Projet 6 – Catégorisez automatiquement des questions

Parcours Data Scientist

Julien Alvarez

22 Août 2019

Table des matières

Table des illustrations

1 Présentation du projet

Le site Stack Overflow est un forum d'entraide dédié aux développeurs informatique. Ses utilisateurs soumettent les problèmes qu'ils rencontrent, et répondent aux questions des autres utilisateurs.

Afin de faciliter les recherches thématiques, le soumissionnaire d'une question a la possibilité d'associer des tags à son post.

L'objectif de ce projet est de développer un système de suggestion de tag pour le site. Il prendra la forme d'un algorithme de machine learning.

Les données d'entrée seront issues d'un outil d'export de données mis à disposition par Stackoverflow: "stackexchange explorer".

Le travail mis en œuvre devra par conséquent collecter les données utiles au projet, en observant l'impact de variations dans les critères choisis pour les générer.

Après les étapes d'exploration et de préparation des données, il pourra être intéressant de comparer les résultats issus d'approches supervisées et non supervisées.

Pour finir, nous mettrons en place une API qui permettra la saisie d'un post (Titre et corps du message), et renverra un ou plusieurs labels prédits.

2 Prétraitement

2.1 Analyse des données

On procède en premier lieu à une analyse temporelle des tags. En effet, les langages de développement évoluent avec le temps. Leur nombre d'utilisateur également. On peut donc supposer que les catégories des posts évoluent avec le temps. Un moyen simple de le vérifier est de réaliser plusieurs requêtes afin d'obtenir les posts par période de 6 mois (par exemple) et de comparer le classement des tags les plus fréquents sur chaque période. On peut constater que les tags évoluent en effet avec le temps : leur fréquence varie avec les évolutions technologiques.

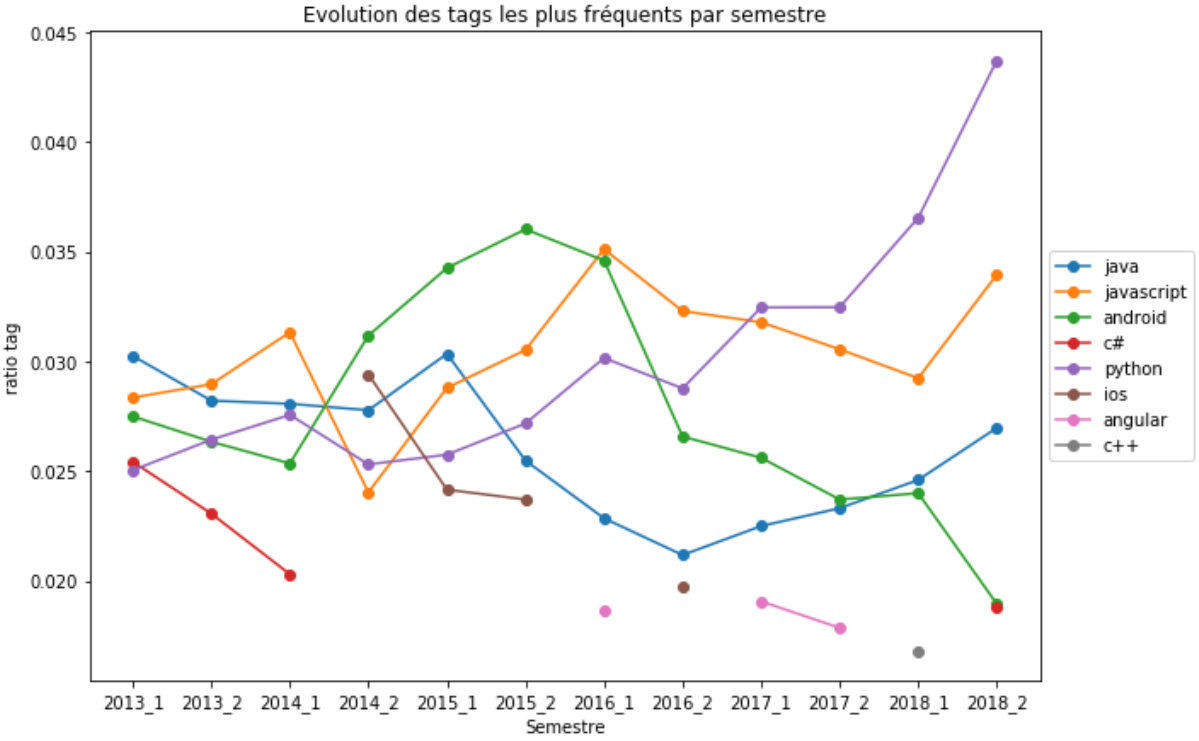

FIGURE 1- EVOLUTION DES TAGS LES PLUS FRÉQUENTS PAR SEMESTRE

Le conséquence d'une telle observation est qu'il apparait judicieux de travailler avec des données récentes : les labels qui seront utilisés pour entraîner notre modèle, et donc susceptibles d'être prédits, correspondront aux technologies susceptibles d'être utilisées par le développeur qui soumet sa question.

2.2 Prétraitements

Il est nécessaire de réaliser des prétraitements sur les documents d'entrée, avant de pouvoir effectuer une modélisation. On va dans un premier temps supprimer les éléments de formatage html, à l'aide de la librairie BeautifulSoup, puis convertir tous les caractères en minuscules (afin de ne pas différencier deux mots identiques si un des deux contient une majuscule, par exemple).

Comme il se présente lui-même, le site dont sont issus les documents étudiés est dédié aux développeurs. Les posts contiennent par conséquent des termes spécifiques tels que C++, .net,… qu'il convient de conserver afin de ne pas perdre d'information. Nous avons donc utilisé un dictionnaire afin de remplacer ces termes par des mots ainsi conservés.

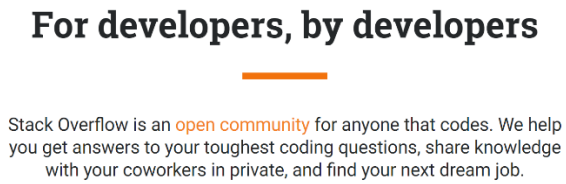

FIGURE 2- PRÉSENTATION DU SITE STACK OVERFLOW

Une fois ces étapes effectuées, les textes sont tokenisés: ils sont convertis en une liste d'éléments. Les éléments qui forment cette liste sont les mots du document qui ne sont pas des stopwords, auxquels on aura appliqué un algorithme de stemming ou lemming.

Les stopwords sont des mots tellement communs qu'il est inutile de les conserver : ils ne permettent pas de distinguer deux documents.

Le processus de stemming (racinisation) consiste à supprimer les affixes des mots afin de ne conserver que leur racine. On obtient alors une version tronquée des mots. Nous avons donc opté pour la lemmatisation, qui utilise des algorithmes différents afin de convertir les mots en lemme (forme canonique).

2.3 Réduction dimensionnelle

Le jeu de données utilisé sur le projet comprend 20236 posts, et un total de 6550 tags différents. Cela représentera un nombre de classes beaucoup trop important à entrainer. De plus, cela signifie que certains de ces tags apparaissent peu de fois. Or l'intérêt de la démarche est de proposer aux utilisateurs des catégories suffisamment générales pour faciliter leur utilisation dans des filtres de recherche, par exemple.

On réalise donc une sélection : si on trace la répartition du nombre de tags en fonction du nombre de fois où ils apparaissent, on obtient la représentation suivante :

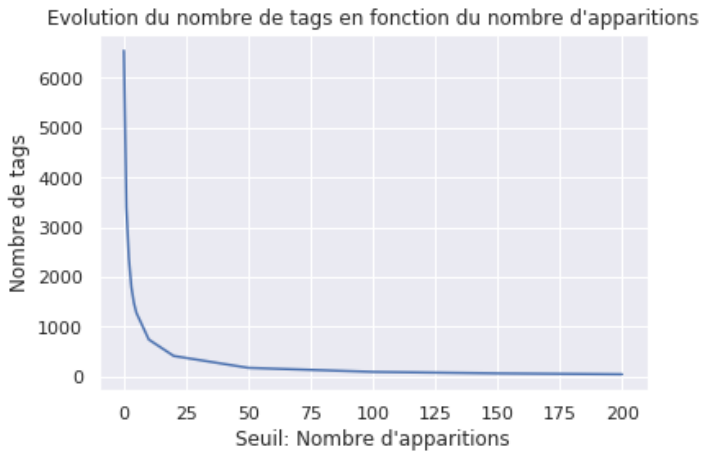

FIGURE 3- DISTRIBUTION DES TAGS

On va déterminer un seuil du nombre d'apparitions (150) qui nous permettra de conserver une soixantaine de tags les plus fréquents. On supprime ensuite de notre jeu de données les posts dont on a supprimé tous les tags lors de cette réduction de dimension. Il nous reste alors un peu plus de 17000 posts.

2.4 Séparation jeux de test et entraînement

On utilisa alors la fonction train_test_split de Scikit-learn afin de séparer les jeux de test et d'entrainement :

```
X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=True, test_size=0.2)
```

On obtient quatre jeux de données: X_train et y_train pour l'entrainement; X_test et y_test pour la mesure de performance.

2.5 Représentation en "Bag of words"

Pour pouvoir réaliser une modélisation sur des documents, il est nécessaire de les vectoriser. Nous allons pour cela les représenter en bag of words, en appliquant les étapes suivantes : tokenisation, comptage, et normalisation. La première étape étant déjà réalisée, nous allons convertir les documents en vecteurs (non normalisés à ce stade) qui comptent les occurrences de chaque mot pour chaque document.

CountVectorizer convertit une collection de documents en matrice de comptage des mots (tokens).

```
# Initialize the "CountVectorizer" object
vectorizer = CountVectorizer(analyzer = "word",
                             tokenizer = None,
                             preprocessor = None,
                             stop_words = None,
                             max_df = 0.5,
                             max_features = 1000
                            )

X_train_count = vectorizer.fit_transform(X_train)
```

On limite le nombre de mots conservés à 1000 (max_features) afin de limiter la dimension de la matrice obtenue. On remarque que l'on pourrait utiliser plusieurs des fonctions de prétraitement à la place du travail de préparation que nous avons effectué plus haut.

Le résultat obtenu est une matrice creuse (à majorité de 0), car chaque document n'utilise qu'une petite partie de l'ensemble du vocabulaire (corpus) rencontré.

2.6 Term Frequency

La normalisation de la matrice obtenue est effectuée avec, TFidftransformer, qui transforme cette matrice creuse en une représentation tf ou tfidf normalisée.

TF (Term-Frequency) indique la fréquence du mot pour chaque document (nombre d'occurrences de ce token dans le document divisé par le nombre total de mots du document).

TFIDF (Term-Frequency - Inverse Document Frequency) multiplie TF par la fréquence de documents contenant le token donné par rapport à la collection de documents. L'objectif visé par cette pondération est de donner un poids plus important aux termes les moins fréquents dans l'ensemble du corpus, considérés comme plus discriminants. On calcule le logarithme du nombre total de documents divisé par le nombre de documents où le mot apparait.

3 Modélisation

Nous avons utilisé deux types d'algorithmes : supervisés (SVC et Random forest), et non supervisé (LDA).

3.1 Algorithmes supervisés

3.1.1 SVC multi-label

Nous implémentons une stratégie One-vs-the-rest avec un algorithme SVC. Cela consiste à entrainer un classifieur f_k pour chaque label. Pour chaque classifieur, le label est alors entrainé contre tous les autres labels.

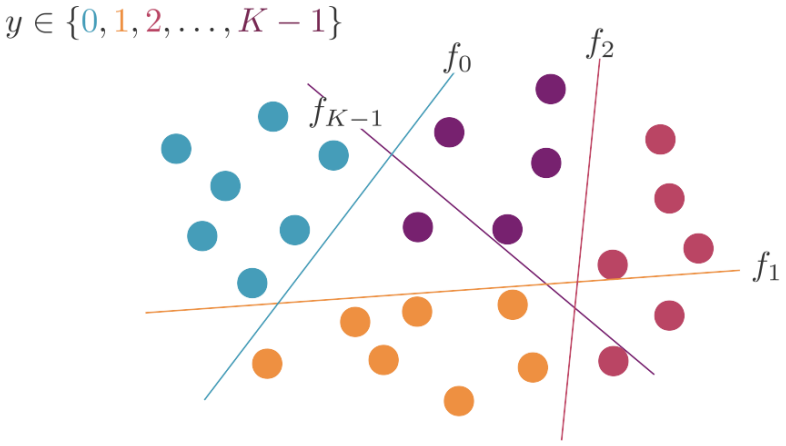

FIGURE 4- ONE-VS-REST

Chaque classifieur SVC (Support Vector Classifier) va alors déterminer l'hyperplan séparateur qui maximise la marge avec les vecteurs de la classe entrainée (One) et ceux des autres classes (Rest).

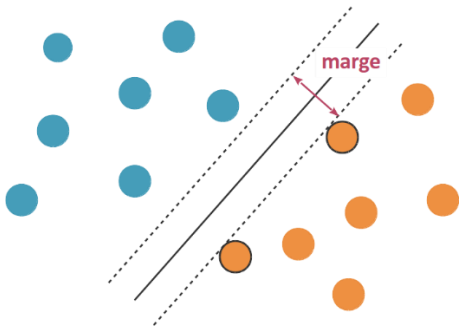

FIGURE 5- MAXIMISATION DE MARGE

Nous utilisons GridSearch afin de régulariser le modèle, et optimiser ses hyperparamètres.

```
SVC_tf = OneVsRestClassifier(SVC(random_state=0))
```

```
gs_svc = GridSearchCV(SVC_tf, param_grid = SVC_param, cv=5,
                      scoring = make_scorer(jaccard_score, average= 'samples'),
                      n_jobs = -1)
```

```
gs_svc = gs_svc.fit(X_train_tf, y_train)
```

3.1.2 Random Forest

Nous testons également Randomforest. C'est un méta-estimateur qui entraine plusieurs classifieurs d'arbre de décision (Decision Tree) sur des échantillons de l'ensemble de données.

Chaque arbre de l'ensemble est construit à partir d'un échantillon créé avec un remplacement de l'ensemble d'apprentissage. De plus, lors de la création de chaque nœud d'un arbre, le meilleur attribut provient soit de toutes les features, soit d'un sous-ensemble aléatoire de taille paramétrable.

La combinaison de ces arbres dans le classifieur Random Forest permet de diminuer la variance, et atténuer la tendance à surentrainer des arbres de décision.

```
RF_model = RandomForestClassifier()
gs_rf = GridSearchCV(RF_model, param_grid = RF_param, cv=5,
                     scoring = make_scorer(jaccard_score, average= 'samples'),
                     n_jobs = -1)
gs_rf = gs_rf.fit(X_train_tf, y_train)
```

3.1.3 Algorithme non-supervisé

3.1.4 Latent Dirichlet allocation (LDA)

3.1.4.1 Principe

La LDA est un modèle génératif probabiliste.

Les observations sont les mots collectés dans les documents. Or chaque mot appartient à un thème (ou topic). On peut donc considérer qu'un document est le mélange de plusieurs thèmes, et que les mots qui le composent sont générés selon la probabilité des thèmes du document.

Le principe de fonctionnement est de fixer le nombre de thèmes, puis effectuer l'apprentissage pour déterminer ces thèmes et les mots qui les composent.

3.1.5 Algorithme semi-supervisé

3.1.5.1 Principe

On utilise les résultats de la LDA (non-supervisé), et on va utiliser le cheminement pour les raccrocher aux tags associés aux posts:

Posts ↔ Topics (score) ↔ mots clés (score) ↔ tags du jeu de données

Cela nous permet ensuite, en ajustant les seuils de décision, de pouvoir associer des tags aux posts, et ainsi de pouvoir appliquer la même métrique que pour l'approche supervisée. Ce qui rend les résultats comparables.

3.1.6 Application

Considérons les matrices suivantes :

$$A = Distance\ (Doc - Topics) = \begin{pmatrix} a_{11} & \cdots & a_{1t} \\ \vdots & \ddots & \vdots \\ a_{d1} & \cdots & a_{dt} \end{pmatrix}$$

Avec d = nombre de documents, et t = nombre de topics.

$$B = Distance\ (Topics - Words) = \begin{pmatrix} b_{11} & \cdots & b_{1w} \\ \vdots & \ddots & \vdots \\ b_{t1} & \cdots & ab_{tw} \end{pmatrix}$$

Avec w = le nombre de mots.

Si on applique un seuil de décision à chacune de ces matrices, on obtient deux matrices de 0 et de 1, signifiant respectivement si un document ou un mot appartient ou non à un topic. Le produit de ces matrices, nous donnera ainsi la relation entre les posts et les mots caractéristiques d'un topic.

Reste ensuite, si l'on veut pouvoir utiliser la même métrique que pour les modèles supervisés, et donc comparer les performances, à relier ces mots aux tags du jeu de données. Nous avons par conséquent réorganisé les colonnes de la matrice afin d'obtenir les mêmes que celles du jeu de test.

3.2 Évaluation des résultats

3.2.1 Coefficient de similarité de Jaccard

La classification étant multi-labels, si on calcule un score de performance sur le résultat exact de chaque post, nous ne prendrons pas en compte les prédictions correctes.

Le coefficient de Jaccard mesure la similarité entre des jeu de données, et est défini par la division de la taille de l'intersection par celle de l'union des deux jeux:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

3.2.2 Comparaison

Algorithme	Term. Freq.	Score
SVC	TF	0.55
SVC	TF-IDF	0.56
RF	TF	0.54
RF	TF-IDF	0.54
LDA	-	0.198

FIGURE 6- TABLEAU DE PERFORMANCE DES MODÈLES

Dans les modèles supervisés, on constate que la SVC appliquée à la matrice de coefficients TF-IDF est la plus performante.

Les temps d'entraînement sont cependant beaucoup plus élevés que pour le Random Forest. Cela peut être un critère de choix de modèle.

On peut également s'intéresser à une autre mesure de la performance : la fonction classification_report de Scikit-learn. Celle-ci permet d'obtenir un score pour chaque label. On peut alors remarquer qu'il y a des différences d'apprentissage selon les labels.

	precision	recall	f1-score	support
.net	1.00	0.11	0.20	44
.net-core	0.68	0.41	0.51	46
amazon-web-services	0.89	0.50	0.64	34
android	0.93	0.78	0.85	411
android-gradle	0.00	0.00	0.00	32
android-studio	0.70	0.36	0.47	92
angular	0.95	0.85	0.90	358
angular-cli	0.80	0.53	0.64	30
apache-spark	0.96	0.79	0.87	29
arrays	0.57	0.24	0.33	51
asp.net	0.00	0.00	0.00	36
...	...	...	...	...
vue.js	0.94	0.62	0.75	48
webpack	0.85	0.62	0.72	47
xcode	0.75	0.61	0.67	102
xcode8	0.00	0.00	0.00	38
micro avg	0.86	0.55	0.67	5599
macro avg	0.76	0.45	0.53	5599
weighted avg	0.81	0.55	0.63	5599
samples avg	0.70	0.60	0.62	5599

FIGURE 7- TABLEAU DE PERFORMANCE PAR LABEL

Pour le modèle semi-supervisé basé sur la LDA, nous avons calculé l'indice de Jaccard en fonction des seuils appliqués sur les matrices doc_topic et topic_word. Le maximum atteint 0.198.

Plusieurs pistes sont possibles pour améliorer ces performances. Notamment dans le lien qui est fait entre les topics trouvés par la LDA, et les labels présents dans les données d'entrée. On pourrait par exemple utiliser Word2Vec, qui va permettre de comparer des représentations vectorielles de mots (calcul d''une distance). L'optimisation d'un seuil de correspondance serait ensuite similaire à la démarche que nous avons suivie.

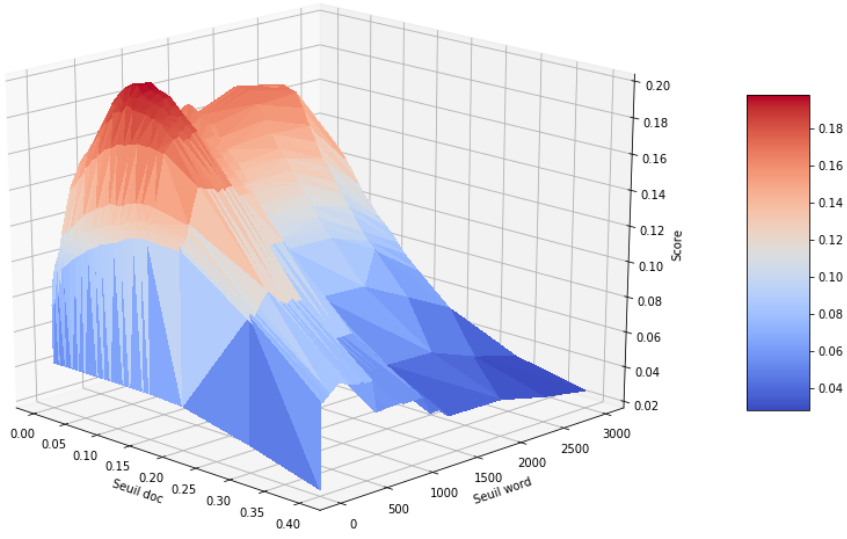

FIGURE 8- PERFORMANCES DU MODÈLE SEMI-SUPERVISÉ EN FONCTION DES SEUILS DE DÉCISION

3.2.3 Modèle conservé

Nous avons choisi d'implémenter la SVC appliquée à la matrice TF-IDF dans notre API, car les performances étaient les meilleures obtenues, malgré un temps d'entraînement plus long.

4 Mise en production

Le modèle est finalement mis en production au moyen d'une API. Pour ce faire, nous avons utilisé le framework Flask, puis déployée sur Heroku:

https://oc-jalvarez-p6.herokuapp.com/

Le code de l'API est disponible sur Github à l'adresse :

https://github.com/jalva80/OC_DS_P6

Les éléments principaux sont:

- Un formulaire WTForms intégré dans une page html;
- Un fichier Utils.py des fonctions utilisées;
- Un fichier Views.py qui exécute la fonction de prédiction lors de la validation du formulaire et affiche les labels prédits;
- Un repertoire joblib_memmap où sont chargés les objets utilisés dans les différentes étapes de prédiction.

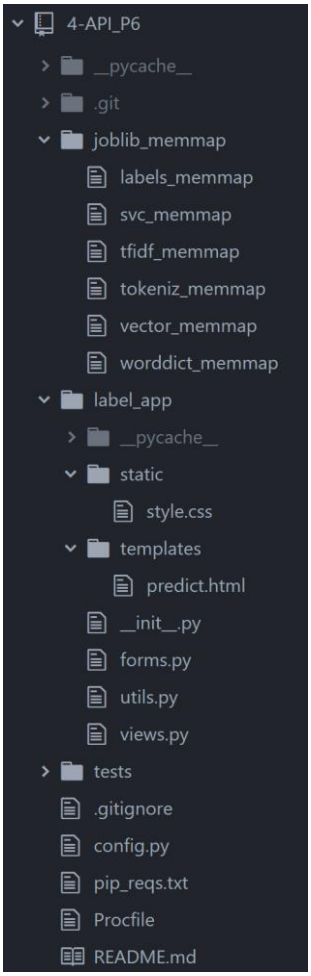

FIGURE 9- ARCHITECTURE DE L'API

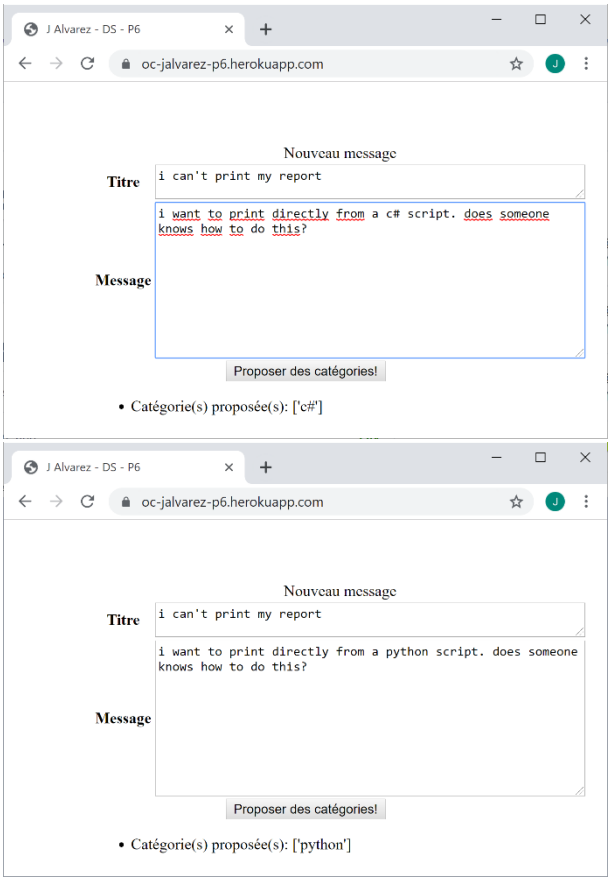

FIGURE 10- EXEMPLES D'UTILISATION DE L'API

5 Conclusion

Transformer des mots saisis sur un clavier en information… La démarche est intéressante et intrigante, et l'exercice d'analyse de texte se révèle tout aussi difficile.

Les résultats sont, dans le cas de ce projet, soumis à des contraintes :

- La qualité des labels donnés par les utilisateurs, et qui servent de référence pour l'entrainement et la mesure de performance (dans le jeu de test)
- Les besoins en ressources machine pour effectuer les calculs, imposant des limitations dans les dimensions du jeu d'entrainement (nombre de features, de posts, etc…) et le paramétrage de la modélisation.
- La spécificité des termes propres aux développeurs, qui implique des traitements spécifiques (termes tels que .net, C, C#, C++ qui ne sont pas des mots à proprement parler, différenciation des révisions, etc…)

Autant de pistes d'amélioration qui laissent espérer qu'il serait possible, en continuant à améliorer les pré-traitements et l'optimisation des modèles, d'obtenir des résultats bien meilleurs.

Et pourtant, il y a déjà une certaine satisfaction à parvenir à reconnaitre une grande part des labels que des utilisateurs ont associés à leur message. Le text mining est sans conteste un domaine attrayant, auquel on prendra sans doute plaisir à se frotter à nouveau.