## **11. Objectif général**

## **Le code présenté constitue une chaîne de traitement (pipeline) de données textuelles. L’objectif est de nettoyer, segmenter, enrichir et analyser un corpus de textes issus d’un fichier CSV (« spooky.csv »). On y retrouve plusieurs étapes : nettoyage du texte (prétraitement), segmentation (tokenization), extraction d’entités nommées, étiquetage grammatical (POS tagging), réduction des formes (lemmatisation et stemming) ainsi que des analyses de fréquences et de sentiments.**

## 

## **2. Nettoyage du texte**

### **a. Chargement et aperçu du jeu de données**

## **python**

## **Copy**

## **import pandas as pd**

## **url = "https://github.com/GU4243-ADS/spring2018-project1-ginnyqg/raw/master/data/spooky.csv"**

## **df = pd.read\_csv(url)**

## **print(df.head(10))**

## 

## **But : Charger le jeu de données depuis un lien GitHub et afficher les 10 premières lignes pour avoir un aperçu des données brutes.**

## **Pourquoi ? Cela permet de vérifier que le fichier est correctement chargé et d’observer la structure initiale (colonnes, exemples de texte, etc.).**

### **b. Téléchargement des ressources NLTK et configuration**

## **python**

## **Copy**

## **import nltk**

## **nltk.download('wordnet')**

## **import re**

## **from nltk.corpus import wordnet**

## 

## **But : Télécharger la ressource « wordnet » de NLTK nécessaire pour la vérification des mots existants et la lemmatisation.**

## **Pourquoi ? WordNet est une base lexicale qui permet de vérifier si un mot existe et fournit des synonymes et d’autres informations lexicales utiles dans le traitement du langage.**

### **c. Gestion des caractères répétitifs**

#### **i. Définition de l’expression régulière et de la fonction de remplacement**

## **python**

## **Copy**

## **repeat\_regexp = re.compile(r'(\w\*)(\w)\2(\w\*)')**

## **repl = r'\1\2\3'**

## 

## **Explication : L’expression régulière cherche des motifs où un caractère (\2) est répété consécutivement (par exemple, « coooooool »). Le remplacement (repl) conserve seulement une occurrence du caractère répété.**

## **Pourquoi ? Les utilisateurs peuvent exagérer la répétition des lettres pour accentuer une émotion (exemple : « coooooool »). Cette étape vise à réduire ces répétitions afin d’uniformiser le texte pour une analyse ultérieure.**

#### **ii. Fonction récursive replace**

## **python**

## **Copy**

## **def replace(word):**

## **if wordnet.synsets(word):**

## **return word**

## **repl\_word = repeat\_regexp.sub(repl, word)**

## **if repl\_word != word:**

## **return replace(repl\_word)**

## **else:**

## **return repl\_word**

## 

## **Fonctionnement :**

## **On teste d’abord si le mot existe dans WordNet (donc s’il est reconnu).**

## **Sinon, on applique l’expression régulière pour réduire les répétitions.**

## **La fonction s’appelle récursivement si le mot modifié est différent, afin de continuer à simplifier jusqu’à ce qu’aucune modification supplémentaire ne soit possible.**

## **Pourquoi ? Pour s’assurer que les mots exagérément allongés soient ramenés à leur forme canonique, ce qui facilite les analyses ultérieures (recherche de synonymes, correspondances, etc.).**

#### **iii. Fonction pour gérer l’ensemble d’un texte**

## **python**

## **Copy**

## **def handle\_repetitive\_chars(text):**

## **words = text.split()**

## **new\_words = []**

## **for word in words:**

## **new\_words.append(replace(word))**

## **return ' '.join(new\_words)**

## 

## **But : Découper le texte en mots, appliquer la fonction replace à chacun d’eux, puis reconstituer le texte.**

## **Pourquoi ? Appliquer ce nettoyage au niveau du texte complet garantit que chaque mot est traité individuellement.**

#### **iv. Application sur le DataFrame**

## **python**

## **Copy**

## **df['text'] = df['text'].apply(handle\_repetitive\_chars)**

## **print(handle\_repetitive\_chars("cooooooool"))**

## **print(df.head(10))**

## 

## **But : Mettre à jour la colonne « text » du DataFrame avec le texte nettoyé et vérifier l’effet de la fonction en testant sur le mot « cooooooool ».**

## **Pourquoi ? Cette transformation prépare les données en éliminant les variations exagérées dues aux répétitions de caractères.**

### **d. Remplacement des homoglyphes**

#### **i. Dictionnaire des homoglyphes**

## **python**

## **Copy**

## **homoglyphs\_map = {**

## **'$': 's', '0': 'o', '1': 'l', '3': 'e', '@': 'a', '5': 's', '7': 't',**

## **'8': 'b', '|': 'i', '¢': 'c', '¡': 'i', 'µ': 'u', 'ß': 'b', '€': 'e',**

## **'!':'i','£':'l',**

## **'Α': 'A', 'Β': 'B', 'Ε': 'E', 'Ζ': 'Z', 'Η': 'H', 'Ι': 'I', 'Κ': 'K',**

## **'Μ': 'M', 'Ν': 'N', 'Ο': 'O', 'Ρ': 'P', 'Τ': 'T', 'Χ': 'X',**

## **'А': 'A', 'В': 'B', 'С': 'C', 'Е': 'E', 'Н': 'H', 'К': 'K', 'М': 'M',**

## **'О': 'O', 'Р': 'P', 'Т': 'T', 'Х': 'X'**

## **}**

## 

## **Explication : Ce dictionnaire mappe certains caractères spéciaux ou similaires (homoglyphes) à leur équivalent alphabétique. Par exemple, « $ » est remplacé par « s », et des lettres issues d’alphabets différents (grec, cyrillique) sont converties en lettres latines.**

## **Pourquoi ? Pour uniformiser le texte et éviter que des variantes graphiques ne perturbent l’analyse (recherche, tokenisation, etc.).**

#### **ii. Fonction de remplacement**

## **python**

## **Copy**

## **def replace\_homoglyphs(text):**

## **return ''.join(homoglyphs\_map.get(char, char) for char in text)**

## 

## **Fonctionnement : Pour chaque caractère du texte, on remplace celui-ci par sa valeur dans le dictionnaire si une correspondance existe, sinon on le conserve.**

## **Pourquoi ? Cette fonction permet de normaliser les caractères, ce qui est essentiel pour garantir la cohérence lors des analyses ultérieures.**

#### **iii. Application et test**

## **python**

## **Copy**

## **print(replace\_homoglyphs("$tupid"))**

## **df['text'] = df['text'].apply(replace\_homoglyphs)**

## 

## **But : Vérifier le bon fonctionnement de la fonction (le test devrait renvoyer « stupid ») puis l’appliquer à tous les textes du DataFrame.**

### **e. Nettoyage des entrées spéciales**

## **python**

## **Copy**

## **from bs4 import BeautifulSoup**

## 

## **def clean\_special\_entries(text):**

## **text = re.sub(r'http\S+|www\S+', ' <URL> ', text)**

## **text = re.sub(r'\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', ' <EMAIL> ', text)**

## **text = BeautifulSoup(text, "html.parser").get\_text()**

## **return text**

## 

## **df['text'] = df['text'].apply(clean\_special\_entries)**

## 

## **Fonctionnement :**

## **Remplace les URLs par la balise « <URL> ».**

## **Remplace les adresses e-mail par « <EMAIL> ».**

## **Utilise BeautifulSoup pour enlever les balises HTML, ne gardant que le texte.**

## **Pourquoi ? Ces transformations assurent que le texte ne contienne pas de données spécifiques comme des liens ou des emails qui pourraient fausser l’analyse sémantique ou les statistiques.**

### **f. Transformation en minuscules et suppression de la ponctuation**

## **python**

## **Copy**

## **df['text'] = df['text'].str.lower()**

## 

## **import string**

## **def remove\_punctuation(text):**

## **return text.translate(str.maketrans('', '', string.punctuation))**

## 

## **df['text'] = df['text'].apply(remove\_punctuation)**

## 

## **But :**

## **Passer tout le texte en minuscules pour éviter les différences dues à la casse.**

## **Supprimer la ponctuation qui peut ne pas être pertinente pour certaines analyses.**

## **Pourquoi ? Cela permet d’uniformiser le texte pour les étapes suivantes (tokenisation, analyse de fréquence, etc.).**

### **g. Suppression des stopwords**

## **python**

## **Copy**

## **from nltk.corpus import stopwords**

## **nltk.download('stopwords')**

## **stop\_words = set(stopwords.words('english'))**

## 

## **def remove\_stopwords(text):**

## **return " ".join([word for word in text.split() if word not in stop\_words])**

## 

## **df['text'] = df['text'].apply(remove\_stopwords)**

## 

## **But : Enlever les mots très fréquents (stopwords) comme « the, and, is », qui n’apportent généralement pas d’information significative.**

## **Pourquoi ? La suppression des stopwords améliore la qualité des analyses en se concentrant sur les mots porteurs de sens.**

### **h. Sauvegarde des données nettoyées**

## **python**

## **Copy**

## **df = df.head(100)**

## **df.to\_csv("cleaned\_spooky.csv", index=False)**

## **print("Preprocessing complete. Data saved as cleaned\_spooky.csv")**

## 

## **But : Limiter le DataFrame aux 100 premiers enregistrements (probablement pour des tests) et sauvegarder le résultat dans un nouveau fichier CSV.**

## **Pourquoi ? Cela permet de conserver une version prétraitée des données pour les étapes de segmentation et d’analyse ultérieures.**

## 

## **3. Segmentation (Tokenization)**

### **a. Préparation de l’environnement**

## **python**

## **Copy**

## **import spacy**

## **nltk.download('punkt')**

## **nltk.download('punkt\_tab')**

## **nlp = spacy.load("en\_core\_web\_sm")**

## **file = 'cleaned\_spooky.csv'**

## **df = pd.read\_csv(file)**

## 

## **But :**

## **Télécharger les ressources nécessaires pour la tokenisation avec NLTK.**

## **Charger le modèle SpaCy pour le traitement du langage.**

## **Charger les données préalablement nettoyées.**

## **Pourquoi ? Ces ressources permettent de segmenter le texte en phrases et en mots, étape cruciale pour les analyses ultérieures.**

### **b. Tokenisation avec NLTK**

## **python**

## **Copy**

## **df = df.dropna(subset=['text'])**

## **df['tokens'] = df['text'].apply(word\_tokenize)**

## **print("First 10 rows after tokenization:")**

## **print(df.head(10))**

## 

## **Fonctionnement : On supprime les lignes où la colonne « text » est vide, puis on applique la fonction word\_tokenize de NLTK à chaque texte.**

## **Pourquoi ? La tokenisation divise le texte en unités (tokens) qui sont ensuite utilisées pour l’analyse sémantique, la recherche de fréquences, etc.**

### **c. Tokenisation basée sur SpaCy et BERT**

#### **i. Tokenisation par règle avec SpaCy**

## **python**

## **Copy**

## **def rule\_based\_tokenization(text):**

## **doc = nlp(text)**

## **return [token.text for token in doc]**

## 

## **df['rule\_based\_tokens'] = df['text'].apply(rule\_based\_tokenization)**

## 

## **But : Utiliser SpaCy pour obtenir une tokenisation qui tient compte de la structure linguistique.**

## **Pourquoi ? SpaCy offre une tokenisation robuste et peut servir pour des analyses nécessitant la segmentation par règles linguistiques.**

#### **ii. Tokenisation subword avec BERT**

## **python**

## **Copy**

## **from transformers import BertTokenizer**

## **tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')**

## 

## **def subword\_tokenization(text):**

## **return tokenizer.tokenize(text)**

## 

## **df['subword\_tokens'] = df['text'].apply(subword\_tokenization)**

## 

## **Fonctionnement : Le tokenizer BERT segmente le texte en sous-mots (subwords) selon le modèle pré-entraîné « bert-base-uncased ».**

## **Pourquoi ? La tokenisation en sous-mots est essentielle pour les modèles de langage modernes (comme BERT) qui traitent les mots rares ou inconnus en les décomposant en unités plus petites.**

## 

## **4. Reconnaissance d'entités nommées (NER) et POS tagging**

### **a. Extraction d’entités nommées**

## **python**

## **Copy**

## **def extract\_entities(text):**

## **doc = nlp(text)**

## **return [(ent.text, ent.label\_) for ent in doc.ents]**

## 

## **df['entities'] = df['text'].apply(extract\_entities)**

## 

## **But : Utiliser SpaCy pour identifier et extraire les entités nommées (personnes, lieux, organisations, etc.) dans le texte.**

## **Pourquoi ? L’extraction d’entités est utile pour comprendre la structure du texte et pour effectuer des analyses spécifiques (ex. étudier la mention d’un auteur ou d’un lieu).**

### **b. Étiquetage grammatical (POS tagging)**

## **python**

## **Copy**

## **def pos\_tagging(text):**

## **doc = nlp(text)**

## **return [(token.text, token.pos\_) for token in doc]**

## 

## **df['pos\_tags'] = df['text'].apply(pos\_tagging)**

## **print(df.head(10))**

## 

## **Fonctionnement : Chaque mot du texte est associé à sa catégorie grammaticale (nom, verbe, adjectif, etc.) grâce à SpaCy.**

## **Pourquoi ? L’étiquetage grammatical permet d’approfondir l’analyse linguistique et peut aider dans des tâches comme la lemmatisation ou l’extraction de syntagmes.**

## 

## **5. Réduction des formes**

### **a. Lemmatisation et Stemming**

## **python**

## **Copy**

## **from nltk.stem import WordNetLemmatizer, PorterStemmer**

## **lemmatizer = WordNetLemmatizer()**

## **stemmer = PorterStemmer()**

## 

## **def lemmatize\_text(text):**

## **return ' '.join([lemmatizer.lemmatize(word) for word in text.split()])**

## 

## **def stem\_text(text):**

## **return ' '.join([stemmer.stem(word) for word in text.split()])**

## 

## **df['lemmatized\_text'] = df['text'].apply(lemmatize\_text)**

## **df['stemmed\_text'] = df['text'].apply(stem\_text)**

## 

## **Explication :**

## **Lemmatisation : Réduit chaque mot à sa forme canonique (lemme) en tenant compte du contexte.**

## **Stemming : Tronque les mots pour obtenir leur radical, souvent de manière plus brutale.**

## **Pourquoi ? Ces techniques réduisent la variance des formes de mots et permettent de regrouper des variantes (ex. « running », « ran », « runs » → « run ») pour des analyses de fréquence ou de similarité.**

## 

## **6. Analyse des fréquences**

### **a. Comptage d’occurrences de mots spécifiques**

## **python**

## **Copy**

## **df['contains\_great'] = df['text'].apply(lambda x: 'great' in x.lower())**

## **great\_counts = df.groupby('author')['contains\_great'].sum()**

## **print(great\_counts)**

## 

## **Fonctionnement : On ajoute une colonne booléenne indiquant si le mot « great » apparaît dans le texte, puis on regroupe par auteur pour compter les occurrences.**

## **Pourquoi ? Cela permet d’identifier quels auteurs utilisent plus fréquemment certains termes, fournissant une analyse stylistique ou thématique.**

### **b. Visualisation avec un graphique Waffle**

## **python**

## **Copy**

## **from pywaffle import Waffle**

## **import matplotlib.pyplot as plt**

## 

## **data = great\_counts.to\_dict()**

## **fig = plt.figure(**

## **FigureClass=Waffle,**

## **rows=5,**

## **columns=10,**

## **values=data,**

## **legend={'loc': 'upper left', 'bbox\_to\_anchor': (1, 1)},**

## **title={'label': 'Occurrences of "great" by Author', 'loc': 'left'}**

## **)**

## **plt.show()**

## 

## **But : Créer un graphique en forme de « waffle chart » qui représente visuellement la fréquence d’un mot par auteur.**

## **Pourquoi ? La visualisation aide à interpréter rapidement les différences entre les auteurs.**

### **c. Comptage et visualisation pour le mot « impossible »**

## **python**

## **Copy**

## **df['contains\_impossible'] = df['text'].apply(lambda x: 'impossible' in x.lower())**

## **impossible\_counts = df.groupby('author')['contains\_impossible'].sum()**

## **print(impossible\_counts)**

## **data = impossible\_counts.to\_dict()**

## 

## **if sum(data.values()) == 0:**

## **print("No data available to plot the waffle chart.")**

## **else:**

## **fig = plt.figure(**

## **FigureClass=Waffle,**

## **rows=5,**

## **columns=10,**

## **values=data,**

## **legend={'loc': 'upper left', 'bbox\_to\_anchor': (1, 1)},**

## **title={'label': 'Occurrences of "impossible" by Author', 'loc': 'left'}**

## **)**

## **plt.show()**

## 

## **Fonctionnement : Même principe que pour « great », mais avec une gestion de l’exception : si aucune occurrence n’est trouvée, le graphique n’est pas affiché pour éviter une erreur de division par zéro.**

## **Pourquoi ? Cela permet d’assurer la robustesse du code même en l’absence de données pour certains mots.**

### **d. Génération de WordClouds par auteur**

## **python**

## **Copy**

## **from wordcloud import WordCloud**

## **for author in df['author'].unique():**

## **text = ' '.join(df[df['author'] == author]['text'])**

## **wordcloud = WordCloud(width=800, height=400).generate(text)**

## **plt.figure(figsize=(10, 5))**

## **plt.imshow(wordcloud, interpolation='bilinear')**

## **plt.title(f'Word Cloud for {author}')**

## **plt.axis('off')**

## **plt.show()**

## 

## **But : Créer des nuages de mots pour chaque auteur afin de visualiser les termes les plus fréquents dans leurs textes.**

## **Pourquoi ? Le WordCloud offre une représentation visuelle intuitive de la répartition des mots et peut révéler des tendances ou particularités stylistiques.**

## 

## **7. Analyse de sentiments**

### **a. Initialisation de l’analyseur VADER**

## **python**

## **Copy**

## **from nltk.sentiment.vader import SentimentIntensityAnalyzer**

## **nltk.download('vader\_lexicon')**

## **sia = SentimentIntensityAnalyzer()**

## 

## **But : Télécharger le lexique VADER et initialiser l’analyseur de sentiments.**

## **Pourquoi ? VADER est un outil efficace pour analyser le sentiment (positif, négatif, neutre) dans des textes en anglais.**

### **b. Extraction des mots à polarité positive et négative**

## **python**

## **Copy**

## **def get\_sentiment\_words(text):**

## **words = text.split()**

## **positive\_words = [word for word in words if sia.polarity\_scores(word)['compound'] > 0]**

## **negative\_words = [word for word in words if sia.polarity\_scores(word)['compound'] < 0]**

## **return positive\_words, negative\_words**

## 

## **df['positive\_words'], df['negative\_words'] = zip(\*df['text'].apply(get\_sentiment\_words))**

## 

## **Fonctionnement : Pour chaque texte, la fonction évalue la polarité de chaque mot. Les mots ayant un score « compound » positif sont considérés comme positifs, et inversement pour les négatifs.**

## **Pourquoi ? Extraire ces mots permet d’identifier quels termes contribuent à une tonalité positive ou négative dans le corpus.**

### **c. Comptage et visualisation par WordCloud**

## **python**

## **Copy**

## **all\_positive\_words = [word for sublist in df['positive\_words'] for word in sublist]**

## **all\_negative\_words = [word for sublist in df['negative\_words'] for word in sublist]**

## 

## **top\_positive\_words = pd.Series(all\_positive\_words).value\_counts().head(100)**

## **top\_negative\_words = pd.Series(all\_negative\_words).value\_counts().head(100)**

## 

## **from wordcloud import WordCloud**

## **import matplotlib.pyplot as plt**

## 

## **positive\_wordcloud = WordCloud(width=800, height=400, background\_color='white', colormap='Greens').generate\_from\_frequencies(top\_positive\_words.to\_dict())**

## **plt.figure(figsize=(10, 5))**

## **plt.imshow(positive\_wordcloud, interpolation='bilinear')**

## **plt.axis("off")**

## **plt.title("Top 100 Positive Words")**

## **plt.show()**

## 

## **negative\_wordcloud = WordCloud(width=800, height=400, background\_color='white', colormap='Reds').generate\_from\_frequencies(top\_negative\_words.to\_dict())**

## **plt.figure(figsize=(10, 5))**

## **plt.imshow(negative\_wordcloud, interpolation='bilinear')**

## **plt.axis("off")**

## **plt.title("Top 100 Negative Words")**

## **plt.show()**

## 

## **Fonctionnement :**

## **On aplatit les listes de mots positifs et négatifs puis on compte leur fréquence.**

## **On génère ensuite des nuages de mots (WordCloud) à partir des 100 mots les plus fréquents pour chaque polarité.**

## **Pourquoi ? Cela permet de visualiser rapidement quels mots dominent dans les sentiments positifs et négatifs, offrant une vue d’ensemble sur le ton du corpus.**

## **. Objectif général**

Le code présenté constitue une chaîne de traitement (pipeline) de données textuelles. L’objectif est de nettoyer, segmenter, enrichir et analyser un corpus de textes issus d’un fichier CSV (« spooky.csv »). On y retrouve plusieurs étapes : nettoyage du texte (prétraitement), segmentation (tokenization), extraction d’entités nommées, étiquetage grammatical (POS tagging), réduction des formes (lemmatisation et stemming) ainsi que des analyses de fréquences et de sentiments.

## **2. Nettoyage du texte**

### **a. Chargement et aperçu du jeu de données**

python

Copy

import pandas as pd

url = "https://github.com/GU4243-ADS/spring2018-project1-ginnyqg/raw/master/data/spooky.csv"

df = pd.read\_csv(url)

print(df.head(10))

* **But :** Charger le jeu de données depuis un lien GitHub et afficher les 10 premières lignes pour avoir un aperçu des données brutes.
* **Pourquoi ?** Cela permet de vérifier que le fichier est correctement chargé et d’observer la structure initiale (colonnes, exemples de texte, etc.).

### **b. Téléchargement des ressources NLTK et configuration**

python

Copy

import nltk

nltk.download('wordnet')

import re

from nltk.corpus import wordnet

* **But :** Télécharger la ressource « wordnet » de NLTK nécessaire pour la vérification des mots existants et la lemmatisation.
* **Pourquoi ?** WordNet est une base lexicale qui permet de vérifier si un mot existe et fournit des synonymes et d’autres informations lexicales utiles dans le traitement du langage.

### **c. Gestion des caractères répétitifs**

#### **i. Définition de l’expression régulière et de la fonction de remplacement**

python

Copy

repeat\_regexp = re.compile(r'(\w\*)(\w)\2(\w\*)')

repl = r'\1\2\3'

* **Explication :**L’expression régulière cherche des motifs où un caractère (\2) est répété consécutivement (par exemple, « coooooool »). Le remplacement (repl) conserve seulement une occurrence du caractère répété.
* **Pourquoi ?** Les utilisateurs peuvent exagérer la répétition des lettres pour accentuer une émotion (exemple : « coooooool »). Cette étape vise à réduire ces répétitions afin d’uniformiser le texte pour une analyse ultérieure.

#### **ii. Fonction récursive replace**

python

Copy

def replace(word):

if wordnet.synsets(word):

return word

repl\_word = repeat\_regexp.sub(repl, word)

if repl\_word != word:

return replace(repl\_word)

else:

return repl\_word

* **Fonctionnement :**
  + On teste d’abord si le mot existe dans WordNet (donc s’il est reconnu).
  + Sinon, on applique l’expression régulière pour réduire les répétitions.
  + La fonction s’appelle récursivement si le mot modifié est différent, afin de continuer à simplifier jusqu’à ce qu’aucune modification supplémentaire ne soit possible.
* **Pourquoi ?** Pour s’assurer que les mots exagérément allongés soient ramenés à leur forme canonique, ce qui facilite les analyses ultérieures (recherche de synonymes, correspondances, etc.).

#### **iii. Fonction pour gérer l’ensemble d’un texte**

python

Copy

def handle\_repetitive\_chars(text):

words = text.split()

new\_words = []

for word in words:

new\_words.append(replace(word))

return ' '.join(new\_words)

* **But :** Découper le texte en mots, appliquer la fonction replace à chacun d’eux, puis reconstituer le texte.
* **Pourquoi ?** Appliquer ce nettoyage au niveau du texte complet garantit que chaque mot est traité individuellement.

#### **iv. Application sur le DataFrame**

python

Copy

df['text'] = df['text'].apply(handle\_repetitive\_chars)

print(handle\_repetitive\_chars("cooooooool"))

print(df.head(10))

* **But :** Mettre à jour la colonne « text » du DataFrame avec le texte nettoyé et vérifier l’effet de la fonction en testant sur le mot « cooooooool ».
* **Pourquoi ?** Cette transformation prépare les données en éliminant les variations exagérées dues aux répétitions de caractères.

### **d. Remplacement des homoglyphes**

#### **i. Dictionnaire des homoglyphes**

python

Copy

homoglyphs\_map = {

'$': 's', '0': 'o', '1': 'l', '3': 'e', '@': 'a', '5': 's', '7': 't',

'8': 'b', '|': 'i', '¢': 'c', '¡': 'i', 'µ': 'u', 'ß': 'b', '€': 'e',

'!':'i','£':'l',

'Α': 'A', 'Β': 'B', 'Ε': 'E', 'Ζ': 'Z', 'Η': 'H', 'Ι': 'I', 'Κ': 'K',

'Μ': 'M', 'Ν': 'N', 'Ο': 'O', 'Ρ': 'P', 'Τ': 'T', 'Χ': 'X',

'А': 'A', 'В': 'B', 'С': 'C', 'Е': 'E', 'Н': 'H', 'К': 'K', 'М': 'M',

'О': 'O', 'Р': 'P', 'Т': 'T', 'Х': 'X'

}

* **Explication :**Ce dictionnaire mappe certains caractères spéciaux ou similaires (homoglyphes) à leur équivalent alphabétique. Par exemple, « $ » est remplacé par « s », et des lettres issues d’alphabets différents (grec, cyrillique) sont converties en lettres latines.
* **Pourquoi ?** Pour uniformiser le texte et éviter que des variantes graphiques ne perturbent l’analyse (recherche, tokenisation, etc.).

#### **ii. Fonction de remplacement**

python

Copy

def replace\_homoglyphs(text):

return ''.join(homoglyphs\_map.get(char, char) for char in text)

* **Fonctionnement :**Pour chaque caractère du texte, on remplace celui-ci par sa valeur dans le dictionnaire si une correspondance existe, sinon on le conserve.
* **Pourquoi ?** Cette fonction permet de normaliser les caractères, ce qui est essentiel pour garantir la cohérence lors des analyses ultérieures.

#### **iii. Application et test**

python

Copy

print(replace\_homoglyphs("$tupid"))

df['text'] = df['text'].apply(replace\_homoglyphs)

* **But :** Vérifier le bon fonctionnement de la fonction (le test devrait renvoyer « stupid ») puis l’appliquer à tous les textes du DataFrame.

### **e. Nettoyage des entrées spéciales**

python

Copy

from bs4 import BeautifulSoup

def clean\_special\_entries(text):

text = re.sub(r'http\S+|www\S+', ' <URL> ', text)

text = re.sub(r'\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', ' <EMAIL> ', text)

text = BeautifulSoup(text, "html.parser").get\_text()

return text

df['text'] = df['text'].apply(clean\_special\_entries)

* **Fonctionnement :**
  + Remplace les URLs par la balise « <URL> ».
  + Remplace les adresses e-mail par « <EMAIL> ».
  + Utilise BeautifulSoup pour enlever les balises HTML, ne gardant que le texte.
* **Pourquoi ?** Ces transformations assurent que le texte ne contienne pas de données spécifiques comme des liens ou des emails qui pourraient fausser l’analyse sémantique ou les statistiques.

### **f. Transformation en minuscules et suppression de la ponctuation**

python

Copy

df['text'] = df['text'].str.lower()

import string

def remove\_punctuation(text):

return text.translate(str.maketrans('', '', string.punctuation))

df['text'] = df['text'].apply(remove\_punctuation)

* **But :**
  + Passer tout le texte en minuscules pour éviter les différences dues à la casse.
  + Supprimer la ponctuation qui peut ne pas être pertinente pour certaines analyses.
* **Pourquoi ?** Cela permet d’uniformiser le texte pour les étapes suivantes (tokenisation, analyse de fréquence, etc.).

### **g. Suppression des stopwords**

python

Copy

from nltk.corpus import stopwords

nltk.download('stopwords')

stop\_words = set(stopwords.words('english'))

def remove\_stopwords(text):

return " ".join([word for word in text.split() if word not in stop\_words])

df['text'] = df['text'].apply(remove\_stopwords)

* **But :** Enlever les mots très fréquents (stopwords) comme « the, and, is », qui n’apportent généralement pas d’information significative.
* **Pourquoi ?** La suppression des stopwords améliore la qualité des analyses en se concentrant sur les mots porteurs de sens.

### **h. Sauvegarde des données nettoyées**

python

Copy

df = df.head(100)

df.to\_csv("cleaned\_spooky.csv", index=False)

print("Preprocessing complete. Data saved as cleaned\_spooky.csv")

* **But :** Limiter le DataFrame aux 100 premiers enregistrements (pour des raisons de test) et sauvegarder le résultat dans un nouveau fichier CSV.
* **Pourquoi ?** Cela permet de conserver une version prétraitée des données pour les étapes de segmentation et d’analyse ultérieures.

## **3. Segmentation (Tokenization)**

### **a. Préparation de l’environnement**

python

Copy

import spacy

nltk.download('punkt')

nltk.download('punkt\_tab')

nlp = spacy.load("en\_core\_web\_sm")

file = 'cleaned\_spooky.csv'

df = pd.read\_csv(file)

* **But :**
  + Télécharger les ressources nécessaires pour la tokenisation avec NLTK.
  + Charger le modèle SpaCy pour le traitement du langage.
  + Charger les données préalablement nettoyées.
* **Pourquoi ?** Ces ressources permettent de segmenter le texte en phrases et en mots, étape cruciale pour les analyses ultérieures.

### **b. Tokenisation avec NLTK**

python

Copy

df = df.dropna(subset=['text'])

df['tokens'] = df['text'].apply(word\_tokenize)

print("First 10 rows after tokenization:")

print(df.head(10))

* **Fonctionnement :**On supprime les lignes où la colonne « text » est vide, puis on applique la fonction word\_tokenize de NLTK à chaque texte.
* **Pourquoi ?** La tokenisation divise le texte en unités (tokens) qui sont ensuite utilisées pour l’analyse sémantique, la recherche de fréquences, etc.

### **c. Tokenisation basée sur SpaCy et BERT**

#### **i. Tokenisation par règle avec SpaCy**

python

Copy

def rule\_based\_tokenization(text):

doc = nlp(text)

return [token.text for token in doc]

df['rule\_based\_tokens'] = df['text'].apply(rule\_based\_tokenization)

* **But :** Utiliser SpaCy pour obtenir une tokenisation qui tient compte de la structure linguistique.
* **Pourquoi ?** SpaCy offre une tokenisation robuste et peut servir pour des analyses nécessitant la segmentation par règles linguistiques.

#### **ii. Tokenisation subword avec BERT**

python

Copy

from transformers import BertTokenizer

tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

def subword\_tokenization(text):

return tokenizer.tokenize(text)

df['subword\_tokens'] = df['text'].apply(subword\_tokenization)

* **Fonctionnement :**Le tokenizer BERT segmente le texte en sous-mots (subwords) selon le modèle pré-entraîné « bert-base-uncased ».
* **Pourquoi ?** La tokenisation en sous-mots est essentielle pour les modèles de langage modernes (comme BERT) qui traitent les mots rares ou inconnus en les décomposant en unités plus petites.

## **4. Reconnaissance d'entités nommées (NER) et POS tagging**

### **a. Extraction d’entités nommées**

python

Copy

def extract\_entities(text):

doc = nlp(text)

return [(ent.text, ent.label\_) for ent in doc.ents]

df['entities'] = df['text'].apply(extract\_entities)

* **But :** Utiliser SpaCy pour identifier et extraire les entités nommées (personnes, lieux, organisations, etc.) dans le texte.
* **Pourquoi ?** L’extraction d’entités est utile pour comprendre la structure du texte et pour effectuer des analyses spécifiques (ex. étudier la mention d’un auteur ou d’un lieu).

### **b. Étiquetage grammatical (POS tagging)**

python

Copy

def pos\_tagging(text):

doc = nlp(text)

return [(token.text, token.pos\_) for token in doc]

df['pos\_tags'] = df['text'].apply(pos\_tagging)

print(df.head(10))

* **Fonctionnement :**Chaque mot du texte est associé à sa catégorie grammaticale (nom, verbe, adjectif, etc.) grâce à SpaCy.
* **Pourquoi ?** L’étiquetage grammatical permet d’approfondir l’analyse linguistique et peut aider dans des tâches comme la lemmatisation ou l’extraction de syntagmes.

## **5. Réduction des formes**

### **a. Lemmatisation et Stemming**

python

Copy

from nltk.stem import WordNetLemmatizer, PorterStemmer

lemmatizer = WordNetLemmatizer()

stemmer = PorterStemmer()

def lemmatize\_text(text):

return ' '.join([lemmatizer.lemmatize(word) for word in text.split()])

def stem\_text(text):

return ' '.join([stemmer.stem(word) for word in text.split()])

df['lemmatized\_text'] = df['text'].apply(lemmatize\_text)

df['stemmed\_text'] = df['text'].apply(stem\_text)

* **Explication :**
  + **Lemmatisation :** Réduit chaque mot à sa forme canonique (lemme) en tenant compte du contexte.
  + **Stemming :** Tronque les mots pour obtenir leur radical, souvent de manière plus brutale.
* **Pourquoi ?** Ces techniques réduisent la variance des formes de mots et permettent de regrouper des variantes (ex. « running », « ran », « runs » → « run ») pour des analyses de fréquence ou de similarité.

## **6. Analyse des fréquences**

### **a. Comptage d’occurrences de mots spécifiques**

python

Copy

df['contains\_great'] = df['text'].apply(lambda x: 'great' in x.lower())

great\_counts = df.groupby('author')['contains\_great'].sum()

print(great\_counts)

* **Fonctionnement :**On ajoute une colonne booléenne indiquant si le mot « great » apparaît dans le texte, puis on regroupe par auteur pour compter les occurrences.
* **Pourquoi ?** Cela permet d’identifier quels auteurs utilisent plus fréquemment certains termes, fournissant une analyse stylistique ou thématique.

### **b. Visualisation avec un graphique Waffle**

python

Copy

from pywaffle import Waffle

import matplotlib.pyplot as plt

data = great\_counts.to\_dict()

fig = plt.figure(

FigureClass=Waffle,

rows=5,

columns=10,

values=data,

legend={'loc': 'upper left', 'bbox\_to\_anchor': (1, 1)},

title={'label': 'Occurrences of "great" by Author', 'loc': 'left'}

)

plt.show()

* **But :** Créer un graphique en forme de « waffle chart » qui représente visuellement la fréquence d’un mot par auteur.
* **Pourquoi ?** La visualisation aide à interpréter rapidement les différences entre les auteurs.

### **c. Comptage et visualisation pour le mot « impossible »**

python

Copy

df['contains\_impossible'] = df['text'].apply(lambda x: 'impossible' in x.lower())

impossible\_counts = df.groupby('author')['contains\_impossible'].sum()

print(impossible\_counts)

data = impossible\_counts.to\_dict()

if sum(data.values()) == 0:

print("No data available to plot the waffle chart.")

else:

fig = plt.figure(

FigureClass=Waffle,

rows=5,

columns=10,

values=data,

legend={'loc': 'upper left', 'bbox\_to\_anchor': (1, 1)},

title={'label': 'Occurrences of "impossible" by Author', 'loc': 'left'}

)

plt.show()

* **Fonctionnement :**Même principe que pour « great », mais avec une gestion de l’exception : si aucune occurrence n’est trouvée, le graphique n’est pas affiché pour éviter une erreur de division par zéro.
* **Pourquoi ?** Cela permet d’assurer la robustesse du code même en l’absence de données pour certains mots.

### **d. Génération de WordClouds par auteur**

python

Copy

from wordcloud import WordCloud

for author in df['author'].unique():

text = ' '.join(df[df['author'] == author]['text'])

wordcloud = WordCloud(width=800, height=400).generate(text)

plt.figure(figsize=(10, 5))

plt.imshow(wordcloud, interpolation='bilinear')

plt.title(f'Word Cloud for {author}')

plt.axis('off')

plt.show()

* **But :** Créer des nuages de mots pour chaque auteur afin de visualiser les termes les plus fréquents dans leurs textes.
* **Pourquoi ?** Le WordCloud offre une représentation visuelle intuitive de la répartition des mots et peut révéler des tendances ou particularités stylistiques.

## **7. Analyse de sentiments**

### **a. Initialisation de l’analyseur VADER**

python

Copy

from nltk.sentiment.vader import SentimentIntensityAnalyzer

nltk.download('vader\_lexicon')

sia = SentimentIntensityAnalyzer()

* **But :** Télécharger le lexique VADER et initialiser l’analyseur de sentiments.
* **Pourquoi ?** VADER est un outil efficace pour analyser le sentiment (positif, négatif, neutre) dans des textes en anglais.

### **b. Extraction des mots à polarité positive et négative**

python

Copy

def get\_sentiment\_words(text):

words = text.split()

positive\_words = [word for word in words if sia.polarity\_scores(word)['compound'] > 0]

negative\_words = [word for word in words if sia.polarity\_scores(word)['compound'] < 0]

return positive\_words, negative\_words

df['positive\_words'], df['negative\_words'] = zip(\*df['text'].apply(get\_sentiment\_words))

* **Fonctionnement :**Pour chaque texte, la fonction évalue la polarité de chaque mot. Les mots ayant un score « compound » positif sont considérés comme positifs, et inversement pour les négatifs.
* **Pourquoi ?** Extraire ces mots permet d’identifier quels termes contribuent à une tonalité positive ou négative dans le corpus.

### **c. Comptage et visualisation par WordCloud**

python

Copy

all\_positive\_words = [word for sublist in df['positive\_words'] for word in sublist]

all\_negative\_words = [word for sublist in df['negative\_words'] for word in sublist]

top\_positive\_words = pd.Series(all\_positive\_words).value\_counts().head(100)

top\_negative\_words = pd.Series(all\_negative\_words).value\_counts().head(100)

from wordcloud import WordCloud

import matplotlib.pyplot as plt

positive\_wordcloud = WordCloud(width=800, height=400, background\_color='white', colormap='Greens').generate\_from\_frequencies(top\_positive\_words.to\_dict())

plt.figure(figsize=(10, 5))

plt.imshow(positive\_wordcloud, interpolation='bilinear')

plt.axis("off")

plt.title("Top 100 Positive Words")

plt.show()

negative\_wordcloud = WordCloud(width=800, height=400, background\_color='white', colormap='Reds').generate\_from\_frequencies(top\_negative\_words.to\_dict())

plt.figure(figsize=(10, 5))

plt.imshow(negative\_wordcloud, interpolation='bilinear')

plt.axis("off")

plt.title("Top 100 Negative Words")

plt.show()

* **Fonctionnement :**
  + On aplatit les listes de mots positifs et négatifs puis on compte leur fréquence.
  + On génère ensuite des nuages de mots (WordCloud) à partir des 100 mots les plus fréquents pour chaque polarité.
* **Pourquoi ?** Cela permet de visualiser rapidement quels mots dominent dans les sentiments positifs et négatifs, offrant une vue d’ensemble sur le ton du corpus.