MCHAREK Ayoub

Catégorisation automatique des questions Stackoverflow

Parcours Data Scientist
Openclassrooms
Projet N° 6

Table des matières

1. Enoncé du projet

1.1 Mise en situation

Stack Overflow est un site de questions et réponses pour les programmeurs professionnels et passionnés. Il s'agit d'un site Web privé, le site phare du Stack Exchange Network, créé en 2008 par Jeff Atwood et Joel Spolsky. Le site propose des questions et réponses sur un large éventail de sujets en programmation informatique. Pour poser une question sur ce site, il faut entrer plusieurs tags de manière à retrouver facilement la question par la suite. Pour les utilisateurs expérimentés, cela ne pose pas de problème, mais pour les nouveaux utilisateurs, il serait judicieux de suggérer quelques tags relatifs à la question posée.

Dans le cadre du 6-ème projet du parcours Data Scientist sur le site Openclassrooms, Je vais devoir développer un système de suggestion de tag pour le site. Celui-ci prendra la forme d'un algorithme de machine learning qui assigne automatiquement plusieurs tags pertinents à une question.

1.2 Données

Stack Overflow propose un outil d'export de données - "stackexchange explorer", qui recense un grand nombre de données authentiques de la plateforme.

En mettant en place quelques requêtes SQL pour explorer le contenue de la base de données et les limites du site web, j'ai décidé de faire 3 extractions consécutives tel que :

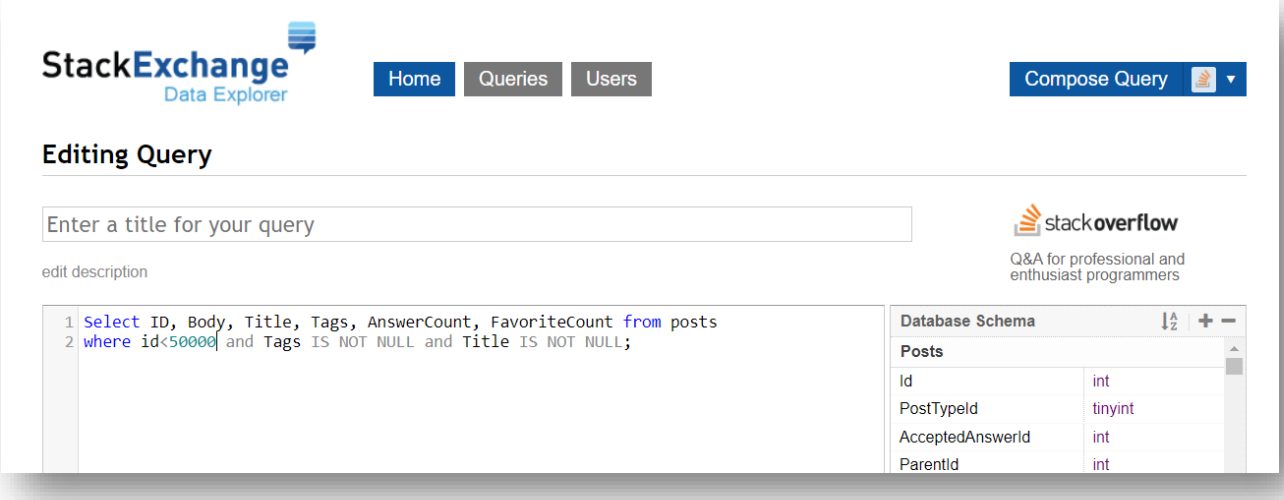


***Figure 1 :** Interface StackExchange et requête SQL*

J'ai ensuite construit un dataset à partir des 3 fichiers csv téléchargés avec les caractéristiques suivantes :

- 103.676 lignes représentant une question chacune
- 5 Variables, 3 qualitatives et 2 quantitatives :
  - Body : les questions posées en format html
  - Title : Les titres de la demande en format chaine de caractères
  - Tags : Les tags en format chaine de caractères
  - AnswerCount : le nombre des réponses à la question
  - FavoritCount : le nombre indiquant la popularité de la question

1.3 Contraintes

Le but principal est de traiter des données non structurer afin de mettre en place un système de suggestion de tag dans la forme d'un algorithme de machine learning qui assigne automatiquement plusieurs tags pertinents à une question.

Deux approches possibles se dégagent de cette tâche :

- Une approche non supervisée : la modélisation de sujets en utilisant le corpus des questions et des titres. Cette approche consiste à détecter les sujets latents abordés dans un corpus de documents, ensuite assigner les sujets détectés à ces différents documents.
- Une approche supervisée : la classification multi-labels en utilisant des tokens extraites du corpus comme variables en entrée et les tags dont on dispose comme variables cibles.

Avant de me lancer dans la modélisation, il va falloir aussi prétraiter les données pour obtenir un jeu de données exploitable.

Il faut savoir que les données textuelles génèrent une énorme quantité de données surtout dans la partie extraction de variable, ainsi, pour optimiser les coûts du calcul et avoir des meilleures performances de mes modèles je vais devoir appliquer des techniques de réductions de dimensions.

Le déploiement de la solution sera sous la forme d'une API.

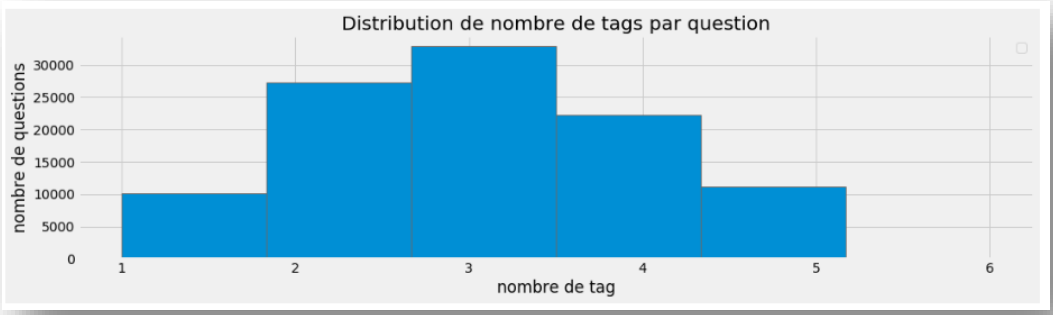
Enfin, pour suivre les modifications du code final à déployer et pour une première fois depuis le début de ma formation, je vais utiliser un logiciel de gestion de versions « Git ».

⇨ Dans la suite du rapport je vais expliquer la démarche et les différentes étapes que j'ai passé par pour répondre à ce projet, je commencerais par l'analyse exploratoire et le nettoyage des données, ensuite je vais détailler les modèles que j'ai réussi à mettre en place et leurs évaluations associées. Enfin j'entame la partie déploiement et prospectives d'améliorations dans la conclusion.

2. Analyse d'exploration et feature engineering

2.1 Analyse exploratoire

Pour l'analyse de mon dataset, j'ai commencé par visualiser la distribution des différentes variables numériques dont je disposais :



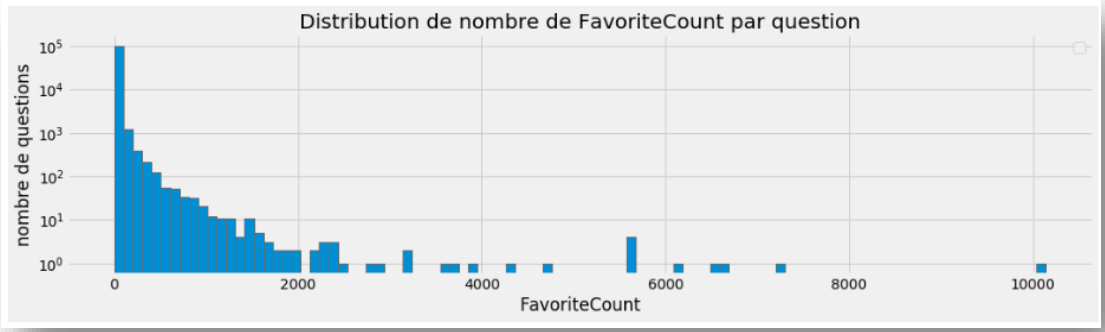
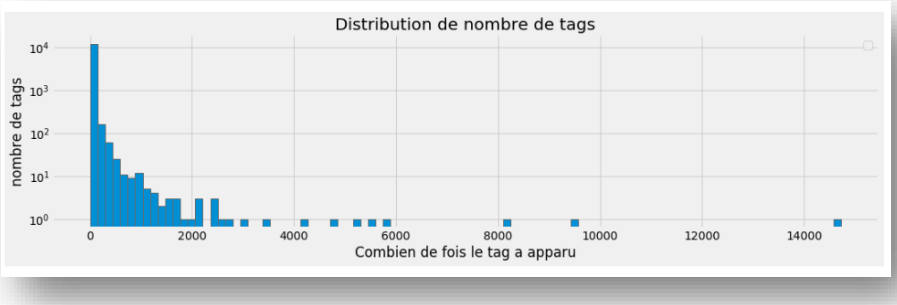


Figure 2 & 3 : Distribution des nombres de tags et de FavoriteCount

J'ai ensuite calculé la fréquence de chaque tag pour mieux comprendre les sujets les plus populaires sur Stackoverflow.




```
'c#': 14725,
'.net': 9511,
'java': 8160,
'c++': 5759,
'asp.net': 5526,
'javascript': 5165,
'python': 4742,
'php': 4212,
'sql': 3388,
'sql-server': 3036,
'jquery': 2716,
'iphone': 2552,
'html': 2497,
'c': 2406,
'windows': 2403,
'asp.net-mvc': 2192,
'wpf': 2137,
'mysql': 2123,
'database': 1933,
```

Figure 4 & 5 : Distribution de l'apparition des Tags par question

On remarque que la variance est assez importante pour les 12.403 tags dans mon dataset.

⇨ L'analyse m'a permis de mieux comprendre le comportement de mes variables, mais j'ai choisi de ne pas prendre en considérations ces résultats pour faire un nettoyage des valeurs aberrantes.

2.2 Nettoyage et feature engineering

Dans cette partie je vais développer les différentes étapes de transformations, nettoyage et préparation des données pour le modèle de machine learning.

a) *Préparation Body : HTML → TEXT*

La première transformation nécessaire est celle du BODY, en effet dans la base de données, les questions sont stockées sous format html, j'ai donc utilisé la bibliothèque **BeautifulSoup** afin d'extraire les questions en format texte.

b) *Création des tokens*

La tokenization désigne le découpage en mots les différentes questions qui constituent mon corpus. Cette étape est indispensable dans le prétraitement des données textes puisque ça va me permettre de passer à l'étape du nettoyage et modélisation.

Dans cette étape j'effectue aussi une normalisation qui me permet de ne pas prendre en compte des détails non importants au niveau local (ponctuation, majuscules etc.)

J'ai utilisé pour ce traitement le tokenizer « **RegexpTokenizer(r'\S+')** » de la bibliothèque **NLTK.**

c) *Stop words, lemming and stemming*

La première manipulation souvent effectuée dans le traitement du texte est la suppression de ce qu'on appelle en anglais les stopwords. Ce sont les mots très courants dans la langue étudiée ("and", "to", "i"... en anglais) qui n'apportent pas de valeur informative pour la compréhension du "sens" de mes questions.

Ensuite le processus de « lemmatisation » qui consiste à représenter les mots sous leur forme canonique. Par exemple pour un verbe, ce sera son infinitif. Pour un nom, son masculin singulier. L'idée étant encore une fois de ne conserver que le sens des mots utilisés dans le corpus.

Enfin le processus de « stemming » ou racinisation. Cela consiste à ne conserver que la racine des mots étudiés. L'idée étant de supprimer les suffixes, préfixes et autres des mots afin de ne conserver que leur origine. C'est un procédé plus simple que la lemmatisation et plus rapide à effectuer puisqu'on n'aura pas à utiliser un dictionnaire.

Pour cette partie j'ai continué à utiliser la bibliothèque **NLTK :**

- Stop Words : nltk.corpus.**stopwords.words('english')**
- Lemmatisation : nltk.stem.**WordNetLemmatizer()**
- Stemming : nltk.stem.porter.**PorterStemmer()**

d) *Bigrammes et Trigrammes*

Les bigrammes sont deux mots qui apparaissent fréquemment ensemble dans le document. Les trigrammes sont 3 mots fréquents.

Quelques exemples dans notre exemple sont : «front_end», «sql_server», «internet_explor_7» etc.

Le modèle **Phrases** de **Gensim** que j'ai utilisé peut construire et implémenter les bigrammes, trigrammes, quadrigrammes et plus encore. Les deux arguments importants de **Phrases** sont **min_count** et **threshold**. Plus les valeurs de ces paramètres sont élevées, plus il est difficile de combiner des mots avec des bigrammes.

L'évaluation de la pertinence de l'output et le choix des paramètres pour cette partie ne sont pas basés sur une méthode scientifique, en effet, j'ai changé ces paramètres plusieurs fois tout en regardant le nombre et le contenu des bi/tri_grammes générés, j'ai ensuite choisi arbitrairement ce qui me semblait le plus correcte pour ce contexte.

e) *Bag of words, TF_IDF and target encoding*

Après avoir vu les différents types de nettoyage du texte possible dans les parties précédentes, je vais maintenant extraire l'information du texte pour le traitement ultérieur par des modèles de machine learning.

Une première approche classique possible est **bag-of-word**, ça consiste en la représentation de chaque question par un vecteur de la taille du vocabulaire avec le nombre d'apparition du mot comme valeur pour chaque variable, ensuite on utilisera la matrice composée de l'ensemble de ces vecteurs qui forment le corpus comme entrée de nos algorithmes.

Une autre manière de pondérer est le **tf-idf** abréviation pour « term frequency – inverse document frequency », cette méthode et contrairement au Bag-of-Words, est destinée à refléter l'importance d'un mot pour un document dans une collection ou un corpus. La valeur tf–idf augmente proportionnellement au nombre de fois qu'un mot apparaît dans le document, ensuite compensée par le nombre de documents dans le corpus qui contiennent le mot, ce qui permet d'ajuster le fait que certains mots apparaissent plus fréquemment en général.

Afin de faire ces transformations, j'ai commencé par une approche manuelle dans laquelle je codais moi-même les fonctions pour générer les fréquences, les fréquences inverses des documents et enfin le tf-idf. Vu que cette méthode n'est pas optimisée, les fonctions prenaient des heures pour s'exécuter, j'ai gardé le code dans le brouillon à la fin du Notebook Analyse mais j'ai utilisé les fonctions prêtes de **sklearn.feature_extraction.text**

```
Cvectorizer = CountVectorizer(tokenizer=text_splitter, max_df=0.5, min_df=10)
Tvectorizer = TfidfVectorizer(tokenizer=text_splitter, max_df=0.5, min_df=10)
```

Ce traitement me permet de faire plus de nettoyage avec deux paramètres :

- max_df : ignorer les tokens qui apparaissent dans plus de 50% des questions (généraux)
- min_df : ignorer les tokens qui apparaissent dans moins de 10 questions (spécifiques)

En ce qui concerne le « target encoding », et puisqu'il peut y avoir plusieurs tags pour une question, j'ai représenté la sortie comme 0 ou 1, où 1 signifie que le token est présent et 0 signifie qu'il est absente. Pour ceci j'utilise le **MultiLabelBinarizer** de **scikit-learn**.

Je finis par enregistrer mes matrices sparce et les différents modèles et fonction de nettoyage que j'ai mis en place pour les utiliser plus tard. Pour ceci j'ai utilisé **pickle.dump & scipy.sparce.save_npz.**

data	signification	dimensions	format
df.csv	Tous les données brutes et transformées	(103 676, 12)	csv
X_C.npz	Matrices Bag of Words output **Count_vectorizer**	(103 676, 31541)	sparce
X_T.npz	Matrices output **TF-IDF**	(103 676, 31541)	sparce
Y.npz	Target output **MultiLabelBinarizer**	(103 676, 12403)	sparce

3. Modélisation et évaluation

3.1 Approche non supervisée

La modélisation automatique de sujet permet de détecter les sujets latents abordés dans un corpus de documents, assigner les sujets détectés à ces différents documents. On peut ensuite utiliser ces sujets pour effectuer des recherches plus rapides, organiser les documents ou les résumer automatiquement.

Dans notre cas, nous allons utiliser une méthode nommée LDA (Latent Dirichlet Allocation). C'est une méthode non-supervisée générative qui se base sur les hypothèses suivantes :

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

Une représentation formelle sous forme de modèle probabiliste graphique est la suivante :

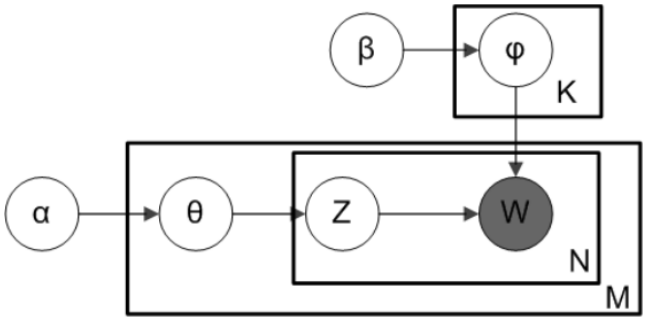


***Figure 6 :** modèle probabiliste graphique*

J'ai tout ce qu'il faut pour former le modèle LDA. En plus du corpus et du bag of words, Je dois aussi également fournir le nombre de sujets.

Pour ce traitement j'ai utilisé **LatentDirichletAllocation** de **sklearn.decomposition**.

La cohérence des sujets constitue une mesure pratique pour juger de la qualité d'un modèle de sujet donné. En effet c'est une mesure la distance relative entre les mots d'un sujet et on peut l'utiliser pour déterminer le nombre optimal des sujets. La librairie **Gensim** offre une fonction pour calculer cet indicateur mais avec **sklearn**, et comme dans notre cas il va falloir le calculer manuellement.

Pour des raisons de temps de traitement énormes pour entrainer les modèles j'ai choisi de ne pas mettre en place cette méthode d'évaluation et je me suis arrêté à la visualisation et l'interprétation.

Pour la visualisation des sujets et maintenant que le modèle LDA est construit, l'étape suivante consiste à examiner les sujets produits et les mots clés associés. Pour ceci j'ai trouvé un outil graphique interactif du package **pyLDAvis** comme on le voit sur les captures d'écrans ci-dessous.

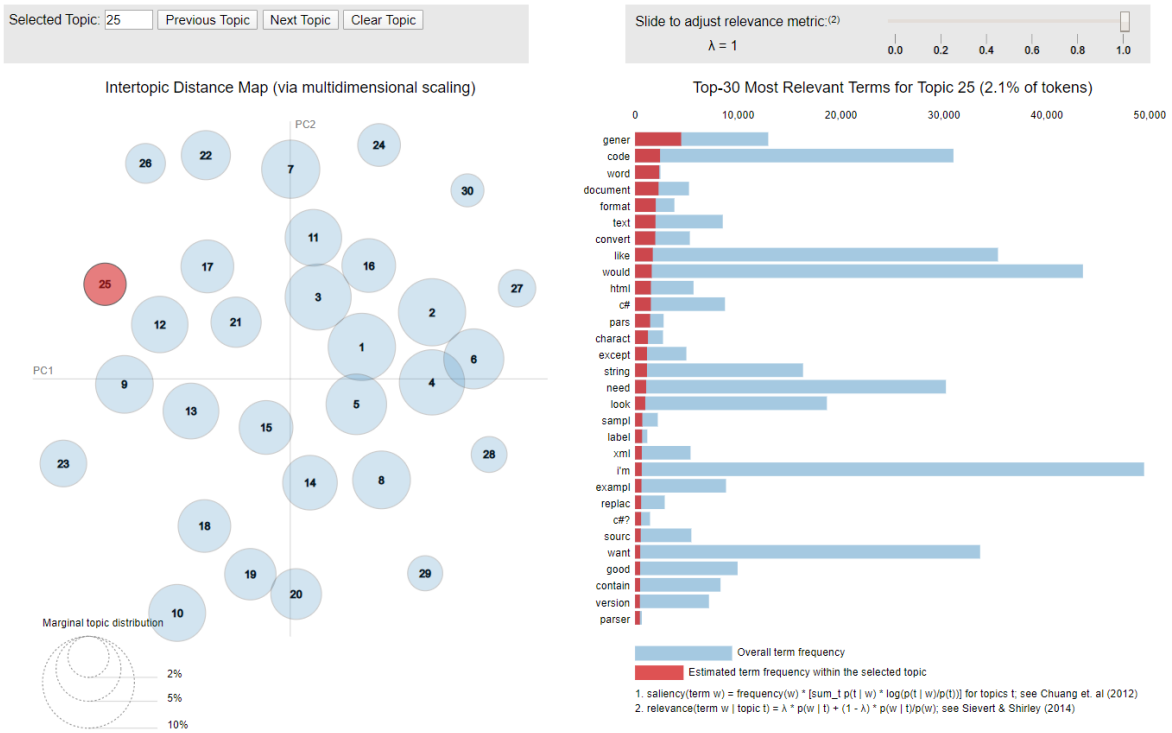


Figure 7 : *Représentation des sujets et des mots associées avec pyLDAvis*

J'ai d'abord remarqué qu'il y a plusieurs tokens qui ne servent à rien dans mes modèles, une possibilité de plus de nettoyage est envisageable.

J'ai varié le paramètre « nombre des sujets » quelques fois entre 20 et 50, à chaque incrémentation le modèle met plus de temps à s'entrainer sans que ça soit nécessairement plus pertinent.

Normalement, pour donner suite à ce travail, on devrait déduire le thème de chaque sujet à partir des mots les plus importants, ensuite labellisé les questions avec les thèmes les plus présents dans chacune. Mais avec un total de 12403 tags ça sera un peu compliqué à mettre en place.

Par ailler j'ai choisi de passer à une approche supervisée.

3.2 Réduction de dimension

Avec 31541 variables dans mes matrices Bag Of Words et TF-IDF, l'entrainement peut prendre un temps de calcul énorme, demandera une mémoire importante à mettre en place et surtout peut engendrer des mauvaises performances, ainsi, et avant de commencer l'entrainement j'utilise une méthode de réduction de dimensions.

Pour ce traitement j'utilise **truncated SVD** de **sklearn**, ce transformateur effectue une réduction de dimensionnalité linéaire au moyen d'une décomposition de valeur singulière tronquée (SVD). Contrairement à l'ACP, cet estimateur ne centre pas les données avant de calculer la décomposition. Cela signifie qu'il peut fonctionner efficacement avec les matrices sparse.

En particulier, la SVD tronquée fonctionne sur les matrices tf-idf. Dans ce contexte, elle est connue sous le nom d'analyse sémantique latente (LSA).

Cet estimateur prend en charge deux algorithmes : un solveur SVD aléatoire rapide et un algorithme «naïf» qui utilise ARPACK comme solveur électronique sur (X * X.T) ou (X.T * X), selon celui qui est le plus efficace.

Un fois que j'ai effectué le calcul du taux de variance cumulé comme le montre le schéma ci-dessous, j'ai choisi de garder :

- 1500 variables de BOW qui représentent plus que 80% de la variance
- 2000 variables du TF_IDF qui représentent plus que 60% de la variance

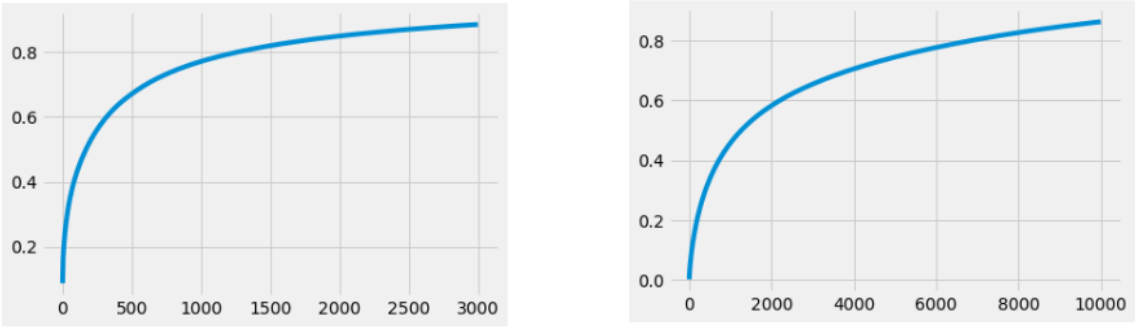


Figure 8 & 9 : variance cumulée pour réduction de dimension BOW & TF_IDF

Pour faire des tests rapides, j'ai aussi généré des matrices de taille réduite (100 variables).

3.3 Approche supervisée

Nous avons un problème de classification multi label, la différence avec une classification multi classes c'est qu'une question peut avoir plus d'un seul label.

En effet, la plupart des algorithmes d'apprentissage traditionnels sont développés pour les problèmes de classification à étiquette unique. Par conséquent, de nombreuses approches dans la littérature transforment le problème multi-étiquettes en plusieurs problèmes mono-étiquettes, de sorte que les algorithmes existants d'une seule étiquette peuvent être utilisés.

Une approche intuitive pour résoudre un problème multi-étiquettes consiste à le décomposer en plusieurs problèmes de classification binaire indépendants (un par catégorie). Pour ceci j'applique une stratégie **OneVsRest**.

L'approche **one-versus-rest** de la classification multi-labels consiste à créer K classifieurs binaires qui séparent chaque classe k de l'union des autres classes. On prédit alors la classe pour laquelle la fonction de décision est maximale.

L'hypothèse principale est que les étiquettes s'excluent mutuellement. Je ne considère aucune corrélation sous-jacente entre les labels de cette méthode.

Une autre préoccupation avec cette stratégie, c'est que lorsque le nombre de classes est énorme, le temps de formation et de prédiction augmente considérablement.

a) *Modèles utilisés :*

Dans un premier temps, et comme indiqué dans la partie précédente, je commence par tester plusieurs classifieurs avec un jeu de données réduit :

Linear SVC : SVM (support Vector Machine) multi classe en optimisant simultanément K classifieurs binaires (méthode de Crammer et Singer). Ce classifieur reformuler le problème de classement comme un problème d'optimisation quadratique. L'idée clé est la notion de marge maximale. La marge est la distance entre la frontière de séparation et les échantillons les plus proches. Ces derniers sont appelés vecteurs supports.

SVM est plus efficace dans les espaces de grande dimension et relativement efficace en mémoire, mais ne fonctionne pas très bien, lorsque l'ensemble de données a plus de bruit.

Random Forest : Les forêts à décision aléatoire sont une méthode d'apprentissage ensembliste pour la classification, le modèle construit une multitude d'arbres de décision au moment de la formation, ensuite produit la classe qui est le mode des classes des arbres individuels.

Les forêts aléatoires réduisent le problème de « over fitting » dans les arbres de décision et réduit également la variance et améliore donc la précision.

Le problème avec les forêts aléatoires c'est que le temps d'entrainement et de prédiction augmente considérablement en choisissant un grand nombre de estimateurs

XGBoost Classifier : XGBoost est une bibliothèque optimisée de renforcement de gradient distribué conçue pour être très efficace, flexible et portable. Il implémente des algorithmes d'apprentissage automatique de classification et de régression dans le cadre du Gradient Boosting avec une méthode d'apprentissage ensembliste. XGBoost fournit un boost d'arborescence parallèle (également connu sous le nom de GBDT, GBM) qui résout de nombreux problèmes de manière rapide et précise.

L'une des forces majeures de cet algo est de pouvoir atteindre une excellente précision avec seulement une mémoire et des exigences d'exécution modestes pour effectuer la prédiction, une fois le modèle formé.

De l'autre côté, comme les forêts aléatoires, les ensembles d'arbres sont très difficiles à interpréter par rapport aux arbres de décision individuels. Aussi les modèles peuvent nécessiter un réglage minutieux du taux d'apprentissage et d'autres paramètres, et le processus de formation peut prendre beaucoup de temps de calculs.

SGDClassifier : SGDClassifier implémente une routine d'apprentissage de descente de gradient stochastique simple qui prend en charge différentes fonctions de perte et pénalités pour la classification. La descente de gradient stochastique (SGD) est une approche simple mais très efficace pour l'apprentissage discriminant des classificateurs linéaires sous des fonctions de perte convexes telles que les machines à vecteur de support (linéaire) et la régression logistique.

Les avantages de la descente de gradient stochastique sont l'efficacité pour les grandes dimensions vu qu'il peut traiter des matrices Sparces et aussi donne facilité de mise en œuvre (nombreuses hyperparamètres pour le tunning).

Le nombre d'hyperparamètres tels que le paramètre de régularisation et le nombre d'itérations peut aussi être considérer comme un inconvénient pour le choix optimal. Enfin, SGDClassifier est sensible à la mise à l'échelle des données (scaling).

b) Scoring et évaluation

Pour **l'évaluation** des classifieurs j'utilise plusieurs indicateurs :

Accuracy : L'un des indicateurs les plus évidents pour la classification qui représente le taux des prédictions correctes :

$$\text{Accuracy} = \frac{\text{True positives} + \text{True negatives}}{\text{True positives} + \text{False positives} + \text{True negatives} + \text{False negatives}}$$

Le problème c'est que dans le cas d'un classifieur multilabel comme le nôtre, cet indicateur sera toujours très faible. Si par exemple on prédit (Python) pour des vrais labels (Python, Pandas), notre score ne changera pas et le modèle va considérer que c'est totalement une fausse prédiction.

F1_score : Le score F1 est la moyenne harmonique de la précision et du rappel, ce score atteint sa meilleure valeur à 1 (précision et rappel parfaits) et la pire à 0.

Le score F comporte une extension utilisée pour évaluer les problèmes de classification avec plus de deux classes. Dans cette configuration, le score final est obtenu par micro-moyenne (biaisée par la fréquence des classes).

Hamming Loss : Cet indicateur est plus adapté aux problèmes de classification multi labels, il représente la fraction des mauvaises étiquettes par rapport au nombre total d'étiquettes,

Il s'agit d'une fonction de perte, donc la valeur optimale est zéro.

% des prédictions non nulles : J'ai remarqué que les premiers modèles que j'ai entrainé ne prédisent aucun label pour un grand pourcentage de mon dataset, ainsi j'ai rajouté cet indicateur comme étant l'un des facteurs de choix principaux.

3.4 Performances, et choix du modèle final

Au départ, j'ai commencé par un test avec le dataset TF_IDF et une réduction de dimension à 100 variables de l'input. Pour ce premier jet, j'ai gardé toutes les questions et j'ai utilisé un modèle de forêts aléatoires avec les paramètres de bases.

```
True tags:      ('date', 'datetime', 'javascript', 'time')
Predicted tags: ('javascript',)

True tags:      ('distro', 'java', 'linux')
Predicted tags: ('java',)

True tags:      ('database', 'php', 'security')
Predicted tags: ('php',)

True tags:      ('.htaccess', 'php', 'xml')
Predicted tags: ('php',)

True tags:      ('.net', 'web-services')
Predicted tags: ('java',)
```

```
mming_loss(y_test, pred_val_tfidf_rf)

0023616887168662005
```

Figure 10 : Premier modèle et prédiction

Pour les quelques Tags prédites, j'estimais que j'avais un bon résultat pour une base line. J'ai supposé que le haut pourcentage des prédictions nulles soit dû à la réduction de dimension intense.

Par erreur de jugement, j'ai décidé de lancer l'entrainement du même modèle sur la totalité de mes dataset BOW et TF_IDF en appliquant une réduction de dimension qui garde 80% de la variance de mes variables.

Modèle	Données à l'entrée	Temps d'entrainement	Temps de prédiction	Accuracy	Hamming Loss	% des prédictions nulles
Random Forest	BOW (103 676, 1500)	7h58min	46min	0.00054	0.00023	> 95%
Random Forest	TF_IDF (103 676, 8000)	18h37min	2h38min	0.00154	0.00023	> 95%

La seule conclusion que j'ai pu tirer, c'est que le modèle avec input TF_IDF est plus performant.

C'est après cette partie que j'ai décidé de rajouter l'indicateur sur la fraction des prédictions nulles.

Pour la suite j'ai décidé de travailler avec un dataset réduit pour tester les différents modèles avant de passer au choix final. Vu que le BOW offre une meilleure variance pour le même nombre de variables réduites, j'ai l'ai choisi comme le modèle de données pour tester les différents classifieurs.

Pour le test des différents classifieurs, j'ai décidé de prendre les hyperparamètres de base, sauf pour le Random Forest, j'ai manuellement changé le nombre d'arbres et la profondeur maximale.

Dans le tableau ci-dessous on note : RF(n_estimator, max_depth) ➔ RandomForest paramètres

Modèle	Temps d'entrainement	Temps de prédiction	Accuracy	Hamming Loss	% des prédictions nulles
RF (10, 4)	1m18s	4m30s	0.00077	0.00096	99.3%
RF (10, 10)	1m20s	4m30s	0.00231	0.00097	93.7%
RF (50, 10)	4m47s	4m58s	0.00154	0.00096	97.1%
RF (50, 20)	4m57s	4m58s	0.00154	0.00096	97.2%
RF (50, 50)	4m57s	4m58s	0.00308	0.00096	96.6%
RF (100, 10)	9m29s	5m09s	0.00231	0.00096	96.9%
RF (100, 20)	9m25s	5m09s	0.00154	0.00096	96%
XGBoost ('gbtree')	4m02s	3s	0.00462	0.00095	87.5%
XGBoost ('dart')	4m21s	3s	0.00462	0.00095	87.5%
XGBoost (' gblinear')	3m17s	3s	0	0.00096	100%
LinearSVC	1m04s	0.4s	0.00925	0.00098	78.1%
SGDClassifier	15s	0.4s	0.00694	0.00148	30%

Le SGDClassifier donne des prédictions non nulles pour 70% et prend beaucoup moins de temps pour s'entrainer, c'est pour ça que c'est le modèle que j'ai choisi malgré ses performances un peu dépassées.

Pour le tuning du modèle avec « **GridSearch »** j'intègre un exemple des données sous le format final TF_IDF réduit vu que la méthode décente de gradient stochastique dépend de la normalisation et donc de la réduction des données.

Le modèle final :

- Données : TF_IDF de dimensions (103 676, 8000) représentant plus que 80% de la variance
- Modèle : SGDClassifier (alpha = 0.00000, penalty = « l2 »)
- Temps d'entrainement : 28h52min
- Temps de prédictions de toutes les questions de test (25000): 1h34min
- Accuracy : 0.07307
- Hamming Loss : 0.000201
- F1_Score : 0.46
- % des prédictions nulles : 12.6%

4. Livrables

4.1 Notebook et rapports

L'ensemble des livrables comporte :

- Notebook Nettoyage et Exploration des données
- Notebook Modélisation non supervisé
- Notebook Modélisation supervisé
- Un repository Git qui contient le code : https://github.com/amcharek/OCP6.git
- Point d'entrée API :
- Un rapport et une présentation PowerPoint

4.2 Code et API

Pour le déploiement de l'API j'ai choisi d'utiliser la plateforme AWS, j'ai utilisé Serverless un framework pour automatiser le déploiement d'une API, avec les services S3 pour stocker les modèles entrainés, Lambda pour la fonction de lecture et de prédiction et enfin API Gateway qui permet de ne pas gérer une instance ou une machine Linux complète dans le Cloud.

Dans le repository Git on trouve les fichiers suivants pour le code finale :

- Le fichier des différentes fonctions utilisées : Tokenization, Traitement Bigrammes, TF_IDF Vectorizer, Transformateur et Classifieur
- Un Script Python pour la Lambda : Importer les différents modèles entrainés, lire la question, traiter et prédire les Tags
- Un fichier YML qui représente l'infrastructure utilisé de AWS pour déployer l'API

5. Conclusion

On note qu'il y'a une marge d'amélioration :

- La partie nettoyage et traitements des tokens
- On peut se limiter aux tags qui reviennent plus que 10 fois
- On peut creuser la partie non supervisée en cherchant un nombre optimal de sujet

En traitant le sujet j'ai dû acquérir des nouvelles compétences en :

- Traitement de langage naturel
- Prétraitement des données non structurées </texte/> pour obtenir un jeu de données exploitable
- Méthode et algorithme de classification multi labels
- Méthode non supervisée pour le traitement de texte
- Utiliser un logiciel de gestion de versions de code, Git
- Infrastructure As A Code, serverless, AWS : Lambda, S3 & API Gateway

6. Bibliographie

- http://www.nltk.org/book/
- https://scikitlearn.org/stable/modules/generated/sklearn.linearzmodel.SGDClassifier.html
- https://www.machinelearningplus.com/nlp/topic-modeling-gensim-python/
- https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html
- https://www.irit.fr/IRIS-site/images/seminairs/Thonet2016.pdf
- https://github.com/susantabiswas/stackoverflow-question-tagger/blob/master/StackOverflow%20Question%20Tagger.ipynb
- https://github.com/priyagunjate/Stack-Overflow-Tag-Prediction/blob/master/Assignment-19.ipynb
- http://theprofessionalspoint.blogspot.com/2019/02/advantages-and-disadvantages-of-random.html
- https://www.datacorner.fr/xgboost/
- https://towardsdatascience.com/optimizing-hyperparameters-in-random-forest-classification-ec7741f9d3f6
- https://gdcoder.com/nlp-tutorial-multilabel-claddification-problem-using-linear-models/
- https://scipy.github.io/old-wiki/pages/SciPyPackages/Sparse.html
- https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.LatentDirichletAllocation.html
- https://towardsdatascience.com/evaluate-topic-model-in-python-latent-dirichlet-allocation-lda-7d57484bb5d0