OpenClassRooms Parcours Data Scientist
Projet 6 :
Catégorisation automatique de questions

Thomas Weber

28 novembre 2018

Table des matières

1 Introduction

Stack Overflow est un site célèbre de questions-réponses liées au développement informatique. Sur ce site, les utilisateurs posent des questions techniques et y associent des tags de manière à pouvoir les classer par thème ou technologie.

L'objectif de ce projet est de développer un système de suggestion de tags. Celui-ci prendra la forme d'un algorithme de machine learning qui assigne automatiquement des tags pertinents à une question. Dans un premier temps nous regarderons les résultats donnés par un modèle supervisé : la régression logistique (nous comparerons aussi ce modèle avec une forêt aléatoire), puis nous testerons un modèle non-supervisé : la LDA.

Avant de pouvoir utiliser les algorithmes de machine learning, il y aura bien sûr toute une étape de préparation des données afin d'extraire les features pertinentes vis-à-vis du problème.

A la fin du projet, on mettra en production sous la forme d'une API les deux modèles.

2 Pré-traitement des données textuelles

2.1 Récupération des données

Stack Overflow propose un outil SQL d'export de données : Stack Exchange Explorer qui nous permet de récupérer un jeu de données rapidement avec la requête suivante :

```
SELECT Id, CreationDate, Body, Title, Tags FROM Posts
WHERE PostTypeId = 1 ORDER BY Rand()
ASC OFFSET 0 ROWS FETCH NEXT 50000 ROWS ONLY
```

La requête récupère 50 000 questions (maximum autorisé par la plateforme) avec le corps de la question, le titre ainsi que les tags associés. Ces questions sont récupérées de manière aléatoire dans le temps avec la fonction Rand() afin d'avoir un échantillon représentatif de tous les tags utilisés (ceux-ci pouvant changer rapidement d'une année sur l'autre avec l'apparation ou la disparation de certaines technologies). Enfin, la condition PostTypeId = 1 dans la requête permet de ne récupérer que les questions et pas les réponses.

2.2 Questions

Pour chaque question le corps du texte et le titre vont subir une série de traitements qui ont pour objectif de passer du texte brut à des features bag-of-words utilisables par les algorithmes. Un bag-of-words est une manière de représenter un texte sous la forme d'une liste des mots qu'il contient, sans soucis du contexte ni de la syntaxe.

Tout d'abord, on retrouve très souvent dans les questions des extraits de code qui sont inclus entre des balises <code>. Ces bouts de code sont difficilement expoitables pour trouver des tags pertinents donc on les supprime, à l'aide de la librairie Beautiful Soup. On supprime aussi toutes les balises HTML, les urls, les chiffres et la ponctuation. Puis on passe le texte en minuscule.

Ensuite on effectue une tokenisation. Cela consiste à séparer tous les mots d'une phrase ou d'un texte et à les transformer en une liste. On utilise la librairie NLTK (Natural Language Toolkit) pour réaliser cette étape.

Les stopwords sont des mots souvent utilisés dans une langue, utiles pour la construction syntaxique mais qui n'apportent pas forcément d'information sur la nature d'un texte. Par exemple, en anglais "be" ou "you". On va les supprimer de notre liste de tokens. Ici on utilise la liste de stopwords en anglais de NLTK ainsi qu'une liste additionnelle faite à la main à partir des observations sur les mots les plus utilisés et qui pourraient être des stopwords.

Enfin la dernière étape est la lemmatisation, qui consiste à ne garder qu'une seule forme pour chaque mot. Par exemple, si un mot est présent au singulier et au pluriel, on ne va garder que la forme au singulier.

2.3 Tags

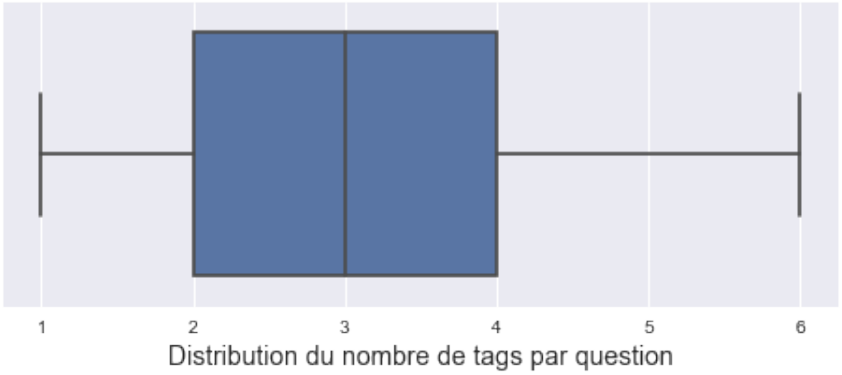
Les tags sont donnés sous la forme d'une chaîne de caractères, séparés les uns des autres par des chevrons :

```
<java><android><android-activity><view><interface>
```

Il faut les séparer pour obtenir une liste de tags qui sera plus facilement exploitable :

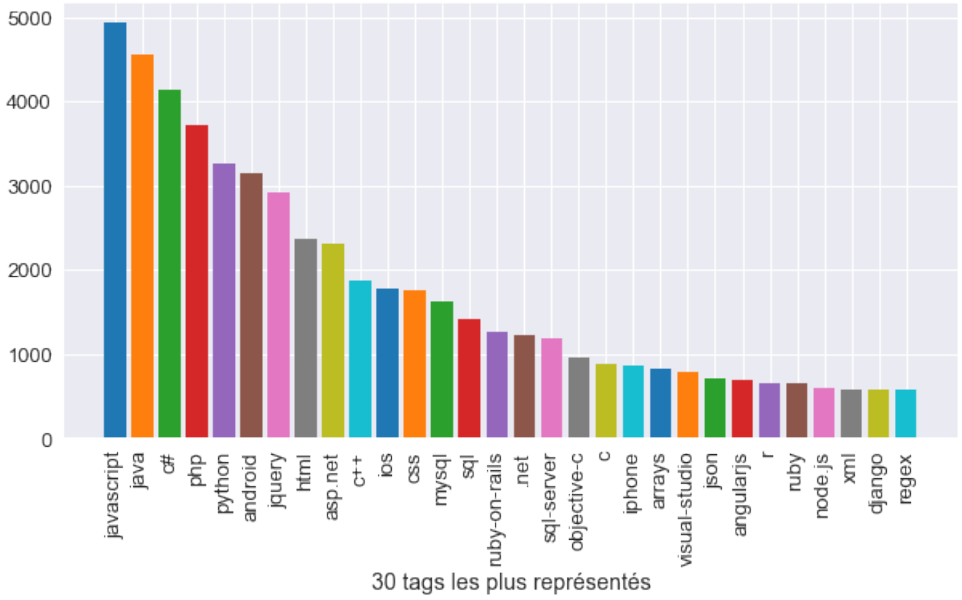
```
[java, android, android-activity, view, interface]
```

Pour chaque question il y a toujours entre 1 et 6 tags.



Distribution du nombre de tags par question

Certains apparaissent trop rarement ou sont trop spécifiques. On ne va donc garder que les tags les plus courants, c'est-à-dire ceux qui apparaissent au moins 150 fois dans le jeu de données. Il reste 114 tags avec le jeu de données utilisé. A titre indicatif, voici les 30 tags les plus utilisés :



3 Préparation des modèles

3.1 Entrées

Pour la régression logistique, on va procéder à une transformation TF-IDF des entrées (Term Frequency - Inverse Document Fréquency). Cette transformation consiste à prendre en compte pour chaque mot son nombre d'occurences dans la question pondéré par l'inverse de sa fréquence d'apparition dans toutes les questions du jeu de données. Cela permet de faire en sorte qu'un mot qui apparait peu souvent dans le jeu de données ait plus de poids qu'un mot qui apparait dans la plupart des questions. Cette pondération est importante pour ce modèle supervisé qui utilise une combinaison linéaire des entrées.

Pour le modèle non-supervisé, on va juste utiliser la TF (Term Frequency) car le modèle utilisé (LDA) a besoin de connaître la distribution des mots pour déterminer des sujets sous-jacents.

Le taille du vocabulaire (nombre de mots) à garder est un paramètre important. En effet, il y a environ 56000 mots dans le jeu de données et les garder tous demanderait trop de ressources pour faire tourner les modèles. Ce paramètre sera déterminé par validation

croisée.

De plus, il y a aussi la possibilité de prendre en compte les co-occurences, c'est-à-dire les groupements de mots ou n-grammes, plutôt qu'uniquement les mots. En effet, certaines combinaisons de deux mots (bi-grammes), trois mots (tri-grammes) ou plus vont réapparaître souvent ensemble et ça peut être intéressant de garder cette information dans les modèles. Ce sera aussi un paramètre qui sera déterminé par validation croisée.

Enfin, le jeu de données est séparé en un jeu d'entraînement (75%) et un jeu de test (25%).

3.2 Sorties

Pour modifier les tags qui sont sous la forme d'une liste on va utiliser la transformation MultiLabelBinarizer de Scikit-learn. Ainsi on va obtenir une colonne par tag et sur chaque ligne (question) un 1 dans la colonne correspondante si le tag est associé à la question ou un 0 sinon.

3.3 Métrique utilisée

Pour mesurer les performances des modèles avec différents paramètres, mais aussi pour comparer les deux modèles, on utilise le score de similarité de Jaccard.

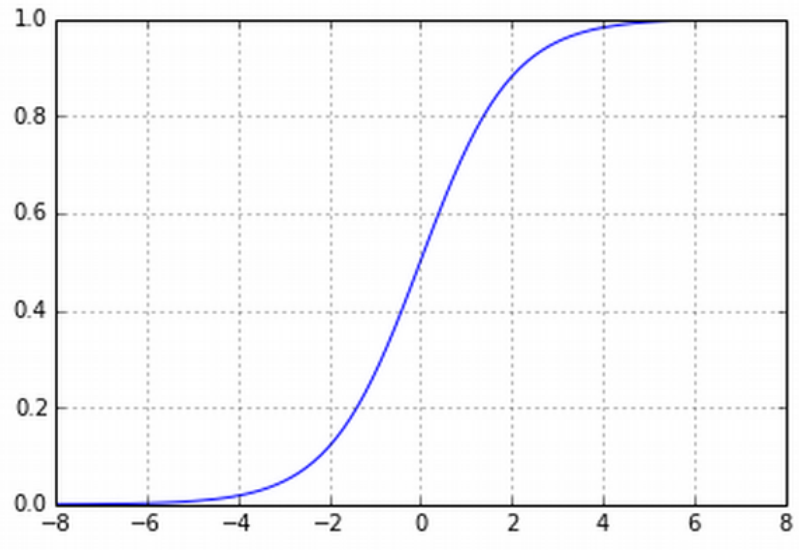
Le score de Jaccard permet de comparer la similarité de deux ensembles, en calculant le cardinal de l'intersection divisé par le cardinal de l'union des deux ensembles.

Ici, les deux ensembles seront la liste des tags prédits par le modèle et la liste des tags réels. Ce score est compris entre 0 et 1 (0 pour des ensembles totalements disjoints et 1 pour des ensembles égaux).

4 Méthode supervisée : Régression logistique

4.1 Présentation du modèle

Le premier modèle testé est le modèle supervisé de la régression logistique. Il s'agit d'un modèle linéaire qui modélise la probabilité qu'une observation appartienne à la classe positive comme la transformation logistique d'une combinaison linéaire des variables : $f(u) = \frac{1}{1+e^{-u}}$



Dans notre cas il s'agit d'un problème de classification multi-label. Il y aura un estimateur pour chaque tag et chaque estimateur va calculer la probabilité qu'une question possède le tag correspondant. Par défaut, la régression logistique considère que le seuil de probabilité pour attribuer un tag est 0.5, seulement en pratique les probabilités dépasse rarement ce seuil donc il a fallu trouver un moyen de pouvoir modifier ce seuil afin que le modèle prédise suffisamment de tags.

4.2 Paramétrage

On vient de le voir, un des paramètres à optimiser sur ce modèle est le seuil de probabilité à partir duquel un tag est associé à une question. Pour ce faire, nous avons modifié la méthode 'predict' de la régression logistique de Scikit-learn afin d'utiliser plutôt la méthode 'predict_proba' puis de lui appliquer un seuil réglable manuellement.

Mais ce n'est pas le seul paramètre à optimiser, il y en a 5 en tout sur lesquels on va s'attarder :

Max_features Le nombre de mots (ou n-grammes) à garder dans les variables d'entrée. Valeurs testées = [1000, 5000, 10000]

Ngram_range Les limites des n-grammes à considérer. Par exemple (1, 2) va garder les mots et les bi-grammes. Valeurs testées = [(1, 1), (1, 2), (1, 3)]

Estimator_penalty La norme de régularisation utilisée. Valeurs testées = ['l1', 'l2']

C Paramètre qui joue sur la force de la régularisation. Valeurs testées = [1, 10, 100]

Threshold Le seuil de probabilité pour lequel la régression logistique considère qu'un tag est associé à une question ou non. Valeurs testées = [0.05, 0.1, 0.2, 0.3]

Afin d'obtenir le modèle avec la meilleure capacité de généralisation et d'éviter le sur-apprentissage, on choisit ces hyper-paramètres par validation croisée.

Les deux premiers sont des paramètres de la TF-IDF alors que les autres sont des paramètres de la régression logistique. Pour pouvoir trouver les paramètres optimaux en une seule validation croisée on utilise un Pipeline qui effectue la TF-IDF et la classification l'un à la suite de l'autre.

Les résultats de la validation croisée donnent comme paramètres optimaux :
max_features = 10000
ngram_range = (1, 3)
estimator_penalty = 'l1'
C = 1
limit = 0.15

4.3 Résultats

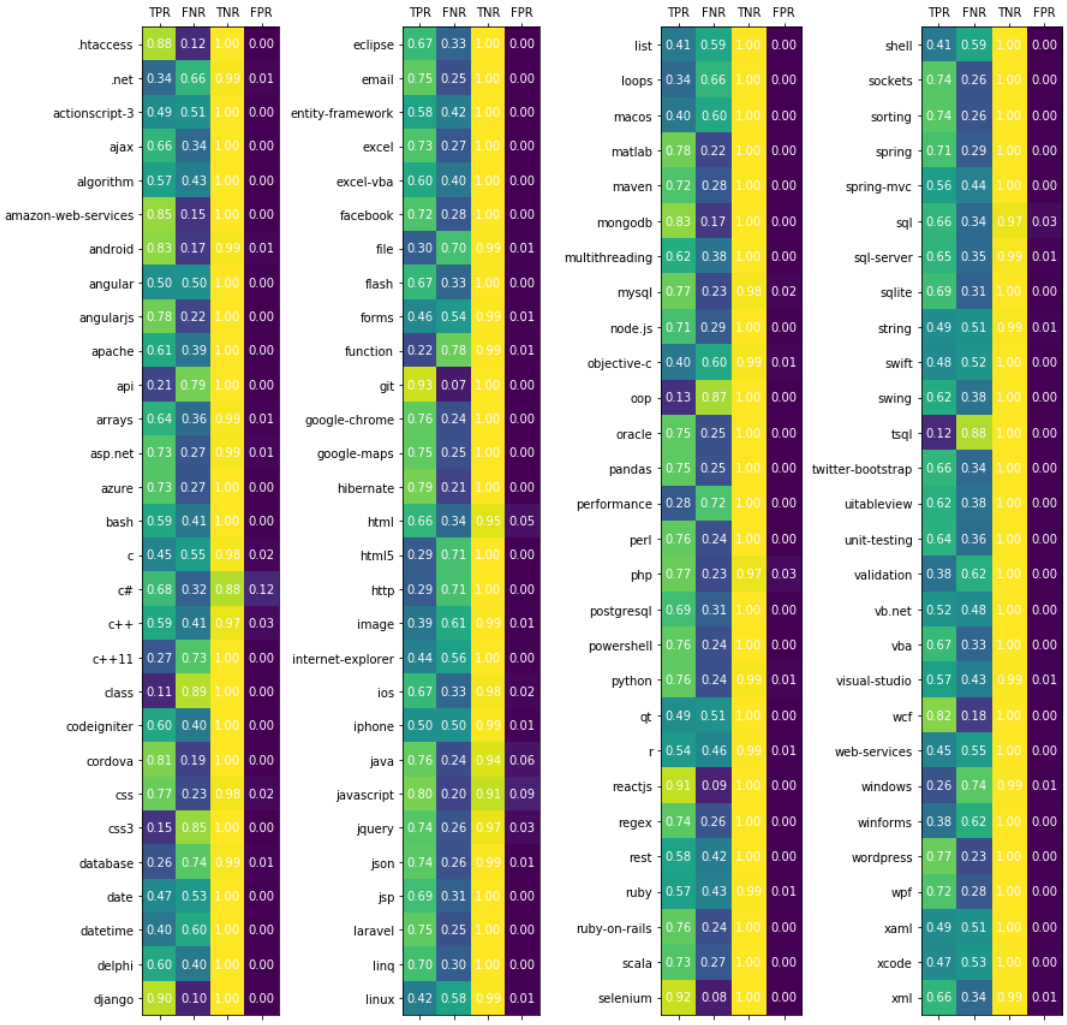
Avec les paramètres retenus par la validation croisée, le score de Jaccard sur le jeu de test est de 0.504.

Sur l'image ci-contre on présente les résultats obtenus pour les premières questions du jeu de test :

Out[13]:

	y_pred	y_true
0	(unit-testing,)	(maven, unit-testing)
1	(c#,)	(c#,)
2	(java,)	(java,)
3	(ios,)	(iphone, objective-c)
4	()	(validation,)
5	(angularjs, html5, java, spring)	(angularjs, spring)
6	(ios, iphone)	(ios,)
7	(c#, c++, visual-studio)	(c++,)
8	(delphi,)	(delphi,)
9	(c++, r)	(file, r)
10	(ruby, ruby-on-rails)	(javascript, ruby, ruby-on-rails)
11	(laravel, php)	(laravel, php)
12	(node.js,)	(javascript, node.js)
13	(javascript, python, regex)	(python, regex)
14	(javascript,)	(css, html, javascript, jquery)
15	(asp.net, c#)	(asp.net, ruby-on-rails)
16	(node.js,)	(node.js,)
17	(git,)	(git,)

On voit que même si les prédictions tombent souvent justes, ça arrive assez fréquemment que des tags manquent voir qu'il n'y en ait pas du tout. L'image ci-dessous peut être interprétée comme une matrice de confusion multi-label. Pour chaque tag on affiche le taux de vrais positifs (TPR), de faux négatifs (TNR), de vrais négatifs (TNR) et de faux positifs (FPR). On se rend compte que le taux de faux positifs est très faible (le modèle prédit rarement de mauvais tags), mais c'est le taux de faux négatifs qui est très variable selon les tags.



4.4 Comparaison avec forêt aléatoire

Nous avons aussi comparé les résultats de la régression logistique avec ceux donnés par une forêt aléatoire. Après une validation croisée, le meilleur score obtenu sur le jeu de test était de 0.463. Ce score étant moins bon, ce sera la régression logistique qui sera mise en

production.

5 Méthode non supervisée : Latent Dirichlet Allocation (LDA)

5.1 Présentation du modèle

Une autre approche possible est d'utiliser un modèle non-supervisé pour faire ressortir des sujets sous-jacents au jeu de données. Nous avons testé la LDA (Latent Dirichlet Allocation), qui est une méthode générative qui se base sur les hypothèses suivantes :

- Chaque document du corpus est un ensemble de mots sans ordre (bag-of-words).
- Chaque document aborde un certain nombre de thèmes dans différentes proportions qui lui sont propres.
- Chaque mot possède une distribution associée à chaque thème. On peut ainsi représenter chaque thème par une probabilité sur chaque mot.

Pour rappel, un modèle génératif définit une probabilité de distribution jointe sur les différentes variables identifiées, à la fois observées (par exemple la distribution des mots) et latentes (par exemple la distribution globale des thèmes ou la distribution des thèmes sur chaque mot). Dans le cas de la LDA, la distribution utilisée est la loi Dirichlet (d'où le nom de la méthode).

5.2 Paramétrage

Pour pouvoir comparer les deux approches, il faut être capable de mesurer le score de Jaccard avec la LDA et donc être en mesure d'extraire des tags à partir de la distribution des thèmes pour chaque question. La solution mise en oeuvre a été de considérer qu'un thème était associé à une question si sa probabilité était supérieure à un certain seuil, qui sera choisi par validation croisée. Ensuite, pour chacun des thèmes associés on regarde les mots les plus réprésentatifs du thème et l'on ne garde que ceux qui sont aussi des tags, le nombre de mots à considérer sera un autre paramètre de la validation croisée.

Enfin, le nombre de thèmes à utiliser ainsi que le 'learning decay' (paramètre qui contrôle la vitesse d'apprentissage du modèle) seront les deux paramètres de la LDA que nous avons cherché à optimiser aussi par validation croisée.

Cet estimateur, que l'on pourrait qualifier de semi-supervisée (puisqu'il se base sur les thèmes de la LDA mais ne garde que les mots qui sont aussi des tags dans l'approche supervisé), a été créé de manière analogue à l'approche supervisée, en modifiant la fonction 'predict' de la LDA sous Scikit-learn.

Les résultats de la validation croisée donnent comme paramètres optimaux :
nombre de thèmes : 10

learning decay : 0.9
seuil de probabilité pour associer un thème à une question : 0.2
nombre de mots à considérer dans chaque thème : 30

5.3 Résultats

Avec les paramètres retenus par la validation croisée, le score de Jaccard sur le jeu de test est de 0.06.

Le score est bien plus faible qu'avec l'approche supervisée et l'on voit vite que ce modèle retourne souvent trop de tags

Out[248]:

	y_pred	y_true
0	(eclipse, file, python)	(maven, unit-testing)
1	(ajax, class, function, javascript, json, php,...	(c#,)
2	(ajax, class, function, javascript, json, php,...	(java,)
3	(ajax, class, function, javascript, json, list...	(iphone, objective-c)
4	(file, function, javascript, list, php, string)	(validation,)
5	(android, api, http)	(angularjs, spring)
6	(api, django, facebook, function, html, image,...	(ios,)
7	(android, api, eclipse, file, http, python)	(c++,)
8	(android, api, database, date, facebook, http,...	(delphi,)
9	(eclipse, file, function, javascript, list, ph...	(file, r)
10	(eclipse, file, python)	(javascript, ruby, ruby-on-rails)
11	(android, api, django, html, http, image, php)	(laravel, php)
12	(android, api, django, eclipse, file, html, ht...	(javascript, node.js)
13	(function, html, javascript, jquery, json, lis...	(python, regex)
14	(eclipse, file, html, javascript, jquery, json...	(css, html, javascript, jquery)
15	(android, api, facebook, http, sql)	(asp.net, ruby-on-rails)

Par contre il est intéressant de regarder les 10 mots les plus représentatifs de chaque thème sous-jacent trouvé par la LDA :

```
Topics in LDA model:
Topic #0:
['app', 'server', 'project', 'web', 'application', 'net', 'url', 'view', 'request', 'client']
Topic #1:
['run', 'error', 'test', 'command', 'version', 'running', 'line', 'thread', 'python', 'program']
Topic #2:
['value', 'array', 'list', 'item', 'data', 'element', 'loop', 'object', 'result', 'map']
Topic #3:
['image', 'code', 'php', 'html', 'page', 'form', 'node', 'js', 'post', 'display']
Topic #4:
['code', 'error', 'function', 'method', 'object', 'call', 'variable', 'class', 'following', 'type']
Topic #5:
['class', 'java', 'module', 'static', 'video', 'android', 'error', 'git', 'interface', 'cell']
Topic #6:
['text', 'jquery', 'string', 'div', 'code', 'json', 'content', 'html', 'size', 'button']
Topic #7:
['table', 'query', 'column', 'row', 'data', 'database', 'sql', 'date', 'mysql', 'xml']
Topic #8:
['user', 'data', 'control', 'application', 'model', 'server', 'time', 'component', 'service', 'set']
Topic #9:
['file', 'page', 'button', 'window', 'code', 'click', 'link', 'folder', 'script', 'read']
```

Comme on peut le voir, quelques thèmes sont effectivement interprétables : le thème n°7 a un lien clair avec les bases de données ou le n°6 avec du développement web front-end (html, js). On peut aussi regarder uniquement les tags parmi les 30 mots les plus représentatifs de chaque thème :

```
Topics in LDA model:
Topic #0:
['android', 'api', 'http']
Topic #1:
['python', 'file', 'eclipse']
Topic #2:
['list', 'string', 'function']
Topic #3:
['image', 'php', 'html', 'django']
Topic #4:
['function', 'class', 'string', 'php', 'javascript', 'json', 'ajax']
Topic #5:
['class', 'java', 'android', 'git', 'matlab']
Topic #6:
['jquery', 'string', 'json', 'html', 'javascript']
Topic #7:
['database', 'sql', 'date', 'mysql', 'xml']
Topic #8:
['api', 'facebook', 'sql']
Topic #9:
['file', 'php', 'javascript']
```

On comprend alors pourquoi ce modèle retourne trop de tags. Si par exemple il attribue pour une question les thèmes n°4 et 5, ça fait déjà 12 tags différents. C'est ce qui explique en partie le score très faible de cette méthode.

Une autre explication est le fait que cette approche ne peut retourner que des tags qui sont aussi dans le texte des questions, alors que souvent ce n'est pas le cas.

6 Comparaison des deux approches

6.1 Quelques Exemples

```
In [253]: compare_results(33745)

Input: d3.js chart showing d3.js chart parse json data local file project d3 chart url local file json array chart showing basi
ng chart advance javascript html json data

Tags: ['javascript', 'json']

Supervised tags: ['javascript', 'jquery', 'json']

Unsupervised tags: ['image', 'php', 'html', 'django']

Most reprensented topics:
Topic #3:
['image', 'code', 'php', 'html', 'page', 'form', 'node', 'js', 'post', 'display']

In [254]: compare_results(32978)

Input: comparison strftime work inconsistenly select today visited url firefox database places.sqlite attempt query accomplish
operator answer query query operator answer work query

Tags: ['sqlite']

Supervised tags: ['sql', 'sqlite']

Unsupervised tags: ['database', 'sql', 'date', 'mysql', 'xml', 'file', 'php', 'javascript']

Most reprensented topics:
Topic #7:
['table', 'query', 'column', 'row', 'data', 'database', 'sql', 'date', 'mysql', 'xml']
Topic #9:
['file', 'page', 'button', 'window', 'code', 'click', 'link', 'folder', 'script', 'read']
```

6.2 Discussion

On voit bien sur les exemples précédents que l'approche non-supervisée est moins efficace dans une optique de prédiction de tags. Cependant elle n'est pas pour autant sans intérêt. En effet, les approches non-supervisées peuvent permettre de faire ressortir de nouveaux sujets qui ne sont pas encore couverts par les tags actuels (comme par exemple lorsqu'une nouvelle technologie apparaît et qu'il commence à y avoir beaucoup de questions sur le sujet). C'est quelque chose qui n'est pas possible avec l'approche supervisée qui va se limiter à prédire des tags déjà existants.

6.3 API

Les deux approches ont été mises à disposition sous la forme d'une API disponible à l'adresse : http://weber-thomas.fr/ocr/project6.

Pour éviter de refaire le paramétrage des modèles à chaque requête, les paramètres des modèles ont été déterminés en local puis transférer sur le serveur avec la librairie joblib.

Enfin, le code de l'API est disponible sur github : https://github.com/serphone/stackoverflow$_tags$