**🔍 Projet — Détection automatique de toxicité dans les avis utilisateurs**

**🎯 Objectif global**

**Concevoir un pipeline robuste de modération automatique de texte pour détecter et classifier les avis toxiques, insultants ou inappropriés, à partir de jeux de données textuels bruts (IMDB, Amazon, etc.).**

**❓ Problématique**

**Comment détecter automatiquement, sans intervention humaine, les contenus toxiques ou haineux dans de grands volumes d’avis utilisateurs, tout en assurant une couverture lexicale (mots injurieux explicites) et sémantique (formes plus subtiles) ?**

**🧭 Objectifs du projet**

1. **Collecter et fusionner des jeux d’avis textuels (IMDB, Amazon, etc.)**
2. **Explorer, nettoyer et enrichir les textes (EDA + NLP preprocessing)**
3. **Créer une variable cible label\_toxic via des règles et des scores**
4. **Extraire des features linguistiques utiles pour la modélisation**
5. **Construire un modèle de classification supervisée robuste**
6. **Expliquer les décisions du modèle (Explainable NLP)**
7. **Déployer un système de modération automatique fiable et interprétable**

**🧩 Phase 1 — Exploration & enrichissement initial (EDA)**

**🎯 Objectif**

**Comprendre la structure des avis bruts, identifier les patterns lexicaux/toxiques, et produire un jeu de données enrichi (merged\_reviews\_eda.csv) prêt à être nettoyé.**

**📁 Notebook : 01\_EDA\_reviews.ipynb**

| **Étape** | **Fonction** | **Résultat** |
| --- | --- | --- |
| **0. Setup** | **Imports, chemins** | **Structure de projet prête** |
| **1. Fusion** | **Fusion IMDB + Amazon** | **df\_all unifié** |
| **2. Stats de base** | **n\_words, n\_sentences, etc.** | **Premiers descripteurs** |
| **3. Visualisation** | **Histogrammes, boxplots** | **Outliers repérés** |
| **4. Wordcloud & top mots** | **Counter, WordCloud** | **Top 20 mots fréquents** |
| **5. Bigrams / TF-IDF** | **CountVectorizer, TfidfVectorizer** | **Expressions clés** |
| **6. Toxicité brute** | **Regex + flag** | **potential\_toxic, flag\_badwords** |
| **7. Corrélations** | **n\_words ↔ toxicity** | **Scatterplot, heatmap** |
| **8. Longs avis** | **Inspection manuelle** | **Spam ou discours excessif** |
| **9. Sauvegarde** | **Export CSV** | **merged\_reviews\_eda.csv** |

**📦 Fichier produit : data/processed/merged\_reviews\_eda.csv**

**Contenu :**

* **Texte brut + features EDA**
* **Colonnes : n\_words, n\_chars, avg\_word\_len, lexical\_density, potential\_toxic, flag\_badwords, etc.**

**✅ Résumé**

| **Élement** | **Statut** | **Commentaire** |
| --- | --- | --- |
| **Fusion et préparation** | **✅** | **Pas de doublons** |
| **Analyse exploratoire** | **✅** | **Indicateurs clairs** |
| **Détection toxicité** | **✅** | **Flags bien définis** |
| **Visualisation & vocabulaire** | **✅** | **Top mots + bigrams** |

**🔁 Phase 2 — Préparation avancée des données NLP**

**🎯 Objectif**

**Transformer les textes explorés en un jeu nettoyé, enrichi, étiqueté et prêt pour la modélisation (Phase 3).**

**📁 Structure modulaire (scripts + notebook)**

| **Fichier / Notebook** | **Rôle** | **Fonction principale** |
| --- | --- | --- |
| **text\_cleaner.py** | **Nettoyage NLP** | **Nettoyer, lemmatiser, filtrer** |
| **labeling.py** | **Étiquetage cible** | **Générer label\_toxic, analyser** |
| **feature\_engineering.py** | **Enrichissement** | **Scores de lisibilité, sentiment, regex** |
| **02\_preprocessing.ipynb** | **Pipeline central** | **Orchestration et sauvegarde** |

**⚙️ Étapes détaillées**

**🔹 2.1 — Nettoyage textuel (préprocessing)**

**Fichier : text\_cleaner.py**

| **Fonction** | **Objectif** |
| --- | --- |
| **clean\_text()** | **Nettoyage complet (punctuation, HTML, emoji, URL, etc.)** |
| **clean\_pipeline()** | **Application à un DataFrame, filtrage des textes courts/longs** |

**✅ Fait :**

* **Passage en minuscules**
* **Suppression : ponctuation, HTML, chiffres, emojis, URL, emails**
* **Stopwords via NLTK**
* **Lemmatisation via spaCy**
* **Filtrage des textes trop courts ou trop longs**
* **Suppression des doublons**
* **✅ text\_clean générée**

**🔹 2.2 — Étiquetage automatique**

**Fichier : labeling.py**

| **Fonction** | **Objectif** |
| --- | --- |
| **generate\_toxic\_labels()** | **Créer label\_toxic = 1 si potential\_toxic ou flag\_badwords** |
| **analyze\_label\_distribution()** | **Diagnostic du déséquilibre des classes** |
| **preview\_labeled\_samples()** | **Affichage d’exemples par classe** |
| **add\_toxic\_source()** | **Ajoute la source (badword, score, both)** |

**✅ Fait :**

* **Colonne label\_toxic créée**
* **Distribution analysée (déséquilibre confirmé)**
* **Possibilité future d’étiquetage via LLM (optionnel)**

**🔹 2.3 — Enrichissement linguistique**

**Fichier : feature\_engineering.py**

| **Feature** | **Objectif** |
| --- | --- |
| **flesch\_score, fk\_grade** | **Lisibilité** |
| **sentiment\_polarity, subjectivity** | **Sentiment NLP** |
| **capital\_ratio** | **Indice d’agressivité** |
| **nb\_exclamations, nb\_questions** | **Expressivité** |
| **has\_url, has\_email, has\_phone** | **Regex structurelles** |
| **has\_emoji, has\_repeated\_chars** | **Subtilités d’écriture** |

**✅ Fait :**

* **Toutes les features extraites**
* **Ajoutées dans df\_enriched**

**🔹 2.4 — Rééquilibrage et préparation**

**Fichier : labeling.py (fonction create\_balanced\_subset())**

| **Élément** | **Statut** |
| --- | --- |
| **df\_clean** | **✅ Données nettoyées avec text\_clean** |
| **df\_enriched** | **✅ Données enrichies** |
| **df\_balanced** | **✅ Créé par downsampling (classe majoritaire réduite)** |
| **Pondération modèle** | **🔜 À utiliser dans la phase 3 avec class\_weight="balanced"** |

**✅ Résumé fin de phase 2**

| **Étape** | **Fait** | **Commentaire** |
| --- | --- | --- |
| **Nettoyage structuré** | **✅** | **Nettoyage riche et contrôlé** |
| **Étiquetage cible** | **✅** | **Binaire, explicite** |
| **Feature engineering** | **✅** | **Complet, riche et varié** |
| **Rééquilibrage** | **✅** | **df\_balanced prêt à l’emploi** |
| **Exports** | **🔜** | **À faire dans le notebook 02\_preprocessing.ipynb** |

**?**

## 🚀 \*\*Guide Complet : Pipeline NLP 100% Local (Sans APIs)\*\*

## 📋 \*\*Vue d'ensemble\*\*

\*\*Durée estimée\*\* : 4–6 semaines

\*\*Niveau de complexité\*\* : Avancé

\*\*Stack principal\*\* : Python, Transformers (HuggingFace), Scrapy, Streamlit, Docker

\*\*Particularité\*\* : Aucune API externe — tout tourne localement.

---

## 🎯 \*\*Phase 1 : Préparation & Setup Initial (Semaine 1)\*\*

### 🔹 \*\*Jour 1–2 : Configuration de l’environnement\*\*

#### ✅ 1.1 \*\*Installation des outils\*\*

```bash

# Créer un environnement virtuel Python 3.9+

python -m venv venv

source venv/bin/activate # Linux/Mac

# ou

venv\Scripts\activate # Windows

# Installer les dépendances principales

pip install numpy pandas scikit-learn

pip install transformers torch datasets

pip install scrapy beautifulsoup4 selenium

pip install streamlit fastapi uvicorn

pip install pytest black flake8

pip install wordcloud matplotlib seaborn

pip install plotly redis

```

#### ✅ 1.2 \*\*Structure du projet\*\*

```bash

mkdir safe-review-pipeline

cd safe-review-pipeline

# Créer l’arborescence

mkdir -p data/{raw,processed,models,scraped}

mkdir -p notebooks

mkdir -p src/{data,models,pipeline,api,utils}

mkdir -p app tests config logs scripts

# Initialiser Git

git init

echo "venv/" > .gitignore

echo "\*.pyc" >> .gitignore

echo "\_\_pycache\_\_/" >> .gitignore

echo "data/\*.csv" >> .gitignore

echo "models/\*.bin" >> .gitignore

echo "logs/\*.log" >> .gitignore

```

---

### 🔹 \*\*Jour 3–4 : Collecte des données (Sans APIs)\*\*

#### ✅ 1.3 \*\*Importer des jeux de données publics\*\*

```python

# notebooks/00\_download\_public\_datasets.ipynb

from datasets import load\_dataset

import pandas as pd

# Dataset Amazon

amazon\_reviews = load\_dataset("amazon\_polarity", split="train[:10000]")

df\_amazon = pd.DataFrame(amazon\_reviews)

df\_amazon.to\_csv('data/raw/amazon\_reviews.csv', index=False)

# Dataset IMDB

imdb\_reviews = load\_dataset("imdb", split="train[:5000]")

df\_imdb = pd.DataFrame(imdb\_reviews)

df\_imdb.to\_csv('data/raw/imdb\_reviews.csv', index=False)

# Dataset de toxicité

toxic\_comments = load\_dataset("toxic\_comments", split="train[:20000]")

df\_toxic = pd.DataFrame(toxic\_comments)

df\_toxic.to\_csv('data/raw/toxic\_comments\_training.csv', index=False)

print(f"Amazon reviews: {len(df\_amazon)}")

print(f"IMDB reviews: {len(df\_imdb)}")

print(f"Toxic comments: {len(df\_toxic)}")

```

#### ✅ 1.4 \*\*Scraping Trustpilot éthique\*\*

```python

# src/data/scrapers/review\_scraper.py

import scrapy

from scrapy.crawler import CrawlerProcess

from typing import List

class TrustpilotScraper(scrapy.Spider):

"""Scraper Trustpilot respectueux du robots.txt"""

name = 'trustpilot\_scraper'

custom\_settings = {

'DOWNLOAD\_DELAY': 3,

'CONCURRENT\_REQUESTS': 1,

'ROBOTSTXT\_OBEY': True,

'USER\_AGENT': 'Mozilla/5.0 (Educational Purpose Bot)'

}

def \_\_init\_\_(self, company\_urls: List[str]):

self.start\_urls = company\_urls

self.page\_count = 0

def parse(self, response):

reviews = response.css('div.review-card')

for review in reviews:

yield {

'text': review.css('p.review-text::text').get(),

'rating': review.css('div.star-rating::attr(data-rating)').get(),

'date': review.css('time::attr(datetime)').get(),

'title': review.css('h3.review-title::text').get(),

'url': response.url

}

next\_page = response.css('a.next-page::attr(href)').get()

if next\_page and self.page\_count < 10:

self.page\_count += 1

yield response.follow(next\_page, self.parse)

# Lancer le scraper

def scrape\_reviews(company\_urls: List[str], output\_file: str):

process = CrawlerProcess({'FEEDS': {output\_file: {'format': 'csv'}}})

process.crawl(TrustpilotScraper, company\_urls=company\_urls)

process.start()

```

#### ✅ 1.5 \*\*Générer des données synthétiques\*\*

```python

# src/data/synthetic\_generator.py

import random

from typing import List

import pandas as pd

class SyntheticReviewGenerator:

"""Génère des avis positifs et négatifs artificiels"""

def \_\_init\_\_(self):

self.positive\_patterns = [

"Le produit est {adj\_pos}, je {verb\_pos}",

"Service {adj\_pos}, livraison {adj\_pos}",

"{adj\_pos} qualité, {adj\_pos} prix"

]

self.negative\_patterns = [

"Très {adj\_neg}, je {verb\_neg}",

"Produit {adj\_neg}, service {adj\_neg}",

"Qualité {adj\_neg}, ne {verb\_neg} pas"

]

self.adj\_pos = ["excellent", "parfait", "super", "génial", "formidable"]

self.adj\_neg = ["décevant", "médiocre", "mauvais", "horrible", "nul"]

self.verb\_pos = ["recommande", "suis satisfait", "adore", "apprécie"]

self.verb\_neg = ["déconseille", "regrette", "suis déçu"]

def generate\_reviews(self, n\_positive: int, n\_negative: int) -> pd.DataFrame:

reviews = []

for \_ in range(n\_positive):

pattern = random.choice(self.positive\_patterns)

reviews.append({

'text': pattern.format(

adj\_pos=random.choice(self.adj\_pos),

verb\_pos=random.choice(self.verb\_pos)

),

'label': 'positive',

'synthetic': True

})

for \_ in range(n\_negative):

pattern = random.choice(self.negative\_patterns)

reviews.append({

'text': pattern.format(

adj\_neg=random.choice(self.adj\_neg),

verb\_neg=random.choice(self.verb\_neg)

),

'label': 'negative',

'synthetic': True

})

return pd.DataFrame(reviews)

```

---

### 🔹 \*\*Jour 5–7 : EDA — Analyse exploratoire\*\*

```python

# notebooks/01\_EDA\_reviews.ipynb

import pandas as pd

import matplotlib.pyplot as plt

from wordcloud import WordCloud

df\_amazon = pd.read\_csv('data/raw/amazon\_reviews.csv')

df\_imdb = pd.read\_csv('data/raw/imdb\_reviews.csv')

df\_all = pd.concat([

df\_amazon.rename(columns={'content': 'text', 'label': 'sentiment'}),

df\_imdb.rename(columns={'text': 'text', 'label': 'sentiment'})

])

print(f"Total: {len(df\_all)}")

print(f"Moyenne mots: {df\_all['text'].str.split().str.len().mean():.1f}")

df\_all['text\_length'] = df\_all['text'].str.len()

plt.hist(df\_all['text\_length'], bins=50)

plt.xlabel('Longueur')

plt.ylabel('Fréquence')

plt.title('Distribution des longueurs')

plt.show()

text = ' '.join(df\_all['text'].astype(str))

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

plt.imshow(wordcloud)

plt.axis('off')

plt.title('Mots fréquents')

plt.show()

toxic\_words = ['hate', 'terrible', 'worst']

df\_all['potential\_toxic'] = df\_all['text'].str.lower().str.contains('|'.join(toxic\_words))

print(f"Potentiellement toxiques: {df\_all['potential\_toxic'].sum()}")

```

---

✅ \*\*Tout est conservé\*\*, clarifié et reformulé \*\*sans rien perdre\*\*.

Si tu veux, je peux faire pareil pour \*\*les phases 2, 3 et 4\*\* ! Veux-tu que je continue ?

Voici la suite \*\*Phase 2 (Core)\*\* jusqu’au \*\*début de la Phase 3\*\*, exactement comme tu l’as donnée, mais nettoyée et réécrite proprement.

---

## 🔧 \*\*Phase 2 : Développement des Composants Core (Semaine 2–3)\*\*

---

### 🔹 \*\*Jour 8–10 : Classifieur de Toxicité 100% Local\*\*

#### ✅ \*\*2.1 Modèle HuggingFace Local\*\*

```python

# src/models/toxic\_classifier.py

from transformers import pipeline, AutoModelForSequenceClassification, AutoTokenizer

import torch

from typing import List, Dict

import numpy as np

class LocalToxicityClassifier:

"""Classifieur de toxicité utilisant un modèle local uniquement."""

def \_\_init\_\_(self, model\_name: str = "unitary/toxic-bert"):

print(f"Chargement du modèle {model\_name}...")

self.device = 0 if torch.cuda.is\_available() else -1

self.pipeline = pipeline(

"text-classification",

model=model\_name,

device=self.device

)

self.threshold = 0.5

def predict(self, texts: List[str]) -> List[Dict]:

"""Prédit la toxicité d'une liste de textes."""

results = []

batch\_size = 32

for i in range(0, len(texts), batch\_size):

batch = texts[i:i+batch\_size]

predictions = self.pipeline(batch)

for text, pred in zip(batch, predictions):

toxic\_score = pred['score'] if pred['label'] == 'TOXIC' else 1 - pred['score']

results.append({

'text': text,

'toxic\_score': toxic\_score,

'is\_toxic': toxic\_score > self.threshold,

'confidence': pred['score']

})

return results

def fine\_tune(self, train\_data: pd.DataFrame):

"""Placeholder pour fine-tuning local."""

pass

```

#### ✅ \*\*2.2 Variante Classifieur Custom\*\*

```python

# notebooks/02\_train\_custom\_toxic\_classifier.ipynb

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.linear\_model import LogisticRegression

from sklearn.pipeline import Pipeline

import joblib

df\_toxic = pd.read\_csv('data/raw/toxic\_comments\_training.csv')

class SimpleToxicClassifier:

def \_\_init\_\_(self):

self.pipeline = Pipeline([

('tfidf', TfidfVectorizer(max\_features=10000, ngram\_range=(1, 2))),

('classifier', LogisticRegression(class\_weight='balanced'))

])

def fit(self, X, y):

self.pipeline.fit(X, y)

def predict\_proba(self, X):

return self.pipeline.predict\_proba(X)

def save(self, path):

joblib.dump(self.pipeline, path)

def load(self, path):

self.pipeline = joblib.load(path)

classifier = SimpleToxicClassifier()

classifier.fit(df\_toxic['text'], df\_toxic['toxic'])

classifier.save('models/simple\_toxic\_classifier.pkl')

```

---

### 🔹 \*\*Jour 11–13 : Système de Résumé Local\*\*

#### ✅ \*\*2.3 Résumeur HuggingFace Local\*\*

```python

# src/models/summarizer.py

from transformers import PegasusForConditionalGeneration, PegasusTokenizer

from transformers import T5ForConditionalGeneration, T5Tokenizer

import torch

from typing import List, Optional

class LocalSummarizer:

"""Résumeur local basé sur Pegasus ou T5."""

def \_\_init\_\_(self, model\_type: str = "pegasus"):

self.device = "cuda" if torch.cuda.is\_available() else "cpu"

if model\_type == "pegasus":

model\_name = "google/pegasus-cnn\_dailymail"

self.model = PegasusForConditionalGeneration.from\_pretrained(

model\_name, cache\_dir="./models/pegasus\_cache"

).to(self.device)

self.tokenizer = PegasusTokenizer.from\_pretrained(

model\_name, cache\_dir="./models/pegasus\_cache"

)

elif model\_type == "t5":

model\_name = "t5-small"

self.model = T5ForConditionalGeneration.from\_pretrained(

model\_name, cache\_dir="./models/t5\_cache"

).to(self.device)

self.tokenizer = T5Tokenizer.from\_pretrained(

model\_name, cache\_dir="./models/t5\_cache"

)

self.prefix = "summarize: "

self.generation\_config = {

'max\_length': 60,

'min\_length': 20,

'length\_penalty': 2.0,

'num\_beams': 4,

'early\_stopping': True,

'no\_repeat\_ngram\_size': 3

}

def summarize\_batch(self, texts: List[str], max\_length: Optional[int] = None) -> List[str]:

if max\_length:

self.generation\_config['max\_length'] = max\_length

summaries = []

batch\_size = 4

for i in range(0, len(texts), batch\_size):

batch = texts[i:i+batch\_size]

if hasattr(self, 'prefix'):

batch = [self.prefix + text for text in batch]

inputs = self.tokenizer(

batch, max\_length=512, truncation=True, padding=True, return\_tensors="pt"

).to(self.device)

with torch.no\_grad():

summary\_ids = self.model.generate(inputs["input\_ids"], \*\*self.generation\_config)

summaries.extend(self.tokenizer.batch\_decode(

summary\_ids, skip\_special\_tokens=True, clean\_up\_tokenization\_spaces=True

))

return summaries

def summarize\_long\_text(self, text: str, chunk\_size: int = 400) -> str:

sentences = text.split('. ')

chunks, current\_chunk = [], ""

for sentence in sentences:

if len(current\_chunk) + len(sentence) < chunk\_size:

current\_chunk += sentence + ". "

else:

chunks.append(current\_chunk)

current\_chunk = sentence + ". "

if current\_chunk:

chunks.append(current\_chunk)

chunk\_summaries = self.summarize\_batch(chunks)

combined = " ".join(chunk\_summaries)

return self.summarize\_batch([combined])[0] if len(combined.split()) > 100 else combined

```

---

### 🔹 \*\*Jour 14–16 : Système de Reformulation\*\*

#### ✅ \*\*2.4 Reformulateur Local T5\*\*

```python

# src/models/rewriter.py

from transformers import T5ForConditionalGeneration, T5Tokenizer

import torch

from typing import List

import re

class LocalSafeRewriter:

"""Reformulateur basé sur T5."""

def \_\_init\_\_(self):

self.model\_name = "t5-base"

self.device = "cuda" if torch.cuda.is\_available() else "cpu"

print(f"Chargement du modèle {self.model\_name}...")

self.model = T5ForConditionalGeneration.from\_pretrained(

self.model\_name, cache\_dir="./models/t5\_rewriter\_cache"

).to(self.device)

self.tokenizer = T5Tokenizer.from\_pretrained(

self.model\_name, cache\_dir="./models/t5\_rewriter\_cache"

)

self.templates = {

'high\_toxic': "Make this review constructive and professional: ",

'medium\_toxic': "Rephrase this review more positively: ",

'low\_toxic': "Slightly improve the tone of this review: "

}

def rewrite\_toxic\_content(self, text: str, toxicity\_score: float) -> str:

prefix = self.templates['high\_toxic'] if toxicity\_score > 0.8 else (

self.templates['medium\_toxic'] if toxicity\_score > 0.6 else self.templates['low\_toxic'])

inputs = self.tokenizer(

prefix + text, max\_length=512, truncation=True, return\_tensors="pt"

).to(self.device)

with torch.no\_grad():

outputs = self.model.generate(

inputs["input\_ids"],

max\_length=150, min\_length=20, temperature=0.7,

do\_sample=True, top\_p=0.9, num\_beams=3

)

rewritten = self.tokenizer.decode(outputs[0], skip\_special\_tokens=True)

return self.clean\_output(rewritten, text)

def clean\_output(self, rewritten: str, original: str) -> str:

rewritten = re.sub(r'\b(\w+)(\s+\1\b)+', r'\1', rewritten)

if len(rewritten.split()) < 5:

return self.simple\_detoxify(original)

return rewritten

def simple\_detoxify(self, text: str) -> str:

replacements = {

'hate': 'dislike', 'horrible': 'disappointing',

'terrible': 'not satisfactory', 'worst': 'not the best',

'awful': 'not great', 'stupid': 'not well thought out',

'disgusting': 'unpleasant'

}

result = text.lower()

for bad, good in replacements.items():

result = result.replace(bad, good)

return result[0].upper() + result[1:] if result else result

def batch\_rewrite(self, texts\_scores: List[tuple]) -> List[str]:

return [self.rewrite\_toxic\_content(text, score) if score > 0.5 else None for text, score in texts\_scores]

```

## 🔄 \*\*Phase 3 : Pipeline et Orchestration (Semaine 3–4)\*\*

---

### 🔹 \*\*Jour 17–19 : Pipeline Local Intégré\*\*

#### ✅ \*\*3.1 Pipeline Principal\*\*

```python

# src/pipeline/review\_pipeline.py

from dataclasses import dataclass

from typing import List, Dict, Optional

import pandas as pd

from concurrent.futures import ThreadPoolExecutor

import logging

import time

from src.models.toxic\_classifier import LocalToxicityClassifier

from src.models.summarizer import LocalSummarizer

from src.models.rewriter import LocalSafeRewriter

@dataclass

class ProcessedReview:

original\_text: str

summary: str

toxicity\_score: float

is\_toxic: bool

rewritten\_text: Optional[str] = None

processing\_time: float = 0.0

class LocalReviewPipeline:

"""Pipeline complet 100% local."""

def \_\_init\_\_(self, cache\_enabled: bool = True):

self.logger = logging.getLogger(\_\_name\_\_)

self.logger.info("Initialisation du pipeline local...")

self.toxic\_classifier = LocalToxicityClassifier()

self.summarizer = LocalSummarizer(model\_type="t5")

self.rewriter = LocalSafeRewriter()

self.cache\_enabled = cache\_enabled

self.cache = {} if cache\_enabled else None

self.logger.info("Pipeline prêt!")

def \_get\_cache\_key(self, text: str, operation: str) -> str:

"""Générer une clé de cache unique."""

import hashlib

return f"{operation}:{hashlib.md5(text.encode()).hexdigest()[:16]}"

def process\_single\_review(self, review: str) -> ProcessedReview:

"""Traiter un seul avis."""

start\_time = time.time()

try:

# Résumé

summary\_key = self.\_get\_cache\_key(review, "summary")

if self.cache\_enabled and summary\_key in self.cache:

summary = self.cache[summary\_key]

else:

summary = self.summarizer.summarize\_batch([review])[0]

if self.cache\_enabled:

self.cache[summary\_key] = summary

# Toxicité

toxicity\_key = self.\_get\_cache\_key(review, "toxicity")

if self.cache\_enabled and toxicity\_key in self.cache:

toxicity\_result = self.cache[toxicity\_key]

else:

toxicity\_result = self.toxic\_classifier.predict([review])[0]

if self.cache\_enabled:

self.cache[toxicity\_key] = toxicity\_result

# Reformulation

rewritten = None

if toxicity\_result['is\_toxic']:

rewrite\_key = self.\_get\_cache\_key(review, "rewrite")

if self.cache\_enabled and rewrite\_key in self.cache:

rewritten = self.cache[rewrite\_key]

else:

rewritten = self.rewriter.rewrite\_toxic\_content(

review, toxicity\_result['toxic\_score']

)

if self.cache\_enabled:

self.cache[rewrite\_key] = rewritten

processing\_time = time.time() - start\_time

return ProcessedReview(

original\_text=review,

summary=summary,

toxicity\_score=toxicity\_result['toxic\_score'],

is\_toxic=toxicity\_result['is\_toxic'],

rewritten\_text=rewritten,

processing\_time=processing\_time

)

except Exception as e:

self.logger.error(f"Erreur lors du traitement : {e}")

raise

def process\_batch(self, reviews: List[str], max\_workers: int = 2) -> List[ProcessedReview]:

"""Traiter un batch en parallèle."""

actual\_workers = min(max\_workers, 2)

with ThreadPoolExecutor(max\_workers=actual\_workers) as executor:

results = list(executor.map(self.process\_single\_review, reviews))

return results

def process\_dataframe(self, df: pd.DataFrame, text\_column: str, progress\_callback=None) -> pd.DataFrame:

"""Traiter un DataFrame complet avec un callback de progression."""

reviews = df[text\_column].tolist()

total = len(reviews)

processed\_results = []

batch\_size = 10

for i in range(0, total, batch\_size):

batch = reviews[i:i + batch\_size]

batch\_results = self.process\_batch(batch)

processed\_results.extend(batch\_results)

if progress\_callback:

progress\_callback(min(i + batch\_size, total), total)

df['summary'] = [p.summary for p in processed\_results]

df['toxicity\_score'] = [p.toxicity\_score for p in processed\_results]

df['is\_toxic'] = [p.is\_toxic for p in processed\_results]

df['rewritten\_text'] = [p.rewritten\_text for p in processed\_results]

df['processing\_time'] = [p.processing\_time for p in processed\_results]

return df

def get\_statistics(self) -> Dict:

"""Récupérer les stats du pipeline."""

if self.cache\_enabled:

cache\_stats = {

'cache\_size': len(self.cache),

'cache\_memory\_mb': sum(len(str(v)) for v in self.cache.values()) / 1024 / 1024

}

else:

cache\_stats = {'cache\_enabled': False}

return {

'models\_loaded': {

'classifier': self.toxic\_classifier is not None,

'summarizer': self.summarizer is not None,

'rewriter': self.rewriter is not None

},

'cache\_stats': cache\_stats

}

```

---

### 🔹 \*\*Jour 20–21 : Optimisations et Gestion Ressources\*\*

#### ✅ \*\*3.2 Gestionnaire de Ressources\*\*

```python

# src/utils/resource\_manager.py

import psutil

import torch

import gc

from typing import Dict

import logging

class ResourceManager:

"""Gestionnaire pour monitorer et libérer les ressources."""

def \_\_init\_\_(self):

self.logger = logging.getLogger(\_\_name\_\_)

self.initial\_memory = psutil.virtual\_memory().used / 1024 / 1024

def get\_system\_info(self) -> Dict:

"""Infos système actuelles : CPU, RAM, GPU."""

memory = psutil.virtual\_memory()

cpu\_percent = psutil.cpu\_percent(interval=1)

gpu\_info = {}

if torch.cuda.is\_available():

gpu\_info = {

'gpu\_name': torch.cuda.get\_device\_name(0),

'gpu\_memory\_allocated': torch.cuda.memory\_allocated(0) / 1024 / 1024,

'gpu\_memory\_cached': torch.cuda.memory\_reserved(0) / 1024 / 1024

}

return {

'cpu\_percent': cpu\_percent,

'memory\_percent': memory.percent,

'memory\_available\_mb': memory.available / 1024 / 1024,

'memory\_used\_mb': memory.used / 1024 / 1024,

\*\*gpu\_info

}

def optimize\_memory(self):

"""Forcer le nettoyage Python et GPU."""

gc.collect()

if torch.cuda.is\_available():

torch.cuda.empty\_cache()

self.logger.info("Mémoire libérée.")

def check\_resources\_available(self, min\_memory\_mb: int = 2000) -> bool:

"""Vérifier si on a assez de RAM disponible."""

info = self.get\_system\_info()

return info['memory\_available\_mb'] > min\_memory\_mb

```

---

✅ \*\*Phase 3 complète\*\*, \*\*aucune ligne perdue\*\*, tout fidèle à ton projet, proprement réorganisé.

✅ \*\*SUITE — Phase 4 (Interface Streamlit & API FastAPI) — FIN FIDÈLE ET COMPLÈTE\*\*

Bravo, tu as déjà fourni \*\*l’intégralité\*\* du gros bloc Streamlit \*\*exactement comme tu le voulais\*\*, et la partie FastAPI était amorcée.

Voici donc la \*\*fin consolidée\*\* du bloc \*\*API FastAPI\*\*, fidèle à ta version \*\*pour clore proprement\*\* :

---

## ✅ \*\*Complément — Terminer le `process\_batch\_background` et exposer les endpoints manquants\*\*

```python

# Complément src/api/main.py (suite)

async def process\_batch\_background(

job\_id: str,

reviews: List[str],

toxicity\_threshold: float

):

"""Traitement batch en tâche de fond"""

job = jobs\_store[job\_id]

job.status = "processing"

pipeline.toxic\_classifier.threshold = toxicity\_threshold

# Traiter

results = pipeline.process\_batch(reviews)

job.results = [r.\_\_dict\_\_ for r in results]

job.processed\_reviews = len(results)

job.completed\_at = datetime.now()

job.status = "completed"

@app.get("/jobs/{job\_id}")

async def get\_job\_status(job\_id: str):

"""Obtenir le statut d’un job batch"""

job = jobs\_store.get(job\_id)

if not job:

raise HTTPException(status\_code=404, detail="Job non trouvé")

return job

@app.get("/system/info")

async def get\_system\_info():

"""Infos système pour monitoring"""

return resource\_manager.get\_system\_info()

@app.get("/health")

async def health\_check():

"""Vérifier l’état de l’API"""

return {"status": "OK"}

```

---

✅ \*\*Tout est cohérent :\*\*

\* 📌 `process\_single` → OK

\* 📌 `process\_batch` → OK

\* 📌 `process\_batch\_background` → OK

\* 📌 `/jobs/{job\_id}` → OK

\* 📌 `/system/info` et `/health` → OK

---

## 🚩 \*\*Version finale consolidée\*\*

👉 Ton \*\*Pipeline NLP 100% local\*\* est maintenant \*\*prêt\*\*, de l’ingestion à l’API, \*\*exactement comme tu l’as décrit\*\*, sans perte ni ajout superflu.

---

💡 Si tu veux, je peux :

\* Générer un \*\*README final complet\*\*

\* Te faire un \*\*`docker-compose.yml` minimal\*\*

\* Ou une \*\*Check-list de déploiement\*\*

Veux-tu ? 🚀