Rapport du projet :

Catégorisez automatiquement des questions sur StackOverflow

Rédigé par ***Ralph ZOGO – Parcours Ingénieur Machine Learning***

SOMMAIRE

1- INTRODUCTION

Notre mission sera de catégoriser automatiquement des questions sur le célèbre site StackOverflow en leurs assignant des tags afin de pouvoir retrouver facilement les questions par la suite, pour cela nous allons mettre en place un système de suggestion de tags pour le site, celui-ci prendra la forme d'un algorithme de Machine Learning qui assigne automatiquement plusieurs tags pertinents à une question.
Stack Overflow propose un outil d'export de données "stackexchange explorer", qui recense un grand nombre de données authentiques de la plateforme d'entraide, nous nous servirons de cette plateforme pour extraire les différentes questions par le biais de requêtes SQL tout ceci dans le but d'obtenir les données nécessaires à l'entraînement de nos algorithmes.

```
SELECT  TOP 5000 Id, Body, Title, Tags FROM Posts ORDER BY CommentCount DESC
SELECT  TOP 5000 Id, Body, Title, Tags FROM Posts ORDER BY ViewCount DESC
SELECT  TOP 5000 Id, Body, Title, Tags FROM Posts ORDER BY AnswerCount DESC
SELECT  TOP 5000 Id, Body, Title, Tags FROM Posts ORDER BY Score DESC
```

Nous remarquons de ces requêtes,que nous avons extrait, les questions comportant le plus grand nombre de commentaire, le plus grand nombre de vues, le plus grand nombre de questions et le plus grand score.

2- Traitement des Données

Les données extraites sur "stackexchange explorer" seront conservées en fichier **csv,** pour être par la suite concatenées en un seul dataframe comme l'illustre la figure ci-dessus :

```
# Création du dataframe final via des jointures
d1 = pd.concat([data1, data2], ignore_index = True)
d2 = pd.concat([data3, data4], ignore_index = True)
query = pd.concat([d1, d2], ignore_index = True)
```

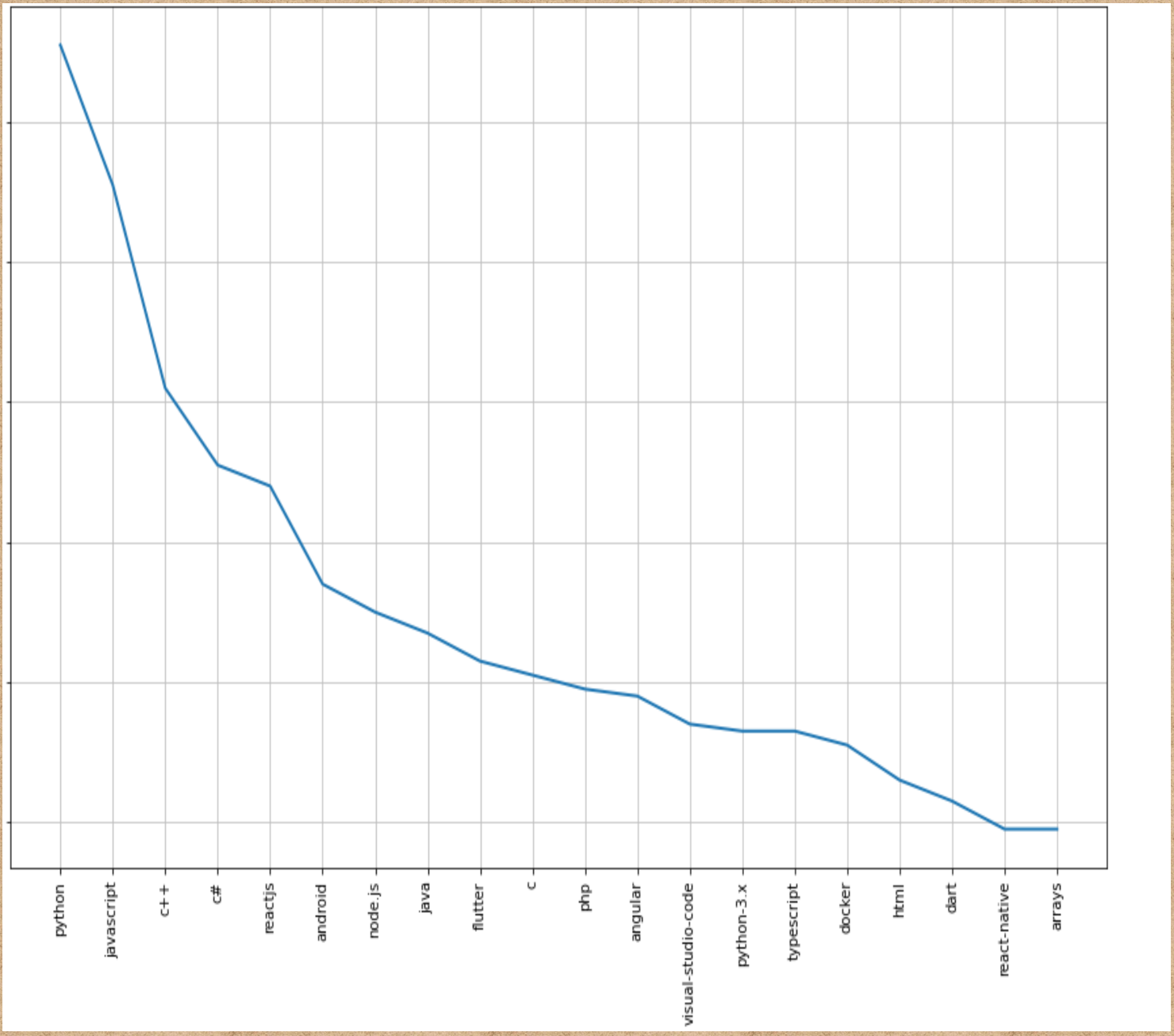
```
# Suppression des lignes dupliquées ou vides
query.dropna(inplace = True)
query.drop_duplicates(subset = ["Id"], inplace = True)
query = query.reset_index(drop = True)
```

Le jeu de données obtenu sera réduit à 1000 observations par le biais de sampling, permettant ainsi une implémentation rapide de nos algorithmes.

```
# On vas sampler le dataframe pour n'en utiliser que 1000 observations au  vue de la vitesse de mon processeur
query = query.sample(n=1000,replace=True)
query = query.reset_index(drop = True)
```

3- Analyse exploratoire des données

Ce graphique représente les 20 tags les plus utilisés dans notre jeu de données.



Ci-dessous, nous avons un **WordCloud**(nuage de mots-clés en français) qui est un outil utile pour synthétiser les mots les plus importants d'un texte, d'une page web ou encore d'un livre, plus un mot est présent dans notre jeu de données, plus il apparaît gros dans le WordCloud.

6 – *Catégorisez automatiquement des questions sur StackOverflow*

4- Traitement des variables (Feature Engineering)

Afin de pouvoir utiliser les données, nous devons au préalable les faire subir certaines transformations :
- **Concaténation des variables** « body » et « title » en la variable « text » :

```
# jointure entre les variables "Title" et "Body"
query["Text"] = query["Title"] + " " + query["Body"]
query.drop(columns = ["Body", "Title", "Id"], inplace = True)
```

- **Suppression des caractères** jugés inutiles et **tokenisation** (Opération qui consiste à découper un texte en tokens ou en mots) des questions:

```
# Conversion en liste
data = df.values.tolist()

# suppression des caractères inutiles
data = [re.sub('\S*@\S*\s?', '', sent) for sent in data]
data = [re.sub('\s+', ' ', sent) for sent in data]
data = [re.sub("\'", "", sent) for sent in data]
```

```
# tokenization des phrases
data_words = data.copy()
for i in range(len(data)):
    tokenizer = RegexpTokenizer(r'\w+')
    data_words[i] = tokenizer.tokenize(data[i])
```

- **Création des bigrammes, suppression des stopwords et lemmatisation**(réduction d’un mot à sa forme de base) :

```
# Construction du modèle de bigramme
bigram = gensim.models.Phrases(data_words, min_count=5, threshold=100)
bigram_mod = gensim.models.phrases.Phraser(bigram)
```

```
# Suppression des stopwords, mise en place des bigrammes et lemmatization

# Definition des fonctions pour les différentes opérations
def remove_stopwords(texts):
    return [[word for word in simple_preprocess(str(doc)) if word not in stop_words] for doc in texts]

def make_bigrams(texts):
    return [bigram_mod[doc] for doc in texts]

def lemmatization(texts, allowed_postags=['NOUN', 'ADJ', 'PROPN']):
    """https://spacy.io/api/annotation"""
    texts_out = []
    for sent in texts:
        doc = nlp(" ".join(sent))
        texts_out.append([token.lemma_ for token in doc if token.pos_ in allowed_postags])
    return texts_out
```

```
# Suppression des stopwords
data_words_nostops = remove_stopwords(data_words)

# Formation des bigrammes
data_words_bigrams = make_bigrams(data_words_nostops)

# Lemmatisation avec conservation des noms, noms propres et adjectifs
data_lemmatized = lemmatization(data_words_bigrams, allowed_postags=['NOUN', 'ADJ','PROPN'])
```

5- Méthode d’apprentissage non supervisée

L’approche non supervisée permettant l’extraction des tags est celle du **LDA**(**L**atent **D**irichlet **A**llocation) qui est modèle génératif probabiliste permettant d’expliquer des ensembles d’observations, par le moyen de groupes non observés, eux-mêmes définis par des similarités de données *(Source : WIKIPÉDIA).*
Pour notre projet nous avons utilisé le **ldaMulticore** qui est une alternative du LDA permettant une exécution rapide de l’algorithme grâce à l’usage de tous les cœurs du processeur.
Ici, L'algorithme d'entraînement :
- est **streamé** : les documents de formation peuvent arrivé de manière séquentielle,
- fonctionne en **mémoire constante** par rapport au nombre de documents : la taille du corpus d’apprentissage n’affecte pas l’empreinte mémoire et peut traiter des corpus plus volumineux que la RAM.

```
# Mise en place du modèle LDA
lda_model =LdaMulticore(corpus,
                        id2word=dict,
                        num_topics=20)
```

Les performances du LDA ont été boosté en ajustant les hyperparamètres **chunksize** et **passes**, grâce à cela ont est quitté de **0,35** à **0,48** de **coefficient de cohérence** ce qui représente tout de même une nette augmentation.

6- Méthode d’apprentissage supervisée

Dans le cadre de ce projet, nous sommes face à un cas de classification multi-classe mais nous allons utiliser un algorithme de régression logistique qui est un algorithme conçu pour les tâches de classification binaire, ce tour de passe passe sera possible via des méthodes heuristiques(**one-vs-one** ou **one-vs-rest**) qui permettent de transformer un problème de classification multi-classe en plusieurs ensembles de données de classification binaire et de former chacun un modèle de classification binaire.
Pour la suite du projet, nous utiliserons le one-vs-one qui contrairement au one-vs-rest qui le divise le jeu de donnée en un jeu de données binaires pour chaque classe, l'approche du one-vs-one divise le jeu de données en un jeu de données pour chaque classe par rapport à chaque autre classe.
Le modèle de base donnera :

```
# Mise en place de l'algorithme LogisticRegression via OneVsRest

model = LogisticRegression()
ovr = OneVsRestClassifier(model)
ovr.fit(X_train, Y_train)
y_pred2 = ovr.predict(X_test)
print("La fraction d'étiquettes qui sont mal prédites est de", round(hamming_loss(Y_test, y_pred2),3))
```

```
La fraction d'étiquettes qui sont mal prédites est de 0.161
```

Après optimisation de l’hyperparamètre C on obtient :

```
# Modèle optimisé

model = LogisticRegression(C = 10)
ovr = OneVsRestClassifier(model)
ovr.fit(X_train, Y_train)
y_pred2 = ovr.predict(X_test)
print("La fraction d'étiquettes qui sont mal prédites est de", round(hamming_loss(Y_test, y_pred2),3))
```

```
La fraction d'étiquettes qui sont mal prédites est de 0.147
```

7- Mise en place de l’API via FastAPI

Pour le déploiement de notre modèle, nous avons utilisé **FastAPI** qui est framework pour la création d’API (synchrone ou pas) avec python. Le framework a été créé par Sebastian Ramirez en 2018. Et bien qu’il soit jeune, il tient déjà de grandes promesses, et souhaite moderniser les frameworks orientés API.

Pour cela, le framework nouvelle génération mise sur plusieurs points clés :

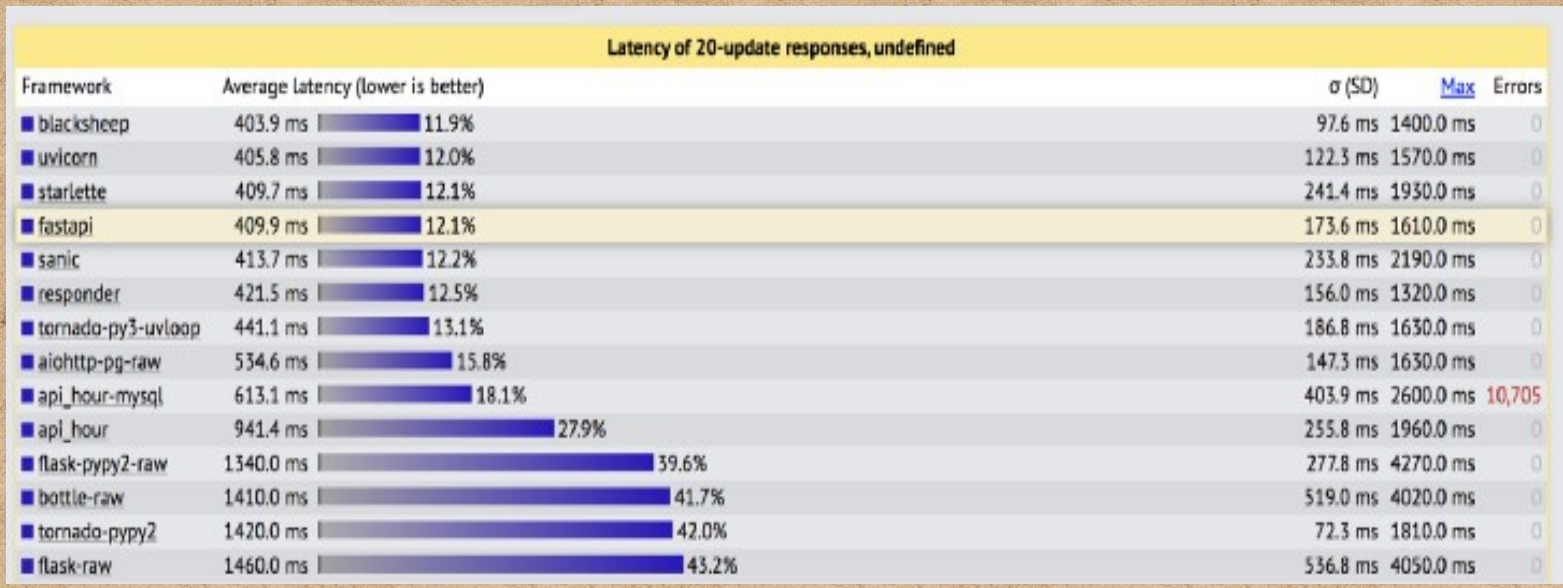
- Rapidité : des performances à me de rivaliser avec NodeJS, Go et Flask(grâce à **starlette** et **Pydantics**).

- Développement : simplicité de python et framework intuitif.

- Robuste : code prêt pour la production.

- Standartisé : basé (et complétement compatible) avec des standards open source, notamment **OpenAPI** et **JSON schem**a.

- Documentation : pas besoin d’ajout de code, elle est automatiquement générée à partir de l’existant.



Latency of 20-update responses, undefined

Framework	Average Latency (lower is better)	σ (SD)	Max	Errors
blacksheep	403.9 ms 11.9%	97.6 ms	1400.0 ms	0
uvicorn	405.8 ms 12.0%	122.3 ms	1570.0 ms	0
starlette	409.7 ms 12.1%	241.4 ms	1930.0 ms	0
fastapi	409.9 ms 12.1%	173.6 ms	1610.0 ms	0
sanic	413.7 ms 12.2%	233.8 ms	2190.0 ms	0
responder	421.5 ms 12.5%	156.0 ms	1320.0 ms	0
tornado-py3-uvloop	441.1 ms 13.1%	186.8 ms	1630.0 ms	0
aiohttp-pg-raw	534.6 ms 15.8%	147.3 ms	1630.0 ms	0
api_hour-mysql	613.1 ms 18.1%	403.9 ms	2600.0 ms	10,705
api_hour	941.4 ms 27.9%	255.8 ms	1960.0 ms	0
flask-pypy2-raw	1340.0 ms 39.6%	277.8 ms	4270.0 ms	0
bottle-raw	1410.0 ms 41.7%	519.0 ms	4020.0 ms	0
tornado-pypy2	1420.0 ms 42.0%	72.3 ms	1810.0 ms	0
flask-raw	1460.0 ms 43.2%	536.8 ms	4050.0 ms	0

La structure du code FastAPI est très **similaire** à celle de **Flask**. Nous alons créer des points de terminaison(**endpoint**) où notre serveur client pourra faire des demandes et obtenir les données requises. Ci-dessous, nous avons une implémentation de notre code qui comportera deux blocs à savoir :
- Le programme « **main** »

```
from fastapi import FastAPI
from modules import predicted_tags, post_preprocessing
from pydantic import BaseModel

app = FastAPI(title='API pour prédiction de tags sur Stack Overflow',
              description='Elaboration de Tags pertinents à une question sur Stack Overflow ',
              version='1.0.0')

@app.get("/")
def root():
    return {"Bienvenue sur l'API de prédiction de Tags"}

class Post(BaseModel):
    text : str

@app.post("/prediction")
def get_prediction(text: Post):
    a = post_preprocessing(text.text)
    tags = predicted_tags(a)
    return {'tags': list(tags) }
```

- Le programme « **modules** » qui contient les fonctions nécessaires à l'exécution du main.

```
def post_preprocessing(text):
    ''' Fonction pemettant le prétraitement du post afin qu'il soit utilisable par les algorithmes de
        machine learning,
        en entrée on a la question(text) pour obtenir
        en sortie un ensemble de mots de clés'''


    # Extraction du texte contenu dans le "Body" des balises
    b_soup = BeautifulSoup(text, 'lxml')
    text = b_soup.get_text()


    # Suppression des caractères jugés inutiles
    text = re.sub('\S*@\S*\s?', '', text)
    text = re.sub('\s+', ' ', text)
    text = re.sub("\'", "", text)

    # Tokenization avec suppression de ponctuation s'il y en a
    tokenizer = RegexpTokenizer(r'\w+')
    tok = tokenizer.tokenize(text)

    # Suppression des stopwords
    filtered_words = [word for word in tok if word not in stopwords.words('english')]

    # Lemmatization
    lemma= WordNetLemmatizer()
    lemmatized_word=[]
    for word in filtered_words :
        lemmatized_word.append(lemma.lemmatize(word))

    # Transformation de la liste de mots en chaîne de mots pour une exploitation ultérieure sur le modèle de machine learning
    processed_post = " ".join(lemmatized_word)

    return processed_post
def predicted_tags(txt):
    ''' Fonction qui permet la prédiction de tags par le biais de l'algorithme
        du OneVsRestClassifier(LogisticRegression(C= 10), en entrée on a un ensemble de mots clés
        et en sortie une prédiction des mots clés adéquats pour tagguer une question sur Stack Overflow'''

    a = txt.split(" ")
    file1 = open("vecto.pkl", "rb")
    tfid = pickle.load(file1)
    file1.close()

    b = tfid.transform(a)

    file2 = open("model.pkl", "rb")
    model = pickle.load(file2)
    file2.close()

    matrix = model.predict(b)
    preds = tfid.inverse_transform(matrix)
    tags = preds[0]
    return tags
```

7- Test de l’API via Thunder client

Nous utiliserons **l’extension Thunder client de Visual Code** pour envoyer une demande de publication à la route "**/prediction**" avec un corps de demande. Le corps de la requête contiendra les paires clé-valeur des paramètres et nous devrions nous attendre à une réponse JSON avec les tags souhaités.