CATÉGORISEZ AUTOMATIQUEMENT DES QUESTIONS.

PAR :

NAMA NYAM GUY ANTHONY

10 Mars 2020

Synthèse de travail

Objectifs

Ce rapport a pour but de présenter les techniques fondamentales du traitement automatique de la langue(Natural Language Processing en anglais) en particulier les traitements et modélisations effectués lors notre mission de développement d'**un système de suggestion des tags** pour le site **Stack Overflow**. Notre objectif est de développer une compréhension des étapes courantes d'une application NLP en parallèle à notre problématique.

- **Introduction.** NLP, les principales difficultés, notre problématique, les applications courantes, les approches utilisées.
- **Analyse textuelle.** Morphologie flexionnelle et dérivationnelle, balises html, abréviations, les formes contractées, les stop words, les ponctuations, les minuscules, les chiffres, les mots non fréquents, les 'mauvais' symboles et les espaces, les catégories grammaticales.
- **Extraction des features.** Comptage des mots(Tf) et pondération des mots(Tf-Idf) ; leurs limites, word embeddings(Glove, distilBert, fastText).
- **Modèlisation machine.** Approche supervisée pour l'extraction des tags(XGBoost, CatBoost, Lightgbm, MultinomialNB, Faiss), approche non supervisée pour l'extraction des mots clés ou keywords(LDA).
- **Evaluation.** Mise en place d'une méthode d'évaluation propre, intégrer la métrique aux classifiers, fine tuning avec RandomizedSearchCV
- **Déploiement.** Gestion de versions avec l'outil Git mise en place avec colab de Google, Google drive et Github. Générer une api pour utilisation.
- **Conclusion.**

Table des matières

1 Introduction

1.1 Qu'est ce que NLP ?

Le 'Natural Language Processing' (NLP) peut être définie comme le traitement automatique du langage humain. C'est un domaine multidisciplinaire entre la linguistique, les sciences cognitives, mathématiques et l'informatique. L'analyse de texte est un domaine d'application majeur pour les algorithmes d'apprentissage machine.

1.2 Les principales difficultés

Les ambiguïtés sont les principales difficultés rencontrées en NLP. Elles sont présentes à tous les niveaux de la subdivision standard de la linguistique :

1. Morphologie : la structure du mot. Elle est de deux types à savoir flexionnelle et dérivationnelle qui sont traités par les algorithmes de racinisation(stemming) et lemmation.
2. Syntaxique : la manière dont les mots sont utilisés pour former une phrase(le déterminant 'le' précèdent un 'nom commun'). l'analyse s'appuie sur les catégories grammaticales ; noms, déterminants, adjectifs, adverbes, etc.
3. Sémantique : sémantique compositionnelle pour la construction du sens des phrases et sémantique lexicale pour le sens individuel des mots(avocat pour fruit ou profession)
4. Pragmatique : le sens dans le contexte.

1.3 Problématique

Pour poser une question sur le site de Stack Overflow, il faut entrer plusieurs tags de manière à retrouver facilement la question par la suite. Il est judicieux de suggérer quelques tags relatifs à la question posée pour les utilisateurs(généralement les débutants) qui ne savent les étiquettes associées à leur question. Il est donc question de développer un système qui assigne automatiquement plusieurs tags pertinents à une question.

1.4 Quelques applications NLP

L'une des applications les actives est le système de questions-réponses(question answering) dont fait référence l'application développée au cours de ce projet. Cette dernière consiste à proposer plusieurs tags(réponses) pour une question posée.
La liste suivante non exhaustive de quelques systèmes usuels proche de notre cas d'application :

- Extraction de sujets
- Classification des sentiments
- Chatbots
- Traduction automatique
- Classification de documents
- Recherche d'information
- Extraction d'information
- Résumé

1.5 Approches

Deux principales approches sont utilisées en *NLP* à savoir l'approche symbolique ou basée sur les règles(connaissance linguistique avancée nécessaire) et l'approche statistique. Cette dernière constitue l'essentielle des techniques utilisées dans ce projet due à l'importante source de données disponibles ainsi que du support matériel. Nous allons développer un algorithme de machine learning.

2 Analyse textuelle de la donnée

C'est l'entrée à presque tout problème *NLP*. Elle consiste au nettoyage et normalisation du texte. Les bibliothèques ***NLTK***, ***SPACY*** et les ***expressions régulières*** constituent les principaux outils de pré-traitement du texte. L'ordre d'utilisation des différentes opérations listées ci-dessous ont une importance. Nous les listons dans l'ordre utilisé.
Les 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.

2.1 Tokenisation

C'est une étape importante qui consiste à diviser un texte en unité(généralement mot) selon un séparateur. Il ne s'agit pas toujours de l'espace et dépend du langage traité. Le manière la plus simple est d'utiliser 'word_tokenizer' de *NLTK*.

```
import nltk
nltk.download('punkt')
```

```
from nltk.tokenize import word_tokenize
text = "Hello, thank you to read this support"
tokens = word_tokenize(text)
print(tokens)
```

```
['Hello', ',', 'thank', 'you', 'to', 'read', 'this', 'support']
```

Pour des tokenisations plus complexes, soit s'intéresser à l'algorithme de tokenisation de *spacy* [1] ou alors définir soit même son propre tokenizer

2.2 Suppression des tags HTML

L'utilisation de la bibliothèque ***BeautifulSoup*** permet largement de supprimer les balises HTML et XML.

```
from bs4 import BeautifulSoup
```

```
soup = BeautifulSoup(html_doc, 'html.parser')  # html_doc is the document with html tags
```

2.3 Suppression des espaces et mise en minuscule

Il s'agit ici de supprimer les espaces d'erreur de début et de fin du texte d'un document ainsi que le casting en minuscule de l'ensemble du texte.

```
text = ' Hello, NLP is for Natural Langage Processing '
print(text.lower().strip())
```

```
hello, nlp is for natural langage processing
```

2.4 Identification abréviations

L'objectif est d'identifier les abréviations(forme courte) et leur correspondant(forme longue). Puis, de soustraire les abréviations dans le texte par leurs correspondants'. L'algorithme de ***Schwartz-Hearst***[5]

Exemple : Programmation orientée objet(POO). POO est la forme courte et Programmation orientée objet la forme longue.

Les abréviations candidates sont déterminées par leur adjacence aux parenthèses. Deux cas répertoriés :

1. https ://spacy.io/usage/linguistic-features#how-tokenizer-works

1. forme longue ‘(‘forme courte‘)’
2. forme courte ‘(‘forme longue‘)’

Une forme courte est candidate s’il a deux mots au plus, de longueur comprise entre 2 et 10 caractères, au moins un de ses caractères est une lettre et le premier caractère est alphanumérique.

Une forme longue est candidate s’il apparaît dans la même phrase que la forme courte, de taille supérieur à $min(|A| + 5, |A| * 2)$, où $|A|$ est le nombre de caractère dans la forme courte.

```
pip install abbreviations  # Installation as a module
```

```
from abbreviations import schwartz_hearst
```

```
pairs = schwartz_hearst.extract_abbreviation_definition_pairs(doc_text='The emergency room (ER) was busy')
pairs
```

```
{'ER': 'emergency room'}
```

A la suite de l’extraction des abréviations par l’algorithme de Schwartz-Hearst, un travail manuel peut être réalisé pour filtrer les abréviations pertinentes.

2.5 Mappage des formes contractées

Pour réaliser à bien cette tâche, cela nécessite un dictionnaire les plus exhaustives possibles des formes contractées et leur correspondant. La bibliothèque python ***re*** est efficace pour rechercher et remplacer ses différents patterns.

```
import re
```

```
text = 'I\'m happy to improve my skills'
re.sub(r"\'m", " am", text)
```

```
'I am happy to improve my skills'
```

2.6 Suppression des ponctuations et symboles spéciaux

Les ponctuations et symboles spéciaux dans le traitement du texte de base comme mot n’apportent pas d’information. Les expressions régulières permettent de les filtrer efficace. Les patterns retrouvés sont remplacés par un espace(’ ’).

```
text = "[didier@gmail.com] 4road street, oyom : +237"
REPLACE_BY_SPACE_RE = re.compile('[/(){}\[\]\|@,;-_+]')
BAD_SYMBOLS_RE = re.compile('[^0-9a-z #]')

text = REPLACE_BY_SPACE_RE.sub(' ', text)
text = BAD_SYMBOLS_RE.sub('', text)

text
```

```
' didier gmailcom  4road street  oyom   237'
```

Les symboles traités ici sont non exhaustives. De plus, il faut être vigilant avec les filtres présentés car dépend aussi de la tâche à réaliser.

2.7 Suppression des mots courants(stop words)

Le conteneur ***Token*** de *spacy* offre d’énorme possibilité pour le traitement efficace de cette tâche. L’attribut *is_stop’* permet d’identifier les tokens considérés comme mot courant ou *stop words*. On peut également personnaliser le vocabulaire des mots considérés comme *stop words*.

```
import spacy
```

```
nlp = spacy.load('en_core_web_sm', disable=["tagger", "parser", "ner", "textcat"])
```

```
nlp.vocab["The"].is_stop = False
word = nlp("The world !")
word[0].is_stop  # word[0] == "The"
```

```
False
```

2.8 Suppression des chiffres

Cette opération consiste à supprimer les nombres ainsi qu'à retirer les chiffres des chaînes de caractères qui n'apportent pas d'informations pertinentes mais qui pourraient alourdir le temps de calcul.

```
words = ['+237', 'Ach4life', '123']
words_no_number = []
for word in words:
    words_no_number.append(''.join([w for w in word if not w.isdigit()]))
words_no_number
```

```
['+', 'Achlife', '']
```

2.9 Suppression des espaces multiples

Après les opérations précédentes, des espaces multiples peuvent être créés. L'opération de suppression appliqué est tout simplement : *text = ' '.join(text.split())*. Où *text* représente le texte avec les espaces multiples.

2.10 Suppression des mots de taille faibles

Généralement, les mots de taille 1 ne sont pas porteurs d'informations ; d'où leurs suppressions. Ses mots sont énormément générés lors de la phase de suppression des ponctuations, symboles spéciaux et chiffres. De le cadre de ce projet, nous avons néanmoins gardé certains à lien avec les étiquettes tels que : c, r, etc.

2.11 Lemmatisation

La **lemmatisation** est un traitement lexical d'un mot pour le mettre dans sa forme canonique.

Exemples :

1. Le lemme *petit* renvoie à 4 formes dérivationnelles : petit, petite, petits, petites
2. Le lemme *aimer* renvoie à de nombreuses formes flexionnelles : a aimé, a été aimé, a été aimée, etc.

Tout comme la suppression des stop words, la bibliothèque Spacy offre un attribut du nom de *'lemma_'* pour renvoyer le lemme d'un mot ou token et qui est particulière adapté pour cela.

```
text = "NAMA is working on Applications of NLP and he keeps organizing local Python meetups."
document = nlp(text)
lemmes = dict()
for token in document:
    lemmes[token] = token.lemma_
print(lemmes)
```

```
{NAMA: 'NAMA', is: 'be', working: 'work', on: 'on', Applications: 'Applications', of: 'of', NLP: 'NLP',
```

2.12 Suppression des mots non fréquents

C'est une étape pour filtrer le bruit qui se présente sous l'aspect de mots non fréquents. Généralement en NLP, le seuil de mots non fréquents tourne autour de 10. Dans ce projet, le seuil est de 3 pour qu'un mot ne soit pas non fréquent. Ce choix est fait sur la base qu'à partir de ce seuil, les mots recensés étaient à majorité reconnus dans la langue anglaise.

2.13 Filtrage des mots par catégorie grammaticale

Cette dernière étape conserve spécialement l'apprentissage des mots clés ou keywords. Comme défini dans le travail de N. Pudota et al. (2010) dans [4], la catégorie grammaticale des uni-grammes candidats doit être : nom, adjectif ou verbe. Une fois de plus, nous utilisons la bibliothèque Spacy pour obtenir l'étiquette grammaticale de chaque mot ou token grâce à l'attribut *'pos_'*.

3 Extraction des features

Il s'agit d'extraire les caractéristiques d'un ensemble de texte dans un format pris en charge par les algorithmes d'apprentissage machine. Ces algorithmes attendent des vecteurs de caractéristiques numériques de taille fixe plutôt que des documents de texte brut de longueur variable. Deux représentations creuses : Bag of words [2] et word embedding

3.1 Bag of words ou comptage de mots

Nous présentons le "Bag of 1-gramme" pour les représentations Tf et Tf-idf. Chaque token(les mots contenus dans les documents) est traité comme une caractéristique ou feature. Le module ***feature_extraction*** de ***sklearn*** permet d'obtenir ces représentations.

3.1.1 Tf

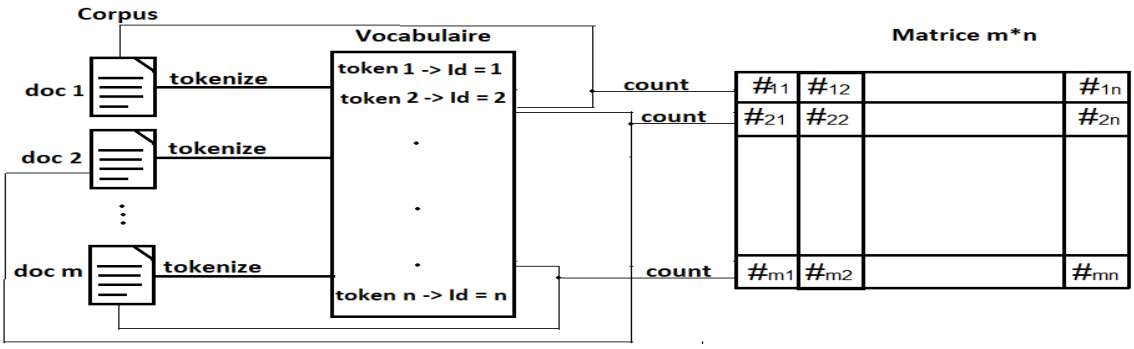

FIGURE 1 – Processus de la représentation TF

$\#_{ij}$ représente le nombre d'occurrence du token j dans le document i

```
# Common Vectorizer usage
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()  # Implements both tokenization and occurrence counting
corpus = [
          'Voici le premier document.',
          'Voici le second document.',
          'Et the troixième.'
          ]
  X = vectorizer.fit_transform(corpus)
```

2. https ://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction

3.1.2 Tf-idf

Dans de large corpus textuel, certains mots(hormis les stop words) très présent impacteront sur la représentation car apportent très peu d'informations significatives sur le contenu réel du document.

Dans le but de réduire l'importance relative de certains mots, il est très courant d'utiliser la transformation tf-idf. Chaque occurrence $\#_{ij}$ sera remplacée par $tf\text{-}idf(t_j, d_i)$ normalisée par la norme 2.

$$tf\text{-}idf(t,d) = tf(t,d) * idf(t), \text{ où :}$$
$tf(t,d)$ correspond à la fréquence du terme ou token t dans le document d.
$idf(t) = \log \frac{1+m}{1+df(t)} + 1$, où m est le nombre total de documents et $df(t)$ est le nombre de documents dans lequel apparait le terme ou token t.

L'utilisation est identique à la précédente transformation en utilisant la classe ***TfidfVectorizer***

3.1.3 Limitations

La représentation d'unigrammes de documents décrits par les occurrences de mots ignore la dépendance de l'ordre des mots. Elle ne capture donc pas les phrases et expressions de plusieurs mots. De plus, la représentation ne tient pas d'éventuelles fautes d'orthographe et dérivations des mots. Une amélioration est d'utiliser les N-grammes sur les mots et les caractères.

Afin d'aborder la tâche plus large de la compréhension du langage naturel, la structure locale des phrases et des paragraphes devrait donc être prise en compte ;

3.2 Plongement de mots ou word embedding

C'est une technique de représentation d'un mot par un vecteur dense de dimension inférieure(20 à 100 en générale) dans un espace avec une forme de similarité entre le sens des mots. L'hypothèse principale est de prendre en compte le contexte dans lequel le mot est trouvé, c'est-à-dire les mots avec lesquels il est souvent utilisé. On appelle cette hypothèse ***distributional hypothesis***.
Exemple : $Vec("Reine") \cong Vec("Homme") + Vec("Femme") + Vec("Roi")$

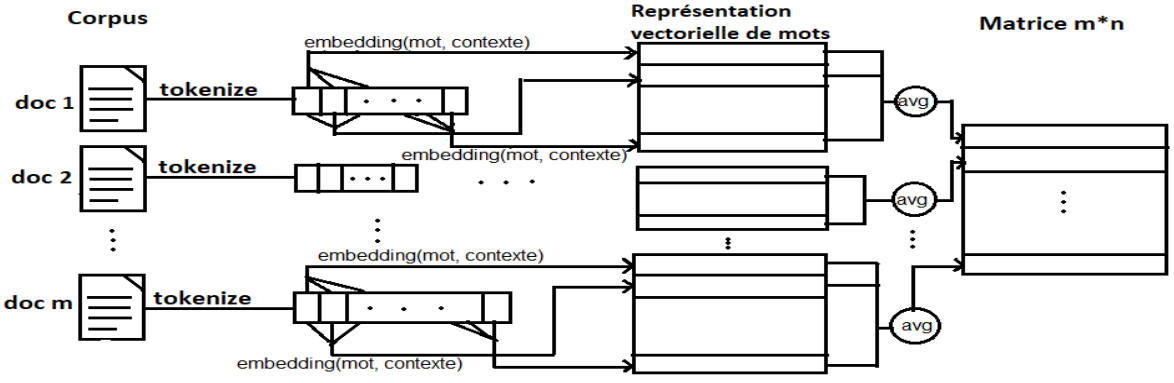

FIGURE 2 – Aperçu général de l'utilisation des méthodes de plongements de mots

avg représente la moyenne des représentations vectorielles de mots.

La figure ci-dessus montre la mise en forme pour alimenter les algorithmes d'apprentissage classiques en utilisant le plongement de mot. Nous nous intéressons à quelques méthodes de plongements de l'état de l'art.

3.2.1 Glove

Glove est un algorithme d'apprentissage non supervisé pour la représentation vectorielle des mots. L'entraînement se fait une matrice statistique X de co-occurrence de mots provenant d'un corpus où les entrées X_{ij} représente le nombre de fois que le mot j apparaît dans le contexte du mot i.

Finalement, le but est de calculer $P_{ij} = P(j|i) = X_{ij}/X_i$ la probabilité que le mot j apparaît dans le contexte du mot i, où $X_i = \sum_k X_{ik}$. Les détails de cet algorithme sont décrits dans [3].

Nous avons la possibilité d'entraîner le modèle [3] pour la représentation vectorielle des mots ou alors d'utiliser les vecteurs de mots pré-entrainés [4]

3.2.2 DistilBert

C'est l'une des méthodes les plus récentes et performantes dans le plongement des mots. C'est une version distribué de **BERT** : plus soft. Elle s'appuie sur des modèles pré-entrainés pour les vecteurs de mots. DistilBert est disponible dans le module ***transformers*** [5] de huggingface.
Dans le cadre de notre projet, cette méthode ne nous a pas donné de meilleurs résultats comparée aux autres méthodes de plongements et même de bag of words. Ceci serait dû certainement d'une part à la limitation du contenu de chaque document à 1500 caractères, mais aussi de l'étape de pré-traitement qui a énormément dépourvu le corpus de son sens juste à un ensemble de mots.

4 Modélisation

Les précédentes représentations sont utilisés dans nos algorithmes afin de pouvoir effectuer différentes tâches de classification à la fois supervisée pour la prédiction des étiquettes(XGBoost, CatBoost, LightGBM, MultinomialNB, Faiss) et non supervisée pour modéliser des sujets(LDA).

4.1 Approche supervisée

Dans cette partie, nous abordons l'algorithme XGBoost pour sa meilleure exactitude(accuracy score) et les modèles Faiss et MultinomialNB pour leurs temps d'exécutions faibles avec une exactitude parfois comparable.

XGBoost(Extreme Gradient Boosting) : la tâche est d'entraîner le modèle(arbre de décision) à trouver les meilleurs paramètres θ qui s'adapte le mieux aux données d'entraînement x_i et les labels y_i. Pour entraîner le modèle, nous avons besoin de définir une fonction objective pour mesurer l'apprentissage au mieux des données d'entraînement. Cette fonction est constituée de deux parties : ***fonction de perte*** et le terme de ***régularisation***.

$$obj(\theta) = L(\theta) + \Omega(\theta)$$

La fonction de perte utilisée est la perte logistique :

$$L(\theta) = \sum_i [y_i ln(1 + e^{-\hat{y}_i}) + (1 - y_i) ln(1 + e^{\hat{y}_i}))]$$

Faiss[2] est une bibliothèque pour la recherche de similarité efficace et le regroupement de vecteurs denses. Les algorithmes utilisés sont implémentés sur le GPU. Il est développé par Facebook AI Research.

On recherche un ensemble de vecteurs de distances euclidiennes minimales.

$$i = argmin_i \|x - x_i\|, \text{ où } \|.\| \text{ est la norme L2}$$

MultinomialNB pour ***Multinomial Naive Bayes*** est un classifier convenable pour les features discrets. La distribution multinomiale nécessite des valeurs positives. Cependant en pratique fonctionne avec la représentation tf-idf ; cela est confirmé dans notre projet. Néanmoins, ne marchera pas avec les plongements des mots.

$$\hat{y} = arg\max_y P(y) \prod_{i=1}^{n} P(x_i|y),$$
où nous pouvons utiliser le ***Maximum A posterio***(MAP) pour estimer $P(y) et P(x_i|y)$.

Pour plus de détails pour la démonstration, voir ici .

3. Classe gensim.models.Word2Vec
4. Vecteur de mots téléchargeable à https ://nlp.stanford.edu/projects/glove/
5. https ://github.com/huggingface/transformers

4.2 Approche non supervisée

Nous avons utilisée la méthode nommée ***LDA(Latent Dirichlet Allocation)***[1] qui est une méthode non-supervisée générative dont la figure suivante donne un aperçu général.

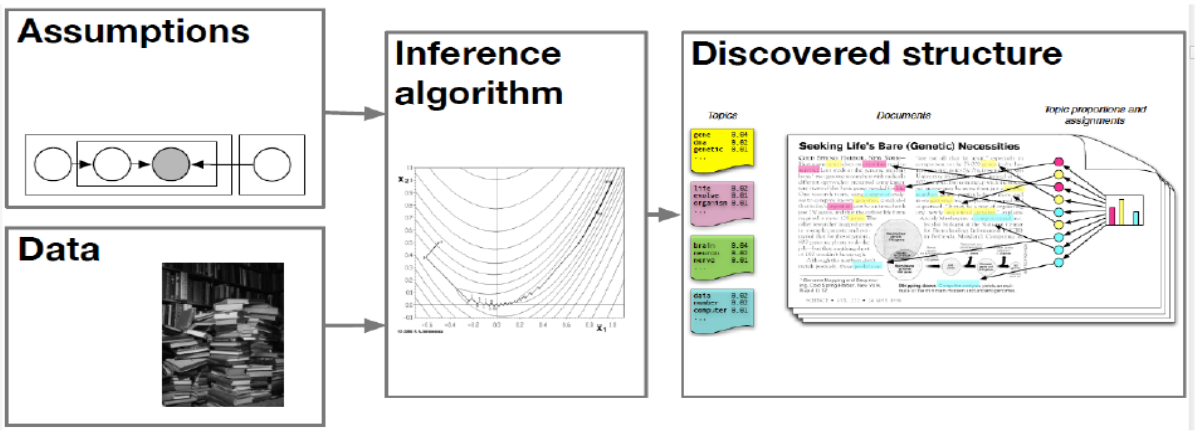

FIGURE 3 – Processus d'inférence de l'algorithme LDA

Hypothèses(exemples) :

- 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 $p(\theta_m)$
- Chaque mot possède une distribution associée à chaque thème $p(\varphi_k)$. On peut ainsi représenter chaque thème par une probabilité sur chaque mot.
- z_n représente le thème du mot w_n

Données : l'ensemble des documents à notre disposition.

Inférence : déterminer les thèmes, les distributions de chaque mot sur les thèmes, la fréquence d'apparition de chaque thème sur le corpus.

Pour effectuer l'inférence de ce modèle, nous avons utilisé la librairie LDA de ***gensim*** [6]. D'autres librairies existent comme celle de ***scikit-learn***.

```
from gensim.models.ldamodel import LdaModel
from gensim.corpora.dictionary import Dictionary
```

```
corpus = ['Voici le premier doc', 'Voici le deuxieme doc', 'Voilà le troixième']
docs_bow = [doc.split() for doc in corpus]
dictionary = Dictionary(data)
corpus_lda = [dictionary.doc2bow(doc_bow) for doc_bow in docs_bow ]
```

```
n_topics = 20
ldaModel = LdaModel(corpus_lda, num_topics=n_topics, id2word=dictionary)
```

Plusieurs évaluations du modèle LDA existe pour déterminer la valeur de l'hyper paramètre n_topics(nombre de sujet).

- L'évaluation quantitative telle que la ***perplexité*** qui mesure la capacité du modèle probabiliste à généraliser.
- L'évaluation qualitative telle que la ***PMI*** qui évalue automatique la cohérence du modèle basé sur le corpus d'entrée.

Cette dernière est l'évaluation utilisée et présente dans la bibliothèque gensim(gensim.models.CoherenceModel)

6. https ://radimrehurek.com/gensim/models/ldamodel.html

5 Évaluation

Il s'agit ici pour nous de présenter la métrique utilisée ainsi que de décrire le processus d'évaluation pour nos algorithmes multi-classes.
La plupart des données à notre disposition contenaient 1 à 5 étiquettes. Nous avons donc décidé d'évaluer l'exactitude(accuracy) de chaque modèle sur la base des 3 premières étiquettes(médiane).

Algorithme 1 own metric

Entrées: classifier, X_test, y_true
Sorties: accuracy

count = 0
pour chaque entrée de notre donnée de test **faire**
  Récupérer les 3 meilleurs labels
  Si la première étiquette trouvée correspond à la vraie étiquette **alors**
    on ajoute 1 à la variable count
  Fin si
  Si la deuxième étiquette trouvée correspond à la vraie étiquette **alors**
    on ajoute 0.5 à la variable count
  Fin si
  Si la troixième étiquette trouvée correspond à la vraie étiquette **alors**
    on ajoute 0.25 à la variable count
  Fin si
fin pour
Retourner count/nombre d'entrée(X_test)

FIGURE 4 – Métrique d'évaluation propre

Dans la suite, On intègre cette métrique à nos différents classifiers [7] ; Pour celà, il va créer nos estimateurs grâce au template suivant :

```
from sklearn.base import BaseEstimator, ClassifierMixin
```

```
class TemplateClassifier(BaseEstimator, ClassifierMixin):

    def fit(self, X, y):

    def predict(self, X):

    def predict_proba(self, X):

    def set_params(self, **parameters):  # important for fine tuning

    def score(self, X, y):
        return own_metric(self, X, y)
```

La mise au point ou ***fine tuning*** de nos différents classifiers est réalisée avec l'implémentation ***RandomizedSearchCV*** [8] de scikit-learn. Il est fortement recommandé d'utiliser des distributions continues pour les paramètres. Comme exemple, nous avons la distribution uniforme(uniform) continue sur $[0, 1]$ et la distribution uniforme discrète(randint) de ***scipy.stats***.

7. https ://scikit-learn.org/stable/developers/develop.html#rolling-your-own-estimator
8. https ://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

6 Déploiement et gestion de code

Le déploiement s'est fait en local avec ***Flask*** qui est un framework d'application web. L'api répond à l'adresse **127.0.0.1 :5000/predict** à la méthode POST. les champs doivent porter les noms(name) respectivement **'title'** et **'body'** pour le titre et le corps du post.
Lors de la création de l'api, plusieurs objets sont intégrés à savoir le dictionnaire et modèle LDA entrainés, extracteur de feature(CountVectoizer) et le modèle de prédiction d'étiquettes entraînés.

Avant toutes prédictions, on retrouve l'étape de pré-traitement de la donnée qui respecte l'ensemble des traitements d'avant entraînement.

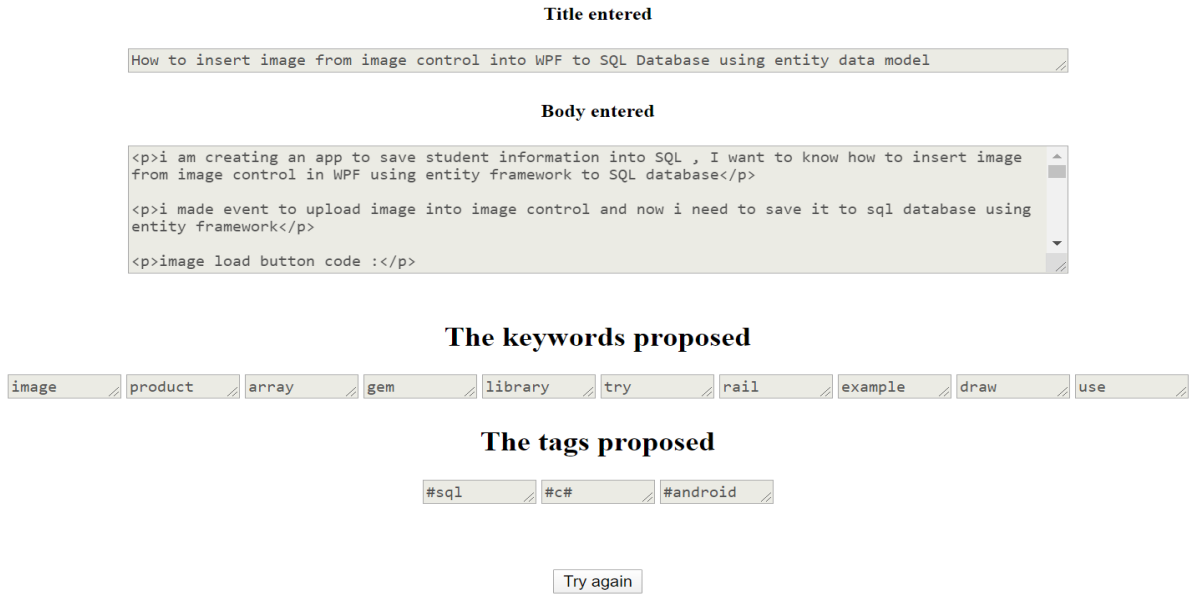

FIGURE 5 – Capture d'écran de prédiction du système développé

L'outil utilisé pour gérer le code est **Git**. Nous l'avons utilisé le triplet (**Google Colab**, **Google Drive**, **Github**). Nous avons d'abord cloné le code disponible sur github [9] dans mon drive google à l'emplacement 'Colab Notebooks' comme suit :

```
from google.colab import drive
drive.mount('/content/drive')
```

```
%cd "drive/My Drive/Colab Notebooks"  # move where you want
```

```
!git clone https://github.com/AnthonyNama/Categoriser-automatiquement-des-questions.git
```

Après la modification de fichiers, la liste des commandes courantes est utilisée pour la mise à jour du dépôt sur Github.

```
!git status  # Effected changes
```

```
!git add -a  # Add all files changes
```

```
!git commit -m "message"  # Commit local changes with message
```

```
!git push origin master  # Update repository github
```

9. https://github.com/AnthonyNama/Categoriser-automatiquement-des-questions

7 Conclusion

Les données textuelles disponibles grandissantes permettent de développer et d'intégrer de nouveaux systèmes NLP. Les systèmes développés intègrent progressivement les niveaux linguistiques tels que sémantique et pragmatique. Ce rapport a pour but de présenter les briques de construction des applications NLP.

Dans ce document, après une brève vue d'ensemble du NLP, nous avons évoqué tour à tour les processus de pré-traitement, d'extraction de features, les algorithmes de modélisation, l'évaluation et le déploiement. Nous pouvons dire qu'il y'a des étapes plus critiques que d'autres qui doivent faire l'objet d'une attention particulière à savoir : le prétraitement, apprentissage des features et la métrique d'évaluation.

De Nombreux pré-traitements existent pour raffiner nos données textuelles. Néanmoins, un traitement dépend de fortement du problème à traiter. De plus, l'ordre d'utilisation de ces traitements est primordial.

L'apprentissage des features dépend directement de l'étape précédente(pré-traitement). Le choix de la représentation utilisée dépend tout aussi des pré-traitements réalisés. Ainsi, si le sens des phrases est réduit à des mots après pré-traitement, se pencher plus sur le comptage de mots sinon vers le plongement de mots.

La métrique utilisée est comparable à une boussole pour avoir une connaissance sur les performances de notre modèle. Il est primordial de savoir choisir voir construire la métrique explicative au problème posé.

De nombreux axes d'exploration manquent à l'appel pour améliorer le modèle final dont nous invoquons entre autres :

1. Manipuler les 'emoji' : **import emoji**
2. Ajouter les bigrammes ou trigrammes à la représentation Bag of words pour améliorer les modèles.
3. Améliorer le modèle avec des features linguistiques(POS, NER, etc.)
4. Utiliser la méthode SVD pour le word embedding.
5. Utiliser l'algorithme NMF pour l'extraction de sujets.

Références

[1] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, pages 601–608. MIT Press, 2002.

[2] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv :1702.08734*, 2017.

[3] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove : Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[4] A ; Baruzzo A ; Tasso C Pudota, N ; Dattolo. A new domain independent keyphrase extraction system. *Digital Libraries, CCIS 2010, 91 : 67-78*, 2010.

[5] A. Schwartz and M. Hearst. A simple algorithm for identifying abbreviations definitions in biomedical text. *Biocomputing*, pages 451–462, 2003.