Système de suggestion de tags

Alexandre BENARD
18/01/2022

Table des matières

Abstract

Il s'agit de mettre en place un système de suggestion de tags pour les questions posées sur le site stackoverflow.com. La démarche consiste à récolter les données à partir de l'outil d'export du site, de les trier et les analyser, entraîner les données et les intégrer à une API.

1/ Récolte des données

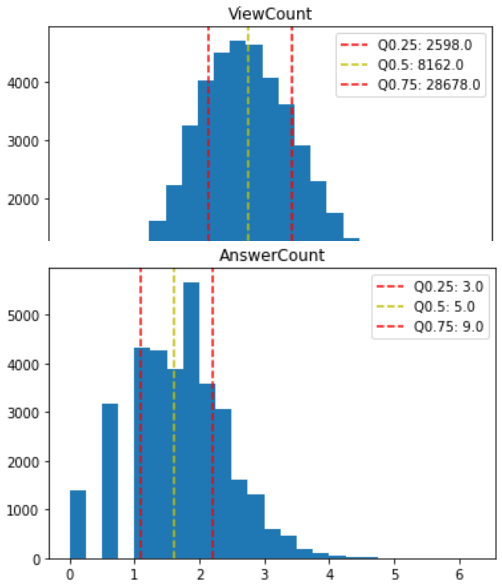
Les données sont fournies par l'outil d'export de données du site stackoverflow.com à cette adresse : https://data.stackexchange.com/stackoverflow/query/new

Parmi toutes les données, les plus pertinentes pour l'objectif proviennent des tables « Post » et « TagSynonyms ». 500 000 individus sont téléchargés à partir de « Posts » et tous ceux de « TagSynonyms », c'est-à-dire

Détail des données téléchargées :

- Table « Posts » :
  - 338 395 individus
  - 23 features :
    - `'Id', 'PostTypeId', 'AcceptedAnswerId', 'ParentId', 'CreationDate', 'DeletionDate', 'Score', 'ViewCount', 'Body', 'OwnerUserId', 'OwnerDisplayName', 'LastEditorUserId', 'LastEditorDisplayName', 'LastEditDate', 'LastActivityDate', 'Title', 'Tags', 'AnswerCount', 'CommentCount', 'FavoriteCount', 'ClosedDate', 'CommunityOwnedDate', 'ContentLicense'`
  - Les features pertinentes pour le projet sont :
    - 'Id', 'Score', 'ViewCount', 'Body', 'Title', 'Tags', 'AnswerCount' et 'FavoriteCount'.
- Table « TagsSynonyms » :
  - 5 365 individus
  - 10 fetures :
    - `'Id', 'SourceTagName', 'TargetTagName', 'CreationDate', 'OwnerUserId', 'AutoRenameCount', 'LastAutoRename', 'Score', 'ApprovedByUserId', 'ApprovalDate'`
  - Les features pertinentes pour le projet sont :
    - 'SourceTagName' et 'TargetTagName'.

2/ Tri et mise en forme des données.



Il s'agit de créer un sous-jeu de données pour chaque table contenant uniquement les features pertinentes (sous-jeux « Posts » contenant 29 101 individus et « TagSynonyms » contenant 5 365 individus).

La suite est effectuée uniquement sur le sous-jeu de données créé à partir de la table « Posts ».

Seuls les individus contenant des tags nous intéressent. On filtre donc le jeu de données pour garder uniquement ceux-ci. Ensuite, la feature 'Body' contient des balises HTML. Ces dernières sont retirées à l'aide de la librairie python BeautyfulSoup et de sa méthode get_text(). Il est maintenant possible de réunir les features 'Title' et 'Body' transformée ensemble et dans cet ordre. Nous nommerons la feature issue de cette réunion 'full_posts'. On peut continuer le tri des individus en ne gardant que ceux dont ayant mis en favori au moins une fois et dont le score est supérieur à zéro (respectivement 'FavoriteCount' > 0 et 'Score' > 0). Les features 'ViewCount' et 'AnswerCount' semblent avoir des valeurs distribuées plus aléatoirement et sur une échelle plus grande. Après, l'affichage d'un histogramme mis au logarithme pour chaque feature, le nombre limite inférieur de vue est fixé à 2 598 (fig. ViewCount) et le nombre de réponses au post à 3 (fig. AnswerCount).

3/ Formatage du texte et des tags

- Texte
  - Utilisation de la librairie python re pour substituer les points de fin de phrase, les suites de plus de deux « . », le caractère « + » par un espace ainsi que les contractions « 's » par « is » et « n't » par « not ».
  - Utilisation de la librairie python nltk et de sa méthode RegexpTokenizer(regex_form).tokenize() pour séparer les mots et les mettre en forme ;
    Avec : `regex_form = r'[a-zA-Z0-9]+\.{1}[a-zA-Z0-9]+|[a-zA-Z]+\#?\+*|[a-zA-Z0-9]+\-*[a-zA-Z0-9]*|\.?[a-zA-Z]+'`
  - Retrait des stopwords anglais fournis par nltk.corpus.stopwords(words('english'))
  - Transformation des mots par leur tag associé grâce au jeu « TagSynonyms ».
  - Transformation des mots et verbes sous leur forme canonique avec la méthode nltk.stem.WordNetLemmatizer.lemmatize(w, 'x') avec 'x' = 'v' pour les verbes et 'x' = 'n' pour les noms.
- Tags
  - Les tags sont écrits entre des chevrons. Il s'agit donc de remplacer les chevrons par des espaces grâce au regex.

4/ Analyse des posts

Pour commencer, il faut définir les mots les plus pertinents du corpus. La librairie gensim convient à cet effet, plus particulièrement la méthode gensim.corpora.Dictionary(). Nous gardons les termes apparaissant plus de vingt fois et dans moins de 50% des éléments du corpus (figure ci-dessous).

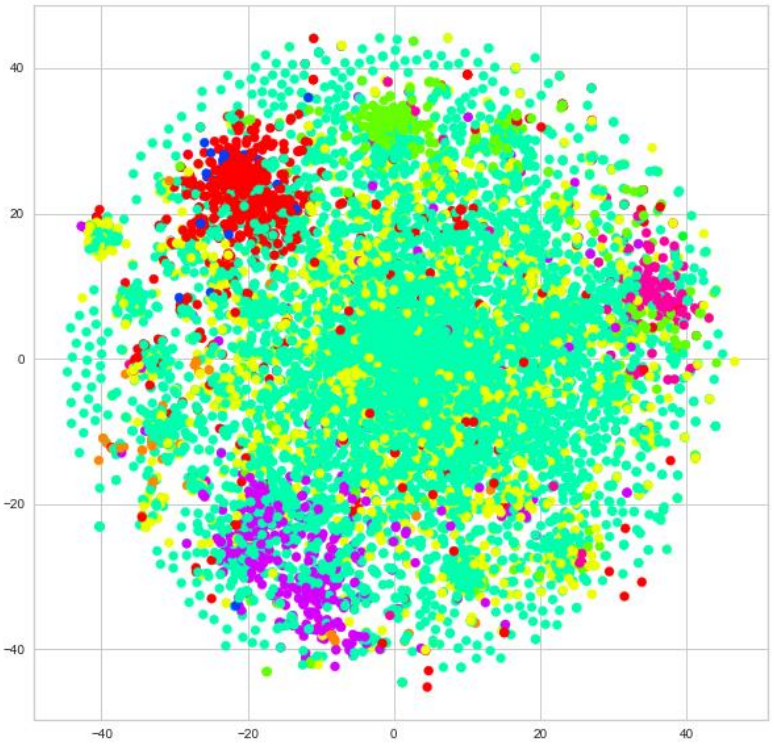
```
from gensim.corpora import Dictionary

docs = datas["stem_words"].apply(lambda x: x.split())

dictionary = Dictionary(docs)
dictionary.filter_extremes(no_below= 20, no_above= 0.5)
```

Il en découle 4005 mots uniques dont le plus fréquent est « file » avec 14557 apparitions.

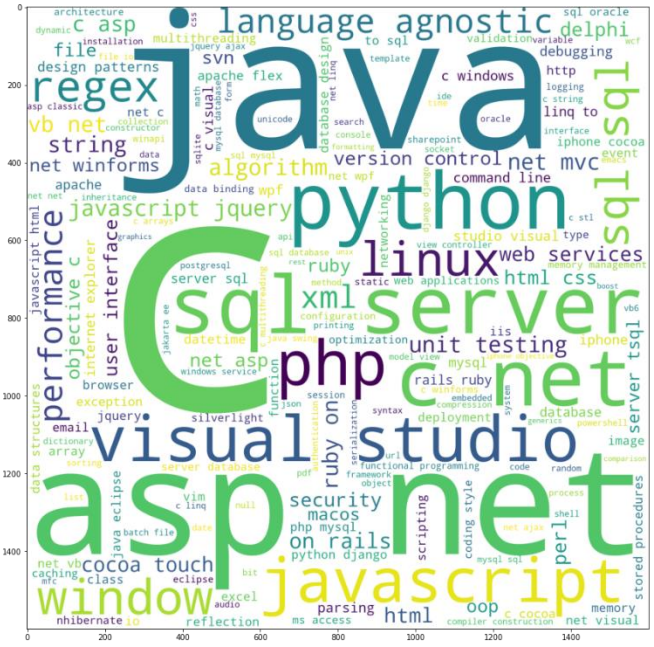
L'utilisation du module wordcloud permet d'afficher un nuage de mots pour une représentation plus visuelle (figure ci-dessous).

Il s'agit maintenant de mettre en évidence d'éventuels clusters de posts grâce à leurs mots. Pour cela, il faut commencer par effectuer un bag of words [1] avec la class CountVectorizer (CV) de scikit-learn (sklearn) suivi d'une réduction de donnée avec la class TruncatedSVD (TSVD) de sklearn. Fixer le paramètre 'n_components' de TSVD à 750 composantes permet d'expliquer 87.6% du jeu initial. On effectue ensuite un KMeans sur les données réduites pour trouver les clusters (12 clusters). Un TSNE est utilisé pour réduire le bag of words en deux dimensions et les clusters sont utilisés pour colorer le graphique. Nous pouvons voir qu'aucun cluster se démarque réellement des autres (cf figure suivante). Néanmoins, on obtient un silhouette score valant 0.138, ce qui signifie que les points sont dans les bons clusters mais très proches des frontières de ces derniers.

[1] Un bag of words est une représentation sous forme de tableau des mots présents dans un corpus de textes. Concrètement, les lignes sont les textes et les colonnes correspondent aux mots dans le corpus. Il y a autant de colonnes que de mots présents. Pour remplir ce tableau, il suffit de compter le nombre d'apparition d'un mot dans le texte et de noter ce nombre dans la colonne correspondante.

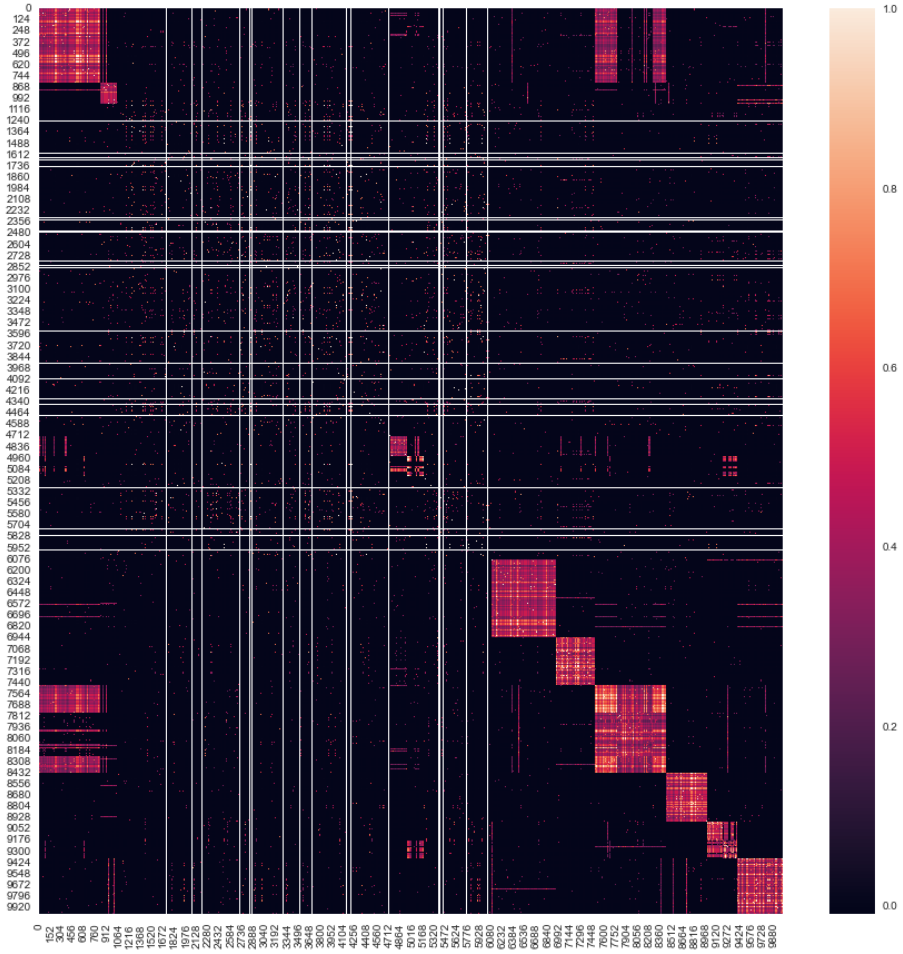


5/ Analyse des tags



La même démarche que précédemment est utilisée pour l'analyse des tags. On obtient 4272 tags uniques dont « c# » est celui qui est le plus présent avec 1 379 apparitions. Le nuage de tags met clairement en avant les tags « C », « Java », « ASP » et « Net » (figure ci-contre).

Une évaluation brute de tags par la méthode de Spearman donne un rho valant 0.021 donc aucune corrélation. La réduction de données, cette fois par un PCA, est expliquée à 82.8% avec 200 composantes. Le KMeans met en avant 9 clusters. L'affichage des clusters après le TSNE est plus clair que pour les posts. D'ailleurs, le silhouette score vaut 0.154. Cette fois, les individus sont triés par clusters avant d'effectuer la corrélation de Spearman. On peut clairement voir que les individus classés par tags de même cluster sont plus corrélés ensemble qu'avec les autres, carrés plus clairs (cf figure ci-dessous).

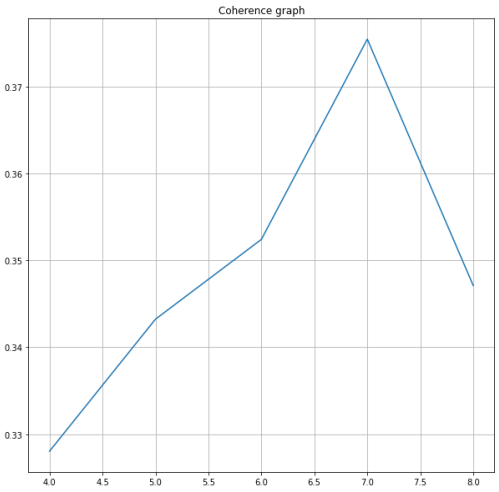


6/ Modélisations

a/ Latent Dirichlet Allocation (LDA)

Pour résumer, LDA imagine un ensemble fixe de thèmes (aussi appelé sujets ou topics). Chaque sujet représente un ensemble de mots. Et l'objectif de LDA est de mapper tous les documents sur les sujets de manière à ce que les mots de chaque document soient principalement capturés par ces sujets imaginaires.

Le processus se déroule en plusieurs étapes :



a. Créer un bag of words du corpus avec gensim.corpora.Dictionary.doc2bow()
b. Créer une matrice des valeurs Tf-idf du corpus avec gensim.models.TfidfModel
c. Définir le nombre de topics optimal avec gensim.models.LdaModel (LDAModel) comme algorithme d'entraînement et gensim.models.coherencemodel.CoherenceModel (CM) pour le score. CM croit jusqu'à former un premier pic. Le nombre optimal de topics à passer en paramètre dans LDAModel est donc le nombre de topics associé à ce pic (fig : Coherence graph).

Processus de prédiction des tags :

a. Récupération des 50 mots les plus importants pour chaque topic (TWW_n) issu du LDAModel et des poids des topics pour chaque post (TPW).

b. Créer d'un dictionnaire ayant pour clé chaque mot du post et en valeur une somme pour tous les mots apparaissant strictement plus de une fois dans le post. Cette somme est calculée de la manière suivante :

$$\sum_{i=0}^{n} TWW_{i,w} * TPW_i$$

Avec $\mathrm{i} = \mathrm{topic}, \mathrm{w} = \mathrm{mot}$.

c. Nous gardons enfin uniquement les mots présents dans le jeu « TagSynonms » et les trions par ordre d'importance descendante.

b/ SGDClassifier

SGDClassifier (SGDC) est un outil qui mêle le Stochastic Gradient Descent (SGD) au SVM, à LogisticRegression et à d'autres algorithmes de regressions. C'est le modèle final choisi pour la prédiction. Les raisons de ce choix sont expliquées dans la partie Processus.

Explications :

Le SGD est une méthode d'optimisation utilisable par de nombreux algorithmes de Machine Learning (ML).

a. Tout d'abord, un gradient est la pente d'une fonction. Il mesure le degré de changement d'une variable en réponse aux changements d'une autre variable. Mathématiquement, le Gradient Descent (GD) est une fonction convexe dont la sortie est la dérivée partielle d'un ensemble de paramètres de ses entrées. Plus le gradient est grand, plus la pente de sortie est raide. Le but est de minimiser la pente de sortie

b. Le terme Stochastic veut dire que le processus est lié à une probabilité aléatoire. Concrètement, alors que le GD prend en compte tout le jeu de données à chaque itération, le SGD prend un échantillon aléatoire parmi le jeu de données.

Processus :

a. Transformation des valeurs observées.
   a. On utilise tout d'abord la class TfidfVectorizer de sklearn sur les posts. Cela permet d'obtenir le poids des mots de chaque individu sous forme de sparse matrix.
   b. Il s'agit ensuite de transformer la matrice en array numpy avec la class TfidfVectorsToArray créée à cet effet.
   c. Ensuite, nous effectuons une réduction de dimensions via la class PCA de sklearn.
   d. Pour chaque individu, il peut exister une ou plusieurs valeurs cibles apparaissant au maximum une fois lorsqu'elle est présente en tant que tag. Nous avons donc affaire à une classification binaire. Nous allons donc utiliser la class OneVsRestClassifier (OVR) de sklearn qui permet de réaliser une classification une à une avec chaque terme cible. La class SGDClassifier (SGDC) de sklearn est passée pour le paramètre estimator de OVR.

b. Transformation des tags cibles en bag of words. Après vérification, il est confirmé que chaque tag 0 ou 1 fois pour chaque individu. Nous faisons donc bien face à une multitude de classifications binaires. L'utilisation de OVR est justifiée.

c. Hyperparamètres de SGDC.
   a. S'agissant d'une classification, l'hyperparamètre 'loss' est choix 'log' (lié à une régression logistique) ou 'modified_huber' (lié au Support Vector Machine).
   b. N'ayant aucune indication, nous testons les valeurs 'None' et 'balanced' pour l'hyperparamètre 'class_weight'.
   c. Enfin, nous essayons les trois choix pour l'hyperparamètre 'penalty' concernant la régularisation, c'est-à-dire 'l2', 'l1' et 'elasticnet' et intégrons plusieurs valeurs pour l'hyperparamètre 'alpha' associé.

d. Méthode de scoring Nous effectuons un clf.predict_proba(X_test). Ensuite, nous trions les valeurs de chaque individu du plus petit au plus grand. Enfin, pour chaque individu, nous mettons à 1 tous les résultats du clf.predict_proba supérieure à la différence entre les deux derniers éléments de la matrice issue du tri et le reste à 0. Nous obtenons une matrice de prédiction MP Il s'agit maintenant de faire la moyenne des résultats égaux entre y_test et MP (figure ci-dessus).

```python
def scoring_function(clf, X, y):
    y_pred = clf.predict_proba(X)

    results_preds = np.zeros(y_pred.shape)
    pred_sorted = np.sort(y_pred, axis= 1)
    diff = pred_sorted[:, -1] - pred_sorted[:, -2]
    results_preds = np.where(y_pred >
        diff.reshape(-1, 1), 1, 0)

    return np.mean(y == results_preds)
```

e. GridSearchCV (GCV) Malheureusement, le GridSearchCV ne fonctionne pas pour ce cas particulier à cause de l'hyperparamètre 'CV' automatiquement défini sur 'StratifiedFold' pour lequel il manque des valeurs d'entraînement. Il a donc fallu le recréer. Pour cela, il a suffi de créer une suite de boucles contenant les hyperparamètres. Concernant, la cross-validation, l'utilisation de ShuffleSplit de sklearn a réglé le problème (figure ci-dessus).

```python
ShuffleSplit(n_splits= 3, train_size= 0.75, random_state= 42)
```

Nous obtenons les résultats suivants :

```
{'score': 0.9912492131616596, 'loss': 'modified_huber', 'class_weight':
None, 'penalty': 'l1', 'alpha': 0.0001, 'l1_ratio': 0.15}
```

Le SGDC est entraîné avec les hyperparamètres fournis par l'équivalent de GCV créé pour l'occasion puis le modèle est sauvegardé avec la méthode dump() du module joblib. Afin d'automatiser les transformations des

valeurs d'entrée, un pipeline (c'est-à-dire une suite de fonctions) a été créé (figure ci-dessous).

```
pipeline = Pipeline(steps=[
    ("tfidf", TfidfVectorizer(token_pattern= r'[\S]+')),
    ("toarray", TfidfVectorsToArray()),
    ("PCA", PCA(n_components= 1500)),
    ("clf", OneVsRestClassifier(
        SGDClassifier(loss= "modified_huber",
        class_weight= None, penalty= "l1", alpha= 0.0001,
        l1_ratio= 0.15, early_stopping= True,
        random_state= 42)))
    ])
```

7/ API

Afin de rendre publique l'utilisation de cet outil de prédiction de tags en fonction de la question posée, le modèle est déployé sur une API web. Les parties frontend et backend fonctionnent de concert.

La module Dash a été utilisé pour l'interface du frontend tandis que FastAPI a été utilisé pour le backend.

Le frontend est basiquement divisé en trois parties :

a. Une entrée pour le titre de la question
b. Une entrée pour la question
c. Une sortie affichant les tags retournés par le backend

Le fonctionnement du backend est très simple aussi. Il récupère les entrées, qui ont été concaténées au préalable, via une méthode http GET (c'est-à-dire accessible depuis l'url). Il effectue la prédiction grâce au modèle précédemment sauvegardé puis, renvoie les tags retenus.