Project Stackoverflow

Mise en place d'un système de tags automatiques des questions de stackoverflow

Efkan TUREDI
03/11/2021

Introduction

Dans ce projet, nous allons construire un système de tags automatisés qui permettra à l'utilisateur de demander des tags pour un texte qu'il soumettra à une API.

Les objectifs d'apprentissage du projet

Nous aborderons ce projet avec les objectifs suivants:

- Mettre en œuvre une approche non supervisée.
- Utiliser une approche supervisée ou non pour extraire des tags à partir des résultats précédents.
- Comparer ses résultats à une approche purement supervisée, après avoir appliqué des méthodes d'extraction de features spécifiques des données textuelles.
- Mettre en place une méthode d'évaluation propre, avec une séparation du jeu de données pour l'évaluation.
- Pour suivre les modifications du code final à déployer, utiliser un logiciel de gestion de versions, par exemple Git.

Table des matières

Récupération des données

Nous devons récupérer nos données sur le site data.stackexchange.com permettant de lancer des requêtes SQL pour récupérer une base de données de questions.

Toutefois, cet outil a des limites. En effet, stackexchange limite le nombre de réponses à 50,000 réponses par requête. Dans le cadre d'une analyse de traitement de langage, ce nombre est insuffisant.

Nous avons décidé de télécharger les questions par lots de 50,000 questions. Ainsi, il sera possible de constituer des BDD plus larges en concaténant ces séries de questions.

Nous avons fait le choix de garder uniquement les questions les plus "populaires" afin de réduire le bruit possiblement plus important apporté par des questions peu pertinentes. Cela est rendu possible directement par les requêtes SQL, comme montré ci-dessous:

```
SELECT TOP(50000) Id, CreationDate, Score, ViewCount, AnswerCount, CommentCour
FROM Posts
WHERE CreationDate BETWEEN CONVERT(datetime, '2020-01-01') AND CONVERT(datetir
AND Score IS NOT NULL
AND ViewCount IS NOT NULL
AND AnswerCount IS NOT NULL
AND CommentCount IS NOT NULL
AND FavoriteCount IS NOT NULL
ORDER BY ViewCount DESC
```

Notre méthodologie a consisté à récupérer les 50,000 questions de chacun des 4 derniers semestres pour les fusionner et avoir notre BDD. Notre prochaine étape consistera à nettoyer et rendre utilisable notre BDD.

Nettoyage des données

Dans ce projet, nous concentrons nos efforts sur 3 colonnes: Title, Body et Tags, qui nécessitent chacune un nettoyage spécifique.

Title	Body	Tags
Error message "DevTools failed to load SourceM...	<p>I'm trying to display an image selected fro...	<javascript><html>
When adding a JavaScript library, Chrome compl...	My code \n<pre class="lang-html pretty...	<google-chrome-devtools>
Could not load dynamic library 'cudart64_101.d...	<p>I just installed the latest version of Tens...	<python><python-3.x><tensorflow><keras><tensor...
SessionNotCreatedException: Message: session n...	<p>I am currently new to robot framework.I am ...	<selenium><google-chrome><selenium-webdriver><...
error NG6002: Appears in the NgModule.imports ...	<p>First time using firestore and I'm getting ...	<angular><google-cloud-firestore><angularfire>

Nettoyage de la colonne Title

Il va nous falloir tokenizer, nettoyer, et lemmatizer ces données. En plus de cela, nous enlèverons certaines données qui seront source de bruits et apportent peu, comme les ponctuations, les chiffres ou les stopwords. Nous faisons aussi le choix d'enlever les tokens qui ont moins de 3 lettres.

Voici des exemples de fonctions que nous utilisons

```
[ ]  regularizer = lambda x : re.sub("[^a-zA-Z]",           # The pattern to search for
                                " ",                  # The pattern to replace it with
                                x)
```

```
[ ]  db['Title'] = db['Title'].apply(lambda x: regularizer(x))
```

```
[ ]  # This tokenizer, makes tokens, put tokens in lower case and remove punctuations
     db['Title'] = db['Title'].apply(lambda x: tokenizer(x))
```

```
[ ]  db['Title'] = db['Title'].apply(lambda x:stop_words_check(x,stop_words))
```

```
[ ]  def list_lemmatizer(list_to_lem):
         return [lemmatizer.lemmatize(token, 'v') for token in list_to_lem]
```

```
[ ]  db['Title'] = db['Title'].apply(lambda x: list_lemmatizer(x))
```

Nettoyage de la colonne Body

Dans cette partie, nous devrons utiliser le package BeautifulSoup4 pour pouvoir récupérer uniquement la partie qui nous intéresse dans le corpus des messages postés. Nous faisons cela via la méthode get_text() de bs4. Ensuite, les nettoyage que nous faisons sont similaires à ceux de la colonne Title.

```
[ ]  from bs4 import BeautifulSoup
```

```
[ ]  db['Body'] = db['Body'].apply(lambda x: BeautifulSoup(x).get_text())
```

```
[ ]  db['Body'] = db['Body'].apply(lambda x: clean_text(x))
```

```
[ ]  db['Body'][14]
```

```
[ ]  db['Body'] = db['Body'].apply(lambda x: tokenizer(x))
```

```
[ ]  db['Body'] = db['Body'].apply(lambda x: stop_words_check(x,stop_words))
```

```
[ ]  db['Body'] = db['Body'].apply(lambda x: list_lemmatizer(x))
```

```
[ ]  db['Body'] = db['Body'].apply(lambda x: " ".join(x))
```

Nettoyage de la colonne Tags

En plus des similarités avec le nettoyage de la colonne Title, il faudra s'adapter aussi à la spécificité de l'état actuel de la colonne Tags avec ces "< / >" étant présent entre chaque tags. Nous avons écrit des fonctions correspondantes pour cela.

De manière plus importante, on a aussi fait le choix de ne garder que les 500 tags les plus récurrents. Cela nous aidera à réduire le bruit lié aux tags trop rares. Si une observation se retrouve sans tags, nous décidons d'enlever cette observation.

```
[ ]  tags_count = pd.Series(chain(*tag_list)).value_counts()
     tags_top_500 = tags_count.nlargest(500)
     tags_top_500
```

```
python              7049
javascript          4767
java                3132
reactjs             3045
android             2827
                    ...
serialization         44
path                  44
paramiko              43
zsh                   43
google-cloud-run      43
Length: 500, dtype: int64
```

Fusion des colonnes Title et Body

Pour rendre nos analyses plus simples et comparables entre supervisée et non supervisée, nous décidons de fusionner les deux colonnes après leurs transformations respectives.

La colonne Title est très intéressante car lorsque l' utilisateur rentre une question dans le système de stackoverflow, le titre est très important pour l'utilisateur car c'est ce qui va permettre d'identifier le problème de manière concise et claire. Potentiellement, les Titles seront plus "identifiants" que les Body.

C’est pourquoi, il sera intéressant de mettre plus de poids sur Title lors de la fusion. D’autant plus que ceci améliore les performances de nos modèles quelque soit le cas. Nous augmentons donc le poids de la colonne Title en le copiant plusieurs fois dans la colonne “Post”, qui est le nom de la colonne issue de la fusion.

Par la suite, nous travaillerons exclusivement sur la colonne Post.

Analyse Supervisée

Séparation des données

Nous séparons nos données en données de test et d'entraînement, en prenant le soin d’encoder les colonnes Tags via MultiLabelBinariser pour les rendre utilisables dans nos algorithmes.

Nous créons aussi des datasets d'entraînements et de tests pour les approches TF-IDF et Bag of Words, que nous utiliserons dans les analyses supervisées et non supervisées respectivement.

Comparaison des analyses supervisées

Nous lançons plusieurs modèles classiques pour voir quels sont ceux qui sont le plus performants en utilisant Accuracy et le Jaccard Score comme critères principaux pour les évaluer. Il en ressort que SGDClassifier() avec un hinge loss avec régularisation L2 (équivalent à un LInearSVC) sont les algorithmes les plus performants avec 50,000 obbservations.

Toutefois, LinearSVC n’est pas “scalable” car la loss squared_hinge, ne se comporte pas très bien quand nous augmentons notre taille de données à 200,000 observations.

A noter que nous avons utilisé ici la méthode MultiOutputClassifier pour faire tourner nos modèles, plutôt que OneVsRest, mais il est cependant tout à fait possible de faire cela avec OneVsRest en paramétrant correctement.

	Test labels	Supervised predicted labels
0	(java, web-scraping)	()
1	(python, python-3.x)	()
2	(batch-file, cmd, powershell, windows)	(powershell,)
3	(angular, javascript, pdf)	()
4	(android,)	(android,)
...	...	...
38167	(java, project-reactor, rest, spring, spring-w...	(java, spring-webflux)
38168	(plot, python)	(python,)
38169	(docker,)	()
38170	(angular, css)	(angular,)
38171	(material-ui, reactjs, typescript)	(reactjs,)

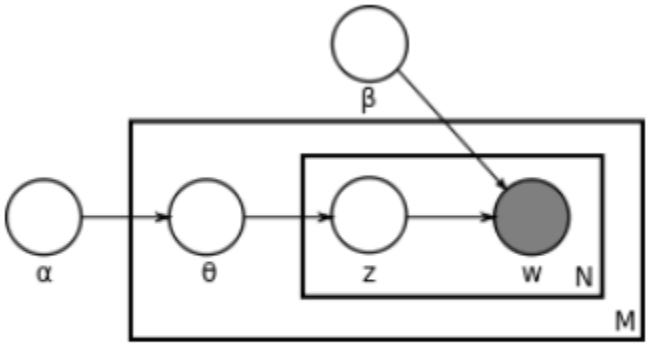
Analyse Non Supervisée

LDA

Dans cette partie nous utilisons l'algorithme de Latent Dirichlet Allocation. Pour faire tourner nos algorithmes, nous utiliserons la structure de dataset en BoW que nous trouvons adaptée.

LDA est un modèle génératif probabiliste. L'algorithme modélise une approche proche d'un clustering. Il permet de regrouper les documents d'un ensemble par k topics (thèmes) et d'associer chaque mot β de chaque document à un des topics. Chaque document est alors composé d'un mélange θ d'un petit nombre de topics.

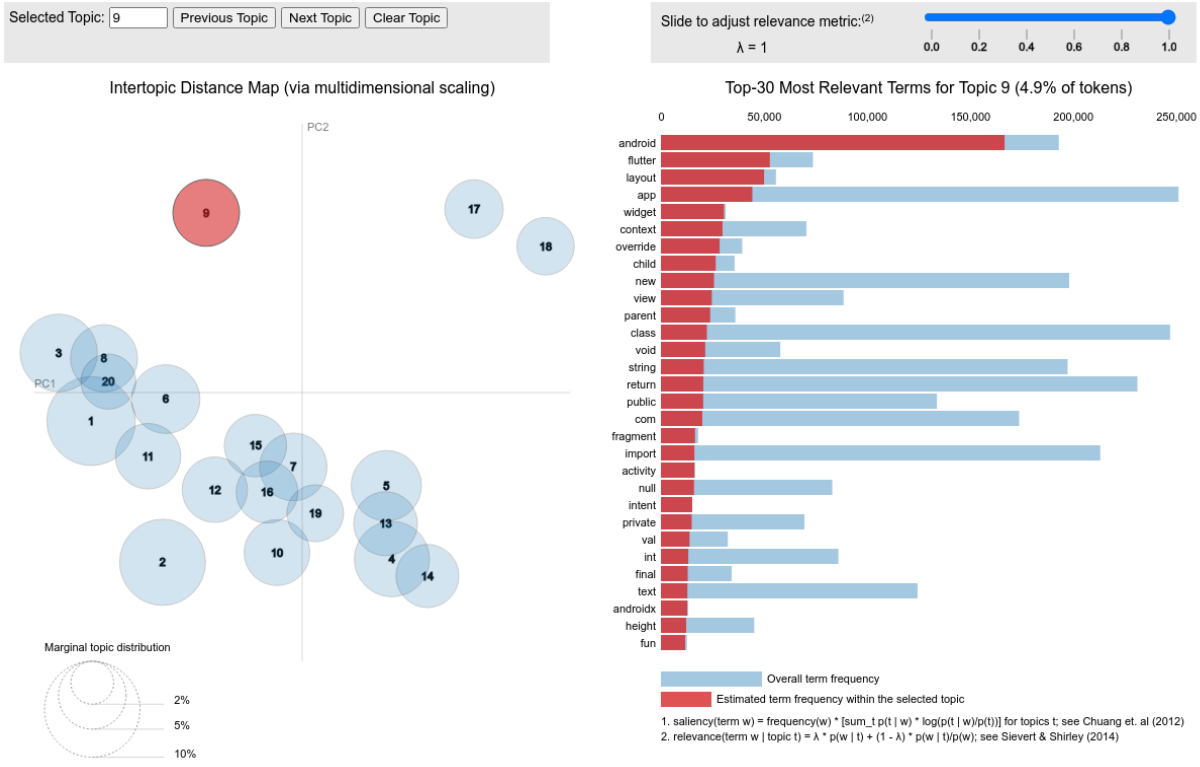
Lors de son initialisation , chaque mot de chaque document est aléatoirement associé à un topic. L'apprentissage consiste à optimiser la probabilité qu'un topic t génère le mot w dans un document d. Voici la représentation graphique du modèle.



Représentation graphique du modèle LDA – source Wikipédia août 2021

Nous faisons le choix d'utiliser l'implémentation de Gensim concernant les algo de LDA, qui nous permettront de bénéficier de la parallélisation du temps de calcul et aussi du fait que cette implémentation est écrite en C++, donc bien plus rapide que du code en pure Python.

Nous utilisons le package pyLDAvis qui est un package spécialement écrit pour pouvoir visualiser les distributions de mots et les topics du corpus de texte qui a été soumis.



Réseau de Neurones avec Keras et Embeddings

Pour utiliser les réseaux de manière rapide, nous décidons de créer un notebook sur Google Colab, afin de tirer profit de sa puissance de calcul des GPU et accélérer les délais d'entraînement.

Nous utiliserons une implémentation de Keras de la méthode des plongements de mots, dont l'intuition est une projection des ensembles de mots dans un espace vectoriel afin d'obtenir des vecteurs denses. Dans ce cas précis, nous utiliserons notre propre plongement de mots avec les couches Embedding, car nous disposons suffisamment de données et nous souhaitons apprendre un plongement de mots spécifique à notre tâche.

Nous n'utilisons pas les deux modèles / algorithmes de word embeddings les plus connus que sont GloVe et Word2vec.

A noter que ceci correspond à notre première expérience avec Keras, donc nous nous baserons largement sur les tutoriels Keras présents en ligne pour la structure de notre réseau de neurones. Voici la structure de notre réseau de Neurones ci-dessous:

```
[75] modelNN = Sequential()
     modelNN.add(Embedding(max_words, 300, input_length=maxlen))
     modelNN.add(Flatten())
     modelNN.add(Dense(2048,activation='relu'))
     modelNN.add(Dropout(0.5))
     modelNN.add(Dense(1024,activation='relu'))
     modelNN.add(Dense(y_encoded.shape[1]))
     modelNN.add(Activation('softmax'))
     modelNN.compile(optimizer='adam', loss='binary_crossentropy', metrics=['categorical_accuracy'])

     modelNN.summary()

     history = modelNN.fit(X_train_embed, y_train_embed,
                           epochs=7,
                           batch_size=1000,
                           validation_data = (X_val_embed,y_val_embed)
                           )
```

Nous obtenons de meilleurs résultats qu'avec les algorithmes de Machine Learning classiques. Notre temps d'entraînement est aussi extrêmement faible comparé aux autres car nous profitons pleinement de la puissance d'un GPU.

Bonus: Comparaison avec les transformers de Hugging Face

Nous décidons d'utiliser les transformers de Hugging Face pour avoir une idée de la valeur ajoutée apportée par l'apprentissage de nos algorithmes. Ils sont fondés sur des mécanismes dits « d'attention » qui ne sont pas nouveaux, mais qui sont ici bien utilisés. Ils offrent aussi des modes d'apprentissage semi-supervisés qui constituent également des atouts majeurs.

Les transformers ont été créés pour faire de la traduction, de la classification ou de la génération de textes. Très vite, on s'est aperçu qu'ils excellaient aussi dans pratiquement toutes les autres tâches de NLP.

En effet, les transformers sont entraînés sur de très grandes bases de données. En l'occurrence, nous utilisons BERT qui a été créé par Google et entraîné sur Wikipedia et BookLibrary.

Nous ne montrons ici que des résultats partiels car l'intérêt de notre projet est de faire de l'apprentissage à partir d'une base de données, mais ces résultats montrent un vrai potentiel d'utilisation intéressant à mieux paramétrer.

	Labels	Predicted_Labels
0	[javascript, html]	[error-handling, for-loop, testing, oop]
1	[python, python-3.x, tensorflow, keras, tensor...	[installation, tensorflow, multiprocessing, go]
2	[selenium, google-chrome, selenium-webdriver, ...	[exception, error-handling, colors, google-chr...
3	[angular, google-cloud-firestore]	[exception, error-handling, import, oop]
4	[java, intellij-idea]	[error-handling, java, testing, express]
5	[javascript, reactjs]	[error-handling, testing, dom, module]
6	[javascript, reactjs, redux, visual-studio-cod...	[error-handling, exception, testing, string]
7	[ios, xcode]	[iphone, mobile, ios, networking]
8	[java, spring-boot]	[error-handling, java, class, groovy]
9	[android, android-studio, kotlin]	[gradle, error-handling, kotlin, performance]

Notre choix se portera donc sur le modèle de réseau de neurones qui délivre des résultats supérieurs

Déploiement avec MLflow et Streamlit

Nous avons choisi MLflow pour la sérialisation et l'exportation du modèle que nous avons choisi. Concernant le choix du dashboard, nous avons opté pour la solution simple proposée par Streamlit.

En lancant mlflow puis streamlit nous sommes capables de faire des requêtes avec localhost.