

Projet Machine Learning

HAI817I - 2024/2025

Classification d'assertions venant d'X (Twitter) selon leur rapport à la science

Groupe 8

Duban Mathis: 22102226

Gonzalez Oropeza Gilles : 21602817

Bernardon Vincent : 22009116

LONGLADE Mickaël : 22105047

Sujet

Ce projet s'inscrit dans le contexte de l'apprentissage supervisé, i.e. les données possèdent des labels. Il vise à trouver les modèles les plus performants pour prédire si des assertions (une assertion est une proposition que l'on avance et que l'on soutient comme vraie) faites par des hommes politiques (par exemple) sont vraies ou fausses.

✓ Installation

Dans cette partie, nous allons installer toutes les librairies dont nous allons avoir besoin pour notre projet.

```
# Installation des librairies pour le projet
!pip install pandas numpy scipy gensim emoji nltk matplotlib seaborn scikit-learn

import warnings # Supprime les warnings
warnings.filterwarnings("ignore", category=FutureWarning)

# Librairies de manipulation de données (graphique, lecture de données,ect...)
import pandas as pd # Lecture de données
```

```
import numpy as np # Array
import seaborn as sns #
import matplotlib.pyplot as plt # Graphique
import sys

# Librairies pour la fonction prepareText
import re # Regular expression
import nltk
import json
from nltk.corpus import stopwords #English stopwords
nltk.download('stopwords') # Téléchargement des stopwords (une seule fois)
from nltk.corpus import wordnet #Mots pour vérifier les suppressions de lettres
nltk.download('wordnet') # Téléchargement de mots existants
import emoji
import inflect # Transformation des chiffres en mots
import re
from googletrans import Translator # Traduction de langues
from nltk.stem import WordNetLemmatizer # Lemmatisation des mots
from nltk.stem import PorterStemmer # Racinisation des mots
nltk.download('punkt') #Tagetisation des mots
nltk.download('punkt_tab') # Tokenisation des mots
nltk.download('averaged_perceptron_tagger')
nltk.download('averaged_perceptron_tagger_eng')
from nltk import pos_tag # Tagination des mots
from nltk.tokenize import word_tokenize
import unicodedata # Suppresion d'accent
import contractions # Transformation des contractions
from collections import Counter
from collections import defaultdict

#Libraries pour l'entraînement du modèle
from sklearn.feature_extraction.text import TfidfVectorizer # Vectorisation
from sklearn.preprocessing import MaxAbsScaler, StandardScaler # Vectorisation
from spellchecker import SpellChecker # dictionnaire phonétique
from sklearn.feature_extraction.text import CountVectorizer # Topic modelling
from sklearn.decomposition import LatentDirichletAllocation #LDA
from sklearn import preprocessing # Upsampling
from imblearn.over_sampling import SMOTE # Upsampling
from imblearn.combine import SMOTETomek
from imblearn.over_sampling import RandomOverSampler # if resampleData doesn't
import sklearn
from sklearn.preprocessing import LabelEncoder #Label encoder

from gensim.models.coherencemodel import CoherenceModel #Coherence model
from sklearn.metrics import classification_report, confusion_matrix, accuracy_s
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e
from sklearn.utils import resample # Upsampling
from wordcloud import WordCloud # Nuage de mot
from gensim.corpora.dictionary import Dictionary # évaluation de cohérenec
from gensim.models import LdaModel # LDA
import os

from sklearn.model_selection import train_test_split, GridSearchCV, KFold, cros
import optuna
```

```
from scipy import stats
```

```
# Classifiers
```

```
from sklearn.tree import DecisionTreeClassifier # Decision TREE classifier
```

```
from sklearn.naive_bayes import MultinomialNB
```

```
from sklearn.svm import SVC
```

```
from sklearn.neighbors import KNeighborsClassifier
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: gensim in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: emoji in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: nltk in /usr/local/lib/python3.11/dist-packa
Requirement already satisfied: matplotlib in /usr/local/lib/python3.11/dist
Requirement already satisfied: seaborn in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/di
Requirement already satisfied: inflect in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: googletrans==4.0.0-rc1 in /usr/local/lib/pyt
Requirement already satisfied: contractions in /usr/local/lib/python3.11/di
Requirement already satisfied: pyspellchecker in /usr/local/lib/python3.11/
Requirement already satisfied: optuna in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: httpx==0.13.3 in /usr/local/lib/python3.11/d
Requirement already satisfied: certifi in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: hstspreload in /usr/local/lib/python3.11/dis
Requirement already satisfied: sniffio in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: chardet==3.* in /usr/local/lib/python3.11/di
Requirement already satisfied: idna==2.* in /usr/local/lib/python3.11/dist-
Requirement already satisfied: rfc3986<2,>=1.3 in /usr/local/lib/python3.11
Requirement already satisfied: httpcore==0.9.* in /usr/local/lib/python3.11
Requirement already satisfied: h11<0.10,>=0.8 in /usr/local/lib/python3.11/
Requirement already satisfied: h2==3.* in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: hyperframe<6,>=5.2.0 in /usr/local/lib/pytho
Requirement already satisfied: hpack<4,>=3.0 in /usr/local/lib/python3.11/d
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/pyt
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/di
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/
Requirement already satisfied: smart-open>=1.8.1 in /usr/local/lib/python3.
Requirement already satisfied: click in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: joblib in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: regex>=2021.8.3 in /usr/local/lib/python3.11
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packa
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.1
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/di
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.1
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/pytho
Requirement already satisfied: more_itertools>=8.5.0 in /usr/local/lib/pyth
Requirement already satisfied: typeguard>=4.0.1 in /usr/local/lib/python3.1
Requirement already satisfied: textsearch>=0.0.21 in /usr/local/lib/python3
Requirement already satisfied: alembic>=1.5.0 in /usr/local/lib/python3.11/
Requirement already satisfied: colorlog in /usr/local/lib/python3.11/dist-p
Requirement already satisfied: sqlalchemy>=1.4.2 in /usr/local/lib/python3.
Requirement already satisfied: PyYAML in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: Mako in /usr/lib/python3/dist-packages (from
```

```

Requirement already satisfied: typing-extensions>=4.12 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: wrapt in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: greenlet>=1 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: anyascii in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: pyahocorasick in /usr/local/lib/python3.11/dist-packages
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!

```

Importation du répertoire de travail sur Google Drive

```

from google.colab import drive
drive.mount('/content/gdrive/')
path='/content/gdrive/My Drive/M1_IMAGINE_ML/ML_Project/'
sys.path.append(path)
%cd $path
%pwd

```

Récupération des données du dataSet présent sur le répertoire Google Drive en ligne

```

# Importation des données
df=pd.read_csv('dataSet/data.csv', sep='\t')
# Lecture des 5 premières lignes pour confirmer la bonne récupération des données
display (df.head())

```

Récupération de nos dictionnaires :

```

# Récupération des monnaies
currency_file=pd.read_csv('dataSet/currency_translation.csv')
display (currency_file.head())
# Création d'un objet dictionnaire pour répertorier les monnaies
currency_dict = dict(zip(currency_file['Currency Symbol'], currency_file['Currency Name']))
# Récupération des abréviations (Slang)
slang_file=pd.read_csv('dataSet/slang_translation.csv')
display (slang_file.head())
# Création d'un objet dictionnaire pour répertorier les abréviations
slang_dict = dict(zip(slang_file['Slang'], slang_file['Meaning']))


```

```
slang_dict = dict(zip(slang_file['Abbreviation'], slang_file['Meaning']))
```

✓ Ingénierie des données

Dans cette partie, on s'intéresse au pré-traitement des données. Afin que chaque élément de notre base soit utilisable et pertinent on va nettoyer, normaliser et transformer nos données afin qu'elles soient préparées et optimales pour nos analyses.

On va ici répertorier tous les éléments qui doivent être traités :

Élément	Exemple	Traitement à effectuer	
Emojis		Suppression des émojis ou remplacement par leur signification	bas
Mention Twitter	@username	Remplacement par un Token	@M
Hashtag	#example	Remplacement par un Token	@H
URL	http://t.co/XGUfUDoLJB	Suppression de l'URL	""
Chiffre	13	Transformation en String	thir
Majuscules	Hello	Suppression de la majuscule	hell
Ponctuation	!	Suppression de la ponctuation	""
Mots répétés	cool cool cool	Normalisation en une seule occurrence	coo
Lettres répétées	that's greeeeeeeeat!	Réduction des répétitions excessives	that
Abréviations	ngl, fr	Remplacement par la version complète	not
Stopwords (déterminants)	the, and, a	Suppression si non pertinent	""
Slang (Argot)	gonna, dunno, wanna	Remplacement par des mots standards	goir
Langue étrangère	bonjour, gracias	Détection et traduction éventuelle	hell
Caractères spéciaux	§, \$, ^	Suppression des caractères	""

Expressions courantes	btw, lol	Remplacement par la version complète	by t
Négation mal formatée	ain't, dunno	Correction grammaticale	am
Émojis en Unicode	\U0001F60D (👊)	Conversion en texte lisible	smi
Symboles de devises	10\$, 10€	Normalisation (ex: "10 euros")	10 c
Accents	cliché	Normalisation (Suppression des accents)	clic
Heures	10AM, 13:30	Remplacement par un token	@T
Numéro de téléphone	+339208373	Remplacement par un token	@P
Expression flottante	14,34 10,000	Conversion en texte lisible	fou

On implémente ici nos fonctions que nous allons utiliser par la suite pour notre ingénierie des données :

```
#Fonction permettant de gérer les caractères spéciaux ayant un sens particulier
def removeSpecialCharacters(word, keepTokens):
    # Replace '+' followed by digits (potential phone numbers) with 'PHONE_NUMB
    if keepTokens:
        word = re.sub(r'\+\d+', 'TOKENPHONENUMBER', word)
    else:
        word = re.sub(r'\+\d+', '', word)

    # Remove / between numbers : 10/10 -> 10 out of 10
    word = re.sub(r'(\d+)/(\d+)', r'\1 out of \2', word)

    # Replace time expressions in HH:MM format
    if keepTokens:
        word = re.sub(r'\b\d{1,2}:\d{2}\b', 'TOKENTIME', word)
    else:
        word = re.sub(r'\b\d{1,2}:\d{2}\b', '', word)

    # Replace numbers followed by 'k' with their full value (e.g., 41916514k ->
    word = re.sub(r'(\d+)k\b', lambda m: str(int(m.group(1)) * 1000), word)
    return word

#Fonction permet de supprimer les répétitions successives de lettres (cas parti
def fixRepeat(word):
    # Reduce excessive repetition to exactly 2 occurrences
    repeat_regex = re.compile(r'(\w*)(\w)\2{2,}(\w*)')
    repl = r'\1\2\2\3'

    if wordnet.synsets(word):
        return word

    repl_word = repeat_regex.sub(repl, word)
    if repl_word != word:
        return fixRepeat(repl_word)

    # Try all combinations of removing one duplicate letter at a time
    candidates = set()
    for i in range(len(repl_word) - 1):
        if repl_word[i] == repl_word[i + 1]:
```

```

        candidates.add(repl_word[:i] + repl_word[i+1:])

# Check if any of the candidates is a valid word
for candidate in candidates:
    if wordnet.synsets(candidate):
        return candidate

# No valid word is found, return the single-letter version
single_letter_version = re.sub(r'(\.)\1', r'\1', repl_word)
return single_letter_version

# Fonction permettant de supprimer les accents
def remove_accents(text):
    return ''.join(c for c in unicodedata.normalize('NFD', text) if unicodedata

```

On crée donc notre fonction **prepareText** permettant de préparer nos données brutes afin de les reformater correctement :

```

stop_words_set = set(stopwords.words('english'))
translator = Translator() # initialisation du traducteur
lemmatizer = WordNetLemmatizer() # initialisation du lematiseur
stemmer = PorterStemmer() # initialisation du "racinisateur"
tokens = {"MENTION", "HASHTAG", "TIME", "PHONENUMBER"} # liste de nos token à i

# Fonction permettant de préparer la chaîne de caractères passée en paramètre
def prepareText(text, keepTokens: bool = True, keepEmojis: bool = True, numbers
    """
    Prépare la chaîne de caractère passée en paramètre

    Parameters
    -----
    text : str
        La chaîne de caractères
    keepTokens : bool, optional
        True si on doit garder les token, False si on doit les supprimer (defau
    keepEmojis : bool, optional
        True si on doit garder les emojis, False si on doit les supprimer (defa
    numbersAsTokens : bool, optional
        True si on doit transformer les chiffres en token, False si on doit les
        Ce token n'est pas supprimé si keepToken vaut False
    translate : bool, optional
        True si on doit traduire le texte en anglais, False si on ne le fait pa

    Returns
    -----
    str
        La chaîne de caractère préparée
    """
    #Majuscule, suppression
    data = str(text).lower()

    #Suppression d'accent
    data = data.encode('ascii', 'ignore').decode('ascii')

```

```

data = remove_accents(data)

#Contraction, on corrige
data = contractions.fix(data)

#Emoji, transformation en String
if (keepEmojis):
    data = emoji.demojize(data)
else:
    data = emoji.replace_emoji(data, replace='')

#Mention Twitter, transformation en Token
if (keepTokens):
    data = re.sub(r'@\w+', 'TOKENMENTION', data)
else:
    data = re.sub(r'@\w+', '', data)

#Hashtag, transformation en Token
if (keepTokens):
    data = re.sub(r'#\w+', 'TOKENHASHTAG', data)
else:
    data = re.sub(r'#\w+', '', data)

#URL, on supprime
data = re.sub(r'https?:\/\/\S+|www\.\S+', '', data)

#Devise, remplacement par sa chaîne de caractères
for symbol, name in currency_dict.items():
    data = re.sub(rf'(\d+){re.escape(symbol)}', r'\1 ' + name, data)

#Special characters that requires more attention than just remove
data = removeSpecialCharacters(data, keepTokens)

#Keep rating expressions (ex : 10/10)
rating_expressions = {}

def replace_match(match):
    key = f"RATING_{len(rating_expressions)}" # Unique placeholder
    rating_expressions[key] = match.group(0) # Store full match
    return key

# Replace rating expressions with placeholders
data = re.sub(r'(\d+|ten|nine|eight|seven|six|five|four|three|two|one) out

#Stopwords, suppression
data = ' '.join([word for word in data.split() if word not in stop_words_se

# Restore full rating expressions
for key, value in rating_expressions.items():
    data = data.replace(key, value)

#Removal of characters and symbols

```



```

#Ponctuation & caracteres speciaux, suppression
data = re.sub(r'^\w\s', '', data)

#Chiffre, transformation en String
if (numbersAsTokens):
    words = data.split()
    data = ' '.join(["number" if word.isdigit() else word for word in words]
else:
    words = data.split()
    data = ' '.join([inflect.engine().number_to_words(word) if word.isdigit()

#Heures, transformation en token
if keepTokens:
    data = re.sub(r'\b(\d{1,2}(:h\d{2})?\s*(am|pm)?)\b', 'TOKENTIME', data)
else:
    data = re.sub(r'\b(\d{1,2}(:h\d{2})?\s*(am|pm)?)\b', '', data)

#Abréviation (Slang)
data = ' '.join([slang_dict.get(word, word) for word in data.split()])

#Mots répétés
data = re.sub(r'\b(\w+)(\s+\1\b)+', r'\1', data)
#Lettres répétés
data = ' '.join([fixRepeat(word) for word in data.split()])

#remplacer TOKEN par @TOKENxxxx correspondant
if (keepTokens):
    for token in tokens:
        data = re.sub(rf'TOKEN{token}', f'{token}', data)

#Traduction du tweet
if translate:
    try:
        data = translator.translate(data, dest='en').text
    except Exception as e:
        pass

return data

```

Exemple du passage de notre fonction

```

#URL
display("http://t.co/XGUfUDoLJB")
display(prepareText("http://t.co/XGUfUDoLJB"))
print("\n")
#Chiffre
display("3")
display(prepareText("3"))
print("\n")
#Majuscule
display("Hello")
display(prepareText("Hello"))
print("\n")

```

```
#Ponctuation
display("Hello!")
display(prepareText("Hello!"))
print("\n")
#Abréviation
display("lol")
display(prepareText("lol"))
print("\n")
#StepWord
display("After planning the project, she carefully researched each step, ensuri
display(prepareText("After planning the project, she carefully researched each
print("\n")
#Emojis
display("😍")
display("keepEmojis=True : " + prepareText("😍", keepEmojis=True))
display("keepEmojis=False : " + prepareText("😍", keepEmojis=False))
print("\n")
#Traductions (dernière étape)
display("我今天去超市买了一些水果")
display("translate=True : " + prepareText("我今天去超市买了一些水果", translate=Tru
display("translate=False : " + prepareText("我今天去超市买了一些水果", translate=Fa
print("\n")
#Mention Twitter
display("as @username said it's bad !")
display("keepTokens=True : " + prepareText("as @username said it's bad !", keep
display("keepTokens=False : " + prepareText("as @username said it's bad !", kee
print("\n")
#Hashtag
display("I went to the theater to see Dune 2 #Dune")
display(prepareText("I went to the theater to see Dune 2 #Dune"))
print("\n")
#Caractères spéciaux
display("$f")
display(prepareText("$f"))
print("\n")
#Devices
display("10$ 10f 10€")
display(prepareText("10$ 10f 10€"))
print("\n")
#Mot répétés
display("Cool Cool Cool Cool Hot Hot Hot")
display(prepareText("Cool Cool Cool Cool Hot Hot Hot"))
print("\n")
#Lettres répétées
display("Steaaaaaaaaak tendeeeeeers beeeeer gooooooose   threeeeeeeee woooooooooooood
display(prepareText("Steaaaaaaaaak tendeeeeeers beeeeer gooooooose   threeeeeeeee w
print("\n")
#Accent
display("cliché")
display(prepareText("cliché"))
print("\n")
#Heures
display("10AM computer 10:30 potatoes 10h30")
display(prepareText("10AM computer 10:30 potatoes 10h30"))
```

```
print("\n")
#Numéro de téléphone
display("+33123456789")
display(prepareText("+33123456789"))
print("\n")
#Expression flottante
display("14,34 10,000")
display("numbersAsTokens=True : " + prepareText("14,34 10,000", numbersAsTokens
display("numbersAsTokens=True : " + prepareText("14,34 10,000", numbersAsTokens
print("\n")
# 10/10
display("10/10")
display(prepareText("10/10"))
print("\n")
#Caractères spéciaux (numéro de téléphone, x/x , 10,0000)
display("N. Lutz ")
display(prepareText("N. Lutz"))
print("\n")
```

On crée une copie de notre set de données de base et on applique notre fonction sur tout nos tweets /!\ cette cellule prend un temps de calcul conséquent car elle crée une copie du CSV avec les données toutes formatées.

```
# File path for the specific dataset
file_path = 'dataSet/precomputed/dataPrepared1101.csv'
```

```
# On évite de traiter à nouveau les données si on a déjà le fichier des données
if os.path.exists(file_path):
    # If the file exists, load it
    dataPrepared = pd.read_csv(file_path)
    print("Le fichier dataPrepared1101 existe déjà. Chargement des données depuis le fichier")
else:
    # If the file does not exist, compute it
    dataPrepared = df.copy()
    dataPrepared['text'] = dataPrepared['text'].apply(prepareText)

    # Save the computed data to disk
    dataPrepared.to_csv(file_path, index=False)
    print("Le fichier dataPrepared1101 n'existe pas. Les données ont été calculées")

    Le fichier dataPrepared1101 existe déjà. Chargement des données depuis le d
```

On supprime toutes les lignes contenant un tweet vide :

```
# Suppression de toutes les lignes vides
dataPrepared = dataPrepared[dataPrepared['text'] != '']
dataPrepared = dataPrepared.dropna(subset=['text'])

# Afficher le nombre de ligne ayant un tweet vide
print("Nombre de lignes contenant un tweet vide : ", len(df[df['text'] == '']))

# Afficher
print("5 premières lignes du dataset :")
display(dataPrepared.head())
```

Une fois notre premier traitement effectué on va effectuer la dernière partie des traitements des données brutes la lemmatisation, racinisation et tagination. Ces étapes permettent d'affiner le texte pour que chaque mot soit réduit à sa forme de base, ce qui est essentiel pour de nombreuses applications de traitement de texte, comme la recherche d'informations ou l'analyse de sentiments.

Voici les étapes que l'on va faire après le formatage :

Lemmatisation : Cette technique consiste à réduire un mot à sa forme canonique (ou lemmé), c'est-à-dire à la forme sous laquelle il apparaît dans le dictionnaire. Exemple better deviendra good.

Racinisation : Cette méthode consiste à réduire un mot à sa racine, c'est-à-dire à enlever les suffixes (ou préfixes) pour obtenir une forme simplifiée du mot. Cela permet de mieux traiter les variations de mot comme runner qui deviendra run.

Tagination (ou étiquetage de parties du discours) : Cette technique consiste à identifier et à étiqueter chaque mot d'un texte en fonction de sa catégorie grammaticale (nom, verbe, adjectif, etc.)

On va commencer par appliquer une tokenisation et une taggenisation sur chacun de nos tweets. Pour cela on va définir 2 fonctions :

```
#Fonction permettant de récupérer le bon tag du mot passé en paramètre
def get_wordnet_pos(word):
    if word.startswith('J'):
        return wordnet.ADJ
    elif word.startswith('V'):
        return wordnet.VERB
    elif word.startswith('N'):
        return wordnet.NOUN
    elif word.startswith('R'):
        return wordnet.ADV
    else:
        return wordnet.NOUN

#Fonction qui applique la tokenisation et une taggenisation sur une phrase pass
def lemmatize_taggenize_sentence(sentence):
    tokens = word_tokenize(sentence) # Tokenisation du texte
    tagged_tokens = pos_tag(tokens) # Étiquetage des mots (POS tagging)
    lemmatized = [lemmatizer.lemmatize(token, get_wordnet_pos(tag)) for token,
    return " ".join(lemmatized)
```

Essayons notre fonction :

```
print(dataPrepared['text'][0])
print(lemmatize_taggenize_sentence(dataPrepared['text'][0]))

knees bit sore guess sign recent treadmilling working
knee bite sore guess sign recent treadmilling work
```

On applique alors notre fonction sur nos données préparées :

```
dataPrepared['text'].apply(lemmatize_taggenize_sentence)
```

On va ajouter dans notre DataFrame un attribut contenant les tweets correctement traités en appliquant les opérations suivantes :

- Normalisation du texte : suppression des variations morphologiques.
- Réduction de la dimensionnalité : un même concept est représenté par un seul mot.
- Amélioration des performances des modèles : les algorithmes de Machine Learning comprennent mieux les relations entre les mots.

La partie subtile c'est que ce genre de traitement peuvent influencer complètement les mots post-traitement. Par exemple unhappiness doit devenir unhappy, pour éviter ce genre d'erreur on doit appliquer une dernière transformation :

```
#Fonction qui permet de ne pas perdre le sens d'un mot traité (i.e unhap)
def refine_stem_lemmatize(token, tag):
    try:
        # Racinisation
        stemmed = stemmer.stem(token)

        # Vérification du préfixe "un"
        if stemmed.startswith("un") and len(stemmed) > 2: # Vérifie que "un" n
            root = stemmed[2:] # Retire le préfixe "un"
            if wordnet.synsets(root): # Vérifie si la racine sans "un" est val
                return f"un{root}"
```

```

        return r"not {root}"

# Vérification de validité du mot racinisé
if not wordnet.synsets(stemmed):
    stemmed = token # Si le mot racinisé est incompréhensible, garde l

# Lemmatisation
lemmatized = lemmatizer.lemmatize(stemmed, get_wordnet_pos(tag))
return lemmatized
except Exception as e:
    # Afficher le mot problématique et son erreur
    print(f"Erreur avec le mot : '{token}' - Exception : {e}")
    raise e # Propager l'exception pour un traitement éventuel

#Fonction qui ajoute dans le dataSet un colonne contenant le texte traité
def process_text_column(text):

    # Tokenisation et traitement
    tokens = word_tokenize(text)
    tagged_tokens = pos_tag(tokens)
    processed_tokens = [refine_stem_lemmatize(token, tag) for token, tag in tagged_tokens]
    return " ".join(processed_tokens)

# Créer une colonne vide pour stocker les textes transformés
dataPrepared['processed_text'] = ""

# Boucle for avec iterrows
for index, row in dataPrepared.iterrows():
    text = row['text'] # Récupérer le texte original

    if pd.notnull(text) and text.strip() != "": # Vérifier que le texte est valide
        try:
            # Appliquer la fonction process_text_column
            dataPrepared.at[index, 'processed_text'] = process_text_column(text)
        except Exception as e:
            print(f"Erreur à l'index {index} avec le texte : {text}")
            print(f"Exception : {e}")
            dataPrepared.at[index, 'processed_text'] = "" # Insérer une chaîne vide
    else:
        dataPrepared.at[index, 'processed_text'] = "" # Gérer les textes nuls

```

On teste notre fonction de traitement final :

```

text = "The runners were running faster than the dogs unhappiness displacement"
tokens = word_tokenize(text)
tagged_tokens = pos_tag(tokens)
stemmed_then_lemmatized = [refine_stem_lemmatize(token, tag) for token, tag in tagged_tokens]
print("racinisation puis lemmatization : ")
print(" ".join(stemmed_then_lemmatized))

```



```
print("\n")

stemmed_tokens = [stemmer.stem(token) for token in tokens]
lemmatized = [lemmatizer.lemmatize(token, get_wordnet_pos(tag)) for token, tag
print("racinetisation : ")
print(" ".join(stemmed_tokens))
print("\n")
print("lemmatization : ")
print(" ".join(lemmatized))

racinetisation puis lemmatization :
The runner be run faster than the dog unhappiness displacement inflexibilit

racinetisation :
the runner were run faster than the dog unhappi displac inflex irrespons ki

lemmatization :
The runner be run faster than the dog unhappiness displacement inflexibilit
```

On remplace dans notre dataSet la colonne processed_text contenant le texte filtré et traité par text afin d'obtenir qu'un seul attribut :

```
#Ajout de l'attribut processed_text sur chaque ligne de notre dataSet
dataPrepared['text'] = dataPrepared['processed_text']
#Suppression de l'attribut processed_text
dataPrepared = dataPrepared.drop(columns=['processed_text'])

# Sauvegarde des données dans un CSV
dataPrepared.to_csv('dataSet/dataPrepared.csv', index=False)

#Affichage des 5 premières lignes
display(dataPrepared.head())
```

✓ Vectorisation via TF-IDF

Dans cette partie, on souhaite continuer notre travail visant à préparer nos données pour les

envoyer aux algorithmes d'apprentissage automatique. Pour cela, on va effectuer une vectorisation via la méthode TF-IDF.

Le principe de la vectorisation est de convertir des données textuelles en une représentation numérique. Cela va permettre aux algorithmes d'apprentissage automatique de comprendre et de traiter le langage humain à partir de nos données préparées.

Dans notre cas, on a décidé d'utiliser les n-grammes (séquence de n mot répétées), cela va nous permettre de récupérer les relations entre les mots et ainsi détecter les mots qui pourraient potentiellement nous conduire vers une fake news ou nous indiquer les mots démontrant qu'un tweet est scientifique.

```
# Récupération des données préparées
dataPrepared = pd.read_csv('dataSet/precomputed/dataPrepared1101.csv')

# Suppression des lignes vides
dataPrepared = dataPrepared[dataPrepared['text'] != '']
dataPrepared = dataPrepared.dropna(subset=['text'])

def vectorize_data(text_series):
    vectorizer = TfidfVectorizer(ngram_range=(1, 2), min_df=0.5, max_df=0.9)
    scaled = MaxAbsScaler().fit_transform(vectorizer.fit_transform(text_series))
    return scaled
#Exemple d'appel
#X = vectorize_data(filtered_data['text'])
```

✓ Topic Modelling via LDA

Une fois nos données vectorisées et compréhensibles pour la machine, nous allons appliquer le principe de *Topic Modelling*.

Le topic modelling est une technique d'apprentissage automatique non supervisé qui identifie et extrait des thèmes ou des sujets latents à partir d'un ensemble de documents textuels (dans notre cas notre ensemble de tweet). Le but de cette étape est d'aider notre futur modèle à labelliser ses données tout en identifiant les sujets principaux.

Dans notre cas, nous utilisons LDA (Latent Dirichlet Allocation), cette technique cherche à découvrir des thématiques cachées (topics) dans un ensemble de documents. On va appliquer le LDA afin d'identifier les topics de chaque tweets et les insérer dans un attribut nommé 'Topic'.

Afin d'obtenir des topics pertinents, il faut faire attention à la répartition de ces derniers, si nous avons un topic trop dominant (ex:30%), cela va écraser la représentation des autres topics. On va alors jouer sur le nombre de topics à représenter afin d'obtenir une répartition plus partagée. De plus nous devons faire attention que chaque topic ait du sens. Le but de cette étape est purement statistique.

```
#Fonction permettant d'afficher les mots les plus courant d'un topic en questio
def print_top_words(model, feature_names, n_top_words=10):
    for topic_idx, topic in enumerate(model.components_):
        top_words = [feature_names[i] for i in topic.argsort()[: -n_top_words -
        display(f"Topic {topic_idx+1}: {'', '.join(top_words)}")

# Transformer TF-IDF en une Matrice Exploitable par LDA
count_vectorizer = CountVectorizer(ngram_range=(1, 2), min_df=5, max_df=0.9)
count_matrix = count_vectorizer.fit_transform(dataPrepared['text'])

# Configuration du modèle LDA pour l'appliquer
n_topics = 15 # Nombre de topics à identifier ( variable à ajuster pour avoir

# Préparer les données tokenisées
tokenized_texts = [text.split() for text in dataPrepared['text']]

# Créer un dictionnaire et un corpus
dictionary = Dictionary(tokenized_texts)
corpus = [dictionary.doc2bow(text) for text in tokenized_texts]

# Entraîner un modèle LDA
lda_model_gensim = LdaModel(corpus=corpus, num_topics=n_topics, id2word=diction

lda_model = LatentDirichletAllocation(n_components=n_topics, random_state=42, m

# Entraînement du modèle
lda_topics = lda_model.fit_transform(count_matrix)

# Obtenir les mots les plus représentatifs de chaque topic
feature_names = count_vectorizer.get_feature_names_out()

# Associer chaque tweet à son topic dominant
topic_assignments = np.argmax(lda_topics, axis=1)

# Ajouter au DataFrame
... ..
```

```
dataPrepared['topic'] = topic_assignments

# Afficher pour chaque répartition son nombre d'itération et son occurrence norm
topic_counts = dataPrepared['Topic'].value_counts(normalize=True)

# Convertir les occurrences en pourcentages
topic_counts_percent = dataPrepared['Topic'].value_counts()

# Créer un tableau avec le nombre d'occurrences et les pourcentages
topic_stats = pd.DataFrame({
    'Représentation (%)': topic_counts,
    'Occurrence': topic_counts_percent
})

display("Visualisation des recurrences et de la répartition des topics :")

# Afficher le tableau
display(topic_stats)

display("Affichage des topics principaux représenté dans nos tweets :")
# Afficher les topics principaux représenté 1,3,0,2,4
print_top_words(lda_model, feature_names, n_top_words=10)

display("Visualisation de la répartition des topics via un graph normalisé :")
# Visualisation de la répartition des topics via un graph normalisé
plt.figure(figsize=(10, 6))
topic_counts.plot(kind='bar', color=colors)
plt.title('Répartition des Topics Dominants', fontsize=16)
plt.xlabel('Topic', fontsize=12)
plt.ylabel('Proportion (%)', fontsize=12)
plt.xticks(rotation=0)
plt.show()
```


On observe que nous avons des topics ayant une répartition +/- équivalente, ce qui signifie que ces 7 topics sont à peu près représentés de la même manière dans nos tweets.

Evaluons la cohérence de nos topics

```
#Evaluons la cohérence de nos topics
coherence_model = CoherenceModel(model=lda_model_gensim, texts=tokenized_texts,
coherence_score = coherence_model.get_coherence())

print(f"Score de cohérence des topics : {coherence_score}")

Score de cohérence des topics : 0.41984850392502715
```

Visualisation du nuage des mots de chaque topics

```
n_topics = len(lda_model.components_)

n_rows = 2
n_cols = (n_topics // n_rows) + (n_topics % n_rows > 0)

fig, axes = plt.subplots(nrows=n_rows, ncols=n_cols, figsize=(n_cols * 5, n_row
axes = axes.flatten()
# Boucle sur chaque topic pour créer le nuage de mots
for topic_idx, topic in enumerate(lda_model.components_):
    # Générer le nuage de mots pour chaque topic
    wordcloud = WordCloud(width=800, height=400).generate(" ".join([feature_nam
    # Afficher le nuage de mots dans le sous-graphe approprié
    axes[topic_idx].imshow(wordcloud, interpolation="bilinear")
    axes[topic_idx].axis("off") # Retirer les axes
    axes[topic_idx].set_title(f"Topic {topic_idx + 1}")
for i in range(n_topics, len(axes)):
    axes[i].axis('off')
plt.tight_layout()
plt.show()
```

✓ Upsampling

Après avoir effectué notre topic modelling afin d'identifier les idées principales de nos tweets, nous allons appliquer un Upsampling.

Dans le domaine des méthodes d'apprentissage automatique, l'upsampling est utilisé afin d'équilibrer nos classes déséquilibrées en augmentant les classes qui sont sous représentées. De ce fait, nous aurons une meilleure répartition de nos classes (**science_related**, **scientific_claim**, **scientific_reference**, **scientific_context**). Voici un extrait de chaque classe dans notre dataset préparé avant l'upsampling :

```
#affichage de chaque classe dans un graph
fig, axes = plt.subplots(2, 2, figsize=(12, 10)) # 2 lignes, 2 colonnes pour 1

# Afficher la répartition de chaque colonne dans un sous-graphe
dataPrepared['science_related'].value_counts().plot(kind='bar', ax=axes[0, 0],
axes[0, 0].set_title('Répartition de science_related')

dataPrepared['scientific_claim'].value_counts().plot(kind='bar', ax=axes[0, 1],
axes[0, 1].set_title('Répartition de scientific_claim')

dataPrepared['scientific_reference'].value_counts().plot(kind='bar', ax=axes[1, 0],
axes[1, 0].set_title('Répartition de scientific_reference')

dataPrepared['scientific_context'].value_counts().plot(kind='bar', ax=axes[1, 1],
axes[1, 1].set_title('Répartition de scientific_context')

plt.tight_layout()
plt.show()
#print la répartition
```



```
# Définition de notre fonction d'upsampling, utilisant SMOTE
def resampleData(X, y):
    combined = SMOTETomek(random_state=42)
    X_resampled, y_resampled = combined.fit_resample(X, y)
    return X_resampled, y_resampled
```

Appliquons l'upsampling :

```
data_lv11 = dataPrepared.copy()
y = data_lv11['science_related']
X_text = data_lv11['text']

#Vectorisation TF-IDF + Scaling
vectorizer = TfidfVectorizer(ngram_range=(1, 2), min_df=5, max_df=0.9)
X_vectorized = vectorizer.fit_transform(X_text)

scaler = MaxAbsScaler()
X_scaled = scaler.fit_transform(X_vectorized)

#Split train/test
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,

X_resampled, y_resampled = resampleData(X_train, y_train)
```

On visualise bien le fait que l'upsampling a bien rééquilibré nos classes et que désormais nous n'avons plus de classes minoritaires.

```
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
sns.countplot(x=y_train)
plt.title('Original Distribution')

plt.subplot(1, 2, 2)
sns.countplot(x=y_resampled)
plt.title('After Resampling')
```

✓ Classification

Commencez à coder ou à générer avec l'IA.

Le but de la classification est de permettre de déterminer le contexte de nos tweets en fonction de leurs caractéristiques.

Pour obtenir les meilleurs résultats possibles, nous allons pour établir 4 méthodes de classification :

- Decision Tree
- Naïve Bayes
- SVC (Support Vector Clustering)
- KNN (k-nearest neighbors)

On rappelle nos classes de tweets :

- science_related
- scientific_claim
- scientific_reference
- scientific_context

Puis nous allons les tester sur 3 tâches de classification :

- {SCIENTIFIQUE} vs. {NON SCIENTIFIQUE} (2 classes, pour la classification de niveau 1)
- {CLAIM, REF} vs. {CONTEXT} (deux classes pour la classification de niveau 2)

- {CLAIM} vs. {REF} vs. {CONTEXT} (trois classes pour la classification niveau 3)

Une fois ce travail réalisé, nous pourrons évaluer les performances de chaque classifieur via plusieurs métriques différentes.

On définis nos fonctions d'affichage de courbes :

```
#Fonction d'affichage des courbes
def plot_curves_confusion(confusion_matrix, class_names):
    plt.figure(1, figsize=(16, 6))
    plt.gcf().subplots_adjust(left=0.125, bottom=0.2, right=1, top=0.9, wspace=

    # Matrice de confusion
    plt.subplot(1, 3, 3)
    sns.heatmap(confusion_matrix, annot=True, fmt="d", cmap='Blues', xticklabel
    plt.xlabel('Predicted', fontsize=12)
    plt.title("Confusion matrix")
    plt.ylabel('True', fontsize=12)
    plt.show()

def plot_curves(scores):
    plt.figure(1, figsize=(16, 6))
    plt.gcf().subplots_adjust(left=0.125, bottom=0.2, right=1, top=0.9, wspace=

    # Plot loss
    plt.subplot(121)
    plt.title('Cross Entropy Loss')
    plt.plot(scores, color='blue')
    plt.ylabel('Loss')
    plt.xlabel('Fold')

    # Plot accuracy
    plt.subplot(122)
    plt.title('Classification Accuracy')
    plt.plot(1 - scores, color='red')
    plt.ylabel('Error Rate')
    plt.xlabel('Fold')

    plt.show()

def plot_curves_results(naive_scores, svc_scores, decision_scores, knn_scores):
    classifiers = ['Naive Bayes', 'SVC', 'Decision Tree', 'KNN']

    fold_scores = [naive_scores, svc_scores, decision_scores, knn_scores]

    # Scores moyens
    plt.figure(figsize=(8, 5))
    mean_scores = [score.mean() for score in fold_scores]
    plt.bar(classifiers, mean_scores, color=['blue', 'orange', 'green', 'red'])
    plt.title('Scores moyens des classifieurs')
```

```
plt.xlabel('Classifieurs')
plt.ylabel('Score moyen')
plt.show()
```

✓ Classification {SCI} vs {NON-SCI} (NIVEAU 1)

On définit nos données d'entraînement pour la classification de niveau 1 :

```
#Copie de nos données d'entrées
data_lv11 = dataPrepared.copy()
y = data_lv11['science_related']
X_text = data_lv11['text']

#Vectorisation TF-IDF + Scaling
vectorizer = TfidfVectorizer(ngram_range=(1, 2), min_df=5, max_df=0.9)
X_vectorized = vectorizer.fit_transform(X_text)

scaler = MaxAbsScaler()
X_scaled = scaler.fit_transform(X_vectorized)

#Split train/test
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,

# Upsampling du jeu d'entraînement SEULEMENT
print(f"sizes of the classes before resampling : {Counter(y_train)}")

X_resampled, y_resampled = resampleData(X_train, y_train)

print(f"sizes of the classes after resampling : {Counter(y_resampled)}")

    sizes of the classes before resampling : Counter({0: 612, 1: 299})
    sizes of the classes after resampling : Counter({0: 603, 1: 603})
```

✓ Recherches des paramètres optimaux des classifieurs

Nous allons utiliser GridSearchCV afin de trouver les paramètres optimaux pour les classifieurs. Nous avons expérimenté avec une petite plage de recherche peu précise au cours du projet afin d'itérer rapidement. Après avoir acquis plus de connaissances, nous avons décidé de pousser la recherche de paramètres ici.

Bien entendu, les résultats sont stockés dans un fichier pour éviter de les recalculer à chaque fois.

```
def perform_gridsearch_and_plot(X_resampled, y_resampled, X_test, y_test, level
    if os.path.exists(filename):
```

```
with open(filename, 'r') as f:
    best_params = json.load(f)
print(f"Loaded best parameters from {filename}")
else:
    best_params = {}

# Decision Tree
param_grid_dt = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [3, 5, 7, 10, 20, 30, 40, 50, None],
    'min_samples_split': [2, 5, 10, 15, 20, 25, 30],
    'min_samples_leaf': [1, 2, 4, 6, 8, 10]
}

dt = DecisionTreeClassifier(random_state=42)
grid_dt = GridSearchCV(dt, param_grid_dt, cv=5, scoring='f1_macro', n_j
grid_dt.fit(X_resampled, y_resampled)

results = grid_dt.cv_results_
params = results['params']
scores = results['mean_test_score']

gini_scores = [s for s, p in zip(scores, params) if p['criterion'] == '
entropy_scores = [s for s, p in zip(scores, params) if p['criterion'] =

plt.figure(figsize=(10, 6))
plt.plot(range(len(gini_scores)), gini_scores, label='Gini', color='ste
plt.plot(range(len(entropy_scores)), entropy_scores, label='Entropy', c
plt.title(f'Decision Tree (Level {level}) - Mean F1 Macro Score Compari
plt.xlabel('Parameter Combination Index', fontsize=12)
plt.ylabel('Mean F1 Macro Score', fontsize=12)
plt.legend()
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()

best_params["DecisionTree"] = grid_dt.best_params_

# Naive Bayes
param_grid_nb = {'alpha': np.logspace(-3, 3, num=100), 'fit_prior': [Tr
nb = MultinomialNB()
grid_nb = GridSearchCV(nb, param_grid_nb, cv=5, scoring='f1_macro')
grid_nb.fit(X_resampled, y_resampled)

scores_nb_true = grid_nb.cv_results_['mean_test_score'][:,2]
scores_nb_false = grid_nb.cv_results_['mean_test_score'][1:,2]
alphas = param_grid_nb['alpha']

plt.figure(figsize=(8, 6))
plt.plot(alphas, scores_nb_true, label='fit_prior=True', color='teal')
plt.plot(alphas, scores_nb_false, label='fit_prior=False', color='darko
plt.title(f'Multinomial Naive Bayes (Level {level}) - F1 Score vs Alpha
plt.xlabel('Alpha', fontsize=12)
plt.xscale('log')
```

```
plt.ylabel('Mean F1 Macro Score', fontsize=12)
plt.legend()
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()

best_params["MultinomialNB"] = grid_nb.best_params_

# KNN
param_grid_knn = {
    'n_neighbors': list(range(1, 31)),
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}
knn = KNeighborsClassifier()
grid_knn = GridSearchCV(knn, param_grid_knn, cv=5, scoring='f1_macro')
grid_knn.fit(X_resampled, y_resampled)

# Organize results by metric and weights
scores_by_metric = defaultdict(lambda: defaultdict(list))

for params, score in zip(grid_knn.cv_results_['params'], grid_knn.cv_re
    metric = params['metric']
    weights = params['weights']
    n_neighbors = params['n_neighbors']
    scores_by_metric[metric][weights].append((n_neighbors, score))

# Plot for each metric
fig, axs = plt.subplots(1, 3, figsize=(18, 5), sharey=True)
metrics = ['euclidean', 'manhattan', 'minkowski']
colors = ['royalblue', 'darkorange']

for i, metric in enumerate(metrics):
    ax = axs[i]
    for j, weights in enumerate(['uniform', 'distance']):
        data = sorted(scores_by_metric[metric][weights], key=lambda x:
            neighbors = [x[0] for x in data]
            scores = [x[1] for x in data]
            ax.plot(neighbors, scores, label=f'weights={weights}', color=co
        ax.set_title(f'KNN - metric={metric}', fontsize=13)
        ax.set_xlabel('n_neighbors', fontsize=11)
        ax.set_ylabel('Mean F1 Macro Score', fontsize=11)
        ax.legend()
        ax.grid(alpha=0.3)

plt.tight_layout()
plt.show()

best_params["KNN"] = grid_knn.best_params_

# SVC
c_values = np.logspace(-3, 3, num=100)
param_grid_svc = {
```

```

        'C': c_values,
        'kernel': ['linear', 'rbf', 'poly'],
        'gamma': ['auto', 'scale']
    }

    svc = SVC(random_state=42)
    grid_svc = GridSearchCV(svc, param_grid_svc, cv=5, scoring='f1_macro',
                           grid_svc.fit(X_resampled, y_resampled)

    results = grid_svc.cv_results_
    params = results['params']
    scores = results['mean_test_score']

    for gamma_value in ['auto', 'scale']:
        plt.figure(figsize=(10, 6))
        for kernel, color in zip(['linear', 'rbf'], ['blue', 'green']): # ker
            kernel_gamma_scores = [
                s for s, p in zip(scores, params)
                if p['kernel'] == kernel and p['gamma'] == gamma_value
            ]
            plt.plot(c_values, kernel_gamma_scores, label=f'kernel={kernel}',
                    plt.title(f'SVC (Level {level}) - gamma={gamma_value}', fontsize=14)
            plt.xlabel('C', fontsize=12)
            plt.xscale('log')
            plt.ylabel('Mean F1 Macro Score', fontsize=12)
            plt.legend()
            plt.grid(alpha=0.3)
            plt.tight_layout()
            plt.show()

    best_params["SVC"] = grid_svc.best_params_

    # Save the best parameters to file
    with open(filename, 'w') as f:
        json.dump(best_params, f, indent=4)
    print(f"Saved best parameters to {filename}")

    # Display all best parameters
    for model_name, params in best_params.items():
        print(f"Best parameters for {model_name} (Level {level}): {params}")

```

```

perform_gridsearch_and_plot(X_resampled, y_resampled, X_test, y_test, 1, "dataS

Loaded best parameters from dataSet/lvl1_parameters.json
Best parameters for DecisionTree (Level 1): {'criterion': 'entropy', 'max_d
Best parameters for MultinomialNB (Level 1): {'alpha': 0.06579332246575682,
Best parameters for KNN (Level 1): {'metric': 'euclidean', 'n_neighbors': 2
Best parameters for SVC (Level 1): {'C': 7.56463327554629, 'gamma': 'scale'

```

```

# Définition de la fonction qui permet seulement de charger les paramètres sauv
def load_models_from_file(filename):

```

```
with open(filename, 'r') as f:
    best_params = json.load(f)

models = {}

if "DecisionTree" in best_params:
    models["DecisionTree"] = DecisionTreeClassifier(**best_params["Decision

if "MultinomialNB" in best_params:
    models["MultinomialNB"] = MultinomialNB(**best_params["MultinomialNB"])

if "KNN" in best_params:
    models["KNN"] = KNeighborsClassifier(**best_params["KNN"])

if "SVC" in best_params:
    models["SVC"] = SVC(**best_params["SVC"], random_state=42)

return models

lvl1_best_params = load_models_from_file("dataSet/lvl1_parameters.json")

print(lvl1_best_params)

{'DecisionTree': DecisionTreeClassifier(criterion='entropy', min_samples_sp
                                     random_state=42), 'MultinomialNB': MultinomialNB(alp
```

▼ Decision Tree

```
best_tree = lvl1_best_params["DecisionTree"]

print("Paramètres :", best_tree.get_params())

best_tree.fit(X_resampled, y_resampled)

# Cross-validation sur données équilibrées
cv_scores = cross_val_score(best_tree, X_resampled, y_resampled, cv=10)
print("Scores CV :", cv_scores)
print("Moyenne CV :", cv_scores.mean())

# Prédiction sur le vrai test (non modifié)
y_pred = best_tree.predict(X_test)
print("\n Accuracy (test) :", accuracy_score(y_test, y_pred))
print(" Classification Report (test) :")
print(classification_report(y_test, y_pred))

# Matrice de confusion
conf_matrix = confusion_matrix(y_test, y_pred)

# Tes fonctions de visualisation (si elles existent)
plot_curves(cv_scores)
plot_curves_confusion(conf_matrix, ['NON-SCI', 'SCI'])
```


✓ Naive bayes

```
best_naive_bayes_classifier = lvl1_best_params["MultinomialNB"]

print("Paramètres :", best_naive_bayes_classifier.get_params())

best_naive_bayes_classifier.fit(X_resampled, y_resampled)

naive_scores = cross_val_score(best_naive_bayes_classifier, X_resampled, y_resa
print("Scores CV :", naive_scores)
print("Moyenne CV :", naive_scores.mean())

y_pred_test = best_naive_bayes_classifier.predict(X_test)

# Rapports
print("\n Accuracy (test) :", accuracy_score(y_test, y_pred_test))
print("Classification Report (test) :")
print(classification_report(y_test, y_pred_test))

# Matrice de confusion
conf_matrix = confusion_matrix(y_test, y_pred_test)

# Visualisation
plot_curves(naive_scores)
plot_curves_confusion(conf_matrix, ['NON-SCI', 'SCI'])
```


✓ SVC

```
best_svc_classifier = lvl1_best_params["SVC"]

print("Paramètres :", best_svc_classifier.get_params())

best_svc_classifier.fit(X_resampled, y_resampled)

svc_scores = cross_val_score(best_svc_classifier, X_resampled, y_resampled, cv=
print("Scores de validation croisée :", svc_scores)
print("Moyenne :", svc_scores.mean())

y_pred_test = best_svc_classifier.predict(X_test)

conf_matrix = confusion_matrix(y_test, y_pred_test)
plot_curves(svc_scores)
plot_curves_confusion(conf_matrix, ['NON-SCI', 'SCI'])
```

✓ KNN

```
best_knn_classifier = lvl1_best_params["KNN"]

print("Paramètres :", best_knn_classifier.get_params())

best_knn_classifier.fit(X_resampled, y_resampled)

knn_scores = cross_val_score(best_knn_classifier, X_resampled, y_resampled, cv=
print("Scores de validation croisée :", knn_scores)
print("Moyenne :", knn_scores.mean())

y_pred_test = best_knn_classifier.predict(X_test)
print("\n Accuracy (test) :", accuracy_score(y_test, y_pred_test))
print("Rapport de classification :\n", classification_report(y_test, y_pred_test))

conf_matrix = confusion_matrix(y_test, y_pred_test)

plot_curves(knn_scores)
plot_curves_confusion(conf_matrix, ['NON-SCI', 'SCI'])
```


✓ Evaluation des classifieurs (Niveau 1)

```
def compute_mean_and_ci(scores):
    mean = scores.mean()
    std = scores.std()
    n = len(scores)
    ci_width = stats.t.ppf(0.975, df=n-1) * (std / np.sqrt(n)) # Half-width (±
    return mean, ci_width

def print_accuracy_with_pm(name, scores):
    mean, ci_width = compute_mean_and_ci(scores)
    pm_percent = (ci_width / mean) * 100
    print(f"{name} : {mean:.4f} ± {pm_percent:.2f}%")

def plot_curves_results_with_ci(scores_list, names=None, colors=None):
    means = []
    cis = []

    for scores in scores_list:
        mean, ci_width = compute_mean_and_ci(scores)
        means.append(mean)
        cis.append(ci_width)

    if names is None:
        names = [f"Model {i+1}" for i in range(len(scores_list))]

    if colors is None:
        colors = ['royalblue', 'seagreen', 'tomato', 'mediumpurple']

    # Plot
    x = np.arange(len(names))
    plt.figure(figsize=(10,6))
    bars = plt.bar(x, means, yerr=cis, capsize=8, color=colors, edgecolor='blac

    plt.xticks(x, names)
    plt.ylabel('Accuracy')
    plt.title('Score moyen des classifieurs')
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.ylim(0, 1)
```

```
plt.show()

# Plot courbes
plot_curves_results_with_ci(
    [naive_scores, svc_scores, cv_scores, knn_scores],
    names=['Naive Bayes', 'SVC', 'Decision Tree', 'KNN']
)

# Print précision ± confiance
print_accuracy_with_pm("Naive Bayes", naive_scores)
print_accuracy_with_pm("SVC", svc_scores)
print_accuracy_with_pm("Decision Tree", cv_scores)
print_accuracy_with_pm("KNN", knn_scores)
```

✓ Classification {CLAIM, REF} vs CONTEXT (Niveau 2)

On définit nos données d'entraînement pour cette classification de niveau 2 :

On utilise nos données d'entraînement pour cette classification de niveau 2 :

```
# Copie des données pour le niveau 2
data_lvl2 = dataPrepared.copy()
# Combiner 'scientific_claim' et 'scientific_reference' en une seule colonne
data_lvl2['claim_or_ref'] = data_lvl2.apply(lambda row: 1 if row['scientific_cl
y = data_lvl2['claim_or_ref'] # la colonne cible pour le niveau 2
X_text = data_lvl2['text']

# Vectorisation TF-IDF + Scaling =====
vectorizer = TfidfVectorizer(ngram_range=(1, 2), min_df=5, max_df=0.9)
X_vectorized = vectorizer.fit_transform(X_text)

scaler = MaxAbsScaler()
X_scaled = scaler.fit_transform(X_vectorized)

# Split train/test =====
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,

print(f"sizes of the classes before resampling : {Counter(y_train)}")

X_resampled, y_resampled = resampleData(X_train, y_train)

print(f"sizes of the classes after resampling : {Counter(y_resampled)}")

    sizes of the classes before resampling : Counter({0: 638, 1: 273})
    sizes of the classes after resampling : Counter({0: 634, 1: 634})
```

✓ Recherche des paramètres optimaux des classifieurs

```
perform_gridsearch_and_plot(X_resampled, y_resampled, X_test, y_test, 2, "dataS

Loaded best parameters from dataSet/lvl2_parameters.json
Best parameters for DecisionTree (Level 2): {'criterion': 'entropy', 'max_d
Best parameters for MultinomialNB (Level 2): {'alpha': 0.002656087782946686
Best parameters for KNN (Level 2): {'metric': 'manhattan', 'n_neighbors': 8
Best parameters for SVC (Level 2): {'C': 17.47528400007683, 'gamma': 'scale

lvl2_best_params = load_models_from_file("dataSet/lvl2_parameters.json")

print(lvl2_best_params)

{'DecisionTree': DecisionTreeClassifier(criterion='entropy', min_samples_sp
random_state=42), 'MultinomialNB': MultinomialNB(alp
```

✓ Decision Tree

```
best_tree = lvl2_best_params["DecisionTree"]
```

```
print("Paramètres :", best_tree.get_params())

best_tree.fit(X_resampled, y_resampled)

# Cross-validation
cv_scores = cross_val_score(best_tree, X_resampled, y_resampled, cv=10)
print("Scores CV :", cv_scores)
print("Moyenne CV :", cv_scores.mean())

# Prédiction
y_pred = best_tree.predict(X_test)
print("\n Accuracy (test) :", accuracy_score(y_test, y_pred))
print("Classification Report (test) :")
print(classification_report(y_test, y_pred))

conf_matrix = confusion_matrix(y_test, y_pred)

plot_curves(cv_scores)
plot_curves_confusion(conf_matrix, ['CONTEXT', 'CLAIM/REF'])
```

✓ Naive bayes

```
best_naive_bayes_classifier = lvl2_best_params["MultinomialNB"]  
  
print("Paramètres :", best_naive_bayes_classifier.get_params())  
  
best_naive_bayes_classifier.fit(X_resampled, y_resampled)  
  
naive_scores = cross_val_score(best_naive_bayes_classifier, X_resampled, y_resa  
print("Scores CV :", naive_scores)  
print("Moyenne CV :", naive_scores.mean())
```

```
y_pred_test = best_naive_bayes_classifier.predict(X_test)

print("\n Accuracy (test) :", accuracy_score(y_test, y_pred_test))
print("Classification Report (test) :")
print(classification_report(y_test, y_pred_test))

conf_matrix = confusion_matrix(y_test, y_pred_test)

plot_curves(naive_scores)
plot_curves_confusion(conf_matrix, ['CONTEXT', 'CLAIM/REF']) # Updated labels f
```

✓ SVC

```
# --- Step 1: RESAMPLE THE TRAINING DATA TO BALANCE CLASSES ---

# Optional: use imblearn if your resampleData() doesn't work well
# ros = RandomOverSampler(random_state=42)
# X_resampled, y_resampled = ros.fit_resample(X_train, y_train)

# Convert to dense format if sparse
X_resampled_dense = X_resampled.toarray() if hasattr(X_resampled, "toarray") el
X_test_dense = X_test.toarray() if hasattr(X_test, "toarray") else X_test

# --- Step 2: SCALE BOTH TRAIN AND TEST DATA USING SAME SCALER ---
scaler = StandardScaler()
X_resampled_scaled = scaler.fit_transform(X_resampled_dense) # fit on train
X_test_scaled = scaler.transform(X_test_dense) # transform test with same scal

# --- Step 3: HYPERPARAMETER SEARCH FOR SVC ---

# --- Step 4: VALIDATE WITH CROSS-VAL ON TRAIN ---

best_svc_classifier = 1v12 best_params["SVC"]
```

```
-----  
print("Paramètres :", best_svc_classifier.get_params())  
  
best_svc_classifier.fit(X_resampled_scaled, y_resampled)  
  
svc_scores = cross_val_score(best_svc_classifier, X_resampled_scaled, y_resampled,  
                              cv=5, scoring='accuracy')  
print("Cross-val scores:", svc_scores)  
print("Mean:", svc_scores.mean())  
  
# --- Step 5: EVALUATE ON TEST SET ---  
  
y_pred_test = best_svc_classifier.predict(X_test_scaled)  
  
print("\nAccuracy (test):", accuracy_score(y_test, y_pred_test))  
print("Classification Report (test):")  
print(classification_report(y_test, y_pred_test))  
  
conf_matrix = confusion_matrix(y_test, y_pred_test)  
plot_curves(svc_scores)  
plot_curves_confusion(conf_matrix, ['CONTEXT', 'CLAIM/REF'])
```

✓ KNN

```
best_knn_classifier = lvl2_best_params["KNN"]

print("Paramètres :", best_knn_classifier.get_params())

best_knn_classifier.fit(X_resampled, y_resampled)

knn_scores = cross_val_score(best_knn_classifier, X_resampled, y_resampled, cv=
print("Scores de validation croisée :", knn_scores)
print("Moyenne :", knn_scores.mean())

y_pred_test = best_knn_classifier.predict(X_test_scaled)
print("\n Accuracy (test) :", accuracy_score(y_test, y_pred_test))
print("Rapport de classification :\n" , classification_report(y_test, y_pred_test))
```

```
print( 'Rapport de classification : m', classification_report(y_test, y_pred_test)

conf_matrix = confusion_matrix(y_test, y_pred_test)

plot_curves(knn_scores)
plot_curves_confusion(conf_matrix, ['CONTEXT', 'CLAIM/REF'])
```


✓ Evaluation des classifieurs (niveau 2)

```
#Evaluation des classifieurs
```

```
# Plot courbes
```

```
plot_curves_results_with_ci(  
    [naive_scores, svc_scores, cv_scores, knn_scores],  
    names=['Naive Bayes', 'SVC', 'Decision Tree', 'KNN']  
)
```

```
# Print précision ± confiance
```

```
print_accuracy_with_pm("Naive Bayes", naive_scores)  
print_accuracy_with_pm("SVC", svc_scores)  
print_accuracy_with_pm("Decision Tree", cv_scores)  
print_accuracy_with_pm("KNN", knn_scores)
```

✓ Classification {CLAIM} VS {REF} VS {CONTEXT} (niveau 3)

On définit nos données d'entraînement pour cette classification de niveau 3 :

```
data_lvl3 = dataPrepared.copy()

def get_level3_label(row):
    if row['scientific_claim'] == 1:
        return 'CLAIM'
    elif row['scientific_reference'] == 1:
        return 'REF'
    elif row['scientific_context'] == 1:
        return 'CONTEXT'
    else:
        return 'NON-SCI'

def apply_level3_label(data):
    data['level3_label'] = data.apply(get_level3_label, axis=1)
    # drop all non sci
    data = data[data['level3_label'] != 'NON-SCI']
    return data

data_lvl3 = apply_level3_label(data_lvl3)
y = data_lvl3['level3_label']
X_text = data_lvl3['text']

vectorizer = TfidfVectorizer(ngram_range=(1, 2), min_df=5, max_df=0.9)
X_vectorized = vectorizer.fit_transform(X_text)
```

```
scaler = MaxAbsScaler()
X_scaled = scaler.fit_transform(X_vectorized)
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42, stratify=y
)

print(f"sizes of the classes before resampling : {Counter(y_train)}")

X_resampled, y_resampled = resampleData(X_train, y_train)

print(f"sizes of the classes after resampling : {Counter(y_resampled)}")

    sizes of the classes before resampling : Counter({'CLAIM': 210, 'REF': 62,
    sizes of the classes after resampling : Counter({'CONTEXT': 210, 'CLAIM': 2
```

✓ Recherche des paramètres optimaux des classifieurs

```
perform_gridsearch_and_plot(X_resampled, y_resampled, X_test, y_test, 3, "dataS
```



```
lvl3_best_params = load_models_from_file("dataSet/lvl3_parameters.json")

print(lvl3_best_params)

{'DecisionTree': DecisionTreeClassifier(criterion='entropy', random_state=4
```

▼ Decision tree

```
best_tree = lvl3_best_params["DecisionTree"]

print("Paramètres :", best_tree.get_params())

best_tree.fit(X_resampled, y_resampled)

# Cross-validation
cv_scores = cross_val_score(best_tree, X_resampled, y_resampled, cv=10)
print("Scores CV :", cv_scores)
print("Moyenne CV :", cv_scores.mean())

# Prediction
y_pred = best_tree.predict(X_test)
print("\n Accuracy (test) :", accuracy_score(y_test, y_pred))
print("Classification Report (test) :")
print(classification_report(y_test, y_pred))

conf_matrix = confusion_matrix(y_test, y_pred)

plot_curves(cv_scores)
plot_curves_confusion(conf_matrix, ['CONTEXT', 'CLAIM', 'REF'])
```


✓ Naive bayes

```
best_naive_bayes_classifier = lvl3_best_params["MultinomialNB"]

print("Paramètres :", best_naive_bayes_classifier.get_params())

best_naive_bayes_classifier.fit(X_resampled, y_resampled)

naive_scores = cross_val_score(best_naive_bayes_classifier, X_resampled, y_resa
print("Scores CV :", naive_scores)
print("Moyenne CV :", naive_scores.mean())

y_pred_test = best_naive_bayes_classifier.predict(X_test)

print("\n Accuracy (test) :", accuracy_score(y_test, y_pred_test))
print("Classification Report (test) :")
print(classification_report(y_test, y_pred_test))

conf_matrix = confusion_matrix(y_test, y_pred_test)

plot_curves(naive_scores)
plot_curves_confusion(conf_matrix, ['CONTEXT', 'CLAIM', 'REF'])
```

✓ **SVC**

" " SVC

```
# ##SVC

X_resampled, y_resampled = resampleData(X_train, y_train)
print("Resampled class distribution:", Counter(y_resampled))

X_resampled_dense = X_resampled.toarray() if hasattr(X_resampled, "toarray") else X_resampled
X_test_dense = X_test.toarray() if hasattr(X_test, "toarray") else X_test

scaler = StandardScaler()
X_resampled_scaled = scaler.fit_transform(X_resampled_dense)
X_test_scaled = scaler.transform(X_test_dense)
clf_SVC = SVC()

best_svc_classifier = lvl3_best_params["SVC"]

print("Paramètres :", best_svc_classifier.get_params())

best_svc_classifier.fit(X_resampled_scaled, y_resampled)

svc_scores = cross_val_score(best_svc_classifier, X_resampled_scaled, y_resampled, cv=5)
print("Cross-val scores:", svc_scores)
print("Mean:", svc_scores.mean())

# --- Step 5: EVALUATE ON TEST SET ---

y_pred_test = best_svc_classifier.predict(X_test_scaled)

print("\nAccuracy (test):", accuracy_score(y_test, y_pred_test))
print("Classification Report (test):")
print(classification_report(y_test, y_pred_test))

conf_matrix = confusion_matrix(y_test, y_pred_test)
plot_curves(svc_scores)
plot_curves_confusion(conf_matrix, ['CONTEXT', 'CLAIM', 'REF'])
```



```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import RandomizedSearchCV, KFold, cross_val_score,
from sklearn.metrics import accuracy_score, classification_report, confusion_ma
from sklearn.preprocessing import StandardScaler

# Assuming X_train, X_test, y_train, y_test from previous level 3 setup
X_resampled, y_resampled = resampleData(X_train, y_train) # use yours
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train.toarray())
X_test_scaled = scaler.transform(X_test.toarray())

best_knn_classifier = lvl3_best_params["KNN"]

print("Paramètres :", best_knn_classifier.get_params())

best_knn_classifier.fit(X_resampled, y_resampled)

knn_scores = cross_val_score(best_knn_classifier, X_resampled, y_resampled, cv=
print("Scores de validation croisée :", knn_scores)
print("Moyenne :", knn_scores.mean())

y_pred_test = best_knn_classifier.predict(X_test)
print("\n Accuracy (test) :", accuracy_score(y_test, y_pred_test))
print("Rapport de classification :\n", classification_report(y_test, y_pred_tes

conf_matrix = confusion_matrix(y_test, y_pred_test)

plot_curves(knn_scores)
plot_curves_confusion(conf_matrix, ['CONTEXT', 'CLAIM', 'REF'])
```

▼ Evaluation des classifieurs (Niveau 3)

```
# Evaluation des classifieurs

# Plot courbes
plot_curves_results_with_ci(
    [naive_scores, svc_scores, cv_scores, knn_scores],
    names=['Naive Bayes', 'SVC', 'Decision Tree', 'KNN']
)

# Print précision ± confiance
print_accuracy_with_pm("Naive Bayes", naive_scores)
print_accuracy_with_pm("SVC", svc_scores)
print_accuracy_with_pm("Decision Tree", cv_scores)
print_accuracy_with_pm("KNN", knn_scores)
```

✓ Optimisation #1 : Utiliser un dataSet plus adapté en faisant varier prepareText

Actuellement quand nous avons lancé notre fonction prepareText nous avons appliqué

Actuellement, quand nous avons lancé notre fonction `prepare_text`, nous avons appliqué tous les paramètres possibles. Cependant, il est possible que nous obtenions un `dataSet` de meilleure qualité suivant les paramètres appliqués. Pour cela, nous allons faire tourner notre fonction de préparation de données avec chaque combinaison de paramètre (2^4 possibilités) et par la suite, nous utiliserons des outils statistiques afin de déterminer quel est le meilleur `dataSet` à utiliser.

Comme précédemment, on va stocker les prétraitements sous forme de fichier afin de gagner du temps lors des réexecutions.

```
# Define preprocessing parameter combinations
combinations = {
    "0000": {"keepTokens": False, "keepEmojis": False, "numbersAsTokens": False
    "0001": {"keepTokens": False, "keepEmojis": False, "numbersAsTokens": False
    "0010": {"keepTokens": False, "keepEmojis": True, "numbersAsTokens": False,
    "0011": {"keepTokens": False, "keepEmojis": True, "numbersAsTokens": False,
    "0100": {"keepTokens": False, "keepEmojis": False, "numbersAsTokens": True,
    "0101": {"keepTokens": False, "keepEmojis": False, "numbersAsTokens": True,
    "0110": {"keepTokens": False, "keepEmojis": True, "numbersAsTokens": True,
    "0111": {"keepTokens": False, "keepEmojis": True, "numbersAsTokens": True,
    "1000": {"keepTokens": True, "keepEmojis": False, "numbersAsTokens": False,
    "1001": {"keepTokens": True, "keepEmojis": False, "numbersAsTokens": False,
    "1010": {"keepTokens": True, "keepEmojis": True, "numbersAsTokens": False,
    "1011": {"keepTokens": True, "keepEmojis": True, "numbersAsTokens": False,
    "1100": {"keepTokens": True, "keepEmojis": False, "numbersAsTokens": True,
    "1101": {"keepTokens": True, "keepEmojis": False, "numbersAsTokens": True,
    "1110": {"keepTokens": True, "keepEmojis": True, "numbersAsTokens": True,
    "1111": {"keepTokens": True, "keepEmojis": True, "numbersAsTokens": True, "
}

# Ensure dataset folder exists
os.makedirs("dataSet", exist_ok=True)

# Dictionary to store all dataframes
dataPrepared = {}

# Process all datasets
for key, params in combinations.items():
    file_path = f"dataSet/precomputed/dataPrepared{key}.csv"

    if os.path.exists(file_path):
        print(f>Loading existing dataset: {file_path}")
        dataPrepared[key] = pd.read_csv(file_path) # Store in dictionary
    else:
        print(f>Processing and saving: {file_path}")
        dataPrepared[key] = df.copy()
        dataPrepared[key]["text"] = dataPrepared[key]["text"].apply(lambda x: p
        dataPrepared[key].to_csv(file_path, index=False)

print("\nAll 16 datasets are ready and stored in `dataPrepared` dictionary!")

Loading existing dataset: dataSet/precomputed/dataPrepared0000.csv
```

```

Loading existing dataset: dataSet/precomputed/dataPrepared0001.csv
Loading existing dataset: dataSet/precomputed/dataPrepared0010.csv
Loading existing dataset: dataSet/precomputed/dataPrepared0011.csv
Loading existing dataset: dataSet/precomputed/dataPrepared0100.csv
Loading existing dataset: dataSet/precomputed/dataPrepared0101.csv
Loading existing dataset: dataSet/precomputed/dataPrepared0110.csv
Loading existing dataset: dataSet/precomputed/dataPrepared0111.csv
Loading existing dataset: dataSet/precomputed/dataPrepared1000.csv
Loading existing dataset: dataSet/precomputed/dataPrepared1001.csv
Loading existing dataset: dataSet/precomputed/dataPrepared1010.csv
Loading existing dataset: dataSet/precomputed/dataPrepared1011.csv
Loading existing dataset: dataSet/precomputed/dataPrepared1100.csv
Loading existing dataset: dataSet/precomputed/dataPrepared1101.csv
Loading existing dataset: dataSet/precomputed/dataPrepared1110.csv
Loading existing dataset: dataSet/precomputed/dataPrepared1111.csv

```

All 16 datasets are ready and stored in `dataPrepared` dictionary!

On applique la vectorisation sur chaque dataSet :

```

for key, dataset in dataPrepared.items():
    print(f"Dataset: {key}")
    file_path = f'dataSet/vectorized_{key}.csv'
    if os.path.exists(file_path):
        print(f"Le fichier {file_path} existe déjà, skipping vectorization for
        continue
    # Nettoyage du dataset courant
    dataset = dataset[dataset['text'] != '']
    dataset = dataset.dropna(subset=['text'])
    dataset['text'] = dataset['text'].fillna('').astype(str)
    dataset['text'] = dataset['text'].apply(lemmatize_taggenize_sentence)
    # Traitement avec gestion d'erreur
    processed_text = []
    for index, text in dataset['text'].items():
        if pd.notnull(text) and text.strip() != "":
            try:
                processed_text.append(process_text_column(text))
            except Exception as e:
                print(f"Erreur à l'index {index} avec le texte : {text}")
                print(f"Exception : {e}")
                processed_text.append("")
        else:
            processed_text.append("")
    dataset['text'] = processed_text # Remplace directement
    # TF-IDF vectorization
    vectorizer = TfidfVectorizer(ngram_range=(1, 2), min_df=5, max_df=0.9)
    vectorized = vectorizer.fit_transform(dataset['text'])
    # Scaling
    scaled = MaxAbsScaler().fit_transform(vectorized)
    # Conversion + sauvegarde
    vectorized_df = pd.DataFrame(scaled.toarray(), columns=vectorizer.get_featu
    vectorized_df.to_csv(file_path, index=False)
    print(f"Vectorized dataset {key} saved.")
    display(vectorized_df.head())

```

```
print("\n" + "-"*50 + "\n")
```

```
Dataset: 0000
Le fichier dataSet/vectorized_0000.csv existe déjà, skipping vectorization
Dataset: 0001
Le fichier dataSet/vectorized_0001.csv existe déjà, skipping vectorization
Dataset: 0010
Le fichier dataSet/vectorized_0010.csv existe déjà, skipping vectorization
Dataset: 0011
Le fichier dataSet/vectorized_0011.csv existe déjà, skipping vectorization
Dataset: 0100
Le fichier dataSet/vectorized_0100.csv existe déjà, skipping vectorization
Dataset: 0101
Le fichier dataSet/vectorized_0101.csv existe déjà, skipping vectorization
Dataset: 0110
Le fichier dataSet/vectorized_0110.csv existe déjà, skipping vectorization
Dataset: 0111
Le fichier dataSet/vectorized_0111.csv existe déjà, skipping vectorization
Dataset: 1000
Le fichier dataSet/vectorized_1000.csv existe déjà, skipping vectorization
Dataset: 1001
Le fichier dataSet/vectorized_1001.csv existe déjà, skipping vectorization
Dataset: 1010
Le fichier dataSet/vectorized_1010.csv existe déjà, skipping vectorization
Dataset: 1011
Le fichier dataSet/vectorized_1011.csv existe déjà, skipping vectorization
Dataset: 1100
Le fichier dataSet/vectorized_1100.csv existe déjà, skipping vectorization
Dataset: 1101
Le fichier dataSet/vectorized_1101.csv existe déjà, skipping vectorization
Dataset: 1110
Le fichier dataSet/vectorized_1110.csv existe déjà, skipping vectorization
Dataset: 1111
Le fichier dataSet/vectorized_1111.csv existe déjà, skipping vectorization
```

On va applique notre Topic Modelling sur chaque dataSet :

```
def apply_lda_and_save(data, key, n_topics=15):
    file_path_lda = f"dataSet/lda_results_{key}.csv"

    if os.path.exists(file_path_lda):
        print(f"LDA results already exist for dataset {key}. Skipping LDA.")
        return pd.read_csv(file_path_lda)

    print(f"Applying LDA to dataset {key}...")

    count_vectorizer = CountVectorizer(ngram_range=(1, 2), min_df=5, max_df=0.9)
    count_matrix = count_vectorizer.fit_transform(data['text'])

    lda_model = LatentDirichletAllocation(n_components=n_topics, random_state=4)
    lda_topics = lda_model.fit_transform(count_matrix)
    topic_assignments = np.argmax(lda_topics, axis=1)
    data['Topic'] = topic_assignments

    data.to_csv(file_path_lda, index=False)
    print(f"LDA results for dataset {key} saved.")
```

```
print('LDA results for dataset {key} saved. ')
return data
```

```
for key in dataPrepared:
    dataPrepared[key] = apply_lda_and_save(dataPrepared[key], key)
```

```
LDA results already exist for dataset 0000. Skipping LDA.
LDA results already exist for dataset 0001. Skipping LDA.
LDA results already exist for dataset 0010. Skipping LDA.
LDA results already exist for dataset 0011. Skipping LDA.
LDA results already exist for dataset 0100. Skipping LDA.
LDA results already exist for dataset 0101. Skipping LDA.
LDA results already exist for dataset 0110. Skipping LDA.
LDA results already exist for dataset 0111. Skipping LDA.
LDA results already exist for dataset 1000. Skipping LDA.
LDA results already exist for dataset 1001. Skipping LDA.
LDA results already exist for dataset 1010. Skipping LDA.
LDA results already exist for dataset 1011. Skipping LDA.
LDA results already exist for dataset 1100. Skipping LDA.
LDA results already exist for dataset 1101. Skipping LDA.
LDA results already exist for dataset 1110. Skipping LDA.
LDA results already exist for dataset 1111. Skipping LDA.
```

On a maintenant tous les dataSet vectorisés, on va chercher quel est le dataSet optimal. Pour cela, on va évaluer la cohérence de chaque dataSet :

```
def compute_topic_coherence(texts, n_topics=15):
    stop_words = set(stopwords.words('english'))
    tokenized_texts = [[word for word in doc.lower().split() if word not in stop_words]
                        for doc in texts]

    # Create Dictionary and Corpus
    dictionary = Dictionary(tokenized_texts)
    corpus = [dictionary.doc2bow(text) for text in tokenized_texts]

    # Gensim LDA model (not scikit-learn)
    lda_model = LdaModel(corpus=corpus,
                        id2word=dictionary,
                        num_topics=n_topics,
                        random_state=42,
                        passes=10)

    # Compute coherence
    coherence_model = CoherenceModel(model=lda_model, texts=tokenized_texts,
                                     dictionary=dictionary, num_topics=n_topics)
    coherence_score = coherence_model.get_coherence()
    return coherence_score

coherence_scores = {}

max_score=0
max_key=""
for key in dataPrepared:
    df = dataPrepared[key]
```

```

coherence = compute_topic_coherence(df['text'].tolist(), n_topics=15)
coherence_scores[key] = coherence
print(f"Dataset '{key}' → Coherence Score: {coherence:.4f}")
if(coherence>max_score):
    max_score=coherence
    max_key=key

print(f"The best option is {max_key} with score {max_score}")

```

```

Dataset '0000' → Coherence Score: 0.3879
Dataset '0001' → Coherence Score: 0.3921
Dataset '0010' → Coherence Score: 0.4180
Dataset '0011' → Coherence Score: 0.4029
Dataset '0100' → Coherence Score: 0.4220
Dataset '0101' → Coherence Score: 0.4098
Dataset '0110' → Coherence Score: 0.4221
Dataset '0111' → Coherence Score: 0.4247
Dataset '1000' → Coherence Score: 0.3653
Dataset '1001' → Coherence Score: 0.3843
Dataset '1010' → Coherence Score: 0.3812
Dataset '1011' → Coherence Score: 0.3644
Dataset '1100' → Coherence Score: 0.3682
Dataset '1101' → Coherence Score: 0.3705
Dataset '1110' → Coherence Score: 0.3621
Dataset '1111' → Coherence Score: 0.3616
The best option is 0111 with score 0.42465753951006896

```

On va utiliser alors pour chaque niveau les classifieurs qui on eu le meilleur score.

✓ Optimisation #2 : on va utiliser Optuna, un outil de recherche d'hyperparamètres plus performant

En plus du dataSet optimisé, on va non pas utiliser SearchGridCV mais optuna qui est à priori plus efficace pour essayer d'obtenir une meilleur accuracy.

Pour chaque niveau de classification, on va appliquer nos 2 optimisations pour le meilleur classifieur obtenu précédement (ne montrant aucun signe de surapprentissage).

✓ Niveau 1 : SVC

On utilise notre meilleur dataSet avec optuna sur le clasifieur SVC pour voir comment nos résultats sont influencés.

```
import pandas as pd
from sklearn.svm import SVC
import optuna

# -- Copie de tes données (adjust to your actual data loading)
data_lv11 = dataPrepared["0111"] # Use the best dataset from optimization #1
y = data_lv11['science_related']
X_text = data_lv11['text']

# -- Vectorisation TF-IDF + Scaling (unchanged)
vectorizer = TfidfVectorizer(ngram_range=(1, 2), min_df=5, max_df=0.9)
X_vectorized = vectorizer.fit_transform(X_text)
scaler_tfidf = MaxAbsScaler()
X_scaled = scaler_tfidf.fit_transform(X_vectorized)

# -- Split (unchanged)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, :

# -- Standard Scaling for the model (unchanged)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train.toarray())
X_test_scaled = scaler.transform(X_test.toarray())

# -- Upsampling to balance (unchanged)
X_df = pd.DataFrame(X_train_scaled)
X_df['label'] = y_train.values
df_majority = X_df[X_df['label'] == 0]
df_minority = X_df[X_df['label'] == 1]
df_minority_upsampled = resample(
    df_minority,
    replace=True,
    n_samples=len(df_majority),
    random_state=42
)
df_upsampled = pd.concat([df_majority, df_minority_upsampled])
X_train_balanced = df_upsampled.drop('label', axis=1).values
y_train_balanced = df_upsampled['label'].values

# -- Optuna: optimization of the SVC
def objective(trial):
    C = trial.suggest_float("C", 1e-3, 1e3, log=True)
    kernel = trial.suggest_categorical("kernel", ["linear", "rbf", "poly"])
    gamma = trial.suggest_categorical("gamma", ["scale", "auto"])
    svc = SVC(C=C, kernel=kernel, gamma=gamma, random_state=42) # Use the best
    scores = cross_val_score(svc, X_train_balanced, y_train_balanced, cv=KFold(n
    return scores.mean()

# -- Create and launch the study
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100) # Adjust n_trials as needed

print("Best hyperparameters:", study.best_params)
print("Best accuracy:", study.best_value)
```

```
# Train the model with best hyperparameters
best_svc = SVC(**study.best_params, random_state=42)
best_svc.fit(X_train_balanced, y_train_balanced)

# Evaluate on the test set
y_pred = best_svc.predict(X_test_scaled)
print(classification_report(y_test, y_pred))

#Affiche la matrice de confusion
conf_matrix = confusion_matrix(y_test, y_pred)
plot_curves_confusion(conf_matrix, ['NON-SCI', 'SCI'])
```


Si on compare avec optuna et sans, on passe d'une accuracy de 0.8899 à 0.9101 (soit une augmentation de 2.02 %)

✓ Niveau 2 : SVC

Pour le niveau 2, c'était SVC le plus pertinent. On va donc le comparer avec optuna

```
import optuna
from sklearn.model_selection import cross_val_score, KFold
from sklearn.svm import SVC
from sklearn.metrics import classification_report, confusion_matrix
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
from sklearn.preprocessing import MaxAbsScaler, StandardScaler
from imblearn.over_sampling import RandomOverSampler
from collections import Counter
from sklearn.utils import resample
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# ... (Your existing code for data loading and preprocessing) ...

# Assuming data_lvl2, y, X_text, X_train, X_test, y_train, y_test are defined f
# Copie des données niveau 2 (sur 0111)
data_lvl2 = dataPrepared["0111"].copy()
# 2. Création de la colonne cible (claim or reference)
data_lvl2['claim_or_ref'] = data_lvl2.apply(lambda row: 1 if row['scientific_cl
y = data_lvl2['claim_or_ref']
X_text = data_lvl2['text']
# 3. Vectorisation TF-IDF + MaxAbsScaler
vectorizer = TfidfVectorizer(ngram_range=(1, 2), min_df=5, max_df=0.9)
X_vectorized = vectorizer.fit_transform(X_text)
scaler = MaxAbsScaler()
X_scaled = scaler.fit_transform(X_vectorized)
# 4. Split train/test
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,

# Upsampling (adjust to your resampling method)
X_df = pd.DataFrame(X_train.toarray())
X_df['label'] = y_train.values
df_majority = X_df[X_df['label'] == 0]
df_minority = X_df[X_df['label'] == 1]
df_minority_upsampled = resample(
    df_minority,
    replace=True,
    n_samples=len(df_majority),
    random_state=42
)
df_upsampled = pd.concat([df_majority, df_minority_upsampled])
X_train_balanced = df_upsampled.drop('label', axis=1).values
y_train_balanced = df_upsampled['label'].values

# StandardScaler for SVC
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_balanced)
X_test_scaled = scaler.transform(X_test.toarray())

def objective(trial):
    C = trial.suggest_float("C", 1e-3, 1e3, log=True)
    kernel = trial.suggest_categorical("kernel", ["linear", "rbf", "poly"])
    gamma = trial.suggest_categorical("gamma", ["scale", "auto"])
    svc = SVC(C=C, kernel=kernel, gamma=gamma, random_state=42)
    scores = cross_val_score(svc, X_train_scaled, y_train_balanced, cv=KFold(n_
    return scores.mean()
```

```
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100)

print("Best hyperparameters:", study.best_params)
print("Best accuracy:", study.best_value)

best_svc = SVC(**study.best_params, random_state=42)
best_svc.fit(X_train_scaled, y_train_balanced)

y_pred = best_svc.predict(X_test_scaled)
print(classification_report(y_test, y_pred))

conf_matrix = confusion_matrix(y_test, y_pred)
plot_curves_confusion(conf_matrix, ['CONTEXT', 'CLAIM/REF'])
```


On passe d'une accuracy de 0.9023 à 0.9169 (une augmentation de 1.46%)

✓ Niveau 3 : SVC

Pour notre niveau 3, c'était SVC le plus performant :

```
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, cross_val_score, KFold
```

```
from sklearn.metrics import classification_report, confusion_matrix
from imblearn.over_sampling import RandomOverSampler
import optuna
import pandas as pd
import numpy as np
from collections import Counter
from sklearn.utils import resample

# Assuming data_lvl3, y, X_text, X_train, X_test, y_train, y_test are defined f
# Copie des données niveau 3 (sur 0111)
data_lvl3 = dataPrepared["0111"].copy()

# Fonction pour assigner le label du niveau 3
def get_level3_label(row):
    if row['scientific_claim'] == 1:
        return 'CLAIM'
    elif row['scientific_reference'] == 1:
        return 'REF'
    elif row['scientific_context'] == 1:
        return 'CONTEXT'
    else:
        return 'NON-SCI'

# Application de la fonction et suppression de 'NON-SCI'
data_lvl3['level3_label'] = data_lvl3.apply(get_level3_label, axis=1)
data_lvl3 = data_lvl3[data_lvl3['level3_label'] != 'NON-SCI']
y = data_lvl3['level3_label']
X_text = data_lvl3['text']

# Vectorisation TF-IDF + MaxAbsScaler
vectorizer = TfidfVectorizer(ngram_range=(1, 2), min_df=5, max_df=0.9)
X_vectorized = vectorizer.fit_transform(X_text)
scaler = MaxAbsScaler()
X_scaled = scaler.fit_transform(X_vectorized)

# Split train/test
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,

# Upsampling avec RandomOverSampler (adjust to your resampling method)
ros = RandomOverSampler(random_state=42)
X_resampled, y_resampled = ros.fit_resample(X_train, y_train)

# StandardScaler for SVC
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_resampled.toarray()) # Fit and transfor
X_test_scaled = scaler.transform(X_test.toarray()) # Transform the testing data

def objective(trial):
    C = trial.suggest_float("C", 1e-3, 1e3, log=True)
    kernel = trial.suggest_categorical("kernel", ["linear", "rbf", "poly"])
    gamma = trial.suggest_categorical("gamma", ["scale", "auto"])
```

```
svc = SVC(C=C, kernel=kernel, gamma=gamma, random_state=42)
scores = cross_val_score(svc, X_train_scaled, y_resampled, cv=KFold(n_split
return scores.mean())

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100)

print("Best hyperparameters:", study.best_params)
print("Best accuracy:", study.best_value)

best_svc = SVC(**study.best_params, random_state=42)
best_svc.fit(X_train_scaled, y_resampled)

y_pred = best_svc.predict(X_test_scaled)
print(classification_report(y_test, y_pred))

conf_matrix = confusion_matrix(y_test, y_pred)
plot_curves_confusion(conf_matrix, ['CONTEXT', 'CLAIM', 'REF'])
```


On passe cette fois de 0.9023 à 0.9523 soit une augmentation de 5% au vu de la matrice de confusion, cela ne semble pas être le meilleur classifieur.

