = pd.DataFrame(lda\_tfidf.transform(matrixnum\_tfidf\_test))

df\_tp1.index = matrixnum\_tfidf\_test.index

# "components\_" : Paramètres de variation pour la distribution topic/mots. Basée sur une Dirichlet, on peut dire que les composants sont "composés" du mot et du nombre de fois qu'il apparait dans le topic.

df\_tp3 = lda\_tfidf.components\_

# Multiplication des deux matrices df\_tp1 (questions/topics) et df\_tp3 (topics/mot) pour obtenir la

# matrice de probabilité questions/mots

df\_mots = df\_tp1.dot(df\_tp3)

df\_mots.columns = names\_tfidf

# Création de la matrice des mots les plus fréquents par document

df\_plus\_frequent = pd.DataFrame()

for i in df\_mots.index:

# On prends les 5 ayant la plus grande occurence

temp = df\_mots.loc[i].nlargest(5)

temp = temp.reset\_index()

df\_plus\_frequent[i] = temp['index']

df\_plus\_frequent = df\_plus\_frequent.T

df\_plus\_frequent.index = df\_tp1.index

# Comptage des bons tags prédit. Premier métrique.

print("Avec mots")

comptage\_metric(data, df\_plus\_frequent, 5)

## Non-supervisé, mais un peu aidé

## Supervisé

# Présentation du modèle final sélectionné ainsi que des performances et améliorations effectuées.