Rapport de projet 6:

Catégoriser automatiquement des questions

Réalisé par: **Taher Haggui**

Sommaire

Introduction:

Stack Overflow est un site célèbre de question-réponses liées au développement informatique. Pour poser une question sur ce site, il faut introduire plusieurs tags. Le problème ne se poserait pas pour les utilisateurs expérimentés. Cependant , il vaut mieux élaborer un système de suggestion de tags pour les nouveaux utilisateurs.

Ainsi ce projet, vient pour répondre à ce besoin et développez un système de suggestion de tags pour la communauté de Stack Overflow. En attendant, un système permettant d'assigner plusieurs tags pertinents à une question.

Nous commençons par extraire une base de données ,qui recense un grand nombre des informations authentiques au site d'entraide. Celle-ci est obtenue moyennant des requêtes SQL.

Par la suite vient la phase d'exploration des données, qui consiste à la prétraitement des textes tout en les récupérant , les arranger dans des listes et ne garder que les caractères alphanumériques. Nous supprimons aussi toutes les mots répétitives et non significatifs (stopwords) , nous font aussi la racinisation des mots qui consiste à garder que les racines des mots tout en supprimant leurs suffixes et leur affixes. Nous préparons aussi dans la phase d'exploration des statistiques telles que bag of words , permettant de calculer la fréquence d'utilisation de chaque mot dans chaque texte . Ainsi la statistique TF-IDF permettant de déterminer les mots spécifiques et les plus représentatives de chaque texte.

Nous entamons par la suite une phase de tests qui consiste tout d'abords à utiliser une approche non-supervisé pour déterminer des tags pertinent pour chaque question. Puis nous testons une approche supervisé qui consiste à déterminer des tags moyennant un machine learning entraîné.

Nous déployons aussi des méthodes spécifiques pour évaluer la performance de chacune de ces deux approches.

I. Phase d'exploration:

Dans cette phase , nous faisons toutes les prétraitement nécessaires des textes à savoir : extraction, séparation, garder que les caractères alphanumériques, transformés les textes en minuscules, supprimer les stopwords, la racinasation des mots etc .

Nous préparons aussi des tableaux de statistiques indispensables pour la phase de tests telles que :

la fréquence d'utilisation de chaque mot dans chaque texte, ainsi que la statistique TF-IDF permettant de distinguer les mots les plus représentatifs de chaque texte. Nous détaillons dans la suite, chacune de ces étapes.

1. Extraction des données :

Pour extraire la base de données utiles pour notre travail. Nous nous sommes rendu sur le site stackexchange et nous avons introduit dans un premier temps la requête SQL suivante :

SELECT * FROM Posts WHERE id <5000

Cette requête, nous a permis d'extraire une base de données avec toutes les variables. Ceci, nous a permis de tester l'utilité de chacune des colonnes dégagées et de sélectionner celles , qui sont utiles pour notre travail. En fait, nous avons trouvé que uniquement les colonnes id, Body et Tags sont les variables les plus pertinents. Ainsi , nous avons affiner notre requête comme suit :

SELECT Id,Body,Tags FROM Posts WHERE id <50000

2. **Nettoyage de textes :**

Cette étape, permet de faire les prétraitement requises pour préparer nos textes. Toutes ces traitement seront détaillées dans la suite.

i. **Décomposition des textes :**

On va utiliser la fonction **tokenize** de la librairie **NLTK** pour décomposer les textes html en tableaux de mots afin de pouvoir effectuer des opérations dessus. Néanmoins, celle-ci permet de conserver les ponctuations, les apostrophes etc. Ceci nous posera un petit problème parce que c'est uniquement les mots qui nous intéressent.

Pour remédier à cela, nous avons utilisé les fameuses **expressions régulières**, qui nous ont permis de garder que les caractères alphanumériques.

Un autre problème se rajoute, c'est que certains mots ont des caractères en majuscule car ils apparaissent au début de phrases. Celui-ci nous a conduit à utiliser la fonction **lower()** de la classe **str** à la récupération de texte.

ii. Racinisation:

La racinisation(ou stemming en anglais), consiste à ne conserver que la racine des mots étudiés. Pour réaliser celui-ci, nous utilisons la fonction **stem** de la librairie **EnglishStemmer** puisqu'il s'agit d'un vocabulaire en anglais. Notons que celle-ci figure de la librairie **NLTK** dans son package **stem.snowbal**l.

iii. Stopwords:

Les Stopwords sont les mots très courants dans la langue ("and", "this ","in"… en anglais) ou dans le corpus étudiés. Ce sont donc les mots qui n'apportent pas de valeur informative pour la compréhension du sens d'un corpus. Nous devions donc supprimer ces mots pour ne garder que les variables informatives.

Pour identifier les Stopwords, nous avons passé par deux étapes à savoir :

-Identifier les mots les plus courants dans le corpus .Pour faire ceci, nous avons calculé la fréquence d'utilisation de chaque mot dans chaque texte. Ainsi nous avons tiré ces fréquences sur toutes les textes moyennant le compteur **Counter()** de la librairie **NLTK**. Après, nous

avons sélectionné les 100 mots les plus utilisés à travers la fonction **most_common()** de compteur Counter().Nous avons supposé par la suite, les mots sélectionnés comme Stopwords**.**

-Nous avons utilisé aussi la liste des Stopwords, prédéfinies par défaut dans NLTK pour les ajouter à la liste des mots, considérés comme stopwords. La liste prédéfinies par NLTK pour la langue anglaise est accessible à travers le module **corpus.stopwords.words('english')** de la librairie NLTK.

La superposition de ces deux listes, nous a permis de définir la liste des mots à considérer comme Stopwords et à les supprimer de nos textes.

3. Bag of words :

Le Bag of words, est la manière la plus simple de représenter un document ou un texte . Celui-ci prends en considération tous les mots utilisés par un texte ou un document sans soucis de contexte (ordre, utilisation etc).

Une représentation bag-of-words classique sera donc celle dans laquelle on représente chaque document par un vecteur de taille de vocabulaire et on utilisera la matrice composée de l'ensemble de ces N documents qui forment le corpus comme entrée de nos algorithmes.En pratique,ça peut être par exemple un vecteur de fréquence d'apparition des différents mots utilisés .

Pour notre cas, nous avons utilisé la librairie **scikit-learn** et plus précisément la fonction **CountVectorizer()** pour établir le bags-of-words ou la fréquence d'utilisation de chaque mot par chaque texte.

Notons que la fonction **CountVectorizer()**,compte plusieurs paramètres important pour affiner les résultats, que nous voulons avoir. Nous trouvons par exemple des paramètres indiquant la fréquence minimum d'apparition des mots à prendre en considération dans un document .Nous pouvons aussi renseigner le pourcentage maximum d'apparition des mots dans tous les documents étudiés. Nous trouvons aussi, des paramètres pour indiquer comment traiter les majuscules ainsi d'autres pour indiquer s'il y a des stopwords à prendre en considération et plein d'autres paramètres très utiles.

Nous devons aussi noter que, CountVectorizer() rendre un résultat sous forme d'un **matrix** lorsque, nous utilisons sa fonction **transform()**.Mais nous avons bien utilisé un résultat sous

forme un **dataframe**, tout en transformant tout d'abords le résultat obtenue à un **array**, moyennant la fonction **matrix.toarray()**.Puis la fonction **get_feature_names()** de CountVectorizer() ,qui permet d'extraire les noms de mots (features) considérés. Ainsi, nous avons fournis notre résultat sous la forme d'un **dataframe**.

4. **TF-IDF :**

Le TF-IDF est une métrique statistique permet d'évaluer l'importance d'un terme contenu dans un document, relativement à une collection ou un corpus. Le poids augmente proportionnellement au nombre d'occurrences du mot dans le document. Il varie également en fonction de la fréquence du mot dans le corpus.

En résumé, le poids s'écrit alors :

poids=fréquence du terme×indicateur similarité

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).

Nous calculons donc le poids tf-idf final par:

poids=fréquence du terme ×idf

Pour calculer le TF-IDF de notre corpus, nous avons fait recours à la librairie **scikit-learn** et nous avons utilisé la fonction **TfidfVectorizer**. Nous avons renseigné ses paramètres, identiques à celles de Countvectorizer. Puis nous avons fait les même transformations, effectuées dans le cas de Bags of words pour rendre notre résultat sous forme d'un dataframe et d'utiliser les **features**, rendus par la fonction **get_feature_names()** de **TfidfVectorizer.**

5. Target data:

Pour déployer l'approche supervisée.Nous avons préparé les étiquettes de données appropriées à chaque texte. Pour ce faire, nous avons utilisé les tags de la colonne « Tags » comme **features**. Ainsi, nous avons établie à l'aide de **CountVectorizer(),** les tags utilisé par chaque texte avec leurs fréquences d'apparition.

II. Phase de tests :

Ici, nous allons déployer deux approches de recommandations de tags. Une approche non supervisée et une supervisée.Ceci tout en tirant profit des préparatifs de la phase d'exploration de données.Donc, nous allons commencer par la méthode non supervisée, que nous allons juger sa performance par la pertinence de ses tags . Nous testons,aussi l'approche supervisée qui permet de prédire les tags d'une texte à l'aide des données entraînées.

1. Méthode non supervisée :

i. Déploiement de méthode :

Pour déployer l'approche non supervisée,nous utilisons les statistiques bag of words et la fonction **LatentDirichletAllocation ()** de **sklearn** , qui va nous permettre de dégager les **topics** les plus utilisés dans notre jeu de données. Nous avons choisi d'identifier un nombre de topics égal à 250 . Nous avons opté pour un nombre de topics qui fait l'arbitrage entre l'optimisation de la performance de méthode et le temps de calcul.

Après avoir entraîner la fonction **LatentDirichletAllocation**, par les statistiques bags of words de notre jeu de données. Nous établissons la représentation de chaque observation par la distribution des topics. Ceci moyennant la fonction **transform()** de **LatentDirichletAllocation**.

Nous choisissons par la suite, les trois Topics les plus représentatifs de chaque observation. Ceci tout en triant, les topics de chaque observation par leur degré de représentativité de chaque texte. La fonction **argsort()**, permet de faire le triage ascendant, qui nous permettra par la suite de choisir les trois derniers topics les plus représentatifs.

Après avoir déterminer les topics représentatifs de chaque observation.Nous avons codé manuellement une fonction, permettant d'identifier les deux mots les plus utilisés de chaque topic. Elle renvoie par la suite ces mots sous forme de tags. Nous appliquons cette fonction à l'ensemble des lignes de notre jeu de données.Notons que notre fonction utilise l'attribut

components_ de **LatentDirichletAllocation(),** qui permet de donner la fréquence d'utilisation de chaque mot par chaque topic.

ii. Performance de méthode non supervisée:

Nous évaluerons la performance de la méthode non supervisée par deux approches :

Approche 1:

Dans cette approche, nous avons cherché de déterminer le pourcentage des tags prévus (tags de chaque question) par rapport au nombre total des tags retournés par la méthode non supervisée. Pour ce faire, nous avons calculé manuellement le score de cette performance. Nous avons trouvé un score de performance de 3.7%. Ceci est équivalente à dire que 3.7 % des tags retournés par cette méthode sont déja utilisé comme tags pour chaque question.

Approche 2:

Nous considérons chaque tags retournée par cette méthode, comme pertinent, s'il fait partie de l'ensemble des tags mentionnées dans la base de données initiale. Donc, nous mesurons la performance de cette méthode par le nombre de tags pertinents en comparaison avec le nombre total des tags, retournées par la méthode non supervisée.

Pour implémenter cette mesure, nous avons créé manuellement une fonction, permettant de calculer le nombre de tags pertinents. Nous avons divisé par la suite ce nombre par le nombre total des tags. Nous avons aboutit à un score de performance de 55 %.

2. Méthode supervisée:

i. Déploiement de méthode :

Pour déployer,l'approche supervisée . Nous avons divisé nos données en données apprentissage et données tests. Nous avons utilisé la fonction train_test_split() de sklearn, ainsi qu'un percentage de 30 % de données d'apprentissage. Notons, que nous avons utilisé les statistiques tf-idf comme données d'entraînement et les données target comme étiquettes de données. Nous avons opté pour la méthode RandomForestClassifier() pour déployer l'approche supervisée. Nous avons utilisé cette méthode avec ses paramètres par défaut.

ii. Performance de méthode supervisée :

Pour mesurer la performance de la méthode supervisée, nous comparons les prédictions de données de testes par cette méthode avec les tags réelles de données de testes. Ainsi, le score de cette méthode , revient à déterminer le pourcentage des tags ,qui correspond aux tags réelles. Cette mesure, nous a donné un score de performance de 0.01 % .

III. API:

Pour illustrer notre travail , nous avons créé une API permettant de mettre en oeuvre l'approche non supervisée. Cette API, prend en entrée le texte d'une question et elle renvoie les tags suggérer par la méthode non supervisée.

En pratique, nous avons créé tout d'abords deux pages html . Une page d'accueil de notre application et une page résultat. Nous avons aussi utilisé le module **Pickle** de python pour pouvoir sauvegarder dans un premier temps ,les objets que nous avons utilisé pour mettre en oeuvre la méthode non supervisée dans la phase de teste, dans un fichier tiers . Dans un second temps, nous avons utilisé ce module pour importer ces objets dans le code de notre API et éviter leur création de nouveau. Notre application, apporte le même prétraitement que celui fait dans notre phase d'exploration de données. Elle utilise par la suite, la méthode **LDA** entrainée sur notre jeu de données pour présenter la distribution de représentation du texte introduit par les différents Topics déjà identifiées par la méthode **LDA**. Enfin , notre API propose 6 nouveaux mots clés qui correspond aux deux mots les plus utilisés par les 3 topics les plus représentatif de notre texte.

Conclusion:

Après avoir tester les deux approches supervisées et non supervisées, nous avons mesuré la performance de ces méthodes par la comparaison de leurs résultats aux tags réelles .Nous avons trouvé que l'approche non supervisée permet de retrouver plus de 3% des tags réelles, tant dis que celle supervisée ne permet pas de retrouver que moins de 0.1% . Ajoutons que

l’approche non supervisée retourne des nouveaux mots clés dont leur pertinence est évaluée à plus de 50% . Ainsi , nous pouvons conclure que l’approche non supervisée permet de suggérer des nouveaux mots clés pertinent . Alors qu’il est trivialement montré que nous ne pouvons pas prédire les tags d’une texte par l'entraînement d’une méthode de machine learning sur un jeu de données supervisées. Nous présentons dans le tableau ci-dessous les résultats de performance de chaque méthode.

Tableau 1: Performances des méthodes

	Tags réels	Pertinence des tags
Approche non supervisée	3.7%	55%
Approche supervisée	0.01%	-