PARCOURS DATA SCIENTIST, PROJET 6

Générateur de tags

Amateur de Stack Overflow, qui vous a souvent sauvé la mise, vous décidez d'aider la communauté en retour. Pour cela, vous développez un système de suggestion de tag pour le site. Celui-ci prendra la forme d'un algorithme de machine learning qui assigne automatiquement plusieurs tags pertinents à une question.

TABLE DES MATIERES

TABLE DES FIGURES

1 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
- Conclusion

2 EXPLICATION DE LA PROBLEMATIQUE, DE SON INTERPRETATION ET DES DEDUCTIONS EFFECTUEES QUANT AUX PISTES DE RECHERCHE POSSIBLES

2.1 LA PROBLEMATIQUE

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

2.2 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, avec une logique différente
    - Supervisé

3 CLEANING EFFECTUE, FEATURE ENGINEERING ET EXPLORATION DES DONNEES

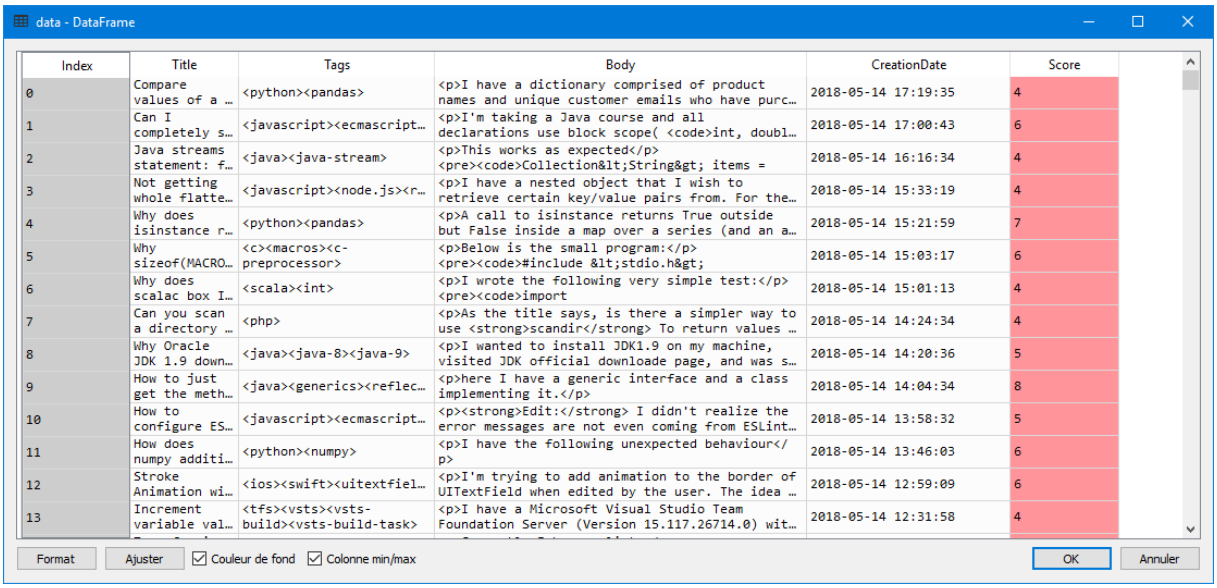
3.1 RECUPERATION DES DONNEES

Les données sont récupérées directement sur le site Stack Overflow. 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 de capacité de traitement 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.



Index	Title	Tags	Body	CreationDate	Score
0	Compare values of a …	<python><pandas>	I have a dictionary comprised of product names and unique customer emails who have purc…	2018-05-14 17:19:35	4
1	Can I completely s…	<javascript><ecmascript…	I'm taking a Java course and all declarations use block scope(<code>int, doubl…	2018-05-14 17:00:43	6
2	Java streams statement: f…	<java><java-stream>	This works as expected <pre><code>Collection<String> items =	2018-05-14 16:16:34	4
3	Not getting whole flatte…	<javascript><node.js><r…	I have a nested object that I wish to retrieve certain key/value pairs from. For the…	2018-05-14 15:33:19	4
4	Why does isinstance r…	<python><pandas>	A call to isinstance returns True outside but False inside a map over a series (and an a…	2018-05-14 15:21:59	7
5	Why sizeof(MACRO…	<c><macros><c-preprocessor>	Below is the small program: <pre><code>#include <stdio.h>	2018-05-14 15:03:17	6
6	Why does scalac box I…	<scala><int>	I wrote the following very simple test: <pre><code>import	2018-05-14 15:01:13	4
7	Can you scan a directory …	<php>	As the title says, is there a simpler way to use scandir To return values …	2018-05-14 14:24:34	4
8	Why Oracle JDK 1.9 down…	<java><java-8><java-9>	I wanted to install JDK1.9 on my machine, visited JDK official downloade page, and was s…	2018-05-14 14:20:36	5
9	How to just get the meth…	<java><generics><reflec…	here I have a generic interface and a class implementing it.	2018-05-14 14:04:34	8
10	How to configure ES…	<javascript><ecmascript…	Edit: I didn't realize the error messages are not even coming from ESLint…	2018-05-14 13:58:32	5
11	How does numpy additi…	<python><numpy>	I have the following unexpected behaviour</ p>	2018-05-14 13:46:03	6
12	Stroke Animation wi…	<ios><swift><uitextfiel…	I'm trying to add animation to the border of UITextField when edited by the user. The idea …	2018-05-14 12:59:09	6
13	Increment variable val…	<tfs><vsts><vsts-build><vsts-build-task>	I have a Microsoft Visual Studio Team Foundation Server (Version 15.117.26714.0) wit…	2018-05-14 12:31:58	4

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

3.2 NETTOYAGE DES DONNEES, FEATURE ENGINEERING ET EXPLORATION DES DONNEES

3.2.1 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 côté 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.

3.2.2 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

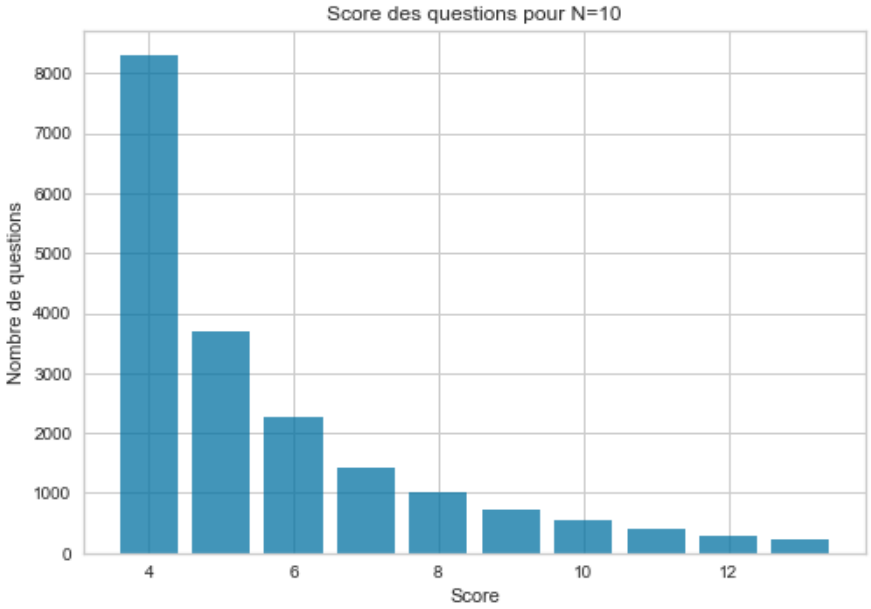


Figure 3 – Graphique : distribution des scores, max = 10

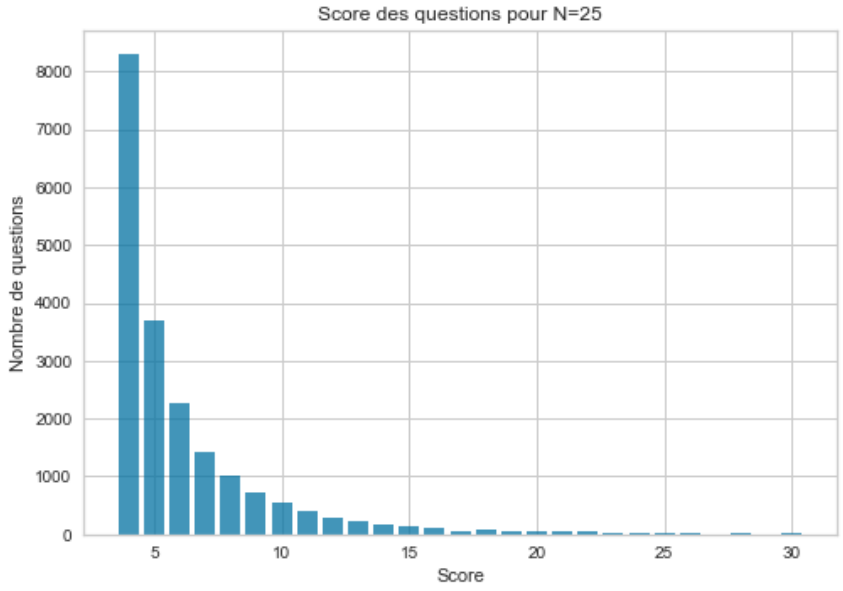


Figure 4 – Graphique : distribution des scores, max = 25

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.

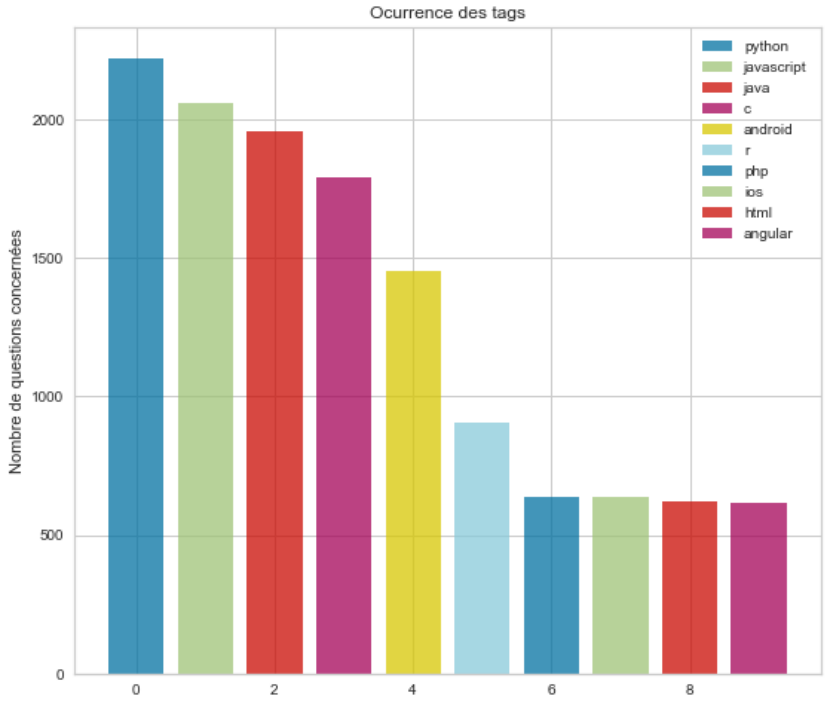


Figure 5 – Graphique : occurrence des tags

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)

3.2.3 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 d'analyse de contenu HTML *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.

3.2.4 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 ».

3.2.4.1 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

3.2.4.2 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 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

3.2.5 Texte nettoyé

Les étapes essentielles du prétraitement du texte ont été réalisées :

- Tokenisation
- Mise en minuscule
- Suppression des stop-words,
- Lemmatisation.

4 EXPLOITATION DU TEXTE NETTOYE

4.1 PREAMBULE

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

4.1.1 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

Ce LDA va donc créer un certain nombre de topics (ici 20) qui pourront être exploités dans la suite du projet.

4.1.2 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.

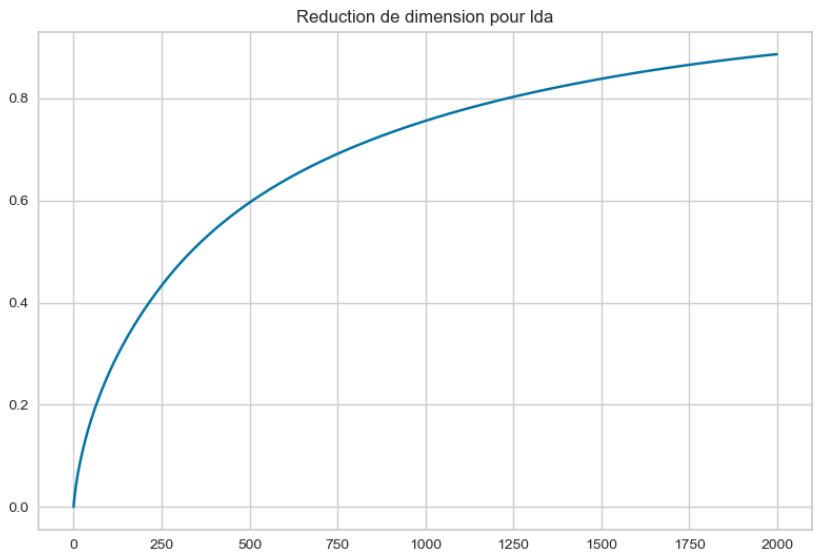


Figure 11 – Graphique : réduction de dimension

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

4.1.3 Visualisation des topics créés

Wordcloud est un générateur de nuage de mots clés et de nuage de tags. Il permet d'avoir une vue « artistique » des topics que LDA a créé. Une vingtaine de topics ont été créés avec le paramètre idoine

configuré pour cela (n_components=20). A noter que ce paramètre peut et va être modifié pour les tests finaux.

Voici un exemple d'un topic avec son wordcloud associé:

Topic 9

['request', 'service', 'google', 'message', 'url', 'client', 'web', 'http', 'spring', 'access', 'browser', 'connection', 'firebase', 'event', 'header']

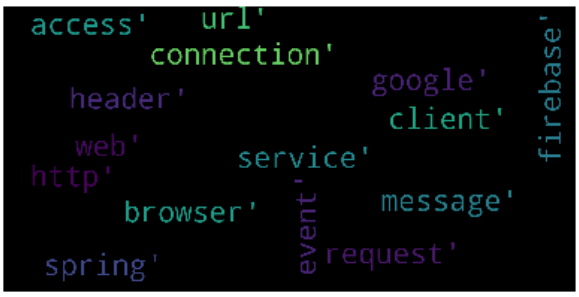


Figure 12 – Graphique : exemple d'un wordcloud

D'après les mots-clefs, on peut penser que ce topic aborde des problèmes de configuration d'explorateur de pages internet.

4.1.4 Métrique

Une méthode d'évaluation doit être crée afin de pouvoir comparer l'efficacité des différents algorithmes et méthodes qui sont utilisées pour ce projet.

Ce métrique suit l'idée suivante. Une évaluation de tags correctement prédit est faite. Cette évaluation va prendre les tags réels et va les comparer aux prédictions. Le compteur incrémente les bonnes prédictions.

Le résultat que l'on obtient est donc le pourcentage de tags prédit que l'on retrouve bien comme tag d'origine.

```
def comptage_metric(data, df_prevision, value):
    # Comptage des bons tags prédits
    count_tag = 0
    for i in df_prevision.index:
        liste_tags = word_tokenize(data.loc[i, 'Tags'])
        for tag in liste_tags:
            for j in range(0, value):
                if tag == df_prevision.loc[i, j]:
                    count_tag = count_tag + 1
```

```
    print(round(count_tag/df_prevision.count().sum()*100, 1), '%')
```

Figure 13 – Code python : fonction comptage_metric

4.2 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')

    # Tentative de réduction de la dimension

    matrixnum_count_num = reduction_dimension(matrixnum_count, 2000,
'lda')

    return matrixnum_count, names_count, lda
```

Figure 14 – 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 15 – Code python : fonction creer_countvectorizer

4.3 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'occurrence, 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')
    # 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 16 – 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 17 – Code python : fonction creer_tfidfvectorizer

4.4 CONCLUSION

Plus pertinente pour notre projet ainsi que plus efficace et bien que les deux furent testées, seule la matrice TF-IDF a été gardée dans ce projet.

En effet, la matrice TF-IDF n'a pas besoin d'être réduite et ne perds donc pas de données. La matrice « bag of words » quant à elle est plus imposante, et aurait dû être réduite pour être exploitée.

5 LES DIFFERENTES PISTES DE MODELISATION EFFECTUEES

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

- Non-supervisé
- Non-supervisé, avec une logique différente
- Supervisé

5.1 NON-SUPERVISE

5.1.1 Concept

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)
```

Figure 18 – Code python : méthode non-supervisée

5.1.2 Un exemple

5.1.2.1 Question posée

"Ways to apply an applicative functor multiple timesHello fellow Haskellers.

Let's say I have an applicative functor (not an instance of monad) I

want to apply multiple times to a pure initial value. For example,

If I want to generalize this to any number of consecutive applications, I can do it with a .

After this definition,

I have an awkward suspicion that the generalization could also be

done with one of the higher-order builtin applicative tools such as or . Can it?

(Edited to take into account the first two comments below.)"

5.1.2.2 Tags réels

- haskell

5.1.2.3 Prédiction des tags

Aucun.

5.1.3 Résultat chiffré

Non supervisé						
n	20			30		
min_df	0,01	0,03	0,05	0,01	0,03	0,05
Avec mots (sans tags)	1%	0,80%	0%	1,10%	0,60%	0%

Figure 19 – Tableau : résultat méthode non-supervisée v1

Les résultats sont très décevants. Cette méthode n'est pas retenue pour créer un générateur de tags.

5.2 NON-SUPERVISE, AVEC UNE LOGIQUE DIFFERENTE

5.2.1 Concept

Cette fois-ci, avec les tags d'origine, on va chercher à déterminer la probabilité d'appartenance d'un tag à un topic.

Pour ce faire, on utilise toujours la première matrice df_tp1 crée par LDA mais la deuxième matrice sera créée de toute pièce en code. Cette deuxième matrice va représenter l'occurrence d'apparition des tags pour chaque topic.

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

```
## PARTIE 2

# 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

# Comptage d'occurence d'apparition des tags pour chaque topic

df_tp2 = pd.DataFrame(columns=liste_tags, index=range(0, 20))

for i in liste_tags:

    mot = ' ' + i + ' '

    mask = data['Tags'].str.contains(str(mot), regex=False)

    temp = df_tp1[mask]

    for j in df_tp1.columns:

        df_tp2.loc[j, i] = temp[j].sum()

# Convertion en float

df_tp2 = df_tp2.astype(float)

# Multiplication des deux matrices df_tp1 (questions/topics) et df_tp2 (topics/tags) pour obtenir la

# matrice de probabilité questions/tags

df_tags = df_tp1.dot(df_tp2)

# Transformation en dataframe

df_tags = df_tags.astype(float)

# Création de la matrice pour afficher les tags les plus fréquents par document

df_prevision = pd.DataFrame()

for i in df_tags.index:

    # On prends les 5 ayant la plus grande occurence
```

```
        temp = df_tags.loc[i].nlargest(5)

        temp = temp.reset_index()

        # On supprime ceux qui ne sont pas dans le body de la question

        for k, j in enumerate(temp['index']):

            mot = ' ' + j + ' '

            if mot not in data.loc[i, 'Body']:

                temp = temp.drop(k)

        df_prevision = df_prevision.append(temp['index'])

    # On récupère les index d'origine

    df_prevision = pd.DataFrame(df_prevision).reset_index(drop=True)

    df_prevision.index = df_tp1.index

    # Comptage des bons tags prédit. Second métrique.

    print("Avec tags")

    comptage_metric(data, df_prevision, 5)
```

Figure 20 – Code python : méthode non-supervisée v2

5.2.2 Un exemple

5.2.2.1 Question posée

"Detecting reference to cell from other workbooks?Is there a way with VBA and/or some formula in Excel to check whether there are other workbooks/sheets referencing a cell? Ideally, also from which workbooks/sheets but if this is not possible, that's also ok.

Lets say I have a workbook with a list of proxy addresses, I want to know if a proxy is already being used by checking if there's any other workbook referencing its cell. This is to have an indicator whether it's a free proxy or already in use.

Any alternative solution that's close to this is also welcome. I'm not per se looking for a full blown solution, but I can get far by pointing me in the right direction. "

5.2.2.2 Tags réels

- Excel
- Vba
- excel-vba
- excel-formula

5.2.2.3 Prédiction des tags

- Excel

Un seul est prédit, mais il peut être considéré comme tout à fait pertinant.

5.2.3 Résultat chiffré

Non supervisé						
n	20			30		
min_df	0,01	0,03	0,05	0,01	0,03	0,05
Avec tags	55%	60,50%	62,90%	54,80%	60,40%	63%

Figure 21 – Tableau : résultat méthode non-supervisée v2

Ces résultats sont nettement meilleurs que ceux observés avec la première méthode. Ils sont également sensibles aux hyperparamètres choisis.

Pour aller plus loin, on pourrait faire une étude complète sur un panel beaucoup plus large de valeurs pour ces hyperparamètre.

5.3 SUPERVISE

5.3.1 Concept

Il s'agit de la troisième grande étape avec un test de manière supervisée.

On utilise l'algorithme RandomForestClassifier qui a fait ses preuves dans le domaine de classification ainsi que l'algorithme SGDClassifier.

Quelques hyperparamètres seront également testés. Pour le RandomForestClassifier, on agira sur le « max_depth » et « n_estimators » car ils font parti des plus importants. Pour le SGDClassifier, on fera varier uniquement le « penalty » pour nous donner une idée de son influence dans ce projet.

L'algorithme RandomForestClassifier appartient à la famille des agrégations de modèles. Le principe est de faire la moyenne des prévisions de plusieurs modèles indépendants pour réduire la variance et donc l'erreur de prévision. Pour construire ces différents modèles, on sélectionne plusieurs échantillons (tirages avec remises). Pour chaque arbre on sélectionne un échantillon d'individus et à chaque étape, la construction d'un nœud de l'arbre se fait sur un sous-ensemble de variables tirées aléatoirement. On se retrouve donc avec plusieurs arbres et donc des prédictions différentes. Pour obtenir la valeur finale, dans le cas d'une classification, on garde la valeur la plus probable.

L'algorithme SGDClassifier est une méthode de descente de gradient (itérative) utilisée pour la minimisation d'une fonction objectif. Le principe est de construire une séquence de modèles de sorte que chaque étape et chaque modèle ajoutés à la combinaison, apparaisse comme un pas vers une meilleure solution. Ce pas est franchi dans la direction du gradient de la fonction perte, afin d'améliorer les propriétés de convergence.

On utilise également la stratégie de classification OneVsRestClassifier. Cette stratégie permet de fiter un classifieur par classe, or nous sommes dans un problème multidimensionnel.

Ensuite, en faisant une prédiction des probabilités, on peut retirer les 5 tags les plus probables parmi ceux existants et les proposer à un nouvel utilisateur pour sa question.

```
def supervise(data, matrixnum_tfidf_train, matrixnum_tfidf_test, df_tags_train, df_tags_test):
    # Choix de l'algorithme de classification
    models = [RandomForestClassifier(n_estimators=10, max_depth=30),
              RandomForestClassifier(n_estimators=10, max_depth=None),
              RandomForestClassifier(n_estimators=30, max_depth=30),
              RandomForestClassifier(n_estimators=30, max_depth=None)]
    for model in models:
        # Nom du classifieur
        name = model.__class__.__name__ + '_' + str(model.max_depth) + '_' + str(model.n_estimators)
        # Choix de l'algorithme de décision
        classif = OneVsRestClassifier(model)
        # Choix entre fit de nouveau ou aller chercher le fit sauvegardé
        classif.fit(matrixnum_tfidf_train, df_tags_train)
        # Predictions
        predictions = classif.predict_proba(matrixnum_tfidf_test)
        predictions = pd.DataFrame(predictions,
                                   index=df_tags_test.index,
                                   columns=df_tags_test.columns)
        df_prevision_finale = pd.DataFrame()
        # Cette partie permet de retirer dans un premier 5 prédictions, puis de vérifier si elles sont dans le body de la question. Si ce n'est pas le cas, on ne les proposera pas.
        for p in predictions.index:
            temp = predictions.loc[p]
            temp = temp[temp>0].nlargest(5).reset_index()
            # On supprime ceux qui ne sont pas dans le body de la question
```

```
            for k, v in enumerate(temp['index']):
                mot = ' ' + v + ' '
                if mot not in data.loc[p, 'Body']:
                    temp = temp.drop(k)
            df_prevision_finale = df_prevision_finale.append(temp['index'])
        df_prevision_finale =
pd.DataFrame(df_prevision_finale).reset_index(drop=True)
        df_prevision_finale.index = matrixnum_tfidf_test.index
        # Comptage des bons tags prédit. Troisième métrique.
        print("Supervisé")
        comptage_metric(data, df_prevision_finale, 5)
        # On termine sur un affichage de questions, tags d'origines et tags prédit.
        for k in df_prevision_finale.index[:20]:
            # On prend les lignes une par une
            ligne = df_prevision_finale.loc[k].copy()
            # Variable list qui va prendre les résultats des prédictions
            predicted_label = []
            for word in ligne:
                if not pd.isna(word):
                    predicted_label.append(word)
            # Impression du resultat
            if len(predicted_label) > 0:
                print(data['Body'].loc[k][:80], "...")
                print('Actual label :', data['Tags'].loc[k])
                print("Predicted label :", predicted_label, "\n")
```

Figure 22 – Code python : fonction supervise

5.3.2 Un exemple

5.3.2.1 Question posée

"Given a string of a million numbers, return all repeating 3 digit numbersI had an interview with a hedge fund company in New York a few months ago and unfortunately, I did not get the internship offer as a data/software engineer. (They also asked the solution to be in Python.)

I pretty much screwed up on the first interview problem...

Question: Given a string of a million numbers (Pi for example), write

a function/program that returns all repeating 3 digit numbers and number of

repetition greater than 1

For example: if the string was: then the function/program would return:

They did not give me the solution after I failed the interview, but they did tell me that the time complexity for the solution was constant of 1000 since all the possible outcomes are between:

000 --> 999

Now that I'm thinking about it, I don't think it's possible to come up with a constant time algorithm. Is it? "

5.3.2.2 Tags réels

- python
- algorithm
- data-structures
- number-theory

5.3.2.3 Prédiction de tags

- algorithm
- android
- perl6
- ruby-on-rails
- templates

Un seul est correct sur les 5 proposés. Aucun autre n'est proposé correctement, ni même s'en approche.

5.3.3 Résultat chiffré

Supervisé								
max_depth	none	none	30	30				
n_estimators	10	30	10	30				
penalty					none	l1	l2	elasticnet
RandomForestClassifier	63,50%	65,40%	60%	65,00%				

SGDClassifier					60,70%	61,70%	64,10%	62,80%

Figure 23 – Tableau : résultat méthode-supervisée

Les résultats sont globalement identiques pour les deux algorithmes utilisés. Comme pour la méthode supervisée, mais à un degré un peu moindre, les résultats sont sensibles aux hyperparamètres qu'il faudra donc tester plus en profondeur.

6 CONCLUSION

6.1 RAPPEL DES RESULTATS

Non supervisé						
n	20			30		
min_df	0,01	0,03	0,05	0,01	0,03	0,05
Avec mots (sans tags)	1%	0,80%	0%	1,10%	0,60%	0%
Avec tags	55%	60,50%	65,90%	54,80%	60,40%	67%

Figure 24 – Tableau : résultat méthode non-supervisée v2

Supervisé								
max_depth	none	none	30	30				
n_estimators	10	30	10	30				
penalty					none	l1	l2	elasticnet
RandomForestClassifier	63,50%	65,40%	60%	65,00%				
SGDClassifier					60,70%	61,70%	64,10%	62,80%

Figure 25 – Tableau : résultat méthode-supervisée

6.2 EXPLICATIONS

Plusieurs critères doivent rentrer en compte :

L'expérience de l'utilisateur est un critère qui n'a pas été exploité dans ce projet mais qui est important. Les résultats peuvent être différents si on entraine le modèle uniquement sur des utilisateurs expérimentés.

De plus, les méthodes supervisées et non-supervisées bien que donnant des résultats assez similaires sur l'échantillon étudié (à 10% près, néanmoins), ont chacune leur avantage. Le supervisé va toujours avoir des meilleures performances tant que le non-supervisé permet de créer une liste tag avec des tags qui n'étaient pas là à l'origine). On peut cependant penser que ce dernier avantage n'est pas forcément très important sur Stack Overflow.