Catégorisez automatiquement des questions

Contenu de la présentation

01 Nettoyage de la donnée

02 Prétraitements de la donnée

03 Modèles

04 Organisation du projet et réalisation

05 Conclusion

01

Nettoyage de la donnée

Scraping de la donnée

Afin de réaliser ce projet, nous avons requêté itérativement dans plusieurs plages de donnée pour éviter les seuils de requête API du site target.
La donnée est sélectionnée selon les critères suivants :

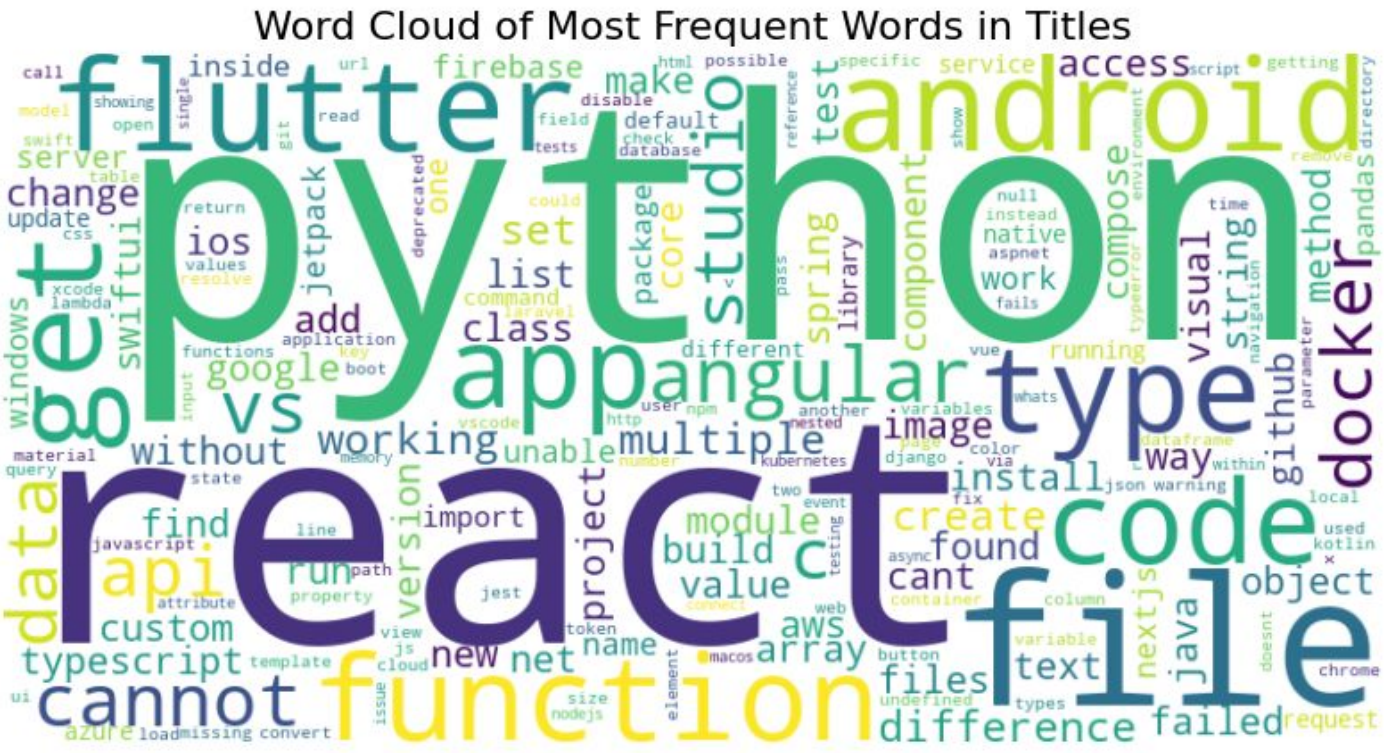
- un score minimum de 10
- au moins 4 tags associés
- au maximum 15 tags associés
- Une question de 10 caractères minimum
- Une question de 100 caractères maximum
- Possède un body

Nettoyage de la donnée

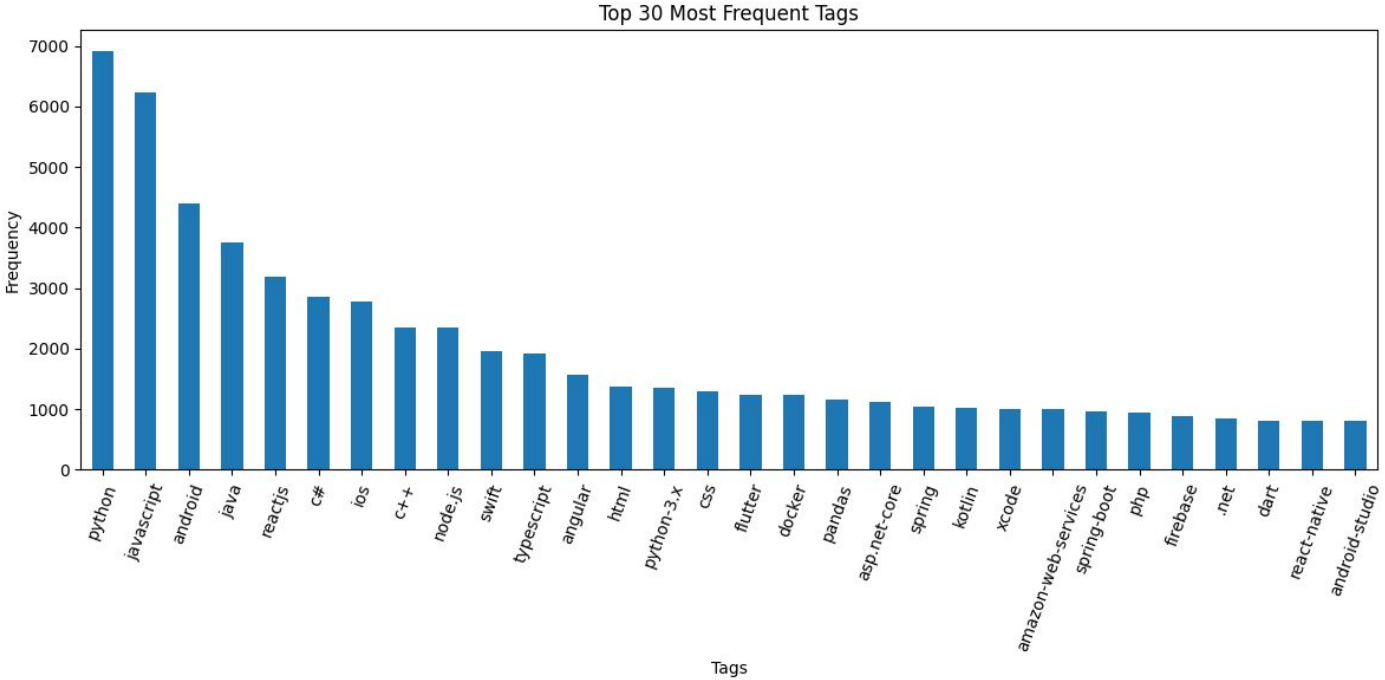
La donnée va être nettoyée de plusieurs façons afin de pouvoir être tester dans différents contextes.

- Commun pour tous :
  - En Anglais
  - Suppression de caractère spéciaux, a l'exception de la combinaison c++ et c#
  - Conversion en minuscule
- Une version "Cleaned" avec le dictionaire de stopword WordNet de NLTK et les mots inutiles les plus fréquents (ici "error", "using", "use")
- Une version "Cleaned strict" avec l'ajout de nouveau mots dans le dictionnaire de stopwords
- Une version "lm" qui utilise un lemmatiseur de rassembler les mots d'une même famille
- Une version "dl" qui ne comporte que le commun

Visualisation de la donnée



Visualisation de la donnée



02

Prétraitements de la donnée

Les différents prétraitements

Le bag of words

- CountVectoriser
- TF-IDF

Word/Sentence Embedding

- Word2Vec
- BERT
- USE

03

Modèles

Les différents modèles testé

Non-superviser dans un LDA

Superviser dans un One Vs Rest Classifier avec :

MultinomialNB

GaussianNB

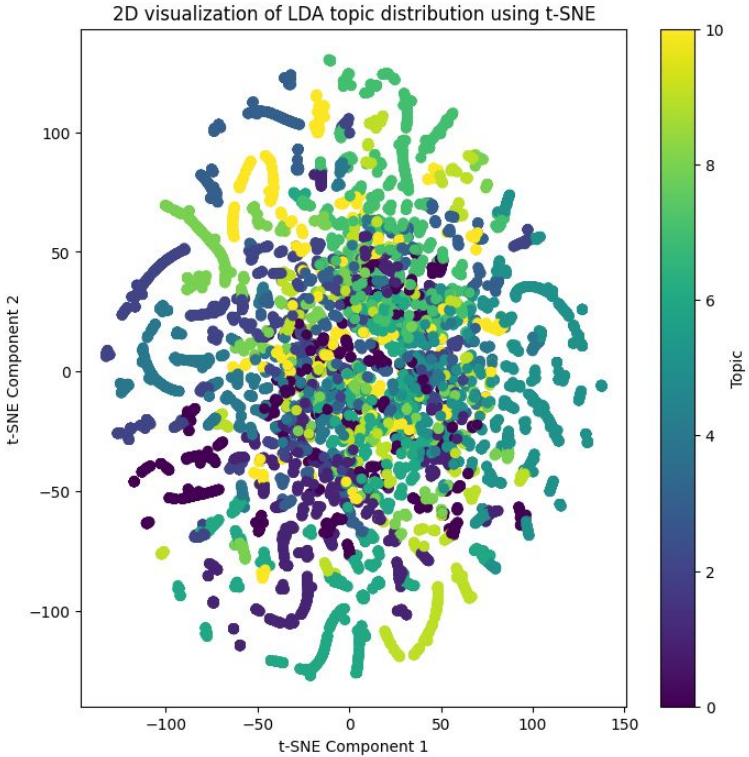
BernoulliNB

SGD Classifier (dans un Calibrated Classifier CV afin d'avoir le predict_proba)

LogisticRegression

SVC/SVM (trop couteux en temp pour nous)

Exemples de visualisations



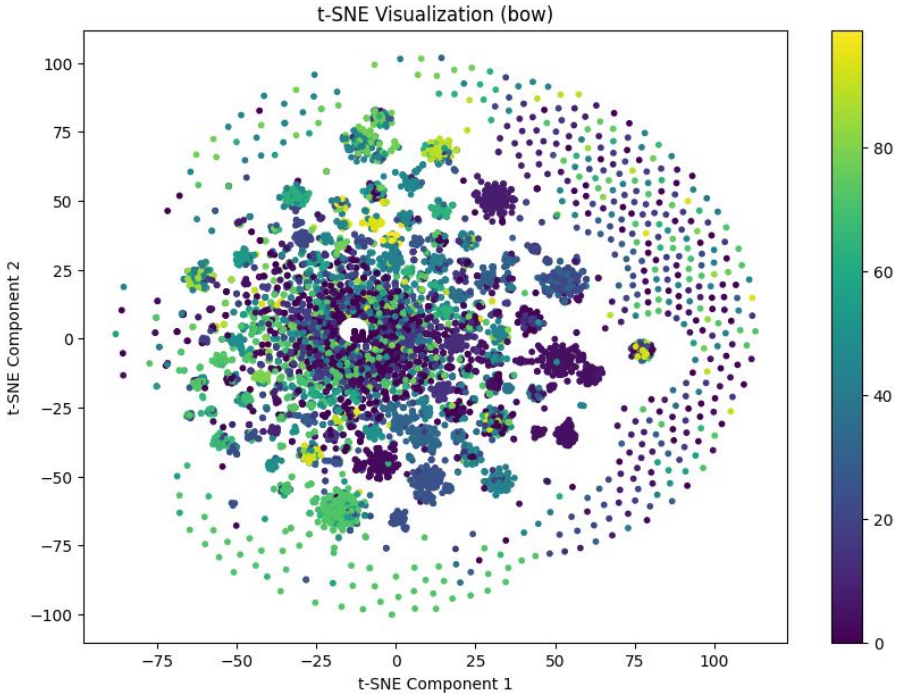
Couverture des tags : 0.000575

Couvertures des mots : 0.005374

Couvertures des mots “cleaned” : 0.016

Ici, le problème, c'est que le modèle ne déduit pas des mots dans les topiques si celui-ci n’est pas expressément écrit dans le texte.
Afin que le modèle ne divague pas trop, on doit limiter le nombre de topiques et ainsi la compréhension du modèle.
Il est également beaucoup plus difficile d’ évaluer la performance sur de grande quantité de data

Exemples de visualisations

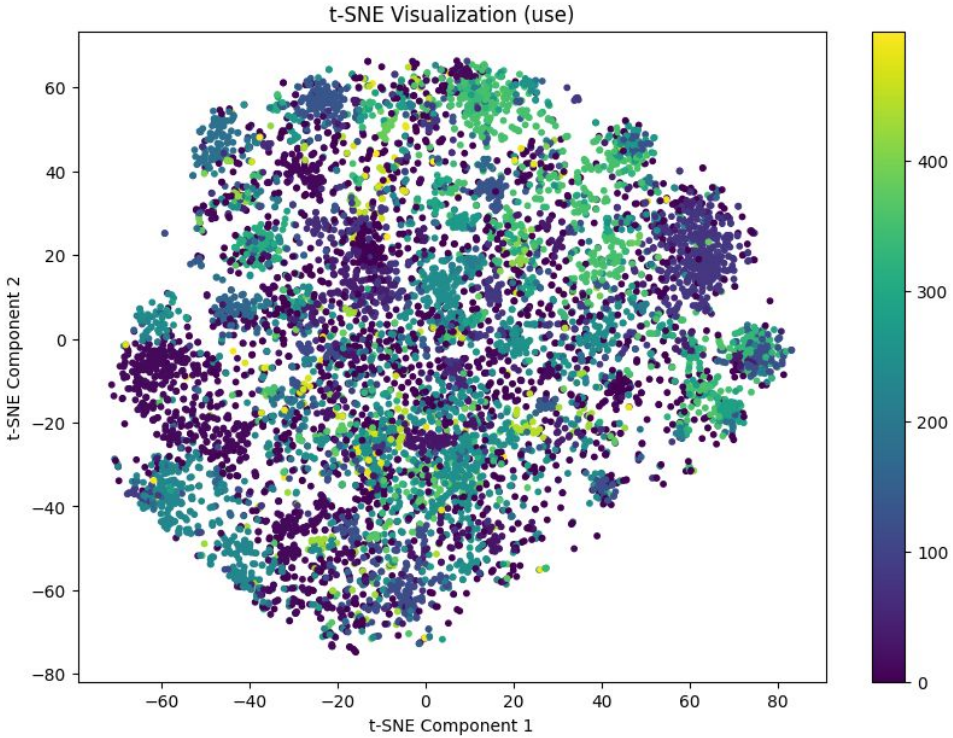


Jaccard Score (bow + Random Forest):
0.4538309892931202

Precision@10 (bow + Random Forest):
0.14223363286264443

Recall@10 (bow + Random Forest):
0.8179832790230736

Exemples de visualisations

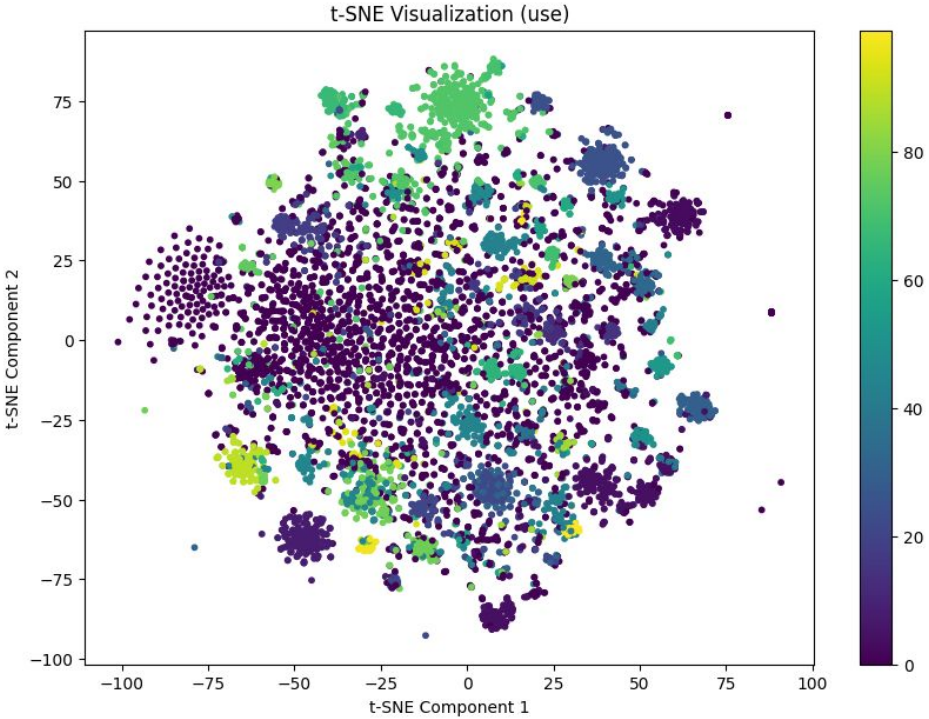


Jaccard Score (use + SGD Classifier):
0.38703141382997497

Precision@10 (use + SGD Classifier):
0.23996075866579464

Recall@10 (use + SGD Classifier):
0.8193227236392704

Exemples de visualisations

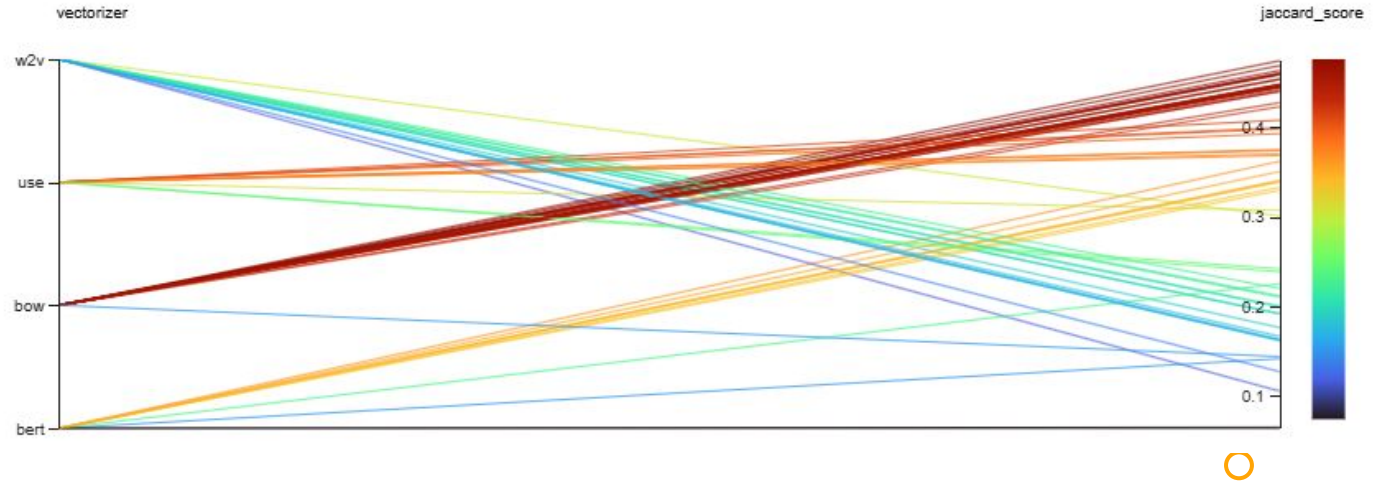


Jaccard Score (use + Logistic Regression): 0.375234521575985

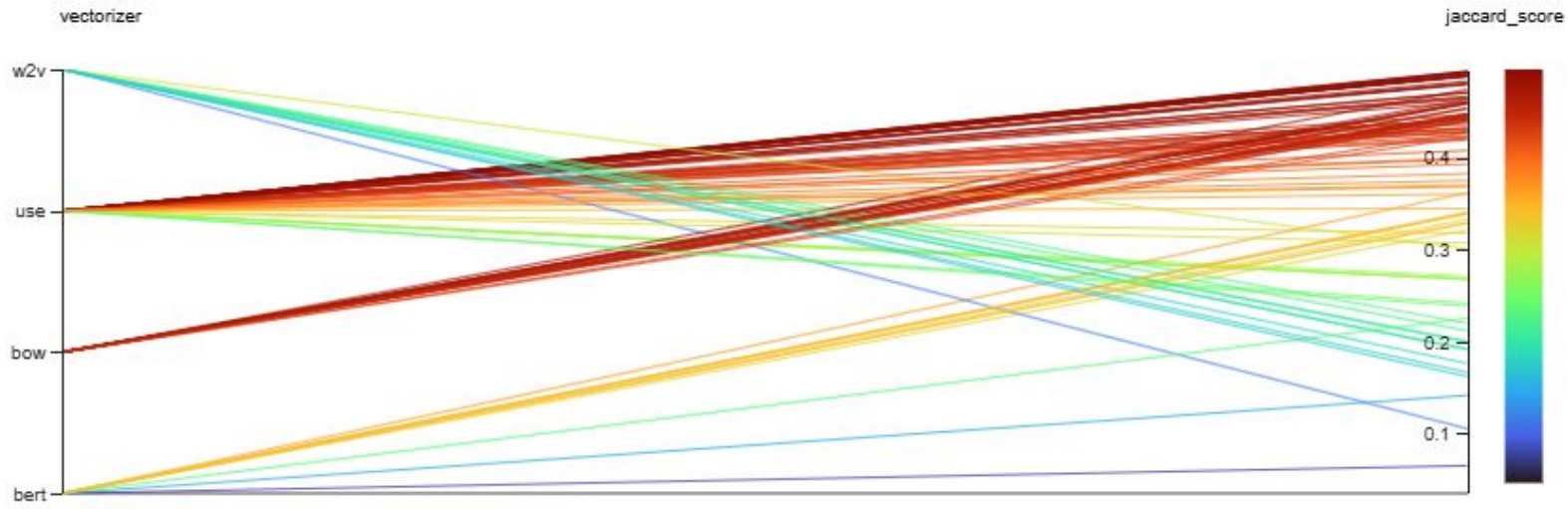
Precision@10 (use + Logistic Regression): 0.14390243902439026

Recall@10 (use + Logistic Regression): 0.8274711168164315

Experiments (sans body)

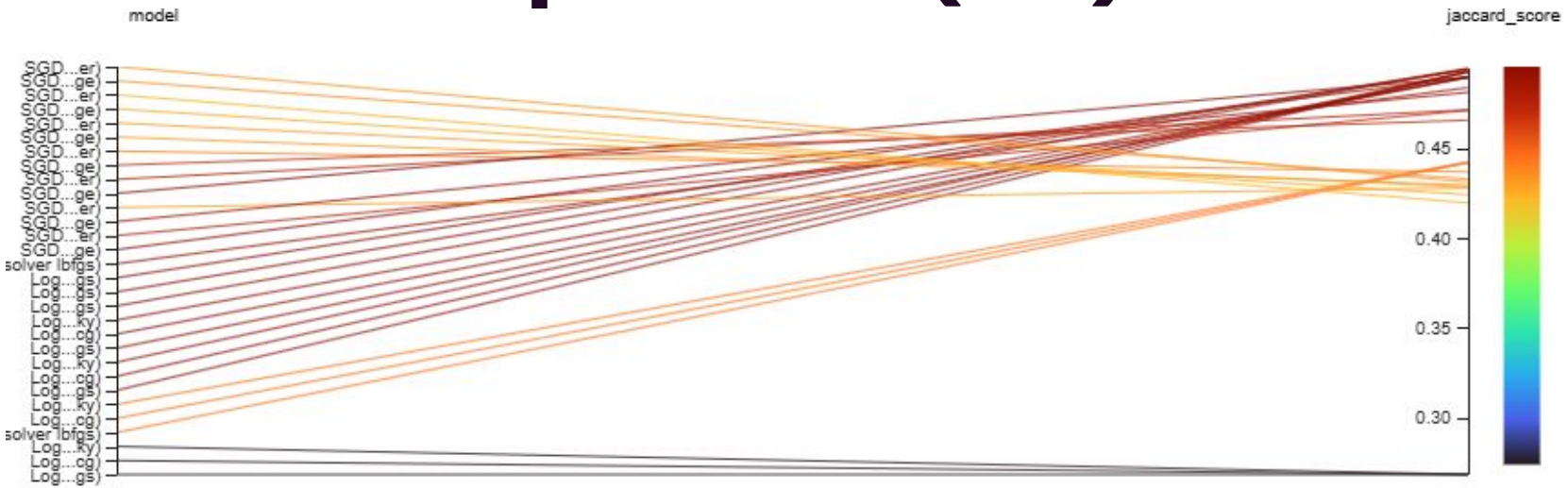


Experiments (avec body)



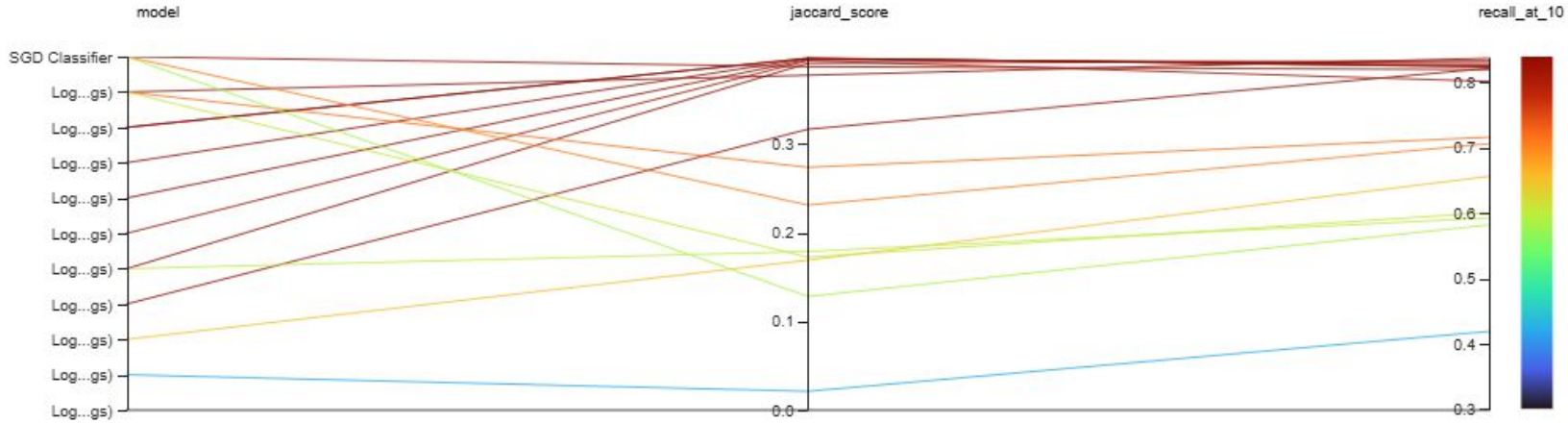
Use s'améliore beaucoup avec plus de contexte et devient vraiment performant

Experiments (USE)



Après test de différents modèles avec plusieurs hyper paramètre le plus performent semble être : LogisticRegression(C 20, penalty l2, solver lbfgs)
score : jaccard_score 0.494
precision_at_10 0.193
prediction_duration (s) 3.467
recall_at_10 0.941
training_duration (s) 251

Experiments (USE 500 tags)



Au vu du recall @ 10 très performant, j'ai décidé de perdre un peu de performance afin de gagner en compréhension globale avec plus de tags référencés et au vu des résultats, je décide de conserver cela.

Modèle final

Parameters (2)

Name	Value
model	LogisticRegression(C 25, penalty l2, solver lbfgs)
vectorizer	use

Metrics (5)

Name	Value
jaccard_score	0.397
precision_at_10	0.241
prediction_duration	18.2
recall_at_10	0.824
training_duration	834.5

04

Organisation du projet et réalisation

Docker et docker compose

Afin de build ce projet, d'avoir un environnement cohérent, peu importe sur quel os il est build, et la maintenance des servies nous buildons chaque services dans son propre container :

- Nginx : server web qui delivre notre interface de test, à terme, il peut filtrer les requêtes au besoin pour faire le lien entre le client et l'api (exemple : refuser les demandes directes sur l'api et ne servir que notre front)
- Flask_app : Api build avec flask qui permet de servir les clients en fonction des routes
- Mlflow : run l'instance de mlflow qui sert les modèles et aussi Mlflow UI pour le suivi des experiments et des modèles enregistrés

Chacun possède des dépendances et des healthcheck

Github actions

Afin de ne pas casser la branche master (principal), en cas de push ou de pull request sur celle-ci, une série de tests s'exécute :

- Un linter python vérifie la qualité du code push

- Les containers son build pour voir si les services se lancent correctement

- Via unittest les routes sont testées pour vérifier que l'api répond bien

- Les images docker sont enfin push sur docker hub afin de faciliter les prochains déploiements

05

Conclusion

Conclusion

Le traitement de la donne via un embedding USE, une classification multiclasse avec OneVsRest et Logistic regression pour 500 tags est plutôt performant.

Un environement dockeriser avec un suivi via mlflow et github action permet un suivie de la pipeline plus efficient.

Cependant, on pourrait réfléchir à certaine améliorations :

- Un suivi des versions pour la data
- Dans une infra hardware plus complexe, Kubernetes pourrait être plus performant, car il permettrait de repartir les worker sur différentes machines
- Avec la route de confirmation et un monitoring avec des outils comme Prometheus nous permettrait de suivre chaque sélection par les clients