ProjetML_G6_Gallerne_Navel_Gilles_Picole-Ollivier

May 26, 2024

1 HAI817 - Machine Learning 1

2 Projet Machine Learning

---

3 2023-2024 - Semestre 2 Master 1 Informatique

4 Détection automatique des fake news à partir de données textuelles

4.0.1 Sujet

4.1 Auteurs :

- 22010416 - Romain Gallerne (IASD)
- 22009176 - Morgan Navel (GL)
- 21809267 - Éric Gilles (GL)
- 22004340 - Richard Picole-Ollivier (GL)

#Installation et pré-traitements

4.2 Installation

Installations et import de toutes les librairies nécessaires.

!! ATTENTION A LA VERSION DE GOOGLE TRANS : 4.0.0-rc1 !!

[ ]:
```
from google.colab import drive
drive.mount('/content/gdrive/')
```

[ ]:
```
# Importation des différentes librairies utiles pour le notebook
!pip install nlputils
!pip install pyLDAvis
```

```
!pip install googletrans==4.0.0-rc1
!pip install --upgrade nltk
```

[5]:
```
#Sickit learn met régulièrement à jour des versions et
#indique des futurs warnings.
#ces deux lignes permettent de ne pas les afficher.
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)

import warnings
import asyncio
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import sys
import string
import re
import pandas as pd
import numpy as np
import sklearn
import unicodedata
import itertools
from nltk.corpus import stopwords
from googletrans import Translator
from sklearn.naive_bayes import GaussianNB
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score
from sklearn.metrics import f1_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.naive_bayes import MultinomialNB
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_fscore_support as score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from xgboost import XGBClassifier
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from gensim.models import CoherenceModel
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import pyLDAvis

import nltk
from nltk import sent_tokenize
from nltk.stem import WordNetLemmatizer
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

Pour pouvoir lire et sauvegarder sur le répertoire Google Drive.

[ ]:
```
my_local_drive='/content/gdrive/MyDrive/Colab Notebooks/ML_FDS/
 ↪HAI817_Project_data/'
# Ajout du path pour les librairies, fonctions et données
sys.path.append(my_local_drive)
# Se positionner sur le répertoire associé
%cd $my_local_drive

%pwd
```

[ ]:
```
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('omw-1.4')
nltk.download('wordnet')
```

##Fonctions de bases des pré-traitements Fonctions essentiels qu'il faut activer pour réaliser les pré-traitement, à faire tourner obligatoirement avant de lancer les pré-traitements.

[ ]:
```
def traduire_texte(texte, translator, source_lang='de', dest_lang='en'):
    segments = [texte[i:i+5000] for i in range(0, len(texte), 5000)]  # Diviser␣
 ↪en segments de 5000 caractères
    traductions = []
    for segment in segments:
      traduction = translator.translate(segment, src=source_lang,␣
 ↪dest=dest_lang)
      traductions.append(traduction.text)
    texte_traduit = ''.join(traductions)
    return texte_traduit

def modif_rating(rating):
    if rating == "True":
      return "true"
    elif rating == "False":
      return "false"
    elif rating == "Partially True" or rating == "Partially False" :
```

```
      return "mixture"
    elif rating == "Other":
      return "other"
    else:
      print("Rating : ",rating)
      raise Exception("Rating indéterminé")

def traduire_df(df_dutch):
  translator = Translator()
  print("Traduction de l'allemand vers l'anglais :")
  for i in range(len(df_dutch)):
    if(i%10==0):
      print(round(i/len(df_dutch)*100,2),"%")
    df_dutch.loc[i,'text'] = traduire_texte(df_dutch.loc[i,'text'], translator)
    df_dutch.loc[i,'rating'] = modif_rating(df_dutch.loc[i,'our rating'])

  display(df_dutch.head())
  return df_dutch
```

[ ]:
```
def sep_ponctuation(tab_token : list, ponctuation : str):
  ponctuation_inutile = string.punctuation.replace("!","").replace("?","")
  a_separe = False
  nouveau_tab = []

  for token in tab_token :
    tup_exc = token.partition(ponctuation)

    if(tup_exc[2] != ''):
        a_separe = True

    for elem in tup_exc :
      if(elem != ''):
        nouveau_tab.append(elem)

  if(a_separe):
    return sep_ponctuation(nouveau_tab, ponctuation)
  else:
    for ponct in ponctuation_inutile:
      for i in (0,max(len(nouveau_tab)-1,0)):
        if(len(nouveau_tab)-1 >= 0):
          nouveau_tab[i] = str(nouveau_tab[i]).replace(ponct,'')
    return nouveau_tab
```

##Traduire le dataframe allemand Traduction du dataframe allemand pour l'enregistrer en version anglais. Ne faire tourner que si l'on souhaite re-générer le dataset traduit.

ATTENTION : Assez long à faire tourner !

```
[ ]: df_dutch = traduire_df(df_dutch)
     df_dutch.to_csv('HAI817_Projet_german_translate.csv', index=False)
```

4.3 Pré-Traitement

Cette partie réalise l'ensemble des pré-traitements et génère une série de fichier prétraités, ne faire tourner que si l'on souhaite regénérer tous les fichiers.

ATTENTION : Assez long à faire tourner !

```
[ ]: #Import des fichiers de base
     df1=pd.read_csv('HAI817_Projet_train.csv', sep=',')
     df2=pd.read_csv('HAI817_Projet_test.csv', sep=',')
     df_dutch=pd.read_csv('HAI817_Projet_german_translate.csv', sep=',')

     df1['text'] = df1['title'] + ' ' + df1['text']
     df2['text'] = df2['title'] + ' ' + df2['text']

     df1 = df1.drop(columns=["public_id"])
     df2 = df2.drop(columns=["ID"])

     df1 = df1.drop(columns=["title"])
     df2 = df2.drop(columns=["title"])
     df_dutch = df_dutch.drop(columns=["our rating"])

     df1 = df1.rename(columns={'our rating': 'rating'})
     df2 = df2.rename(columns={'our rating': 'rating'})
```

```
[ ]: def preTraitement(
                    dataset_allemand=False, # utilisation du dataset allemand pour␣
      ↪augmenter nos données
                    removedigit=False, # supprimer les nombres
                    token_exclamation_interrogation=False, # tokeniser les "!" et "?"
                    removestopwords=False, # supprimer les stopwords
                    lemmatisation_racinisation=False, # lematisation des termes et␣
      ↪conserver la racine
                    df1=df1,
                    df2=df2,
                    df_dutch=df_dutch
                   ):

         df1['text'].fillna('',inplace=True)
```

```
  df2['text'].fillna('',inplace=True)
  df_dutch['text'].fillna('',inplace=True)

  if dataset_allemand: #On utilise le dataset en allemand pour augmenter nos␣
↪données
    df = pd.concat([df1, df2, df_dutch], ignore_index=True)
  else:
    df = pd.concat([df1, df2], ignore_index=True)

  """
    PARTIE PRE-TOKENISATION
  """

  for i in range(len(df)):
    texte = str(df.loc[i,'text'])

    # suppression des caractères spéciaux
    texte = re.sub(r'[^\w\s]',' ', texte)
    # suppression de tous les caractères uniques
    texte = re.sub(r'\s+[a-zA-Z]\s+', ' ', texte)
    # substitution des espaces multiples par un seul espace
    texte = re.sub(r'\s+', ' ', texte, flags=re.I)

    # minuscule
    texte = texte.lower()

    # retirer les chiffres
    if removedigit:
      texte = re.sub(r'\d+', '', texte)

    """
      PARTIE POST-TOKENISATION
    """

    # découpage en tokens
    tokens = texte.split()

    # suppression ponctuation
    if(token_exclamation_interrogation):
      words = sep_ponctuation(sep_ponctuation(tokens,'?'),'!')
    else:
      table = str.maketrans('', '', string.punctuation)
      words = [token.translate(table) for token in tokens]

    # suppression des stopwords
    stop_words = set(stopwords.words('english'))
```

```
    stop_words.
↪update(['always','try','go','get','make','would','really','like','came','got','know','come'
    non_stop_words = ['d', 'o', 't']
    for word in non_stop_words:
        stop_words.discard(word)

    if removestopwords:
      words = [word for word in words if not word in stop_words]

    # lemmatisation & racinisation
    if lemmatisation_racinisation:
        lem = nltk.stem.wordnet.WordNetLemmatizer()
        words = [lem.lemmatize(word)for word in words]
        ps = nltk.stem.porter.PorterStemmer()
        words=[ps.stem(word) for word in words]

    df.loc[i,'text'] = ' '.join(words)

  return df
```

[ ]:
```
num = 0
tabs = list(itertools.product([False, True], repeat=5))

# Parcourir chaque combinaison
for tab in tabs:
  num += 1
  df_preTraite = preTraitement(tab[0],tab[1],tab[2],tab[3],tab[4])
  name = ''
  for i in range(len(tab)):
    if(tab[i]): name += 'T'
    else: name += 'F'
  df_preTraite.to_csv('HAI817_Projet_Traitement_'+name+'.csv', index=False)
  print(name,"exporté (",num/0.32,"% )")
```

5 Topic Modeling

##Modélisation des différents topic

Fichier Pré-traité

Importation du fichier déjà pré-traité.

[ ]:
```
import re
import spacy
import gensim
import string
import nltk
```

```
from nltk.corpus import stopwords
from nltk.corpus import wordnet
import gensim
from gensim.utils import simple_preprocess
from gensim.models import Phrases
from gensim.models.phrases import Phraser
from gensim import corpora
from gensim import models
nlp = spacy.load("en_core_web_sm", disable=['parser', 'ner'])
stop_words = set(stopwords.words('english'))

def MyCleanTextsforLDA(texts,
                       min_count=1, # nombre d'apparitions minimale pour un␣
 ↪bigram
                       threshold=2,
                       no_below=1, # nombre minimum d'apparitions pour être dans␣
 ↪le dictionnaire
                       no_above=0.5, # pourcentage maximal (sur la taille totale␣
 ↪du corpus) pour filtrer
                       stop_words=stop_words
                       ):

    allowed_postags=['NOUN', 'ADJ', 'VERB', 'ADV']
    sentences=texts.copy()

    # suppression des caractères spéciaux
    sentences = [re.sub(r'[^\w\s]', ' ', str(sentence)) for sentence in␣
 ↪sentences]
    # suppression de tous les caractères uniques
    sentences = [re.sub(r'\s+[a-zA-Z]\s+', ' ', str(sentence)) for sentence in␣
 ↪sentences]
    # substitution des espaces multiples par un seul espace
    sentences = [re.sub(r'\s+', ' ', str(sentence), flags=re.I) for sentence in␣
 ↪sentences]

    # conversion en minuscule et split des mots dans les textes
    sentences = [sentence.lower().split() for sentence in sentences]

    # utilisation de spacy pour ne retenir que les allowed_postags
    texts_out = []
    for sent in sentences:
        if len(sent) < (nlp.max_length): # si le texte est trop grand
            doc = nlp(" ".join(sent))
            texts_out.append(" ".join([token.lemma_ for token in doc if token.
 ↪pos_ in allowed_postags]))
        else:
```

```
            texts_out.append(sent)
    sentences=texts_out

    # suppression des stopwords
    words =[[word for word in simple_preprocess(str(doc)) if word not in␣
↪stop_words] for doc in sentences]


    # recherche des bigrammes
    bigram = Phrases(words, min_count, threshold,delimiter=' ')
    bigram_phraser = Phraser(bigram)


    # sauvergarde des tokens et des bigrammes
    bigram_token = []
    for sent in words:
        bigram_token.append(bigram_phraser[sent])


    # creation du vocabulaire
    dictionary = gensim.corpora.Dictionary(bigram_token)


    # il est possible de filtrer des mots en fonction de leur occurrence␣
↪d'apparitions
    dictionary.filter_extremes(no_below, no_above)
    # et de compacter le dictionnaire
    dictionary.compactify()
    corpus = [dictionary.doc2bow(text) for text in bigram_token]

    # recuperaction du tfidf plutôt que uniquement le bag of words
    tfidf = models.TfidfModel(corpus)
    corpus_tfidf = tfidf[corpus]

    return corpus, corpus_tfidf, dictionary, bigram_token
```

[ ]:
```
df_source = pd.read_csv('HAI817_Projet_Traitement_TTTTT.csv', sep=',')

df_fake = df_source[df_source["rating"] == "false"]
df_true = df_source[df_source["rating"] == "true"]
df_mixture = df_source[df_source["rating"] == "mixture"]
df_other = df_source[df_source["rating"] == "other"]


df_source.reset_index(drop = True, inplace = True)
print(df_source.shape)
```

```
(2462, 2)
```

[ ]:
```python
import gensim
from gensim import models

def get_best_coherence_values(corpus, dictionary, listtokens, start=5, stop=15,␣
 ↪step=2):
    coherence_values = []
    model_list = []
    for num_topics in range(start, stop, step):
        lda_model = gensim.models.LdaMulticore(corpus=corpus,
                                               id2word=dictionary,
                                               num_topics=num_topics,
                                               random_state=100,
                                               chunksize=100,
                                               passes=10,
                                               per_word_topics=True)

        coherence_model_lda = CoherenceModel(model=lda_model, texts=listtokens,␣
 ↪dictionary=dictionary, coherence='c_v')
        model_list.append(lda_model)
        coherence_values.append(coherence_model_lda.get_coherence())
    return model_list, coherence_values
```

[ ]:
```python
# Calculer le nombre d'occurrences de chaque valeur dans la colonne 'our rating'
value_counts = df_source['rating'].value_counts()

true_count = value_counts.get("true", 0)
false_count = value_counts.get("false", 0)
mixture_count = value_counts.get("mixture", 0)
other_count = value_counts.get("other", 0)
# Afficher le nombre d'occurrences pour chaque valeur
print("Occurrences de 'true':", true_count)
print("Occurrences de 'false':", false_count)
print("Occurrences de 'mixture':", mixture_count)
print("Occurrences de 'other':", other_count)
```

```
Occurrences de 'true': 664
Occurrences de 'false': 1084
Occurrences de 'mixture': 511
Occurrences de 'other': 203
```

Calcul de la cohérence des différentes classes de données : “true”, “false”, “mixture”, “other”

[ ]:
```
df_fake = df_source[df_source["rating"] == "false"]
df_true = df_source[df_source["rating"] == "true"]
df_mixture = df_source[df_source["rating"] == "mixture"]
df_other = df_source[df_source["rating"] == "other"]

corpus_fake, corpus_tfidf_fake, dictionary_fake, token_fake =␣
 ↪MyCleanTextsforLDA(df_fake.text,stop_words=stop_words)
corpus_true, corpus_tfidf_true, dictionary_true, token_true =␣
 ↪MyCleanTextsforLDA(df_true.text,stop_words=stop_words)
corpus_mixture, corpus_tfidf_mixture, dictionary_mixture, token_mixture =␣
 ↪MyCleanTextsforLDA(df_mixture.text,stop_words=stop_words)
corpus_other, corpus_tfidf_other, dictionary_other, token_other =␣
 ↪MyCleanTextsforLDA(df_other.text,stop_words=stop_words)
```

[ ]:
```
#Coherence values pour donnnées FAKE
start=6
stop=15
step=1

model_list_fake, coherence_values_fake = get_best_coherence_values(
    dictionary=dictionary_fake,
    corpus=corpus_fake,
    listtokens=token_fake,
    start=start,
    stop=stop,
    step=step)
```

[ ]:
```
#Coherence values pour donnnées TRUE
start=6
stop=15
step=1
model_list_true, coherence_values_true = get_best_coherence_values(
    dictionary=dictionary_true,
    corpus=corpus_true,
    listtokens=token_true,
    start=start,
    stop=stop,
    step=step)
```

[ ]:
```
#Coherence values pour donnnées MIXTURE
start=6
stop=15
step=1
model_list_mixture, coherence_values_mixture = get_best_coherence_values(
    dictionary=dictionary_mixture,
    corpus=corpus_mixture,
    listtokens=token_mixture,
```

```
    start=start,
    stop=stop,
    step=step)
```

[ ]:
```
#Coherence values pour donnnées OTHER
start=6
stop=15
step=1
model_list_other, coherence_values_other = get_best_coherence_values(
    dictionary=dictionary_other,
    corpus=corpus_other,
    listtokens=token_other,
    start=start,
    stop=stop,
    step=step)
```

[ ]:
```
import pyLDAvis.gensim_models as gensimvis

print ("Calcul de la cohérence et de la perplexité du modèle pour le corpus : ")

print('\n')
print('Cohérence pour news "vrai" : ', str(coherence_values_fake))
print('Cohérence pour news "fausse" : ', str(coherence_values_true))
print('Cohérence pour news "mixture" : ', str(coherence_values_true))
print('Cohérence pour news "other" : ', str(coherence_values_true))
print('\n')
```

```
Calcul de la cohérence et de la perplexité du modèle pour le corpus :


Cohérence pour news "vrai" :  [0.30986333436585367, 0.3581869120790963,
0.3839641501180163, 0.42197847901575436, 0.4749022873610732,
0.44102120928394994, 0.48063727685265817, 0.48287280265118154,
0.5182303401898596]
Cohérence pour news "fausse" :  [0.3476063642765608, 0.3496531904812456,
0.38784031721044177, 0.4051681033050882, 0.4115627972345087,
0.45716132740405124, 0.45144639438866085, 0.43547910185902083, 0.50386950041314]
Cohérence pour news "mixture" :  [0.3476063642765608, 0.3496531904812456,
0.38784031721044177, 0.4051681033050882, 0.4115627972345087,
0.45716132740405124, 0.45144639438866085, 0.43547910185902083, 0.50386950041314]
Cohérence pour news "other" :  [0.3476063642765608, 0.3496531904812456,
0.38784031721044177, 0.4051681033050882, 0.4115627972345087,
0.45716132740405124, 0.45144639438866085, 0.43547910185902083, 0.50386950041314]
```

Recherche du meilleur modèle pour chaque classe

[ ]:
```
#Meilleur modèle de la classe FAKE
best = max(coherence_values_fake)
print("Best coherence: "+str(best))
for i in range(len(coherence_values_fake)):
  if(coherence_values_fake[i] == best):
    model_fake = model_list_fake[i]
visu_data_fake = gensimvis.prepare(model_fake, corpus_fake, dictionary_fake)
pyLDAvis.display(visu_data_fake)
```

```
Best coherence: 0.5182303401898596
```

[ ]:
```
<IPython.core.display.HTML object>
```

[ ]:
```
#Meilleur modèle de la classe TRUE
best = max(coherence_values_true)
print("Best coherence: "+str(best))
for i in range(len(coherence_values_true)):
  if(coherence_values_true[i] == best):
    model_true = model_list_true[i]
visu_data_true = gensimvis.prepare(model_true, corpus_true, dictionary_true)
pyLDAvis.display(visu_data_true)
```

```
Best coherence: 0.50386950041314
```

[ ]:
```
<IPython.core.display.HTML object>
```

[ ]:
```
#Meilleur modèle de la classe MIXTURE
best = max(coherence_values_mixture)
print("Best coherence: "+str(best))
for i in range(len(coherence_values_mixture)):
  if(coherence_values_mixture[i] == best):
    model_mixture = model_list_mixture[i]
visu_data_mixture = gensimvis.prepare(model_mixture, corpus_mixture,␣
 ↪dictionary_mixture)
pyLDAvis.display(visu_data_mixture)
```

```
Best coherence: 0.4628398951477889
```

[ ]:
```
<IPython.core.display.HTML object>
```

[ ]:
```
#Meilleur modèle de la classe OTHER
best = max(coherence_values_other)
print("Best coherence: "+str(best))
```

```
for i in range(len(coherence_values_other)):
  if(coherence_values_other[i] == best):
    model_other = model_list_other[i]
visu_data_other = gensimvis.prepare(model_other, corpus_other, dictionary_other)
pyLDAvis.display(visu_data_other)
```

```
Best coherence: 0.5050079616386773
```

[ ]:
```
<IPython.core.display.HTML object>
```

#Equilibrage des jeux de données (Sans Topic Moding)

###Vrai & Faux

Réduction du dataframe fake tant qu'il est supérieur au dataframe vrai

[ ]:
```
def equilibrer_vraifaux(df_source):
  df_fake = df_source[df_source["rating"] == "false"]
  df_true = df_source[df_source["rating"] == "true"]

  df_fake_vf = df_fake.copy()
  df_true_vf = df_true.copy()

  while (df_fake_vf.size > df_true_vf.size):
    random_index = np.random.choice(df_fake_vf.index)
    # Supprimer la ligne correspondant à l'indice aléatoire
    df_fake_vf = df_fake_vf.drop(random_index)

  while (df_fake_vf.size < df_true_vf.size):
    random_index = np.random.choice(df_true_vf.index)
    # Supprimer la ligne correspondant à l'indice aléatoire
    df_true_vf = df_true_vf.drop(random_index)

  #print(df_fake_vf.size)
  #print(df_true_vf.size)

  return pd.concat([df_fake_vf,df_true_vf], ignore_index=True)
```

###(Vrai, Faux) & Other

Réduction du dataframe (fake,vrai) tant qu'il est supérieur au dataframe other

[ ]:
```
def equilibrer_other(df_source):
  df_fake = df_source[df_source["rating"] == "false"]
  df_true = df_source[df_source["rating"] == "true"]
  df_other = df_source[df_source["rating"] == "other"]

  df_fake_copy = df_fake.copy()
  df_true_copy = df_true.copy()
```

```
  df_other_copy = df_other.copy()

  while (df_fake_copy.size > df_other_copy.size/2):
    random_index = np.random.choice(df_fake_copy.index)
    # Supprimer la ligne correspondant à l'indice aléatoire
    df_fake_copy = df_fake_copy.drop(random_index)

  while (df_true_copy.size > df_other_copy.size/2):
    random_index = np.random.choice(df_true_copy.index)
    # Supprimer la ligne correspondant à l'indice aléatoire
    df_true_copy = df_true_copy.drop(random_index)

  df_faketrue = pd.concat([df_true_copy, df_fake_copy], ignore_index=True)
  df_faketrue["rating"] = [1]*len(df_faketrue)

  while (df_faketrue.size > df_other_copy.size):
    random_index = np.random.choice(df_faketrue.index)
    # Supprimer la ligne correspondant à l'indice aléatoire
    df_faketrue = df_faketrue.drop(random_index)

  while (df_faketrue.size < df_other_copy.size):
    random_index = np.random.choice(df_other_copy.index)
    # Supprimer la ligne correspondant à l'indice aléatoire
    df_other_copy = df_other_copy.drop(random_index)

  df_other_copy["rating"] = [0]*len(df_other_copy)

  #print(df_faketrue.size)
  #print(df_other_copy.size)

  return pd.concat([df_faketrue,df_other_copy], ignore_index=True)
```

###Vrai & Faux & Mixture & Other Réduction des dataframe (fake,vrai,mixture) tant qu'il est supérieur au dataframe other

[ ]:
```
def equilibrer_quatreclasses(df_source):
  df_fake = df_source[df_source["rating"] == "false"]
  df_true = df_source[df_source["rating"] == "true"]
  df_other = df_source[df_source["rating"] == "other"]
  df_mixture = df_source[df_source["rating"] == "mixture"]

  df_fake_copy = df_fake.copy()
  df_true_copy = df_true.copy()
  df_other_copy = df_other.copy()
  df_mixture_copy = df_mixture.copy()

  while (df_fake_copy.size > df_other_copy.size):
```

```
    random_index = np.random.choice(df_fake_copy.index)
    # Supprimer la ligne correspondant à l'indice aléatoire
    df_fake_copy = df_fake_copy.drop(random_index)
  df_fake_copy["rating"] = [0]*len(df_fake_copy)

  while (df_true_copy.size > df_other_copy.size):
    random_index = np.random.choice(df_true_copy.index)
    # Supprimer la ligne correspondant à l'indice aléatoire
    df_true_copy = df_true_copy.drop(random_index)
  df_true_copy["rating"] = [1]*len(df_true_copy)

  while (df_mixture_copy.size > df_other_copy.size):
    random_index = np.random.choice(df_mixture_copy.index)
    # Supprimer la ligne correspondant à l'indice aléatoire
    df_mixture_copy = df_mixture_copy.drop(random_index)
  df_mixture_copy["rating"] = [2]*len(df_mixture_copy)

  df_other_copy["rating"] = [3]*len(df_other_copy)

  #print(df_fake_copy.size)
  #print(df_true_copy.size)
  #print(df_other_copy.size)
  #print(df_mixture_copy.size)

  return pd.concat([df_fake_copy,df_true_copy,df_other_copy,df_mixture_copy],␣
 ↪ignore_index=True)
```

6 Vectorisation et Modèles

##Initialisation (à faire tourner absolument)

[ ]:
```
def enregistrerModele_Default(pretraitement, classifieur, n_split, accuracy,␣
 ↪precision, recall, fmesure, matrice, cheminFichier):

  try :
    df_results = pd.read_csv(cheminFichier, sep=',')

    results_dict = {'pretraitement': df_results.get('pretraitement').tolist(),
                    'classifieur': df_results.get('classifieur').tolist(),
                    'n_split': df_results.get('n_split').tolist(),
                    'accuracy': df_results.get('accuracy').tolist(),
                    'precision': df_results.get('precision').tolist(),
                    'recall': df_results.get('recall').tolist(),
                    'fmesure': df_results.get('fmesure').tolist(),
                    'matrice': df_results.get('matrice').tolist()
                    }
```

```
    results_dict['pretraitement'].append(pretraitement)
    results_dict['classifieur'].append(classifieur)
    results_dict['n_split'].append(n_split)
    results_dict['accuracy'].append(accuracy)
    results_dict['precision'].append(precision)
    results_dict['recall'].append(recall)
    results_dict['fmesure'].append(fmesure)
    results_dict['matrice'].append(matrice)

  except FileNotFoundError:
    results_dict = {'pretraitement': [pretraitement],
                    'classifieur': [classifieur],
                    'n_split': [n_split],
                    'accuracy': [accuracy],
                    'precision': [precision],
                    'recall': [recall],
                    'fmesure': [fmesure],
                    'matrice': [matrice],
                    }

  # Créer un DataFrame à partir du dictionnaire
  df_results = pd.DataFrame(results_dict)

  # Enregistrement du DataFrame dans un fichier CSV
  pretraitementTab = list(pretraitement)
  avance = 0
  for index in range(len(pretraitementTab)):
    if(pretraitementTab[index]=='T'):
      avance += pow(2,4-index)
  avance /= 0.32

  print(pretraitement, classifieur,"enregistrés. (",avance,"% )")
  df_results.to_csv(cheminFichier, index=False)
```

[ ]:
```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.model_selection import train_test_split, cross_val_score, KFold
from sklearn.preprocessing import LabelEncoder
vectorizer = TfidfVectorizer()
label_encoder = LabelEncoder()

# Vectorisation des données textuelles
def Vectorisation(df_source):
    X = vectorizer.fit_transform(df_source['text'].values.astype('U'))
    y = label_encoder.fit_transform(df_source['rating'].values.astype('U'))
    return X,y
```

##Fonction Classifieur

[ ]:
```
def Classifieur_Default(pretraitement, classification, name, X, y):
    nb_split = 5
    seed = 30

    if(name == "KNN"):
      model = KNeighborsClassifier()
    elif(name == "CART"):
      model = DecisionTreeClassifier()
    elif(name == "SVM"):
      model = SVC()
    elif(name == "RFO"):
      model = RandomForestClassifier()
    elif(name == "MultinomialNB"):
      model = MultinomialNB()
    elif(name == "Xgboost"):
      model = XGBClassifier()
    else:
      raise Exception("Classifieur indéterminé")

    # Validation croisée
    kfold = KFold(n_splits=nb_split, shuffle=True, random_state=seed)

    # Initialiser des listes pour stocker les résultats
    accuracies = []
    precisions = []
    recalls = []
    f1_scores = []
    confusion_matrices = []

    for train_index, test_index in kfold.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        # Entraînement du meilleur modèle sur les données d'entraînement
        model.fit(X_train, y_train)

        # Prédiction des étiquettes sur les données de test
        y_pred = model.predict(X_test)

        # Calculs des différentes mesures
        accuracies.append(accuracy_score(y_test, y_pred))
        precisions.append(precision_score(y_test, y_pred, average='weighted'))
        recalls.append(recall_score(y_test, y_pred, average='weighted'))
        f1_scores.append(f1_score(y_test, y_pred, average='weighted'))
        confusion_matrices.append(confusion_matrix(y_test, y_pred))
```

```
    # Moyenne des mesures
    accuracy = np.mean(accuracies)
    precision = np.mean(precisions)
    recall = np.mean(recalls)
    f1 = np.mean(f1_scores)
    confusion_matrix_result = np.mean(confusion_matrices, axis=0)

    enregistrerModele_Default(pretraitement, name, nb_split, accuracy,␣
↪precision, recall, f1, confusion_matrix_result,␣
↪'HAI817_Default_'+classification+'.csv')
```

##Baseline avec les classes Vrai & Faux **ATTENTION : Peut être long à faire tourner !**

[ ]:
```
def test_base_Vrai_Faux(classifieur):
  num = 0
  tabs = list(itertools.product([False, True], repeat=5))

  # Parcourir chaque combinaison
  for tab in tabs:
    num += 1
    name = ''
    for i in range(len(tab)):
      if(tab[i]): name += 'T'
      else: name += 'F'
    df_preTraite = pd.read_csv('HAI817_Projet_Traitement_'+name+'.csv', sep=',')
    df_preTraite = equilibrer_vraifaux(df_preTraite)
    X,y = Vectorisation(df_preTraite)
    Classifieur_Default(name, 'Vrai_Faux', classifieur, X, y)
```

[ ]:
```
#KNeighborsClassifier
test_base_Vrai_Faux("KNN")
#DecisionTreeClassifier
test_base_Vrai_Faux("CART")
#SVM
test_base_Vrai_Faux("SVM")
#RandomForestClassifier
test_base_Vrai_Faux("RFO")
#MultinomialNB
test_base_Vrai_Faux("MultinomialNB")
#Xgboost
test_base_Vrai_Faux("Xgboost")
```

##Baseline avec les classes VraiFaux & Other **ATTENTION : Peut être long à faire tourner !**

[ ]:
```
def test_base_VraiFaux_Other(classifieur):
  num = 0
  tabs = list(itertools.product([False, True], repeat=5))

  # Parcourir chaque combinaison
  for tab in tabs:
    num += 1
    name = ''
    for i in range(len(tab)):
      if(tab[i]): name += 'T'
      else: name += 'F'
    df_preTraite = pd.read_csv('HAI817_Projet_Traitement_'+name+'.csv', sep=',')
    df_preTraite = equilibrer_other(df_preTraite)
    X,y = Vectorisation(df_preTraite)
    Classifieur_Default(name, 'VraiFaux_Other', classifieur, X, y)
```

[ ]:
```
#KNeighborsClassifier
test_base_VraiFaux_Other("KNN")
#DecisionTreeClassifier
test_base_VraiFaux_Other("CART")
#SVM
test_base_VraiFaux_Other("SVM")
#RandomForestClassifier
test_base_VraiFaux_Other("RFO")
#MultinomialNB
test_base_VraiFaux_Other("MultinomialNB")
#Xgboost
test_base_VraiFaux_Other("Xgboost")
```

##Baseline avec les classes Vrai & Faux & Other & Mixture **ATTENTION : Peut être long à faire tourner !**

[ ]:
```
def test_base_Vrai_Faux_Other_Mixture(classifieur):
  num = 0
  tabs = list(itertools.product([False, True], repeat=5))

  # Parcourir chaque combinaison
  for tab in tabs:
    num += 1
    name = ''
    for i in range(len(tab)):
      if(tab[i]): name += 'T'
      else: name += 'F'
    df_preTraite = pd.read_csv('HAI817_Projet_Traitement_'+name+'.csv', sep=',')
    df_preTraite = equilibrer_quatreclasses(df_preTraite)
    X,y = Vectorisation(df_preTraite)
    Classifieur_Default(name, 'Vrai_Faux_Other_Mixture', classifieur, X, y)
```

[ ]:
```
#KNeighborsClassifier
test_base_Vrai_Faux_Other_Mixture("KNN")
#DecisionTreeClassifier
test_base_Vrai_Faux_Other_Mixture("CART")
#SVM
test_base_Vrai_Faux_Other_Mixture("SVM")
#RandomForestClassifier
test_base_Vrai_Faux_Other_Mixture("RFO")
#MultinomialNB
test_base_Vrai_Faux_Other_Mixture("MultinomialNB")
#Xgboost
test_base_Vrai_Faux_Other_Mixture("Xgboost")
```

#Trouver les meilleurs paramètres

Nous allons maintenant essayer de trouver les meilleurs paramètres afin d'obtenir le meilleur modèle possible. Pour cela, nous allons identifier pour chaque classification, quelle sont les meilleurs classifieurs. Nous chercherons alors les meilleurs paramètres sur ces classifieurs.

##Initialisation

[ ]:
```
def enregistrerModele_Params(pretraitement, classifieur, best_params,␣
↪kfold_Tab, size, testsize, accuracy, precision, recall, fmesure, matrice,␣
↪cheminFichier):

 try :
   df_results = pd.read_csv(cheminFichier, sep=',')

   results_dict = {'pretraitement': df_results.get('pretraitement').tolist(),
                   'classifieur': df_results.get('classifieur').tolist(),
                   'best_params': df_results.get('best_params').tolist(),
                   'kfold': df_results.get('kfold').tolist(),
                   'size': df_results.get('size').tolist(),
                   'testsize': df_results.get('testsize').tolist(),
                   'accuracy': df_results.get('accuracy').tolist(),
                   'precision': df_results.get('precision').tolist(),
                   'recall': df_results.get('recall').tolist(),
                   'fmesure': df_results.get('fmesure').tolist(),
                   'matrice': df_results.get('matrice').tolist()
                   }

   results_dict['pretraitement'].append(pretraitement)
   results_dict['classifieur'].append(classifieur)
   results_dict['best_params'].append(best_params)
   results_dict['kfold'].append(kfold_Tab)
   results_dict['size'].append(size)
   results_dict['testsize'].append(testsize)
   results_dict['accuracy'].append(accuracy)
```

```
    results_dict['precision'].append(precision)
    results_dict['recall'].append(recall)
    results_dict['fmesure'].append(fmesure)
    results_dict['matrice'].append(matrice)

  except FileNotFoundError:
    results_dict = {'pretraitement': [pretraitement],
                    'classifieur': [classifieur],
                    'best_params': [best_params],
                    'kfold': [kfold_Tab],
                    'size': [size],
                    'testsize': [testsize],
                    'accuracy': [accuracy],
                    'precision': [precision],
                    'recall': [recall],
                    'fmesure': [fmesure],
                    'matrice': [matrice]
                    }

  # Créer un DataFrame à partir du dictionnaire
  df_results = pd.DataFrame(results_dict)

  # Enregistrement du DataFrame dans un fichier CSV
  pretraitementTab = list(pretraitement)
  avance = 1
  for index in range(len(pretraitementTab)):
    if(pretraitementTab[index]=='T'):
      avance += pow(2,4-index)
  avance /= 0.32

  print(pretraitement, classifieur,"enregistrés. (",avance,"% )")
  df_results.to_csv(cheminFichier, index=False)
```

[ ]:
```
from sklearn.model_selection import RandomizedSearchCV, KFold, cross_val_score,␣
 ↪cross_validate
from sklearn.metrics import make_scorer, accuracy_score, precision_score,␣
 ↪recall_score, f1_score, confusion_matrix

def Classifieur_Params(pretraitement, classification, name, X, y):
    nb_split = 5
    iterations = 20
    seed = 30
    testsize = 0.2
    size = 1-testsize

    if name == "SVM":
        # Grille de paramètres pour SVM
```

```
        param_grid = {'C': [0.01, 0.1, 1, 10, 100],
                      'gamma': ['scale', 'auto', 0.001, 0.01, 0.1, 1, 10],
                      'kernel': ['rbf', 'linear', 'poly', 'sigmoid'],
                      'degree': [2, 3, 4, 5],
                      'coef0': [0.0, 0.1, 0.5, 1.0]}
        random_search = RandomizedSearchCV(SVC(),␣
↪param_distributions=param_grid, n_iter=iterations, random_state=seed,␣
↪scoring='accuracy', cv=nb_split)

    elif name == "Xgboost":
        # Grille de paramètres pour XGBoost
        param_grid = {'max_depth': [3, 4, 5, 6, 7],
                      'learning_rate': [0.01, 0.05, 0.1, 0.2, 0.3],
                      'n_estimators': [100, 200, 300, 400, 500],
                      'subsample': [0.6, 0.7, 0.8, 0.9, 1.0],
                      'colsample_bytree': [0.6, 0.7, 0.8, 0.9, 1.0],
                      'gamma': [0, 0.1, 0.2, 0.3, 0.4]}
        random_search = RandomizedSearchCV(XGBClassifier(),␣
↪param_distributions=param_grid, n_iter=iterations, random_state=seed,␣
↪scoring='accuracy', cv=nb_split)

    elif name == "MultinomialNB":
        # Grille de paramètres pour MultinomialNB
        param_grid = {'alpha': [0.0, 0.01, 0.1, 0.5, 1.0, 2.0, 5.0, 10.0],
                      'fit_prior': [True, False]}

        random_search = RandomizedSearchCV(MultinomialNB(),␣
↪param_distributions=param_grid, n_iter=iterations, random_state=seed,␣
↪scoring='accuracy', cv=nb_split)


    elif name == "RFO":
        # Grille de paramètres pour RFO
        param_grid = {'n_estimators': [50, 100, 200, 300, 400, 500],
                      'max_features': ['autp', 'sqrt', 'log2'],
                      'max_depth': [None, 10, 20, 30, 40, 50],
                      'min_samples_split': [2, 5, 10],
                      'min_samples_leaf': [1, 2, 4],
                      'bootstrap': [True, False]}

        random_search = RandomizedSearchCV(RandomForestClassifier(),␣
↪param_distributions=param_grid, n_iter=iterations, random_state=seed,␣
↪scoring='accuracy', cv=nb_split)

    elif name == "KNN":
        # Grille de paramètres pour KNN
        param_grid = {
```

```
            'n_neighbors': [1, 3, 5, 7, 9, 11, 13, 15],
            'weights': ['uniform', 'distance'],
            'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
            'leaf_size': [10, 20, 30, 40, 50],
            'p': [1, 2]}

        random_search = RandomizedSearchCV(KNeighborsClassifier(),␣
↪param_distributions=param_grid, n_iter=iterations, random_state=seed,␣
↪scoring='accuracy', cv=nb_split)

    elif name == "CART":
        # Grille de paramètres pour CART
        param_grid = {
            'criterion': ['gini', 'entropy'],
            'splitter': ['best', 'random'],
            'max_depth': [None, 10, 20, 30, 40, 50],
            'min_samples_split': [2, 5, 10, 20],
            'min_samples_leaf': [1, 2, 4, 10],
            'max_features': [None, 'sqrt', 'log2'],
            'class_weight': [None, 'balanced']}

        random_search = RandomizedSearchCV(DecisionTreeClassifier(),␣
↪param_distributions=param_grid, n_iter=iterations, random_state=seed,␣
↪scoring='accuracy', cv=nb_split)

    else:
        raise Exception("Classifieur indéterminé")

    # Entraînement du modèle sur les données
    random_search.fit(X, y)
    #pd.DataFrame(random_search.cv_results_).to_csv('export.csv')
    best_model = random_search.best_estimator_
    best_model.set_params(**(random_search.best_params_))

    # Validation croisée
    kfold = KFold(n_splits=nb_split, shuffle=True, random_state=seed)

    # Initialiser des listes pour stocker les résultats
    accuracies = []
    precisions = []
    recalls = []
    f1_scores = []
    confusion_matrices = []

    for train_index, test_index in kfold.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]
```

```
        # Entraînement du meilleur modèle sur les données d'entraînement
        best_model.fit(X_train, y_train)

        # Prédiction des étiquettes sur les données de test
        y_pred = best_model.predict(X_test)

        # Calculs des différentes mesures
        accuracies.append(accuracy_score(y_test, y_pred))
        precisions.append(precision_score(y_test, y_pred, average='weighted'))
        recalls.append(recall_score(y_test, y_pred, average='weighted'))
        f1_scores.append(f1_score(y_test, y_pred, average='weighted'))
        confusion_matrices.append(confusion_matrix(y_test, y_pred))

    # Moyenne des mesures
    accuracy = np.mean(accuracies)
    precision = np.mean(precisions)
    recall = np.mean(recalls)
    f1 = np.mean(f1_scores)
    confusion_matrix_result = np.mean(confusion_matrices, axis=0)

    enregistrerModele_Params(pretraitement, name, random_search.best_params_,␣
↪[iterations, seed, nb_split], size, testsize, accuracy, precision, recall,␣
↪f1, confusion_matrix_result, 'HAI817_BestParams_'+classification+'.csv')
```

##Classification VRAI/FAUX

[ ]:
```
def test_params_Vrai_Faux(classifieur):
  num = 0
  tabs = list(itertools.product([False, True], repeat=5))

  # Parcourir chaque combinaison
  for tab in tabs:
    num += 1
    if(num > 0):
      name = ''
      for i in range(len(tab)):
        if(tab[i]): name += 'T'
        else: name += 'F'
      df_preTraite = pd.read_csv('HAI817_Projet_Traitement_'+name+'.csv',␣
↪sep=',')
      df_preTraite = equilibrer_vraifaux(df_preTraite)
      X,y = Vectorisation(df_preTraite)
      Classifieur_Params(name, 'Vrai_Faux', classifieur, X, y)
```

[ ]:
```
#SVM
test_params_Vrai_Faux("SVM")
```

[ ]:
```
#RandomForestClassifier
test_params_Vrai_Faux("RFO")
```

[ ]:
```
#Xgboost
test_params_Vrai_Faux("Xgboost")
```

##Classification VRAIFAUX/OTHER

Classifieur Accuracy Precision Recall FMesure * KNN 0.52712 0.53565 0.52712 0.49382 * CART 0.55068 0.56295 0.55068 0.54733 * RFO 0.61520 0.62747 0.61520 0.61414 * SVM 0.62175 0.65367 0.62175 0.60992 * MultiNB 0.59136 0.65195 0.59136 0.56986 * Xgboost 0.59797 0.60157 0.59797 0.59878

Il semblerait donc qu'il soit intéréssant de chercher les meilleurs paramètres de la classification VRAI/FAUX sur SVM, RFO et MultinomialNB.

[ ]:
```
warnings.filterwarnings('always')  # "error", "ignore", "always", "default",␣
 ↪"module" or "once"
warnings.filterwarnings("ignore", category=FutureWarning)
warnings.filterwarnings("ignore", category=DeprecationWarning)
def test_params_VraiFaux_Other(classifieur):
  num = 0
  tabs = list(itertools.product([False, True], repeat=5))

  # Parcourir chaque combinaison
  for tab in tabs:
    num += 1
    name = ''
    for i in range(len(tab)):
      if(tab[i]): name += 'T'
      else: name += 'F'
    df_preTraite = pd.read_csv('HAI817_Projet_Traitement_'+name+'.csv', sep=',')
    df_preTraite = equilibrer_other(df_preTraite)
    X,y = Vectorisation(df_preTraite)
    Classifieur_Params(name, 'VraiFaux_Other', classifieur, X, y)
```

[ ]:
```
#MultinomialNB
test_params_VraiFaux_Other("MultinomialNB")
```

[ ]:
```
#SVM
test_params_VraiFaux_Other("SVM")
```

[ ]:
```
#RandomForest
test_params_VraiFaux_Other("RFO")
```

##Classification VRAI/FAUX/OTHER/MIXTURE

```
[ ]: warnings.filterwarnings('always')  # "error", "ignore", "always", "default",␣
      ↪"module" or "once"
     warnings.filterwarnings("ignore", category=FutureWarning)
     warnings.filterwarnings("ignore", category=DeprecationWarning)
     def test_params_Vrai_Faux_Other_Mixture(classifieur):
       num = 0
       tabs = list(itertools.product([False, True], repeat=5))

       # Parcourir chaque combinaison
       for tab in tabs:
         num += 1
         if(num > 0):
           name = ''
           for i in range(len(tab)):
             if(tab[i]): name += 'T'
             else: name += 'F'
           df_preTraite = pd.read_csv('HAI817_Projet_Traitement_'+name+'.csv',␣
      ↪sep=',')
           df_preTraite = equilibrer_quatreclasses(df_preTraite)
           X,y = Vectorisation(df_preTraite)
           Classifieur_Params(name, 'Vrai_Faux_Other_Mixture', classifieur, X, y)
```

```
[ ]: #SVM
     test_params_Vrai_Faux_Other_Mixture("SVM")
```

```
[ ]: #RandomForestClassifier
     test_params_Vrai_Faux_Other_Mixture("RFO")
```

```
[ ]: #XGboost
     #test_params_Vrai_Faux_Other_Mixture("Xgboost")
```

6.1 #Pipeline

Création des pipelines pour les 3 classifieurs

6.2 ##DfNormalisers

```
[ ]: def preTraitement_pipeline(X,
                    removedigit=False, # supprimer les nombres
                    token_exclamation_interrogation=False, # tokeniser les "!" et "?"
                    removestopwords=False, # supprimer les stopwords
                    lemmatisation_racinisation=False, # lematisation des termes et␣
      ↪conserver la racine
                   ):
```

```
    texte = str(X)

    """
      PARTIE PRE-TOKENISATION
    """

    # suppression des caractères spéciaux
    texte = re.sub(r'[^\w\s]',' ', texte)
    # suppression de tous les caractères uniques
    texte = re.sub(r'\s+[a-zA-Z]\s+', ' ', texte)
    # substitution des espaces multiples par un seul espace
    texte = re.sub(r'\s+', ' ', texte, flags=re.I)

    # minuscule
    texte = texte.lower()

    # retirer les chiffres
    if removedigit:
      texte = re.sub(r'\d+', '', texte)

    """
      PARTIE POST-TOKENISATION
    """

    # découpage en tokens
    tokens = texte.split()

    # suppression ponctuation
    if(token_exclamation_interrogation):
      words = sep_ponctuation(sep_ponctuation(tokens,'?'),'!')
    else:
      table = str.maketrans('', '', string.punctuation)
      words = [token.translate(table) for token in tokens]

    # suppression des stopwords
    stop_words = set(stopwords.words('english'))
    stop_words.
↪update(['always','try','go','get','make','would','really','like','came','got','know','come'
    non_stop_words = ['d', 'o', 't']
    for word in non_stop_words:
        stop_words.discard(word)

    if removestopwords:
      words = [word for word in words if not word in stop_words]

    # lemmatisation & racinisation
    if lemmatisation_racinisation:
```

```
        lem = nltk.stem.wordnet.WordNetLemmatizer()
        words = [lem.lemmatize(word)for word in words]
        ps = nltk.stem.porter.PorterStemmer()
        words=[ps.stem(word) for word in words]

    sentence = ' '.join(words)

    return sentence
```

[ ]:
```
class DfNormalizer(BaseEstimator, TransformerMixin):
    def __init__(self,
                 removedigit=False, # supprimer les nombres
                 token_exclamation_interrogation=False, # tokeniser les "!" et "?
 ↪"
                 removestopwords=False, # supprimer les stopwords
                 lemmatisation_racinisation=False # lematisation des termes et␣
 ↪conserver la racine
                 ):

        self.removestopwords=removestopwords
        self.removedigit=removedigit
        self.token_exclamation_interrogation=token_exclamation_interrogation
        self.removestopwords=removestopwords
        self.lemmatisation_racinisation=lemmatisation_racinisation

    def transform(self, X, **transform_params):
        # Nettoyage du texte
        X=X.copy() # pour conserver le fichier d'origine
        return [preTraitement_pipeline(
                            text,
                            removedigit=self.removedigit,
                            token_exclamation_interrogation=self.
 ↪token_exclamation_interrogation,
                            removestopwords=self.removestopwords,
                            lemmatisation_racinisation=self.
 ↪lemmatisation_racinisation)
        for text in X]

    def fit(self, X, y=None, **fit_params):
        return self

    def fit_transform(self, X, y=None, **fit_params):
        return self.fit(X).transform(X)

    def get_params(self, deep=True):
        return {
            'removedigit':self.removedigit,
```

```
            'token_exclamation_interrogation':self.
↪token_exclamation_interrogation,
            'removestopwords':self.removestopwords,
            'lemmatisation_racinisation':self.lemmatisation_racinisation
        }

    def set_params (self, **parameters):
        for parameter, value in parameters.items():
            setattr(self,parameter,value)
        return self
```

6.3 ##Mise en place des Piplines

[ ]:
```
from sklearn.pipeline import Pipeline
```

[ ]:
```
#Import des fichiers de base
df1=pd.read_csv('HAI817_Projet_train.csv', sep=',')
df2=pd.read_csv('HAI817_Projet_test.csv', sep=',')
df_dutch=pd.read_csv('HAI817_Projet_german_translate.csv', sep=',')

df1['text'] = df1['title'] + ' ' + df1['text']
df2['text'] = df2['title'] + ' ' + df2['text']

df1 = df1.drop(columns=["public_id"])
df2 = df2.drop(columns=["ID"])

df1 = df1.drop(columns=["title"])
df2 = df2.drop(columns=["title"])
df_dutch = df_dutch.drop(columns=["our rating"])

df1 = df1.rename(columns={'our rating': 'rating'})
df2 = df2.rename(columns={'our rating': 'rating'})
```

[ ]:
```
def pipeline_function(
    df_source,
    removedigit,
    token_exclamation_interrogation,
    removestopwords,
    lemmatisation_racinisation
):

  X = df_source["text"].copy()
  y = df_source["rating"].copy()

  pipeline = Pipeline([('df_pretraite', DfNormalizer(removedigit=removedigit,
```

```
                                                                    ␣
↪token_exclamation_interrogation=token_exclamation_interrogation,
                                                                    ␣
↪removestopwords=removestopwords,
                                                                    ␣
↪lemmatisation_racinisation=lemmatisation_racinisation,
                                                                     )),
                    ('tfidf', TfidfVectorizer()),
                    ('clf', SVC())])

#grid_param = [{'clf': [SVC()],
#                'clf__C': [0.01, 0.1, 1.0, 10.0, 100.0],
#                'clf__gamma': ['scale', 'auto', 0.001, 0.01, 0.1, 1, 10],
#                'clf__degree': [2, 3, 4, 5],
#                'clf__coef0': [0.0, 0.1, 0.5, 1.0],
#                'clf__kernel': ['rbf', 'linear', 'poly', 'sigmoid']
#               }]

grid_param = [{'clf': [SVC()],
               'clf__C': [100],
               'clf__gamma': [0.01],
               'clf__kernel': ['rbf']
              }]

gd_sr = GridSearchCV(pipeline,
                     param_grid=grid_param,
                     scoring='accuracy',
                     cv=5,
                     n_jobs=1,
                     return_train_score=True)

# Entraînement du meilleur modèle sur les données d'entraînement
gd_sr.fit(X, y)

print('meilleurs paramètres', gd_sr.best_params_,'\n')
print('meilleur estimateur',gd_sr.best_estimator_,'\n')

return gd_sr
```

[ ]:
```
import pickle
df_avec_allemand = pd.concat([df1, df2, df_dutch], ignore_index=True)

df_vf = equilibrer_vraifaux(df_avec_allemand)
df_vfo = equilibrer_other(df_avec_allemand)
df_vfom = equilibrer_quatreclasses(df_avec_allemand)
```

[ ]:
```
filename = "Vrai_Faux_Modele.pkl"
pickle.dump(pipeline_function(df_vf,False,False,False,True), open(filename,␣
↪'wb'))
print("Sauvegarde du modèle")
```

```
meilleurs paramètres {'clf': SVC(C=100, gamma=0.01), 'clf__C': 100,
'clf__gamma': 0.01, 'clf__kernel': 'rbf'}

meilleur estimateur Pipeline(steps=[('df_pretraite',
DfNormalizer(lemmatisation_racinisation=True)),
                ('tfidf', TfidfVectorizer()), ('clf', SVC(C=100, gamma=0.01))])

Sauvegarde du modèle
```

[ ]:
```
filename = "VraiFaux_Other_Modele.pkl"
pickle.dump(pipeline_function(df_vfo,False,True,True,False), open(filename,␣
↪'wb'))
print("Sauvegarde du modèle")
```

```
meilleurs paramètres {'clf': SVC(C=10, gamma=1), 'clf__C': 10, 'clf__gamma': 1,
'clf__kernel': 'rbf'}

meilleur estimateur Pipeline(steps=[('df_pretraite',
                 DfNormalizer(removestopwords=True,
                              token_exclamation_interrogation=True)),
                ('tfidf', TfidfVectorizer()), ('clf', SVC(C=10, gamma=1))])

Sauvegarde du modèle
```

[ ]:
```
filename = "Vrai_Faux_Other_Mixture_Modele.pkl"
pickle.dump(pipeline_function(df_vfom,False,True,False,False), open(filename,␣
↪'wb'))
print("Sauvegarde du modèle")
```

```
meilleurs paramètres {'clf': SVC(C=100, gamma=0.01), 'clf__C': 100,
'clf__gamma': 0.01, 'clf__kernel': 'rbf'}

meilleur estimateur Pipeline(steps=[('df_pretraite',
                 DfNormalizer(token_exclamation_interrogation=True)),
                ('tfidf', TfidfVectorizer()), ('clf', SVC(C=100, gamma=0.01))])

Sauvegarde du modèle
```

#Exportation en PDF

[14]:
```
!jupyter nbconvert --to pdf "ProjetML_G6_Gallerne_Navel_Gilles_Picole-Ollivier.
↪ipynb"
```