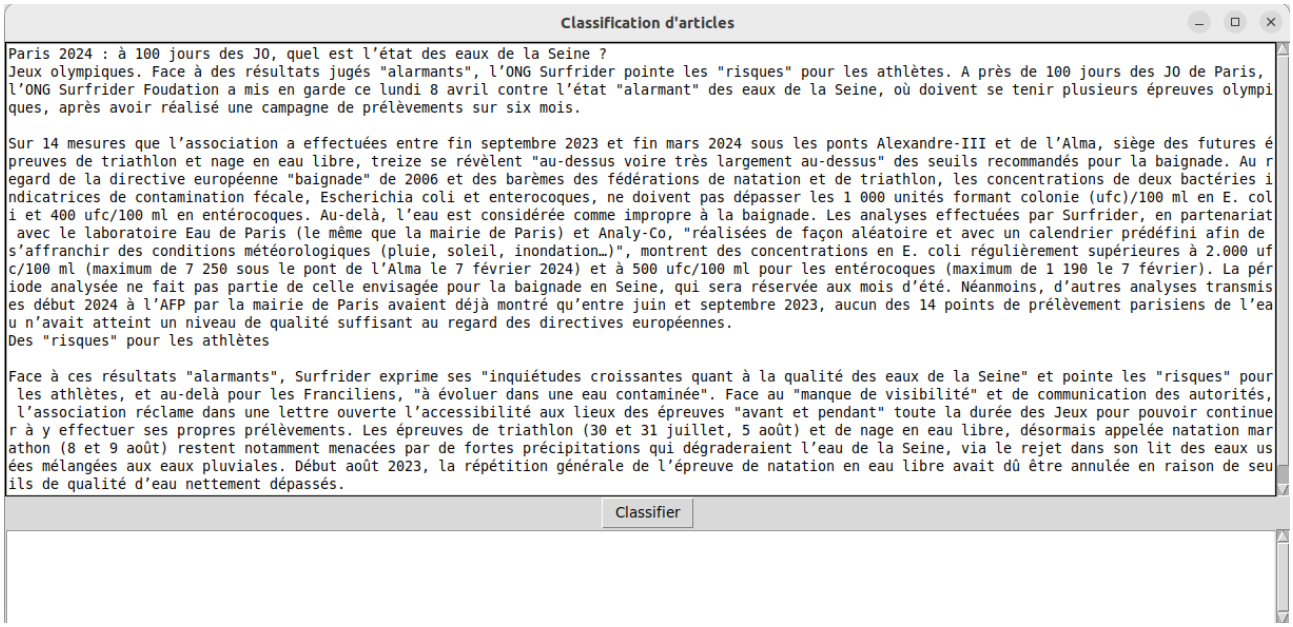
Projet : Classification d'Articles

Système de classification d'articles en catégories



L3 – Fouille de données, Ingénierie des langues et Développement de logiciel libre

Sommaire

Introduction

Ce projet de classification d'articles est réalisé dans le cadre d'un projet de fouille de données et d'ingénierie des langues de troisième année de licence d'informatique. Ce projet implique de la collecte de données en langage humain, du traitement automatique de la langue et de l'analyse de ces données.

Ce projet consiste un un logiciel qui permet de suggérer une catégorie d'articles parmi quatre catégories à partir d'un texte donné par l'utilisateur. Lorsque le texte n'est pas d'une des quatre catégories suivantes : économie, politique, sciences-sante et environnement, le logiciel pourra estimer qu'il est d'une catégorie 'Autre'. Le logiciel répond alors au besoin d'organiser des documents numériques de manière cohérente.

Architecture du Projet

Le langage de programmation utilisé pour ce logiciel est **Python**. Les librairies suivantes ont été utilisées :

- **MechanicalSoup :** permet d'envoyer des requêtes HTTP et de naviguer dans les pages web en utilisant un objet "browser" qui maintient l'état de la session. Cette librairie a permis de WebScrapper des articles en récupérant le code html des pages web visitées. Ces articles sont nécessaires à l'entrainement des modèles d'apprentissage.

- **Requests :** est utilisée pour créer une session persistante qui peut être réutilisée pour toutes les requêtes envoyées à travers MechanicalSoup, optimisant ainsi la gestion des connexions et le maintien de l'état entre les requêtes.

- **CSV :** utilisée pour enregistrer les données extraites lors du scraping dans un fichier CSV.

- **Pandas :** est utilisée pour lire, traiter, et écrire des fichiers CSV contenant des données d'articles provenant de différentes sources (L'Express, Le Monde, Slate). Elle permet de facilement filtrer, transformer, et agréger des données.

- **Re :** permet d'effectuer des recherches et des manipulations de chaînes de caractères à l'aide d'expressions régulières. Elle est utilisée pour supprimer les balises HTML des articles bruts.

- **Unidecodedata :** est utilisée pour retirer les accents des données d'articles.

- **spaCy** : bibliothèque pour le traitement du langage naturel. Elle est utilisée pour tokeniser le texte, retirer la ponctuation, les espaces, les mots vides (stop words) personnalisés et pour lemmatiser le texte.

- **scikit-learn**, composants utilisés :
    - **TfidfVectorizer** : Convertit une collection de documents bruts (les articles) en une matrice de caractéristiques TF-IDF pour avoir un format que les algorithmes de machine learning peuvent traiter.

    - **train_test_split :** Divise le dataset en sous-ensembles d'entraînement et de test.

- **SVC :** Implémente le modèle qui classe les textes en catégories.

- **make_pipeline** : Simplifie le processus de construction de pipelines de traitement en chaînant ensemble le Vectoriseur TF-IDF et le modèle d'apprentissage.

- **GridSearchCV :** Optimise les hyperparamètres d'un modèle en exécutant une recherche exhaustive sur une grille d'hyperparamètres spécifiée, en utilisant la validation croisée pour évaluer la performance.

- **OneClassSVM :** Implémente le modèle qui détecte les textes qui n'appartiennent pas aux catégories : politique, economie, sciences-sante et environnement.

- **joblib :** permet de sauvegarder et charger le modèle de machine learning entrainé.

Le développement a été réalisé sans IDE.
Le programme dispose d'une interface graphique réalisée avec Tkinter.

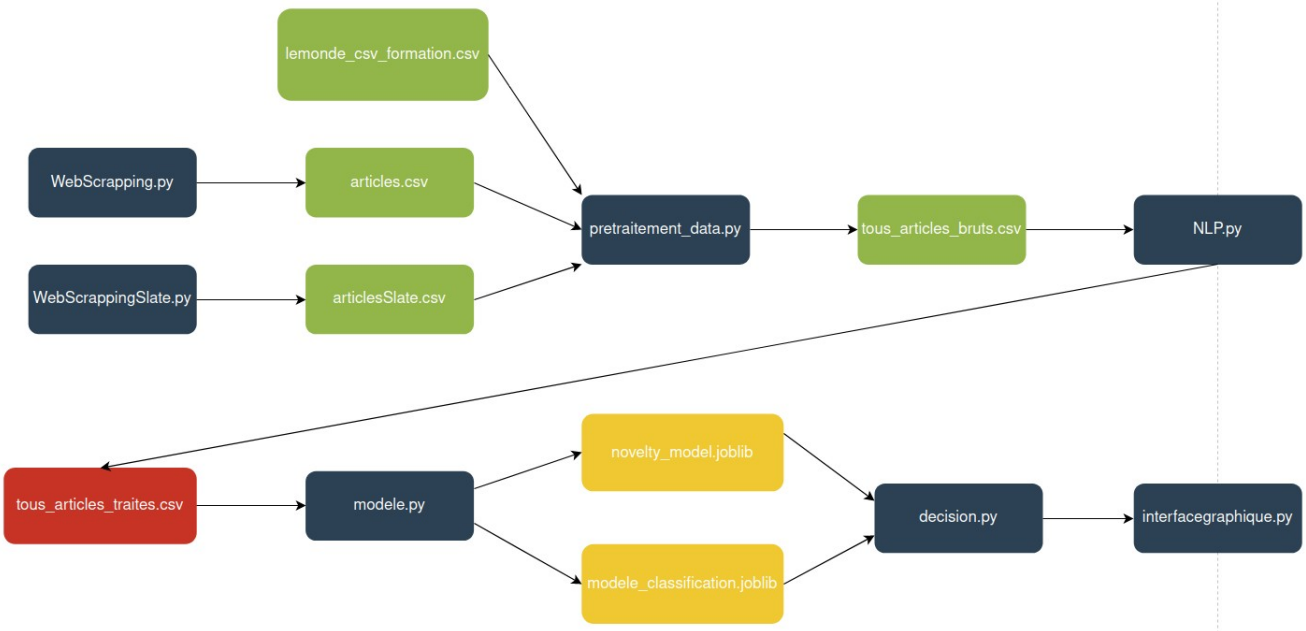


Figure 1 : Schéma des fichiers du projet

Décision de conception

Ce projet étant un projet de classification supervisée, il faut fournir aux modèles d'apprentissage utilisés une base de données sur laquelle s'entrainer. La source utilisée pour créer le dataset composé d'articles est "L'express" : https://www.lexpress.fr/ pour sa disponibilité d'articles gratuitement. Il existe une base de données d'articles du site "LeMonde" disponible sur kaggle mais les articles ont été récupérés sur une courte période (quelques mois de l'année 2016). Ils seront utiles ultérieurement.

Pour créer la base de données d'articles, il faut scraper les articles sur le site L'express. Concernant les articles, le choix suivant a été fait : scraper un maximum d'articles disponibles dans les "listes"

sur le site en partant du plus récent vers le plus ancien (environ 2020) au lieu d'aller dans les archives et choisir une journée de publication d'articles (de toutes catégories) puis récuperer tous les articles et le faire pour plusieurs journées. La première option aboutira à du machine learning sur une zone temporelle compacte et récente.

Les pages web contenant les listes d'articles d'une catégorie sont structurées de la manière suivante : une trentaine d'aperçu d'articles sont disponibles par page, il est possible de trouver les liens vers ces articles à partir de cette page. Pour avoir plus d'articles, nous faisons une nouvelle requête sur le serveur, vers la page suivante (la suite de la liste).
Pour cela, nous utilisons MechanicalSoup, qui permet de créer une instance d'un navigateur non graphique qui ouvre des liens. On peut ensuite récuperer l'html de la page avec l'attribut soup de l'objet MechanicalSoup. Les methodes find() et find_all() sont les principales pour trouver les balises voulues dans l'html.
A partir de la page contenant les listes, nous cherchons la balise qui contient la "fenêtre principale" d'information sur le site, ensuite nous récupérons les liens contenus dans cette fenêtre car ce sont des liens vers les articles. Ces liens sont présents en double (dans l'image et le texte de l'aperçu), nous cherchons alors seulement ceux du texte grâce à leur différence qui est la présence de l'attribut "class" dans celui du texte.
Pour ne récuperer que les articles gratuits, nous remarquons que les articles premium ont une indication spéciale sur la page web (un éclair jaune), et dans le code html cela se traduit par un attribut "class" égal à une certaine valeur. Nous décidons de ne pas récuperer les liens dont l'attribut "class" est égal à cette valeur.

Pour finir, nous utilisons les url de chacun des articles d'une catégorie pour les ouvrir avec une instance de mechanicalsoup.Browser() et nous récuperons leur titre, leur date de publication, et le contenu de l'article. Nous écrivons ensuite ces données dans un fichier CSV.

Nous supprimons les entrées dans notre dataset dont les articles n'ont pas de contenu grâce à la librairie Pandas, qui est très efficace pour traiter des données dans un fichier CSV.

Traitement du langage naturel

Le traitement du langage naturel (NLP) est essentiel dans la classification des articles en catégories, car il permet aux modèles de machine learning d'analyser les nuances du texte. Grâce à des techniques de prétraitement et d'extraction de caractéristiques adaptées, la NLP facilite la conversion des textes bruts en formats structurés, rendant possible une classification précise et efficace.

Pour le traitement du français, spaCy a été choisi pour les raisons suivantes. La librairie spaCy offre de bonnes performances et une grande précision pour les langues comme le français et fournit des modèles pré-entraînés spécifiques au français. De plus, elle est conçue pour être utilisé dans des applications en temps réel, ce qui est idéal pour un système de classification automatique.

Plusieurs modèles d'apprentissage ont été essayés dans le cadre de ce projet. Suivant le modèle d'apprentissage testé, différents traitements du langage produisent différents résultats de prédiction. Nous présentons dans la suite tous les traitements testés. Les tests spécifiques pour chaque modèle seront présentés ultérieurement.

La librairie SpaCy est utilisée pour retirer du texte une liste de mots pérsonnalisée, mais aussi retirer la ponctuation et les espaces du texte. Enfin, SpaCy lemmatise les token restants.

De plus, le traitement implique une fonction qui retire les balises html dans le texte récupéré et une autre qui retire les accents. Plusieurs expressions régulières sont utilisées pour gérer le texte, il y a notamment des expressions régulières qui :

- transforme le contenu des articles en minuscule
- garde uniquement les caractères ASCII ou accentués
- retire les “mots” qui commencent par des chiffres et contiennent des lettres
- retire les nombres

TF-IDF

La “Term Frequency” (TF ou fréquence de termes) est la fréquence d'un mot dans un document. Elle indique l'importance du mot dans ce document spécifique. En pratique, c’est le nombre de fois qu'un mot apparaît dans le document divisé par le nombre total de mots dans ce document.

L’ “Inverse Document Frequency” (IDF ou inverse de la fréquence dans les documents) mesure l'importance d'un mot en tenant compte de la fréquence à laquelle il apparaît dans l'ensemble du corpus. Les mots qui apparaissent fréquemment dans de nombreux documents seront considérés comme moins importants. L'IDF d'un mot est calculé comme le logarithme du nombre total de documents divisé par le nombre de documents contenant le mot.

Enfin, le score TF-IDF d'un mot dans un document est le produit de sa fréquence de terme (TF) et de son inverse de fréquence de document (IDF). Ce score représente l'importance d'un mot dans un document par rapport à un corpus.

Un vecteur TF-IDF est nécessaire dans la classification de textes. Il permet de représenter chaque article sous forme de vecteur pondéré, ce qui met en évidence les mots les plus pertinents et réduit l'impact des termes fréquemment utilisés mais peu significatifs. En outre, ce vecteur est important car il peut être fourni aux modèles d’apprentissage pour l’entrainement et la prédiction.

Dans ce projet, nous appliquons alors un vectoriseur TF-IDF à notre ensemble de données traité. Après application du vectoriseur TF-IDF sur les articles et titres, nous nous rendons compte en affichant une partie du vecteur que certains mots sont des caractères spéciaux, notamment des caractères d’autres alphabets.

```
# Convertir la matrice TF-IDF en DataFrame pour une meilleure lisibilité
tfidf_df = pd.DataFrame(tfidf_matrix.toarray(), columns=tfidf_vectorizer.get_feature_names_out())

# Afficher les scores TF-IDF pour les premiers documents
print(tfidf_df.head())
```

```
    00       000  0000  0002  001  002  0024_new3769ct  003  004  005  ...  \
0  0.0  0.000000   0.0   0.0  0.0  0.0             0.0  0.0  0.0  0.0  ...
1  0.0  0.108453   0.0   0.0  0.0  0.0             0.0  0.0  0.0  0.0  ...
2  0.0  0.000000   0.0   0.0  0.0  0.0             0.0  0.0  0.0  0.0  ...
3  0.0  0.027283   0.0   0.0  0.0  0.0             0.0  0.0  0.0  0.0  ...
4  0.0  0.000000   0.0   0.0  0.0  0.0             0.0  0.0  0.0  0.0  ...

   תהיה  של  שיגיע  קשה  קיבלנו  קודמת  あと2日  打上げまで  TOUS  UNIS
0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
1   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
2   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
3   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
4   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

[5 rows x 40964 columns]
```

Figure 2: Mots trouvés dans la matrice TF-IDF

La question à se poser avec ces caractères/mots est la pertinence de leur présence dans le vecteur, ils pourraient ne pas être pertinents pour la classification et donc n'avoir aucune utilité dans le vecteur. Dans ce cas, ils sont une utilisation inutile de l'espace du vecteur.
Etant donné que les articles récupérés sont en français, nous pouvons commencer par supposer que ces caractères n'ont pas d'importance dans le but de classification de textes.
D'autres sont des chiffres/nombres sans apparente pertinence non plus. Pour connaitre la pertinence de ces termes, nous implémenterons le modèle avec puis sans, puis nous regarderons lequel présente les meilleurs résultats de prédiction.

Choix du modèle de classification multiclasse

Il existe plusieurs modèles qui sont adaptés à la tache de classification multiclasse. Parmis ceux-ci, plusieurs ont été testés dans le cadre de ce projet : la Machine à Vecteurs de Support (SVM), les Forêts Aléatoires, les Gradient Boosting Machines (GBM) et enfin BERT.

Forêt Aléatoire

Pour commencer, nous choisissons la forêt aléatoire avec ce code :

```python
df = pd.read_csv('Dataset/tous_articles_traites.csv', encoding='utf-8')
texts = df['Content'].tolist()
#creer le vecteur IF-IDF avec titre+article

y = df['Category']
tfidf_vectorizer = TfidfVectorizer()
X = tfidf_vectorizer.fit_transform(texts)

# Diviser les données en ensembles d'entraînement et de test
# ensemble de test 20%
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, shuffle=True)

modele = RandomForestClassifier(random_state=42)

# Entrainement
modele.fit(X_train, y_train)

y_pred = modele.predict(X_test)

# % reponses correctes
print("Précision :", accuracy_score(y_test, y_pred))
# vrais positifs correctement identifiés
print("Rappel :", recall_score(y_test, y_pred, average='macro'))  # 'macro' pour multi-classe
# moyenne harmonique des 2
print("Score F1 :", f1_score(y_test, y_pred, average='macro'))

# Matrice de confusion
print("Matrice de confusion :\n", confusion_matrix(y_test, y_pred))
```

Figure 3 : Code du modèle Forêt Aléatoire

Les résultats qui suivent sont présentés les uns à la suite des autres en gardant les modifications de code qui augmentent les performances par rapport aux résultats précedents, pour la suite des résultats. A l'inverse les modifications qui diminuent les performances par rapport aux résultats précédents sont retirées pour les codes suivants (et donc pour les résultats suivants).

Ces résultats sont présentés à l'aide de trois métriques :

Precision : C'est le rapport entre le nombre de prédictions correctes (vrai positif + vrai négatif) et le nombre total de prédictions. Il est calculé ainsi : (vrai positif + vrai négatif)/ nombre de prédictions total.

Rappel : Le rappel est le rapport entre le nombre de prédictions correctes pour une classe et le nombre total d'instances appartenant à cette classe (vrai positif / (vrai positif + faux négatif)). Un rappel élevé signifie que le modèle est capable de trouver la plupart des échantillons appartenant à la classe cible.

Score F1 : Le score F1 est la moyenne harmonique de la précision et du rappel (2 * (Précision * Rappel) / (Précision + Rappel)). C'est une mesure équilibrée qui prend en compte les deux métriques. Le score F1 est particulièrement utile lorsque les classes sont déséquilibrées ou lorsque l'on veut éviter d'accorder plus d'importance à l'une des deux métriques.

Résultats :

Précision : 0.8018433179723502
Rappel : 0.8020360571193859
Score F1 : 0.8014858058758104
Matrice de confusion :
[[87 14 12 3]
[13 75 8 1]

[9 2 93 3]
[4 10 7 93]]

En retirant les mots commençant par des chiffres :

Précision : 0.7949308755760369
Rappel : 0.7957518160623442
Score F1 : 0.7940237866708455
Matrice de confusion :
[[83 15 15 3]
[12 75 7 3]
[5 5 94 3]
[5 8 8 93]]

En retirant les accents :

Précision : 0.804147465437788
Rappel : 0.8052168000651881
Score F1 : 0.8036444926384004
Matrice de confusion :
[[85 15 13 3]
[11 77 7 2]
[8 2 94 3]
[3 10 8 93]]

En retirant les nombres :

Précision : 0.815668202764977
Rappel : 0.8150887729813219
Score F1 : 0.8147227318194245
Matrice de confusion :
[[91 11 12 2]
[12 74 7 4]
[7 1 96 3]
[6 8 7 93]]

Puis en gardant seulement les mots composés uniquement de caractères alphabétiques :

Précision : 0.8018433179723502
Rappel : 0.8009887018245635
Score F1 : 0.8003515367255539
Matrice de confusion :
[[88 12 12 4]
[15 72 7 3]
[6 4 94 3]
[4 9 7 94]]

Avec ces résultats, nous pouvons essayer d'améliorer le modèle en ajustant les hyperparamètres. Pour ce faire, nous utilisons la recherche aléatoire (Random Search) pour trouver les meilleurs

hyperparamètres du modèle RandomForestClassifier, comme n_estimators, max_depth, min_samples_split, et min_samples_leaf.

Le code est alors complété par ces lignes :

```
df = pd.read_csv('Dataset/tous_articles_traites.csv', encoding='utf-8')
texts = df['Content'].tolist()
#creer le vecteur IF-IDF avec titre+article

y = df['Category']
tfidf_vectorizer = TfidfVectorizer()
X = tfidf_vectorizer.fit_transform(texts)

# Diviser les données en ensembles d'entraînement et de test
# ensemble de test 20%
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, shuffle=True)

modele = RandomForestClassifier(random_state=42)

# Définit l'espace des hyperparamètres à tester
param_distributions = {
    'n_estimators': [50, 100, 200, 300, 400],
    'max_depth': [None, 10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10, 15, 20],
    'min_samples_leaf': [1, 2, 4, 6, 8]
}

# Configurer la recherche aléatoire
random_search = RandomizedSearchCV(estimator=modele, param_distributions=param_distributions, n_iter=100,
cv=5, n_jobs=-1, verbose=2, random_state=42, scoring='accuracy')

# Exécuter la recherche aléatoire
random_search.fit(X_train, y_train)
meilleur_modele = random_search.best_estimator_

y_pred = meilleur_modele.predict(X_test)

# % reponses correctes
print("Précision :", accuracy_score(y_test, y_pred))
# vrais positifs correctement identifiés
print("Rappel :", recall_score(y_test, y_pred, average='macro'))  # 'macro' pour multi-classe
# moyenne harmonique des 2
print("Score F1 :", f1_score(y_test, y_pred, average='macro'))

# Matrice de confusion
print("Matrice de confusion :\n", confusion_matrix(y_test, y_pred))
```

Figure 4: Code des Forêts Aléatoires avec ajustement des hyperparamètres

Avec ces ajustements, les résultats du modèle sur l'ensemble de test n'étaient pas suffisants (environ 0.82 sur les différentes métriques), alors nous avons décidé d'augmenter la taille du dataset. Un dataset lemonde trouvé sur kaggle a été ajouté pour compléter les catégories qui manquaient d'articles dans le dataset L'Express (sans le sous-echantillonage à 542).
Cependant, ce dataset n'a pas d'articles "science-sante" alors il a fallu scrapper des articles de ces 2 catégories sur Slate : https://www.slate.fr/.
En ajoutant ces articles, nous nous retrouvons avec un dataset de 1169 articles de chaque catégorie.

Avec le nouveau dataset et sans traitements du langage:

Précision : 0.790625
Rappel : 0.787755307593201
Score F1 : 0.7886805859615963
Matrice de confusion :
[[201 24 22 7]
[32 149 11 20]
[33 12 197 8]
[10 12 10 212]]

En retirant les accents :

Précision : 0.7980769230769231
Rappel : 0.7981813642627897
Score F1 : 0.7971159954807685
Matrice de confusion :
[[178 16 19 6]
[30 168 16 21]
[23 8 198 4]
[9 29 8 203]]

La mise en minuscules ne fait aucune différence et le fait de retirer les caractères non (ASCII + caractères accentués) donne de moins bons résultats, retirer les mots qui commencent par des chiffres et contiennent des lettres et nombres seuls donne de moins bons résultats aussi.

Ajustement hyperparamètres :

Précision : 0.7895299145299145
Rappel : 0.7891365068240208
Score F1 : 0.7883424444606195
Matrice de confusion :
[[169 22 23 5]
[35 166 16 18]
[21 6 203 3]
[7 34 7 201]]

SVM

Après ces résultats, la décision prise a été le changement de modèle car la limite de la forêt aléatoire est atteinte. Nous essayons alors le SVM :

```python
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score, confusion_matrix
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import make_pipeline
from sklearn.svm import SVC
from joblib import dump
from sklearn.metrics import classification_report

# Chargement des données
df = pd.read_csv('tous_articles_traites.csv', encoding='utf-8')
texts = df['Content'].tolist()
y = df['Category']

# Diviser les données en ensembles d'entraînement et de test
# ensemble de test 20%
X_train, X_test, y_train, y_test = train_test_split(texts, y, test_size=0.2, random_state=42, shuffle=True)

# Créer le modèle
pipeline = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

#pipeline.fit(X_train, y_train)

# Définir la grille des hyperparamètres à tester
parameters = {
    'tfidfvectorizer__max_df': (0.5, 0.75, 1.0),
    'tfidfvectorizer__ngram_range': [(1, 1), (1, 2)],  # unigrammes ou bigrammes
    'svc__C': [0.1, 1, 10, 100],
}
# Créer l'instance de GridSearchCV
grid_search = GridSearchCV(pipeline, parameters, cv=5, n_jobs=-1, verbose=1, scoring='accuracy')

grid_search.fit(X_train, y_train)
print("Meilleur score de validation croisée (accuracy) : {:.3f}".format(grid_search.best_score_))

meilleur_modele = grid_search.best_estimator_
y_pred = meilleur_modele.predict(X_test)

# % reponses correctes
print("Précision :", accuracy_score(y_test, y_pred))
# vrais positifs correctement identifiés
print("Rappel :", recall_score(y_test, y_pred, average='macro'))  # 'macro' pour multi-classe
# moyenne harmonique des 2
print("Score F1 :", f1_score(y_test, y_pred, average='macro'))

# Matrice de confusion
print("Matrice de confusion :\n", confusion_matrix(y_test, y_pred))
```

Figure 5: Code du SVC avec ajustement des hyperparamètres

Résultats (sans ajustements des hyperparamètres) :

Précision : 0.8290598290598291
Rappel : 0.8284667098273144
Score F1 : 0.8287239761819258
Matrice de confusion :
[[178 22 15 4]
[23 191 5 16]
[29 6 195 3]
[8 23 6 212]]

SVM (Ajustement hyperparamètres avec GridSearchCV) :

Meilleur score de validation croisée (accuracy) : 0.850
Précision : 0.8461538461538461
Rappel : 0.844849006984842
Score F1 : 0.8450349656949792
Matrice de confusion :

[[176 21 18   4]
 [22 194   5 14]
 [24   5 201   3]
 [3 21   4 221]]

Sans traitements avec les RegEx :

Précision : 0.8536324786324786
Rappel : 0.8521156659843321
Score F1 : 0.8522043221448066
Matrice de confusion :
[[178 16 17   8]
 [22 198   6   9]
 [26   6 197   4]
 [5 13   5 226]]

Nous revoyons la NLP.

sans token.is_stop de SpaCy :

Meilleur score de validation croisée (accuracy) : 0.847
Précision : 0.8643162393162394
Rappel : 0.8638650152024556
Score F1 : 0.8637393357732706
Matrice de confusion :
[[193 15   9   2]
 [23 199   4   9]
 [30   6 191   6]
 [7 12   4 226]]

Sans lemmatisation :

Meilleur score de validation croisée (accuracy) : 0.848
Précision : 0.8589743589743589
Rappel : 0.8581302120580172
Score F1 : 0.8582307951645333
Matrice de confusion :
[[187 18 12   2]
 [20 202   4   9]
 [30   7 190   6]
 [10 11   3 225]]

Dans le code du SVC, nous utilisons un pipeline, qui est une chaîne de transformations de données et de modélisation qui s'exécute dans un ordre spécifique. L'idée est de simplifier le processus de codage en permettant l'exécution séquentielle de plusieurs étapes de traitement de données et de modélisation à travers un seul objet. Le pipeline est utile pour le prétraitement des données, la réduction de dimensionnalité, et l'application de modèles d'apprentissage automatique de manière automatisée.

En utilisant un pipeline, nous nous assurons que toutes les étapes de prétraitement sont appliquées de manière cohérente à la fois sur les données d'entraînement et de test, évitant ainsi des erreurs courantes telles que la fuite de données lors de la validation croisée.

SVC avec noyau linéaire

Le SVC (Support Vector Classifier) est un type de SVM utilisé pour la classification. Les SVM sont des modèles d'apprentissage supervisés qui cherchent à trouver l'hyperplan qui sépare le mieux les différentes classes dans l'espace des caractéristiques. Ils sont particulièrement efficaces dans les espaces de haute dimension et quand la frontière de décision n'est pas nécessairement linéaire.

Le "noyau" dans un SVM est une fonction utilisée pour transformer l'espace des caractéristiques afin de faciliter la séparation linéaire des données. Un "noyau linéaire" signifie que l'on cherche une séparation linéaire dans l'espace des caractéristiques original, sans transformation supplémentaire. En d'autres termes, le modèle essaie de trouver l'hyperplan qui sépare les classes avec la plus grande marge possible dans l'espace original, sans appliquer de transformations non linéaires sur les données.

Hyperparamètres

Les hyperparamètres sont importants pour ajuster le comportement du vectoriseur de texte TfidfVectorizer et du modèle de classification SVC (Support Vector Classifier). Chacun de ces paramètres influence différemment la performance du modèle d'apprentissage automatique.

Le max_df est utilisé pour éliminer les termes qui apparaissent trop fréquemment, aussi considérés comme non informatifs. Il s'agit d'un seuil pour ignorer les termes qui ont une fréquence documentaire (document frequency) plus élevée que la valeur spécifiée.
Il peut être un nombre réel entre 0.0 et 1.0, représentant un pourcentage, ou un nombre entier pour spécifier un nombre exact de documents. Par exemple, max_df=0.5 signifie "ignorer les termes qui apparaissent dans plus de 50% des documents".

Le ngram_range définit les limites des différentes tailles d'n-grams à inclure dans les vecteurs de caractéristiques. Un n-gram est une séquence de n mots consécutifs dans le texte.
C'est un tuple (min_n, max_n), où min_n est la taille minimale d'n-grams utilisés et max_n la taille maximale. Par exemple, ngram_range=(1, 2) inclura à la fois les unigrammes (mots uniques) et les bigrammes (paires de mots consécutifs).

Le paramètre C est un paramètre de régularisation pour le modèle SVC. Il contrôle la marge d'erreur tolérée dans la séparation des classes par l'hyperplan du modèle.
C'est un nombre réel positif. Des valeurs plus petites spécifient une régularisation plus forte, donc une tolérance plus élevée pour les erreurs de classification et une marge plus large, tandis que des valeurs plus élevées entraînent une régularisation moins stricte, c'est à dire moins de tolérance pour les erreurs, cherchant à classer correctement tous les exemples d'entraînement avec une marge plus étroite.
Il permet d'équilibrer la complexité du modèle et sa capacité à bien ajuster les données d'entraînement avec sa capacité à généraliser à de nouvelles données non vues. Un C trop élevé peut conduire à un surajustement, alors qu'un C trop faible peut conduire à un sous-ajustement.

GridSearchCV

GridSearchCV va chercher parmi un ensemble d'hyperparmamètres, les meilleurs pour notre modèle SVC.
La fonction prend en paramètre pipeline qui est l'estimateur ou le modèle sur lequel la recherche d'hyperparamètres est effectuée. Dans ce cas, le pipeline comprend un prétraitement avec TfidfVectorizer suivi d'un modèle de classification avec SVC.
Puis, elle prend en paramètre 'parameters', qui est une liste de dictionnaires spécifiant les hyperparamètres à tester et leurs plages de valeurs possibles.
Ensuite, cv=5 détermine la stratégie de validation croisée utilisée pour l'évaluation du modèle. Ici, cv=5 signifie que la validation croisée k-fold sera utilisée avec 5 folds. Cela signifie que les données seront divisées en 5 ensembles ou "folds", où le modèle sera entraîné sur 4 d'entre eux et testé sur le 5ème, et ce processus sera répété 5 fois, à chaque fois avec un ensemble différent utilisé comme test.
n_jobs=-1 spécifie le nombre de travaux à exécuter en parallèle. n_jobs=-1 signifie que tous les processeurs disponibles seront utilisés. Cela peut accélérer la recherche, surtout lorsqu'il y a de nombreux ensembles d'hyperparamètres à évaluer.
verbose=1 contrôle le niveau de verbosité; plus la valeur est élevée, plus il y a de messages. verbose=1 indique que des messages seront affichés pour indiquer la progression de la recherche.
scoring='accuracy' définit la métrique de performance à utiliser pour évaluer la qualité des modèles. Dans ce cas, 'accuracy' est utilisée, ce qui correspond au pourcentage de prédictions correctes parmi l'ensemble total de cas. La recherche de grille utilisera cette métrique pour comparer les performances des modèles avec différents ensembles d'hyperparamètres et sélectionnera celui qui produit la meilleure précision.

BERT

Le code pour l'implémentation du modèle neuronal BERT est le suivant :

```python
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd
from sklearn.model_selection import train_test_split
from transformers import AutoTokenizer, TFAutoModelForSequenceClassification
import numpy as np
from datasets import Dataset
import tensorflow as tf
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.losses import SparseCategoricalCrossentropy

# préparer un texte pour le modèle BERT
def convert_example_to_feature(text):
    return tokenizer(text,
                     padding="max_length", # ajoute des tokens de remplissage pour que tous les textes/tokenizations aient la même longueur
                     truncation=True, # Tronque les textes plus longs que max_length.
                     max_length=124, # longueur maximale des séquences
                     return_tensors="tf") # retourne des tenseurs TensorFlow

# Chargement des données
df = pd.read_csv('DataBrut/tous_articles_bruts.csv', encoding='utf-8')
texts = df['Content'].tolist()
y = df['Category']

# Diviser les données en ensembles d'entrainement et de test
# ensemble de test 20%
X_train, X_test, y_train, y_test = train_test_split(texts, y, test_size=0.2, random_state=42, shuffle=True)

# encode les etiquettes en valeur numerique
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
# transform pour garder le meme encodage que pour y_train
y_test_encoded = label_encoder.transform(y_test)

# tokenizer pré-entrainé pour BERT
tokenizer = AutoTokenizer.from_pretrained('bert-base-multilingual-cased')

#
X_train_encoded = [convert_example_to_feature(text) for text in X_train]
X_test_encoded = [convert_example_to_feature(text) for text in X_test]

# Création des datasets Tensorflow
train_dataset = tf.data.Dataset.from_tensor_slices(({"input_ids": [feature["input_ids"].numpy()[0] for feature in X_train_encoded],
                                                     "attention_mask": [feature["attention_mask"].numpy()[0] for feature in X_train_encoded]},
                                                    y_train_encoded))

test_dataset = tf.data.Dataset.from_tensor_slices(({"input_ids": [feature["input_ids"].numpy()[0] for feature in X_test_encoded],
                                                    "attention_mask": [feature["attention_mask"].numpy()[0] for feature in X_test_encoded]},
                                                   y_test_encoded))

# Préparation pour l'entrainement
BATCH_SIZE = 8
# melange a chaque epoque
train_dataset = train_dataset.shuffle(len(X_train_encoded)).batch(BATCH_SIZE).prefetch(tf.data.experimental.AUTOTUNE)
test_dataset = test_dataset.batch(BATCH_SIZE).prefetch(tf.data.experimental.AUTOTUNE)

# Chargement du modèle BERT pré-entraîné pour TensorFlow
model = TFAutoModelForSequenceClassification.from_pretrained('bert-base-multilingual-cased', num_labels=len(set(y_train_encoded)))

# Configuration de l'optimiseur et compilation du modèle
optimizer = Adam(learning_rate=5e-5)
model.compile(optimizer=optimizer, loss=SparseCategoricalCrossentropy(from_logits=True), metrics=['accuracy'])

# Entraînement
model.fit(train_dataset, epochs=6, validation_data=test_dataset)

# Évaluation
results = model.evaluate(test_dataset)
print("Test loss, Test accuracy:", results)
```

Figure 6: Implémentation de BERT

Il débute par l'ouverture du fichier CSV avec le contenu des articles sans traitements du langage. Ce choix est vérifié par expérience, c'est-à-dire que c'est le dataset qui produit les meilleurs résultats.

Ensuite, les catégories sont encodées en valeurs numériques avec une instance de l'objet LabelEncoder, pour que le modèle puisse travailler dessus.

Nous chargeons ensuite un tokenizer préentrainé pour le modèle BERT et nous transformons les contenu d'articles avec ce tokenizer.

Nous devons ensuite créer des dataset Tensorflow avec ces tableaux. Les dictionnaires {"input_ids": ..., "attention_mask": ...} fournissent les entrées nécessaires au modèle BERT. Les input_ids sont les identifiants des tokens du texte, et attention_mask indique au modèle quels tokens doivent être pris en compte et lesquels sont des paddings. La conversion .numpy()[0] est utilisée pour extraire les valeurs numpy des tenseurs TensorFlow et s'assurer que les données sont au bon

format pour être utilisées dans le dataset. Cette conversion est nécessaire car les fonctions de tokenisation de Hugging Face retournent des tenseurs et nous avons besoin de les convertir en un format compatible avec tf.data.Dataset.

Par la suite, nous appliquons shuffle(len(X_train_encoded)) sur le datatset tensorflow, ce qui mélange aléatoirement les exemples dans l'ensemble d'entraînement. len(X_train_encoded) est utilisé comme taille de la mémoire tampon de mélange, ce qui signifie que le mélange est effectué sur la totalité de l'ensemble d'entraînement. Cela aide à réduire le risque de surapprentissage en s'assurant que le modèle ne rencontre pas les mêmes séquences d'exemples à chaque époque, ce qui favorise une meilleure généralisation.

.batch(BATCH_SIZE) regroupe les données en lots de taille spécifiée par BATCH_SIZE. Cela signifie que les données seront présentées au modèle en groupes de 8 exemples à la fois pendant l'entraînement et l'évaluation.

.prefetch(tf.data.experimental.AUTOTUNE) permet au pipeline de données de préparer de manière asynchrone le prochain lot pendant que le modèle travaille sur le lot actuel. tf.data.experimental.AUTOTUNE permet à TensorFlow de choisir automatiquement le nombre optimal de lots à prétraiter. Cela peut réduire les goulets d'étranglement potentiels et améliorer l'efficience de l'entraînement en réduisant le temps d'attente du GPU/TPU pour les nouvelles données.

Pour l'ensemble de test, le mélange n'est pas nécessaire, donc seulement batch et prefetch sont appliqués, suivant le même raisonnement que pour l'ensemble d'entraînement.

AutoTokenizer fait partie de la bibliothèque Transformers de Hugging Face et permet de charger automatiquement le tokenizer correspondant à un nom de modèle spécifié. AutoTokenizer peut charger BertTokenizer quand nous utilisons un modèle BERT, mais il est également compatible avec d'autres architectures de modèles pré-entraînés.

model = TFAutoModelForSequenceClassification.from_pretrained('bert-base-multilingual-cased', num_labels=len(set(y_train_encoded)))
est utilisée pour charger un modèle BERT pré-entraîné et l'adapter pour une tâche de classification de séquences avec un nombre spécifique de labels. Cela signifie que le modèle est configuré pour prédire un ensemble fixe de catégories basé sur l'entrée textuelle, ce qu'il nous faut pour des tâches comme la catégorisation de documents.

Adam(learning_rate=5e-5) utilise l'optimiseur Adam avec un taux d'apprentissage spécifique. Adam est utilisé pour l'entraînement des réseaux de neurones en raison de sa gestion efficace des taux d'apprentissage.

SparseCategoricalCrossentropy(from_logits=True) est une fonction de perte utilisée pour les problèmes de classification à plusieurs classes où les étiquettes sont des entiers, et from_logits=True signifie que la sortie du modèle ne passe par une fonction d'activation softmax.

metrics=['accuracy'] indique au modèle de suivre l'exactitude de la classification pendant l'entraînement.

Ci-dessous se trouvent les résultats de l'application du modèle dans différents contextes.

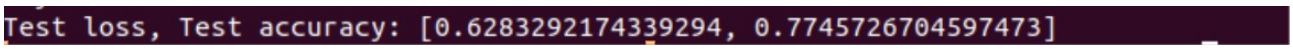


Figure 7 : BERT avec traitement NLP

Figure 8 : BERT sans traitement NLP

Figure 9 : BERT avec 210 max_length au lieu de 128

Figure 10 : BERT avec modele ‘bert-base-multilingual-cased’ sur dataset traité

Figure 11 : BERT avec modele ‘bert-base-multilingual-cased’ sur dataset brut

Figure 12 : BERT avec 4 epochs

La précision de BERT avec 10 epochs est 0.266 et avec 6 epochs est 0.7628.

Ces résultats étant inférieurs à ceux du SVC, nous utiliserons le SVC.

Interface graphique

La méthode .pack() est l'un des gestionnaires de géométrie de Tkinter. Elle organise les widgets en blocs avant de les placer dans le parent (la fenêtre). Par défaut, .pack() place les widgets verticalement dans leur conteneur parent, du haut vers le bas, et les centre horizontalement.

Dans ce cas, chaque widget sera ajouté l'un après l'autre, du haut vers le bas de la fenêtre, et centré par rapport à l'axe horizontal. C'est ce qui se passe dans notre code, où zone_texte, bouton_predire, et zone_resultat sont placés dans cet ordre.

classify est la méthode qui sera appelée lorsque l'utilisateur clique sur le bouton pour générer une prédiction.

self.zone_texte.get("1.0", "end-1c") récupère le texte saisi par l'utilisateur dans la zone_texte. "1.0" signifie à partir de la première ligne, premier caractère, et "end-1c" signifie jusqu'à la fin du texte moins un caractère pour éviter d'inclure le caractère de nouvelle ligne automatiquement ajouté à la fin du texte dans le widget.

self.zone_resultat.config(state=tk.NORMAL) configure la zone_resultat pour qu'elle soit modifiable avant de mettre à jour le texte.

self.zone_resultat.delete("1.0", tk.END) efface le contenu actuel de la zone_resultat. zone_resultat.insert(tk.END, 'Notre modèle de classification place votre texte dans la catégorie : ' + categorie_predite + ".")) insère le nouveau texte de prédiction dans la zone_resultat.

Enfin, zone_resultat.config(state=tk.DISABLED) remet la zone_resultat en état non modifiable, empêchant l'utilisateur de changer le texte de prédiction.

One Class SVM

Pour le repérage des textes qui n'appartiennent à aucune des catégories sur lesquelles s'est entrainé le SVC, nous entrainons un 'One Class SVM' qui est un modèle permettant de détecter de la "nouveauté" en s'entrainant sur le dataset qu'il considérera comme contenant des textes "normaux". Ce modèle sera ajouté en complément du SVC. Pour tester les capacités du double modèle, nous scrappons de nouveau les mêmes catégories qu'au début du projet sur le site l'express mais en bloquant la récupération des articles déjà récuperés auparavant. Pour les articles appartenant à des catégories sur lesquelles ne s'est pas entrainé le SVC, nous récupérons les articles non-utilisés du dataset lemonde trouvé sur kaggle. Nous utilisons les catégories sport et culture. Le dataset de test sera composé des deux datasets évoqués, échantillonés avec un nombre d'articles équivalents dans chacune des catégories. Ce nouveau dataset permettra de tester la performance du double modèle.

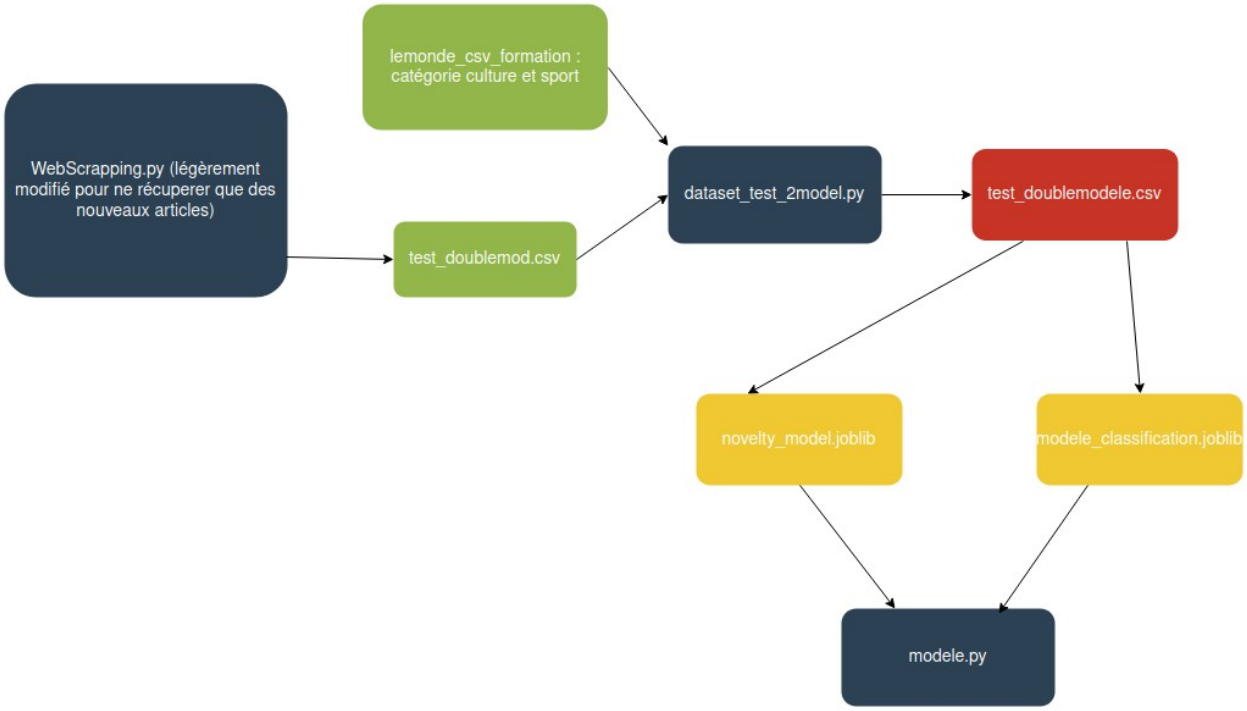


Figure 13 : Création du dataset de test des performances des deux modèles

Tout d'abord, testons le modèle One Class SVM seul pour connaitre sa performance de détection de nouveauté.

Utilisation de gridsearch pour trouver les meilleurs hyperparamètres du One Class SVM

```python
# Chargement des données
df = pd.read_csv('Dataset/tous_articles_traites.csv', encoding='utf-8')
texts = df['Content'].tolist()
y = df['Category']

#pipeline pour le modele qui detecte l'appartenance à aucune des catégories
oc_svm_pipeline = make_pipeline(TfidfVectorizer(), OneClassSVM(gamma='auto', nu=0.1))

# Définition de la grille des paramètres
param_grid = {
    'oneclasssvm__gamma': ['scale', 'auto', 0.1, 0.01, 0.001],
    'oneclasssvm__nu': [0.01, 0.05, 0.1, 0.2, 0.5]
}
y = [1 for cat in df['Category']]

grid_search = GridSearchCV(oc_svm_pipeline, param_grid, scoring=make_scorer(f1_score, pos_label=-1), cv=3)

# Entraîne le modèle One-Class SVM
grid_search.fit(texts, y)


test = pd.read_csv('Dataset/test_doublemodele.csv', encoding='utf-8')
X_test = test['Content'].tolist()

y_test = [-1 if cat in ['societe', 'sport', 'culture'] else 1 for cat in test['Category']]

predictions = grid_search.best_estimator_.predict(X_test)

# Évaluation des performances
print("Meilleurs paramètres:", grid_search.best_params_)
print("Score F1 pour anomalies:", f1_score(y_test, predictions, pos_label=-1))
print("Précision pour anomalies:", precision_score(y_test, predictions, pos_label=-1))
print("Rappel pour anomalies:", recall_score(y_test, predictions, pos_label=-1))
```

Figure 14 : Recherche des meilleurs hyperparamètres pour One Class SVM

Dans notre programme, nous configurons le Grid Search avec make_scorer(f1_score, pos_label=-1), pour indiquer que le Score F1 doit être calculé en considérant les anomalies (marquées avec -1 dans notre cas) comme la classe positive. Cela permet d'optimiser le modèle pour la performance sur les anomalies plutôt que sur les données normales.

```
doryan@doryan-Swift-SF314-42:~/L3Info/ProjetFDD_IDL/Fouille_Apprentissage$ pytho
n3 modele.py
Meilleurs paramètres: {'oneclasssvm__gamma': 'scale', 'oneclasssvm__nu': 0.01}
Score F1 pour anomalies: 0.5500131960939562
Précision pour anomalies: 0.9529035208047554
Rappel pour anomalies: 0.38657020960860694
doryan@doryan-Swift-SF314-42:~/L3Info/ProjetFDD_IDL/Fouille_Apprentissage$
```

Figure 15 : Résultats de la recherche des meilleurs hyperparamètres pour One Class SVM

Nous appliquons alors le One Class SVM sur l'ensemble de test composé des articles les plus récents de l'express avec les 4 catégories connues par le SVC et des articles du dataset lemonde avec des catégories nouvelles (culture et sport).

```
doryan@doryan-Swift-SF314-42:~/L3Info/ProjetFDD_IDL/Fouille_Apprentissage
n3 modele.py
              precision    recall  f1-score   support

     Nouveau       0.73      0.41      0.53       126
      Normal       0.76      0.92      0.83       252

    accuracy                           0.75       378
   macro avg       0.75      0.67      0.68       378
weighted avg       0.75      0.75      0.73       378

doryan@doryan-Swift-SF314-42:~/L3Info/ProjetFDD_IDL/Fouille_Apprentissage
```

Figure 16 : Performance du One Class SVM sur l'ensemble de test

De nouveau, pour valider l'utilisation de ce modèle dans le logiciel final, nous essayons d'autres manières de détecter des nouveautés et comparons leur performance.

Il existe plusieurs manières de détecter de la nouveauté dans un texte en s'entrainant sur un dataset de données connues, notamment l'Isolation Forest, le Local Outlier Factor (LOF) et le DBSCAN.

Le premier test est réalisé avec DBSCAN:

```python
# Test data
test = pd.read_csv('Dataset/test_doublemodele.csv', encoding='utf-8')
texts_test = test['Content'].tolist()
vectorizer = TfidfVectorizer()
X_test = vectorizer.fit_transform(texts_test)

# Application de DBSCAN sur les données de test
dbscan_test = DBSCAN(eps=0.3, min_samples=3)
clusters_test = dbscan_test.fit_predict(X_test)

predicted_labels_test = [1 if label == -1 else 0 for label in clusters_test]
true_labels_test = [1 if cat in ['sport', 'culture'] else 0 for cat in test['Category']]

precision = precision_score(true_labels_test, predicted_labels_test)
recall = recall_score(true_labels_test, predicted_labels_test)
f1 = f1_score(true_labels_test, predicted_labels_test)

print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')
```

Figure 17 : Code de DBSCAN

```
doryan@doryan-Swift-SF314-42:~/L3Info/ProjetFDD_IDL/Fouille_Apprentissage$ pytho
n3 modele.py
Precision: 0.3333333333333333
Recall: 1.0
F1 Score: 0.5
```

Figure 18 : Résultats de DBSCAN

Nous observons un mauvais score de précision et un score de rappel parfait, dans notre cas la précision est trop importante car nous voulons pouvoir compter sur la prédiction qu'un texte est nouveau. Nous préférons alors dans ce cas le One Class SVM.

Le second test est réalisé avec l'Isolation Forest :

```
# Chargement des données
df = pd.read_csv('Dataset/tous_articles_traites.csv', encoding='utf-8')
texts = df['Content'].tolist()

vectorizer = TfidfVectorizer()
X_train_transformed = vectorizer.fit_transform(texts)
X_train_transformed = X_train_transformed.tocsr()  # Conversion en format CSR

iso_forest = IsolationForest(n_estimators=100, random_state=42)
iso_forest.fit(X_train_transformed)  # Entrainement avec les données au bon format


test = pd.read_csv('Dataset/test_doublemodele.csv', encoding='utf-8')
X_test = test['Content'].tolist()
X_test_transformed = vectorizer.transform(X_test)
X_test_transformed = X_test_transformed.tocsr()  # Conversion en format CSR

y_test = [-1 if cat in [ 'sport', 'culture'] else 1 for cat in test['Category']]

# Prédiction avec le modèle Isolation Forest sur les données de test
predicted_labels = iso_forest.predict(X_test_transformed)

# Modification de seuil
scores = iso_forest.decision_function(X_test_transformed)
threshold = np.percentile(scores, 30)
predicted_labels = np.where(scores < threshold, -1, 1)

# Évaluation de la performance du modèle One-Class SVM
print(classification_report(y_test, predicted_labels, target_names=['Nouveau', 'Normal']))
```

Figure 19 : Code de l'isolation forest

Les résultats sont copiés ci-dessus :

	precision	recall	f1-score	support
Nouveau	0.42	0.13	0.20	126
Normal	0.68	0.91	0.78	252
accuracy			0.65	378
macro avg	0.55	0.52	0.49	378
weighted avg	0.59	0.65	0.58	378

Les résultats sont tous en dessous des résultats que propose le One Class SVM.

Le troisième test est réalisé avec LOF :

```python
# Chargement des données
df = pd.read_csv('Dataset/tous_articles_traites.csv', encoding='utf-8')
texts = df['Content'].tolist()

# Vectorisation des textes
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(texts)

# Initialisation du modèle LOF
lof = LocalOutlierFactor(n_neighbors=50, novelty=True, contamination=0.1)

# Entrainement du LOF
lof.fit(X_train)

# Chargement et préparation des données de test
test = pd.read_csv('Dataset/test_doublemodele.csv', encoding='utf-8')
X_test = vectorizer.transform(test['Content'].tolist())
y_test = [-1 if cat in ['sport', 'culture'] else 1 for cat in test['Category']]

# Prédiction avec le modèle LOF sur les données de test
predicted_labels = lof.predict(X_test)
predicted_labels = [1 if label == -1 else -1 for label in predicted_labels]  # pour correspondre à l'attendu

# Évaluation de la performance du modèle LOF
print(classification_report(y_test, predicted_labels, target_names=['Nouveau', 'Normal']))
```

Figure 20 : Code du LOF

	precision	recall	f1-score	support
Nouveau	0.19	0.44	0.27	126
Normal	0.18	0.06	0.09	252
accuracy			0.19	378
macro avg	0.18	0.25	0.18	378
weighted avg	0.18	0.19	0.15	378

Figure 21 : Résultats du LOF

De la même manière que précédemment les résultats ne sont pas aussi bons que ceux du One Class SVM.

Nous combinons alors notre modèle SVC et notre One Class SVM pour finaliser le projet. Pour cela nous avons deux choix : d'une nous pouvons appliquer le modèle One Class SVM au texte saisi par l'uilisateur puis selon la réponse, afficher qu'il est d'une catégorie autre ou alors fournir ce texte au modèle SVC dans le cas opposé.
Cependant, nous pouvons aussi appliquer chacun des deux modèles puis récupérer la probabilité à laquelle est estimée l'appartenance à une catégorie par le modèle SVC. Cela nous permettrait de determiner un seuil de confiance. Dans le cas ou ce seuil de confiance est dépassé, nous faisons confiance au SVC, sinon nous appelons le One Class SVM qui fait une prédiction. S'il prédit une nouveauté, nous estimerons que le texte est d'une catégorie 'Autre' sinon nous estimerons que la catégorie est celle prédite par le SVC.

Nous testons par expérience laquelle de ces deux manières d'appliquer les modèles donne les meilleurs résultats. Nous trouvons que la seconde donne de meilleurs résultats et ceux ci sont :

	precision	recall	f1-score	support
Autre	0.96	0.40	0.57	126
economie	0.70	0.83	0.76	63
environnement	0.65	0.97	0.78	63
politique	0.54	0.79	0.65	63
sciences-sante	0.78	0.81	0.80	63
accuracy			0.70	378
macro avg	0.73	0.76	0.71	378
weighted avg	0.77	0.70	0.69	378

Figure 22 : Performance du double modèle

On voit sur la figure ci-dessus une exactitude de 70%.

Paquetage logiciel .deb

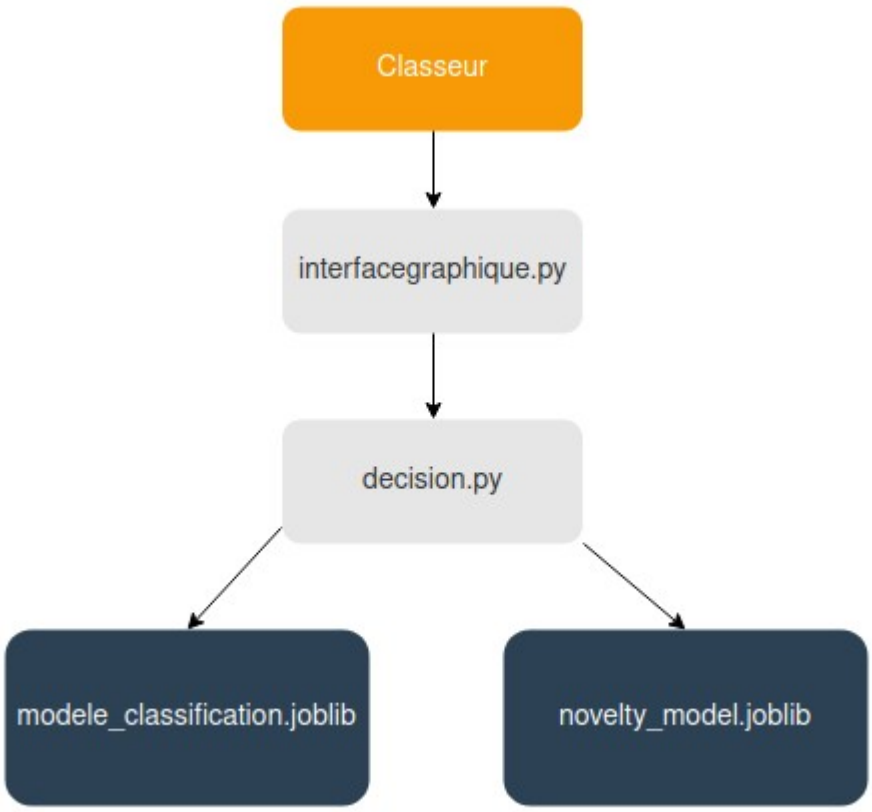


Figure 23 : Schéma de fonctionnement du logiciel

Il nous faut ensuite créer un package .deb qui nous permettra de distribuer le logiciel. Nous organisons alors nos fichiers nécessaires selon les règles des paquets .deb.

Nous créons un nouveau répertoire pour le projet nommé ‘classification-articles’, dans lequel nous créons d’autres répertoires :

classification-articles/
├── DEBIAN

```
|   ├── control       # Fichier contenant les métadonnées du paquet et les dépendances
|   └── postinst      #  Script post-installation qui installe les dépendances Python
├── usr
|   ├── bin
|   |   └── Classeur     # script qui execute le code Python
|   ├── lib
|   |   └──  classification_articles
|   |      ├── interfacegraphique.py  # Code Python principal
|   |      └── decision.py           # Code Python utilisé par le principal
|   └── share
|      └──  classification_articles
|         ├── modele_classification.joblib  # Fichier de modèle SVC
|         ├── novelty_model.joblib          # Fichier de modèle One Class SVM
|         └── requirements.txt              # Liste des dépendances Python
|      └── doc
|         └── classification_articles
|            └── README.md
```

Nous créons ensuite le paquet .deb avec la commande :
dpkg-deb --build classification-articles

Documentation Utilisateur

Installation du projet

Pour pouvoir exécuter le logiciel, il faut installer certaines dépendances. Vous trouverez ci-dessous un guide étape par étape pour le faire.

1. Vérifier l'installation de Python3

Ouvrez votre terminal puis tapez la commande suivante pour vérifier que Python est installé sur votre machine :

```
python3 --version
```

Si Python 3 est installé, cette commande affichera la version de Python 3 installée. Si vous obtenez un message d'erreur indiquant que Python n'est pas installé, vous pouvez l'installer avec :

```
sudo apt-get update
sudo apt-get install python3
```

2. Vérifier l'installation de "python3-pip"

Pour vérifier si pip pour Python 3 est installé, ouvrez un terminal si ce n'est pas déjà fait et tapez :

```
pip3 –version
```

Si pip3 est installé, cela affichera la version de pip installée. Sinon, vous pouvez l'installer en utilisant :

```
sudo apt-get install python3-pip
```

3. Vérifier l'installation de "python3-tk"

Pour vérifier si python3-tk est installé, ouvrez un terminal si ce n'est pas déjà fait et tapez :

```
python3 -c "import tkinter"
```

Si la commande s'exécute sans erreurs, python3-tk est installé. Si vous obtenez une erreur, vous pouvez installer python3-tk avec :

```
sudo apt-get install python3-tk
```

Une fois ces étapes terminées, vous pouvez passer à l'étape de l'installation du logiciel.

4. Installer le logiciel

Avant de procéder à l'installation, il est essentiel de naviguer jusqu'au répertoire contenant le fichier d'installation. Utilisez la commande “cd” (change directory) pour vous déplacer dans l'arborescence de vos fichiers jusqu'à l'emplacement du fichier ”classification-articles.deb”.

Une fois dans le bon dossier, installez le logiciel en tapant :

```
sudo dpkg -i classification-articles.deb
```

Pour lancer le programme, tapez à présent dans le terminal :

```
Classeur
```

Fonctionnalité du projet

Pour utiliser le programme saisissez le contenu d’un article dans la zone de texte de la fenêtre qui apparait, comme le montre la figure ci-dessous.

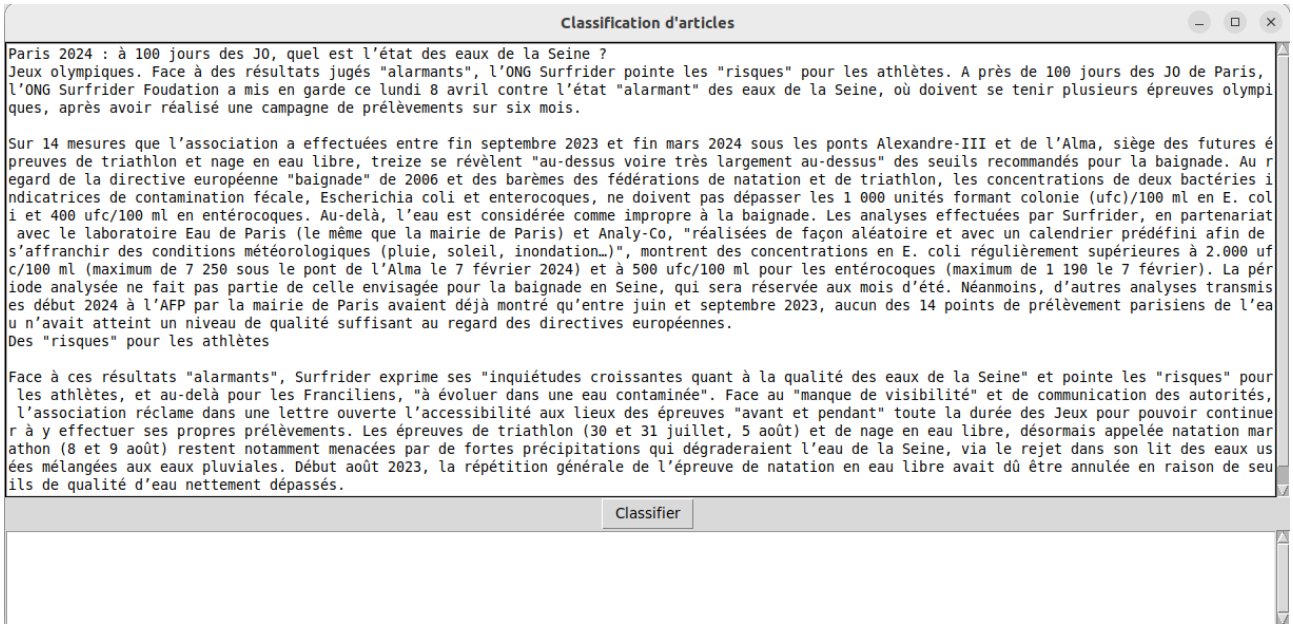


Figure 24 : Exemple de saisie de texte

Vous pouvez ensuite cliquer sur le bouton “Classifier”.

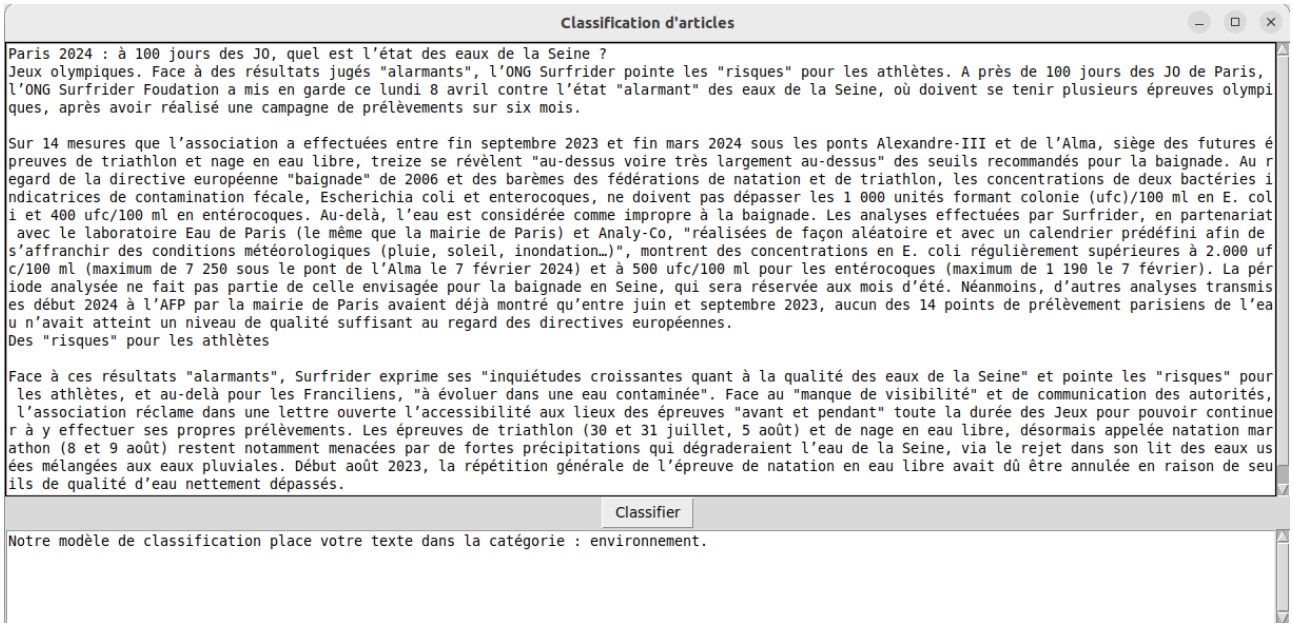


Figure 25 : Classification du texte saisi

Dans cet exemple, le programme estime que le texte est dans la catégorie environnement.
Vous pouvez ensuite sélectionner le texte puis le supprimer, en saisir un autre et le classifier de nouveau.

Les catégories que le logiciel pourra proposer sont : “politique”, “économie”, “environement”, “sciences-santé” et “autre”.

Conclusion

Ce projet de classification d'articles à l'aide de modèles de machine learning et de NLP en Python a été une expérience enrichissante. J'ai appris à manipuler des bibliothèques de traitement du langage naturel comme spaCy, et à intégrer des modèles d'apprentissage automatique, tels que ceux fournis par scikit-learn, pour traiter et analyser efficacement de grands ensembles de données textuelles. La gestion des données non structurées et l'amélioration des algorithmes pour augmenter la précision de classification ont été des défis particulièrement formateurs. Ce projet a également renforcé ma compréhension des implications pratiques de la NLP, soulignant l'importance d'une préparation spécifique des données pour obtenir des résultats fiables.

SOURCES

Nettoyage des articles :
https://blog.octo.com/nettoyage-du-texte-en-nlp-moins-de-vocabulaire-moins-de-bruit

WebScrapping :
https://realpython.com/python-web-scraping-practical-introduction/

Forêt Aléatoire :
https://towardsdatascience.com/a-practical-guide-to-implementing-a-random-forest-classifier-in-python-979988d8a263

Modele SVC :
https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

Modele BERT :
https://www.kaggle.com/code/harshjain123/bert-for-everyone-tutorial-implementation

One Class SVM:
https://scikit-learn.org/stable/auto_examples/svm/plot_oneclass.html

DBSCAN :
https://datascientest.com/machine-learning-clustering-dbscan

Isolation Forest :
https://blog.paperspace.com/anomaly-detection-isolation-forest/

LOF:
https://scikit-learn.org/stable/auto_examples/neighbors/plot_lof_outlier_detection.html