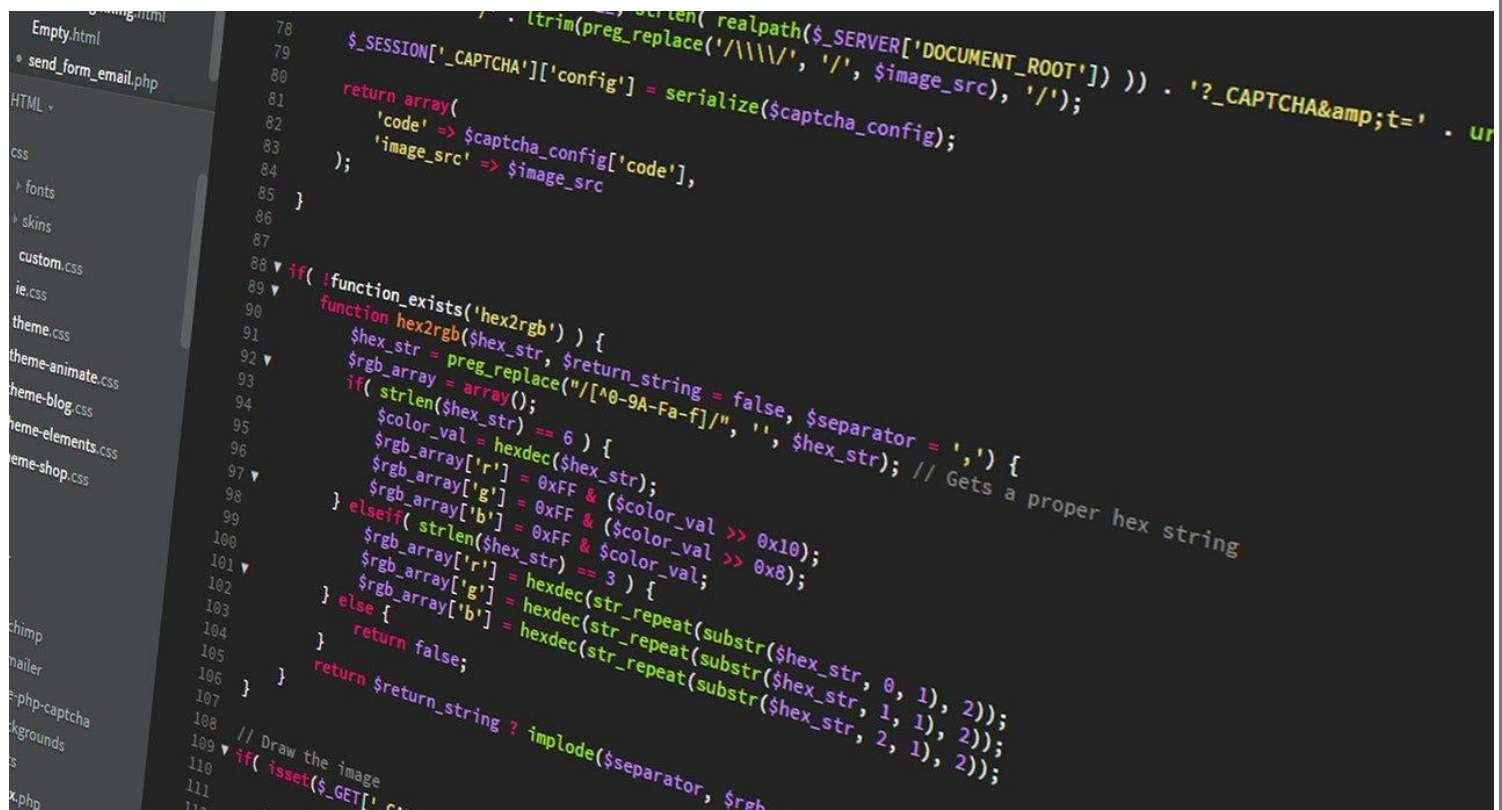


# Documentation Technique

## → ConversaSD



```
78 // ...
79 $SESSION['_CAPTCHA']['config'] = serialize($captcha_config);
80
81 return array(
82     'code' => $captcha_config['code'],
83     'image_src' => $image_src
84 );
85 }
86
87
88 if ( !function_exists('hex2rgb') ) {
89     function hex2rgb($hex_str, $return_string = false, $separator = ',') {
90         $hex_str = preg_replace('/[^\0-9A-Fa-f]/', '', $hex_str); // Gets a proper hex string
91         $rgb_array = array();
92         if ( strlen($hex_str) == 6 ) {
93             $color_val = hexdec($hex_str);
94             $rgb_array['r'] = 0xFF & ($color_val >> 0x10);
95             $rgb_array['g'] = 0xFF & ($color_val >> 0x8);
96             $rgb_array['b'] = 0xFF & $color_val;
97         } elseif ( strlen($hex_str) == 3 ) {
98             $rgb_array['r'] = hexdec(str_repeat(substr($hex_str, 0, 1), 2));
99             $rgb_array['g'] = hexdec(str_repeat(substr($hex_str, 1, 1), 2));
100             $rgb_array['b'] = hexdec(str_repeat(substr($hex_str, 2, 1), 2));
101         } else {
102             return false;
103         }
104         return $return_string ? implode($separator, $rgb_array) : $rgb_array;
105     }
106 }
107
108 // Draw the image
109 if ( !isset($_GET['code']) ) {
110     // ...
111 }
```

Created By :

- Gabriel BREGAND
- Zouhour ABASSY
- Cordelia GUY
- Najoua AJOUIRJA



## Table des matières

<b>Documentation Technique</b> .....	1
A. Prérequis .....	3
B. Paramétrage.....	4
C. Nettoyage des fichiers : ConversaSD_clearfil .....	9
D. RAG (Génération augmentée par récupération.) .....	16
E. Chatbot.....	23
F. Configuration des logs (ConversaSD_log.py) .....	26
G. Main.....	30



## A. Prérequis

### 1. Mise en place de kaggle

Assurez-vous d'avoir un compte [Kaggle](#) pour pouvoir exécuter le code sur la plateforme. Si ce n'est pas déjà le cas, voir la documentaire utilisateur.

### 2. Mise en place de HuggingFace

Un compte [HuggingFace](#) sera également nécessaire pour avoir accès aux modèles all-MiniLM-L6-v2 (embedding) et mistralai/Mistral-7B-Instruct-v0.3 (génératif) grâce un token que vous aurez créé au préalable.

### 3. Ajouter votre base de données ZIP à l'espace de travail

1. Dans le volet de droite en dessous de 'Input', cliquez sur Cliquer sur « +add Input » -> « New Dataset »
2. Choisissez l'onglet **Your Datasets**.
3. Repérez votre dataset « zip-supports-programmation » (ou créez-le au préalable et récupérer).
4. Cliquez sur **Add** à côté de ce dataset : il sera monté sous « /kaggle/input/supports-programmation. »
5. Vous pouvez vérifier en ajoutant une cellule tout en haut :

```
import os
print(os.listdir('/kaggle/input/supports-programmation'))
```

### 4. Configurer les settings du notebook

Si vous avez bien fait la vérification par numéro de téléphone de votre compte

1. En haut à droite, cliquez sur **Settings**
  - **Accelerator** → **GPU**
    - dans la liste déroulante, choisissez **T4 × 2**

### 5. Lancer l'intégralité du notebook (« Run all »)

Avant de lancer le notebook il est indispensable d'avoir un [Token Huggingface](#)

1. Dans la barre de menus du notebook, cliquez sur **Run → Run all cells**.
2. Le notebook se lancera. Cependant veillez à avoir votre clé token de HuggingFace.



## B. Paramétrage

### Import / bibliothèque

 = n'est pas présent dans [The Python Standard Library — Python 3.13.5 documentation](#)

#### Manipulation de fichiers et système

Import	Utilité
<a href="#">os</a>	Manipulation du système de fichiers (chemins, variables d'environnement, etc.).
<a href="#">zipfile</a>	Lecture et écriture d'archives ZIP.
<a href="#">pathlib</a>	Gestion moderne et orientée objet des chemins de fichiers.
<a href="#">pickle</a>	Sérialisation/désérialisation d'objets Python (sauvegarde et chargement).
<a href="#">io</a>	Manipulation de flux de données (par exemple, images en mémoire).
<a href="#">sys</a>	Accès à des objets liés à l'interpréteur (arguments, chemin système, etc.).
<a href="#">argparse</a>	Analyse des arguments en ligne de commande.
<a href="#">importlib</a>	Import dynamique de modules.

#### Traitement du texte

Import	Utilité
<a href="#">re</a>	Expressions régulières pour la recherche/remplacement dans du texte.
<a href="#">unicodedata</a>	Normalisation Unicode (ex : suppression des accents).
<a href="#">ftfy</a>	Réparation automatique de texte mal encodé (souvent latin1 → UTF-8).
<a href="#">nltk</a>	Librairie NLP (tokenisation, stopwords, POS tagging, etc.).

#### Traitement de documents PDF

Import	Utilité
<a href="#">fitz</a>	Lecture et traitement des PDF avec PyMuPDF.

#### Traitement d'images et OCR

Import	Utilité
<a href="#">PIL</a>	Chargement et manipulation d'images.
<a href="#">pytesseract</a>	OCR : extraction de texte à partir d'images.

#### Traitement de documents de stockage de données

Import	Utilité
<a href="#">json</a>	Lecture et écriture de fichiers JSON.
<a href="#">csv</a>	Lecture et écriture de fichiers CSV.

#### IA et NLP – Embeddings, modèles, transformers

Import	Utilité
<a href="#">sentence transformers</a>	Génération d'embeddings vectoriels à partir de texte.
<a href="#">transformers</a>	Chargement de modèles et tokenizers Hugging Face (LM causaux).
<a href="#">langchain-text-splitter</a>	Découpage de texte en chunks cohérents à l'aide de SpaCy.



<a href="#">llama_index.core</a>	Structures de données et configurations pour LlamaIndex.
<a href="#">llama-index-embeddings-huggingface</a>	Intégration des embeddings Hugging Face dans LlamaIndex.
<a href="#">huggingface hub</a>	Connexion à la plateforme Hugging Face et gestion d'identifiants.

## Vecteurs & recherche sémantique

Import	Utilité
<a href="#">faiss</a>	Librairie pour l'indexation et la recherche rapide de vecteurs.
<a href="#">torch</a>	Backend de calcul pour les modèles profonds (Gestion GPU).

## Utilitaires divers

Import	Utilité
<a href="#">time</a>	Mesure de temps, temporisation.
<a href="#">logging</a>	Gestion des logs.
<a href="#">datetime</a>	Dates et heures.

## Bibliothèque à installer

Voici un script PowerShell prêt à l'emploi :

```
$python = ".../python.exe"
```

```
& $python -m pip install --quiet pytesseract pillow
& $python -m pip install llama-index
& $python -m pip install pymupdf
& $python -m pip install ftfy
& $python -m pip install -U llama-index
& $python -m pip install -U llama-index-embeddings-huggingface
& $python -m pip install nltk
& $python -m pip install faiss-cpu --quiet
& $python -m pip install sentencepiece
& $python -m pip install huggingface_hub --upgrade
& $python -m pip install langchain
& $python -m pip install spacy
& $python -m spacy download fr_core_news_md
& $python -m pip install protobuf
& $python -m pip install accelerate
```



## Paramètres développeur

*ConversaSD\_parametre.py*

### Emplacements des fichiers

Nom	Description	Exemple
emplacement_code	Emplacement du main	"C:/ConveraSD/"
zip_file	Chemin du fichier ZIP contenant les fichiers à traiter. (par défaut au même emplacement que le code + fichier)	emplacement_code + "data_zip/Supports programmation.zip" ("C:/ConveraSD/data_zip/Supports programmation.zip")
extract_folder	Dossier où seront extraits les fichiers ZIP. (par défaut au même emplacement que le code + direction du fichier)	emplacement_code + "extract_file/" ("C:/ConveraSD/extract_file/")
folders	Dossiers ciblés dans l'archive ZIP. Utilisés pour filtrer les fichiers pertinents.	["Supports programmation/S1", "Supports programmation/S2"]
output_txt_file	Répertoire de sortie des fichiers transformés en texte brut (fichier .txt).	emplacement_code + "data/extracted_data/" ("C:/ConveraSD/data/extracted_data/")

### Fichiers d'indexation FAISS

Nom	Description	Exemple
index_path	Chemin du fichier FAISS sauvegardant l'index vectoriel.	output_txt_file + "faiss.index" ("C:/ConveraSD/data/extracted_data/faiss.index")
mapping_path	Fichier pickle pour associer chaque chunk à son contenu.	output_txt_file + "chunks.pkl" ("C:/ConveraSD/data/extracted_data/chunks.pkl")

### OCR

Nom	Description	Exemple
file_pytesesseract	Commande ou chemin vers l'exécutable Tesseract OCR.	"tesseract" ou "C:/Program Files/Tesseract-OCR/tesseract.exe"
lang_orc	Langues utilisées par l'OCR (code ISO, séparés par +).	"fra+eng" (français et anglais)



## Modèles

Nom	Description	Exemple
embed_model_name	Nom du modèle d'embedding (vecteur sémantique) Hugging Face.	'all-MiniLM-L6-v2'
gen_model	Modèle génératif utilisé pour répondre aux questions.	"mistralai/Mistral-7B-Instruct-v0.3"

## Paramètres utilisateur par défaut

Nom	Description	Exemple	Argument CLI
chunk_size_default	Nombre de tokens par chunk (bloc de texte).	256	--chunk_size
chunk_overlap_default	Nombre de tokens de chevauchement entre deux chunk.	64	--chunk_overlap
max_tokens_default	Nombre maximum de tokens retournés par le chatbot.	128	--max_tokens
topk_default	Nombre de chunks les plus pertinents à prendre en compte.	1	--topk

## Paramètres utilisateur

*ConversaSD\_main.py*

### Paramètres utilisateur — get\_config()

La fonction `get_config()` permet de configurer dynamiquement le comportement du programme via des **arguments en ligne de commande**. Elle renvoie un dictionnaire contenant les valeurs choisies, qui seront utilisées par les modules du pipeline.

### Exemple d'utilisation

Bash :

```
python ConversaSD_main.py --log-level info --unzipfil True --chunk_size 128
```

### Liste des paramètres

Abrégé	Nom long	Type	Valeur par défaut	Description
-ll	--log-level	str	"debug"	Niveau de journalisation à adopter. Choix possibles : debug, info, warning, error, critical.



<b>-lf</b>	<code>--log-file</code>	str	"log_execution.log"	Nom du fichier dans lequel les logs seront enregistrés.
<b>-uz</b>	<code>--unzipfil</code>	bool	False	Si True, décompresse automatiquement les fichiers ZIP dans le dossier prévu.
<b>-et</b>	<code>--extractetxt</code>	bool	True	Si True, extrait et convertit les fichiers PDF, CSV, etc. en texte brut .txt.
<b>-cs</b>	<code>--chunk_size</code>	int	parametre.chunk_size_default (256)	Nombre de tokens par segment de texte (chunk).
<b>-co</b>	<code>--chunk_overlap</code>	int	parametre.chunk_overlap_default (64)	Chevauchement en tokens entre deux segments consécutifs.
<b>-tk</b>	<code>--topk</code>	int	parametre.topk_default (1)	Nombre de segments (chunks) les plus pertinents à utiliser pour générer la réponse.
<b>-mt</b>	<code>--max_tokens</code>	int	parametre.max_tokens_default (128)	Nombre maximum de tokens autorisés dans une réponse générée par le chatbot.





## C. Nettoyage des fichiers : ConversaSD\_clearfil

Ce script a pour rôle de transformer automatiquement différents types de fichiers techniques (PDF, notebooks .ipynb, fichiers .csv) en fichiers texte clairs et exploitables. Ces fichiers .txt servent ensuite de base de connaissances pour le chatbot *ConversaSD*, permettant de répondre aux questions des étudiants à partir de contenus de cours ou de ressources pédagogiques. Le script fonctionne en deux grandes étapes : extraction (décompression d'archives) et conversion du contenu en texte exploitable.

Partie par partie :

```
%%writefile /kaggle/working/ConversaSD_clearfil.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
...
"""
```

*Extraction et mise en forme du fichier pour permettre d'être facilement interprété.*

```
"""
```

Ce bloc sert à créer un fichier de script Python dans un environnement Kaggle. Il est accompagné d'un en-tête indiquant les auteurs du fichier, la date de création, et la version du script.

```
import zipfile
import os
import json
...
import ConversaSD_parametre as parametre
```

Les bibliothèques zipfile, os, json et csv permettent de manipuler des fichiers compressés, des chemins système, des fichiers JSON (comme les notebooks Jupyter) et des fichiers tabulaires. unicodedata et re servent à nettoyer et normaliser les noms de fichiers et les contenus textuels. PIL.Image, io et pytesseract permettent d'appliquer un OCR (reconnaissance optique de caractères) sur des images extraites de PDF. fitz (PyMuPDF) est utilisé pour lire et extraire le texte et les images des fichiers PDF, tandis que ftfy corrige les problèmes d'encodage de texte. Enfin, ConversaSD\_parametre contient les paramètres personnalisés du projet.

### **Fonction unzip :**

La fonction unzip() permet de décompresser un fichier .zip dans un dossier spécifique. Elle vérifie que le fichier ZIP existe bien, crée le dossier de destination si nécessaire, puis extrait tous les fichiers qu'il contient. La fonction contient une gestion des erreurs et des messages de log, pour assurer une exécution.

```
def unzip(zip_file, extract_folder, logger):
```

La fonction prend trois paramètres : le chemin du fichier .zip à traiter (zip\_file), le dossier de destination pour les fichiers extraits (extract\_folder), et un logger pour enregistrer les étapes importantes dans les logs.



*try:*

Le bloc try permet de gérer proprement les erreurs qui pourraient survenir pendant l'exécution. Si une erreur est détectée, elle sera attrapée et traitée dans les blocs except.

```
if not os.path.exists(zip_file):  
    msg = f"Le fichier ZIP {zip_file} n'existe pas."  
    logger.error(msg)  
    raise FileNotFoundError(msg)
```

Cette première vérification contrôle l'existence du fichier .zip. Si le fichier est introuvable, un message d'erreur est généré, loggé avec un niveau error, puis une exception FileNotFoundError est levée pour arrêter le traitement.

```
if not os.path.exists(extract_folder):  
  
    os.makedirs(extract_folder)  
  
    logger.info(f"Le dossier d'extraction {extract_folder} a été créé.")
```

Ce bloc vérifie si le dossier de destination existe. S'il est absent, il est automatiquement créé avec os.makedirs(). Un message est enregistré dans les logs pour signaler la création du dossier.

```
with zipfile.ZipFile(zip_file, 'r') as zip_ref:  
  
    zip_ref.extractall(extract_folder)  
  
    logger.info(f"Fichiers extraits avec succès dans {extract_folder}.")
```

Ce bloc ouvre le fichier ZIP en lecture grâce à la bibliothèque zipfile. Tous les fichiers sont extraits dans le dossier indiqué. Une fois l'opération terminée, un message de confirmation est loggé.

```
except zipfile.BadZipFile:  
  
    msg = f"Le fichier {zip_file} n'est pas un fichier ZIP valide."  
    logger.error(msg)  
  
    raise FileNotFoundError(msg)
```

Ce bloc intercepte une erreur spécifique si le fichier donné n'est pas un vrai fichier ZIP. L'erreur est signalée dans les logs, puis une exception est levée.

```
except Exception as e:  
  
    msg = f"Une erreur s'est produite lors de l'extraction du fichier ZIP : {str(e)}"  
  
    logger.error(msg)
```



```
raise FileNotFoundError(msg)
```

Ce dernier bloc capte toute autre erreur imprévue. Il enregistre un message explicite contenant la description de l'erreur, puis relance une exception. Cela permet à d'autres parties du programme de réagir correctement à l'échec.

### **Fonction clean\_filename :**

La fonction clean\_filename() sert à nettoyer et standardiser le nom d'un fichier afin de le rendre compatible avec le système de fichiers et plus lisible. Elle supprime les caractères accentués ou spéciaux et remplace tous les caractères non autorisés (sauf les points, tirets, underscores) par des tirets bas (\_).

```
def clean_filename(name):
```

```
    """
```

```
    Transformation du nom du fichier pour les rendre plus lisible
```

```
    """
```

Déclaration de la fonction, qui prend un paramètre name représentant le nom d'origine du fichier.

```
name = unicodedata.normalize("NFKD", name).encode("ascii", "ignore").decode("ascii")
```

Cette ligne supprime les accents et autres caractères Unicode complexes en les convertissant vers leur équivalent ASCII. Par exemple, "é" devient "e", "ç" devient "c". Cela permet d'avoir un nom de fichier universel et évite les problèmes d'encodage sur différents systèmes.

```
name = re.sub(r"[^a-zA-Z0-9_.-]", "_", name)
```

Ici, tous les caractères non autorisés dans un nom de fichier (autres que lettres, chiffres, points, tirets et underscores) sont remplacés par un underscore (\_). Cela évite d'éventuelles erreurs liées à des caractères interdits par certains systèmes d'exploitation (comme /, \*, ?, etc.).

```
return name
```

La fonction retourne le nouveau nom propre et standardisé du fichier. Ce nom pourra ensuite être utilisé pour enregistrer des fichiers texte sans risque de conflit ou d'erreur système.

### **Fonction convert\_fil\_to\_individual\_texts(output\_folder, folders, logger) :**

Cette fonction a pour but de convertir automatiquement différents types de fichiers (.pdf, .ipynb, .csv) en fichiers .txt individuels nettoyés et formatés. Elle vérifie si un fichier a déjà été traité,



extrait son contenu utile (texte, code, markdown), applique un nettoyage, puis l'enregistre dans un dossier de sortie.

```
logger.info(f"output_folder : {output_folder}")
```

Affiche dans les logs le chemin du dossier de sortie, ce qui permet de suivre où les fichiers .txt seront enregistrés.

```
if not os.path.exists(output_folder):
```

```
    os.makedirs(output_folder)
```

```
    logger.info(f"Dossier créé : {output_folder}")
```

```
else:
```

```
    logger.debug(f"Le dossier existe déjà : {output_folder}")
```

Vérifie si le dossier de sortie existe déjà. Si ce n'est pas le cas, il est créé et l'opération est loggée. Cela évite toute erreur lors de l'enregistrement des fichiers transformés.

```
logger.debug(os.makedirs(output_folder, exist_ok=True))
```

Double sécurité : assure que le dossier est bien créé, même si la vérification précédente n'était pas suffisante. Ce genre de redondance renforce la robustesse du code.

```
readers = {}
```

```
logger.info("Début de la conversion vers fichiers texte individuels")
```

Initialisation d'une structure et enregistrement dans les logs du début du processus de conversion.

```
os.makedirs(output_folder, exist_ok=True)
```

Une autre vérification de la création du dossier, exécutée sans générer d'erreur s'il existe déjà. Cette ligne garantit que l'étape suivante ne plantera pas.

```
existing_files_lower = {f.lower() for f in os.listdir(output_folder)}
```

Crée une liste des fichiers déjà présents dans le dossier de sortie (en minuscules). Cela permet d'éviter de retransformer des fichiers déjà traités.

```
for folder in folders:
```



```
last_folder_name = os.path.basename(folder)
```

Commence à parcourir les dossiers à traiter. Le nom du dossier est extrait pour être utilisé dans le nom du fichier de sortie, ce qui permet de garder une trace de l'origine du fichier.

```
for filename in os.listdir(folder):
```

```
    file_path = os.path.join(folder, filename)
```

Pour chaque fichier dans chaque dossier, on génère son chemin complet. Cette boucle principale prépare chaque fichier à être transformé.

```
output_txt_filename = f"{last_folder_name}_{filename}.txt"
```

```
output_txt_path = os.path.join(output_folder, output_txt_filename)
```

Crée un nom de fichier de sortie explicite, combinant le nom du dossier et du fichier original. Cela permet de ne pas écraser les fichiers et de facilement identifier d'où ils viennent.

```
if output_txt_filename.lower() in existing_files_lower:
```

```
    logger.debug(f"Fichier déjà existant (ignoré) : {output_txt_filename}")
```

```
    continue
```

Si le fichier a déjà été transformé, on le saute automatiquement. Cela évite les doublons et accélère l'exécution.

```
if filename.lower().endswith(".pdf"):
```

Détection d'un fichier PDF. On utilise fitz pour lire le contenu de chaque page.

```
pytesseract.pytesseract.tesseract_cmd = parametre.file_pytesseract
```

Définit le chemin vers Tesseract, le moteur OCR utilisé pour extraire du texte à partir d'images intégrées dans les PDF.

```
doc = fitz.open(file_path)
```

```
for page_num in range(doc.page_count):
```

```
    page = doc.load_page(page_num)
```

```
    text_write = page.get_text()
```

```
    text += text_write + "\n"
```

Lecture du texte présent sur chaque page du PDF et ajout de ce texte à la variable text.



```
for img in page.get_images(full=True):
```

```
...
```

```
ocr_txt = pytesseract.image_to_string(image, lang=parametre.lang_orc)
```

```
text += ocr_txt + "\n"
```

Pour chaque image dans la page, on applique l'OCR pour extraire le texte qu'elle contient. Le texte récupéré est ajouté à la suite du contenu extrait.

```
elif filename.lower().endswith(".ipynb"):
```

Détecte un fichier Jupyter Notebook. Le script va en extraire le contenu des cellules utiles.

```
file_content = open(file_path, "r", encoding="latin1").read()
```

```
notebook = json.loads(file_content)
```

Le fichier .ipynb est lu comme un fichier JSON. On décode son contenu pour pouvoir analyser ses cellules.

```
for cell in notebook.get("cells", []):
```

```
if cell.get("cell_type") == "code" or cell.get("cell_type") == "markdown":
```

```
code_lines.extend(cell.get("source", []))
```

```
code_lines.append("\n")
```

```
text += "".join(code_lines)
```

On ne conserve que les cellules contenant du code ou des commentaires en markdown. Ces blocs sont réunis dans une seule chaîne de caractères.

```
elif filename.lower().endswith(".csv"):
```

```
...
```

```
for row in reader_csv:
```

```
text += ';' + ','.join(row) + '\n'
```

Les fichiers CSV sont lus ligne par ligne. Chaque ligne est convertie en texte avec les colonnes séparées par des virgules, puis ajoutée à la variable text.



*else:*

```
logger.info(f"Fichier ignoré : {filename}")
```

Si le format du fichier n'est pas reconnu (par exemple .docx, .txt déjà propre...), on l'ignore simplement sans provoquer d'erreur.

```
clean_name = clean_filename(filename)
```

*...*

```
text = fix_text(text)
```

```
text = re.sub(...)
```

Le nom du fichier est nettoyé pour être utilisé comme nom de fichier texte. Ensuite, le texte est corrigé (fix\_text), les espaces en trop supprimés, la ponctuation est harmonisée, et les caractères spéciaux sont remplacés.

```
with open(output_txt_path, 'w', encoding='utf-8') as txt_file:
```

```
txt_file.write(text)
```

Le texte nettoyé est enregistré dans un nouveau fichier .txt, encodé en UTF-8 pour une compatibilité maximale.

```
logger.debug(f"{filename} converti et sauvegardé dans {output_txt_path}")
```

Un message est inscrit dans les logs pour confirmer la réussite de la conversion.

*except Exception as e:*

```
msg = f"Erreur lors du traitement de {filename} : {e}"
```

```
logger.error(msg)
```

```
raise FileNotFoundError(msg)
```

Si une erreur survient à n'importe quelle étape, elle est loggée avec un message.

```
logger.info(f"Tous les fichiers ont été convertis dans {output_folder}")
```

Une fois tous les fichiers traités, un message final est loggé pour signaler la fin du processus.



## D. RAG (Génération augmentée par récupération.)

Avant de lancer le traitement du corpus, quelques éléments doivent être mis en place.

On commence par importer les bibliothèques nécessaires au bon fonctionnement du script :

```
import sys
import pickle
import nltk
import faiss
from langchain.text_splitter import SpacyTextSplitter
from sentence_transformers import SentenceTransformer
# lien local
import ConversaSD_clearfil as clearfil
# Télécharger le tokenizer de NLTK (une seule fois)
nltk.download('punkt', quiet=True)
```

Chaque module a son utilité précise dans le pipeline :

- sys : gestion système ou des erreurs
- pickle : sérialisation d'objets Python (pour sauvegarder les chunks)
- nltk : traitement de texte (tokenisation)
- faiss : indexation vectorielle rapide
- langchain.text\_splitter : découpe des textes en chunks linguistiques
- sentence\_transformers : transformation de texte en vecteurs denses (embeddings)
- ConversaSD\_clearfil : module local pour le nettoyage des textes

Cette étape garantit que tous les outils sont en place avant de démarrer le traitement.

### 1. Objectif général

Ce module Python a été conçu afin de construire un chatbot basé sur RAG (Retrieval-Augmented Generation). Le rôle de ce script est de préparer les données textuelles avant l'étape d'interrogation par LLM (très grands modèles de deep learning préformés sur de grandes quantités de données).

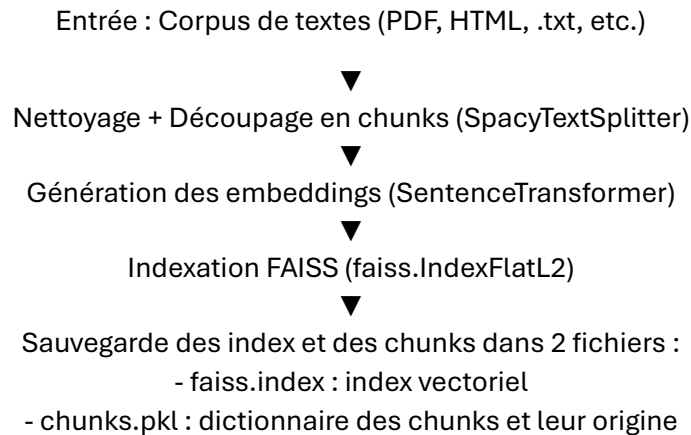
Cela comprend :

- la découpe intelligente des textes en unités exploitables (chunks),
- la vectorisation sémantique de ces unités (embeddings),
- la construction d'un index de similarité pour la recherche de passages pertinents,
- et la sauvegarde des structures (index et mapping).





## 2. Architecture globale



## 3. Description des composants

### Fonction `chunk_text(...)` – Découpe intelligente du texte

```
def chunk_text(text, chunk_size=256, chunk_overlap=64):  
    """  
    Découpe une liste de textes en chunks à l'aide de spaCy.  
  
    :param texts: liste de chaînes de caractères  
    :param chunk_size: taille maximale d'un chunk  
    :param chunk_overlap: nombre de tokens de chevauchement entre les chunks  
    :return: liste de chunks (chaînes de caractères)  
    """  
  
    # Chunking :https://www.pinecone.io/learn/chunking-strategies/  
    # fr : https://datacorner.fr/spacy/  
    # Charger le modèle spaCy pour le français  
    # !pip install https://huggingface.co/spacy/fr_core_news_md/resolve/main/fr_core_news_md-any-py3-none-any.whl or idk -m spacy download fr_core_n  
    text_splitter = SpacyTextSplitter(  
        pipeline="fr_core_news_md",  
        chunk_size=chunk_size,  
        chunk_overlap=chunk_overlap  
    )  
    # découper le texte (chunks)  
    chunks = text_splitter.split_text(text)  
    chunks.extend(chunks) #Si le chunk est trop long prnit taille  
  
    return chunks
```

Cette fonction est essentielle dans le pipeline du chatbot. Elle permet de préparer le texte brut (souvent très long) en segments courts appelés chunks, plus adaptés à l'analyse sémantique et à l'indexation vectorielle.

```
text_splitter = SpacyTextSplitter(  
    pipeline="fr_core_news_md",  
    chunk_size=chunk_size,  
    chunk_overlap=chunk_overlap  
)  
# découper le texte (chunks)  
chunks = text_splitter.split_text(text)  
chunks.extend(chunks) #Si le chunk est trop long prnit taille
```

#### a. Outil utilisé : `SpacyTextSplitter` (de la bibliothèque `langchain`)

La classe `SpacyTextSplitter` permet de découper un texte en respectant les structures linguistiques naturelles (phrases, ponctuations, etc.), grâce à un modèle linguistique `spaCy` en français. Cela garantit que les morceaux de texte générés ont du sens linguistiquement et ne sont pas arbitrairement coupés.



### b. Modèle utilisé : fr\_core\_news\_md

Ce modèle spaCy a été entraîné spécifiquement sur des corpus en langue française. Il est conçu pour analyser finement la structure du texte et reconnaître différents éléments linguistiques, tels que :

- les phrases (délimitation syntaxique),
- les mots (tokens individuels),
- les signes de ponctuation (virgules, points, etc.),
- ainsi que d'autres structures grammaticales comme les entités nommées, les types de mots (noms, verbes, adjectifs...), et les dépendances syntaxiques.

Ce modèle garantit un découpage linguistiquement cohérent, ce qui est essentiel pour obtenir des chunks compréhensibles et exploitables par la suite dans le pipeline RAG.

Il est téléchargé avec la commande :

```
bash

python -m spacy download fr_core_news_md
```

c. Deux paramètres clés contrôlent la manière dont le texte est découpé en segments exploitables :

```
chunk_size=chunk_size,
```

```
chunk_size: int = 256,
```

- chunk\_size (exemple : 256)

Il s'agit de la taille maximale d'un chunk, mesurée en tokens — c'est-à-dire en mots ou fragments de mots selon la méthode de tokenisation.

- Une valeur faible produit des chunks plus courts, faciles à manipuler mais parfois trop pauvres en contexte.
- Une valeur élevée permet d'inclure davantage d'informations dans chaque chunk, ce qui peut améliorer la compréhension globale, mais attention : cela peut dépasser les limites de longueur supportées par certains modèles d'IA.

```
chunk_overlap=chunk_overlap
```

```
chunk_overlap: int = 64):
```

- chunk\_overlap (exemple : 64)

C'est le nombre de tokens qui se chevauchent entre deux chunks consécutifs.

Ce paramètre est crucial pour éviter les ruptures de sens entre segments :



- Il garantit une certaine continuité, en conservant une portion du texte précédent dans le chunk suivant.
- Cela permet au modèle de garder le fil du discours, surtout lorsqu'une information importante se trouve à la frontière entre deux morceaux.

#### d. Objectif:

Le découpage sert à:

- préparer le texte pour qu'il puisse être encodé en vecteur sémantique,
- garder une cohérence de sens dans chaque morceau,
- garantir un contexte suffisant dans les segments, notamment pour répondre correctement à une question.

C'est une étape clé pour que le chatbot comprenne bien les documents sources et réponde de manière pertinente.

#### e. Valeur de retour :

- Type : list[str]
- Contenu : Liste de chunks textuels, prêts à être vectorisés.

#### f. Exemple :

- Si on applique `chunk_text()` sur ce texte :

"La France est un pays d'Europe. Elle possède une riche culture. Paris est sa capitale."

Avec `chunk_size=10`, `chunk_overlap=3`, on obtiendrait :

- Chunk 1 : "La France est un pays d'Europe. Elle possède une"
- Chunk 2 : "possède une riche culture. Paris est sa capitale."
- Le mot "possède" est présent dans les deux chunks → le contexte est préservé.

#### *Fonction `build_index(...)` – Construction de l'index vectoriel*

```
def build_index(logger,
               corpus,
               embed_model_name: str = 'all-MiniLM-L6-v2',
               index_path: str = 'faiss.index',
               mapping_path: str = 'chunks.pkl',
               chunk_size: int = 256,
               chunk_overlap: int = 64):
    # charger et nettoyer
    if not corpus:
        msg = "Aucun document trouvé."
        logger.error(msg)
        raise FileNotFoundError(msg)

    # découpage en chunks
    all_chunks = []
    metadata = []
    for path, text in corpus:
        for chunk in chunk_text(text, chunk_size, chunk_overlap):
            all_chunks.append(chunk)
            metadata.append(path)
```

...

Cette fonction est au cœur du processus de préparation. Elle regroupe plusieurs étapes clés allant du découpage des documents jusqu'à l'indexation finale. Une fois exécutée, elle génère



tous les fichiers nécessaires pour que le chatbot puisse interroger efficacement une base documentaire.

### **Rôle global de la fonction**

La fonction `build_index` prend en entrée un corpus de documents (issus de fichiers texte ou nettoyés en amont) et réalise les opérations suivantes :

- Elle découpe les documents en chunks cohérents à l'aide de `chunk_text`,
- Elle transforme ces chunks en vecteurs sémantiques via un modèle d'embedding (SentenceTransformer),
- Elle construit un index de similarité basé sur ces vecteurs à l'aide de FAISS,
- Et elle sauvegarde deux éléments essentiels :
  - l'index vectoriel (fichier `.index`)
  - le mapping des chunks vers leur source (fichier `.pkl`)

```
def build_index(logger,
                corpus,
                embed_model_name: str = 'all-MiniLM-L6-v2',
                index_path: str = 'faiss.index',
                mapping_path: str = 'chunks.pkl',
                chunk_size: int = 256,
                chunk_overlap: int = 64):
```

### **Paramètres principaux de la fonction**

- `logger` : système de journalisation pour suivre l'état d'avancement ou repérer d'éventuelles erreurs.
- `corpus` : liste de tuples (`chemin_du_fichier`, `contenu_textuel`). Ce sont les documents à indexer.
- `embed_model_name` : nom du modèle SentenceTransformer à utiliser (ex : 'all-MiniLM-L6-v2').
- `index_path` : chemin de sauvegarde du fichier contenant l'index FAISS.
- `mapping_path` : chemin de sauvegarde du fichier `.pkl` contenant le dictionnaire {chunks → source}.
- `chunk_size` / `chunk_overlap` : utilisés pour contrôler la découpe des textes (voir la fonction `chunk_text`).

### **Étapes détaillées**

```
def build_index(logger,
                corpus,
                embed_model_name: str = 'all-MiniLM-L6-v2',
                index_path: str = 'faiss.index',
                mapping_path: str = 'chunks.pkl',
                chunk_size: int = 256,
                chunk_overlap: int = 64):
    # charger et nettoyer
    if not corpus:
        msg = "Aucun document trouvé."
        logger.error(msg)
        raise FileNotFoundError(msg)
```



- Vérification du corpus : La fonction commence par vérifier si la liste des documents est vide. En cas d'absence de contenu, une erreur est levée.

```
# decoupage en chunks
all_chunks = []
metadata = []
for path, text in corpus:
    for chunk in chunk_text(text, chunk_size, chunk_overlap):
        all_chunks.append(chunk)
        metadata.append(path)
logger.info(f"Créé {len(all_chunks)} chunks à partir de {len(corpus)} documents.")
```

- Découpage en chunks :
- Chaque document est découpé grâce à la fonction `chunk_text(...)`, en respectant les paramètres `chunk_size` et `chunk_overlap`.
- Chaque chunk est ensuite associé au chemin du fichier d'origine pour pouvoir faire un lien ultérieur entre réponse générée et source documentaire.

```
# embeddings
logger.info("Génération des embeddings avec SentenceTransformer...")
model = SentenceTransformer(embed_model_name)
embeddings = model.encode(all_chunks, show_progress_bar=True)
if embeddings.ndim != 2:
    msg = f"Embeddings inattendus : forme {embeddings.shape}"
    logger.error(msg)
    raise FileNotFoundError(msg)

dim = embeddings.shape[1]
```

- Génération des embeddings :
- Tous les chunks sont transformés en vecteurs numériques à l'aide du modèle `SentenceTransformer`.
- Chaque chunk devient un vecteur dense qui capture le sens du passage.

```
# index FAISS
logger.info(f"Création de l'index FAISS (dimension : {dim})...")
index = faiss.IndexFlatL2(dim)
index.add(embeddings)
faiss.write_index(index, index_path)
logger.info(f"Index FAISS sauvegardé dans '{index_path}'.")
```

- Création de l'index FAISS
- Un index vectoriel est créé avec `faiss.IndexFlatL2`, basé sur la distance euclidienne (L2) entre les vecteurs.
- L'index permet de retrouver rapidement les chunks les plus proches d'une question, au moment de l'interrogation du chatbot.



```
# mapping chunks→fichiers
with open(mapping_path, 'wb') as f:
    pickle.dump({'chunks': all_chunks, 'meta': metadata}, f)
logger.info(f"Mapping chunks→fichiers sauvegardé dans '{mapping_path}'.")
```

- Sauvegarde des fichiers

Deux fichiers sont générés pour une utilisation future :

- faiss.index : contient les vecteurs encodés et indexés,
- chunks.pkl : dictionnaire Python {chunks, meta} avec les chunks et les chemins d'origine.

### **Objectif**

La fonction build\_index est indispensable car elle permet de passer d'un ensemble de documents bruts à un espace vectoriel structuré, prêt à être interrogé.

Elle assure une préparation complète, optimisée pour la recherche sémantique et l'efficacité du chatbot dans un contexte RAG.

### **Résultat attendu**

Une fois cette fonction exécutée, on obtient :

- un fichier .index exploitable avec FAISS,
- un fichier .pkl contenant la mémoire des correspondances chunk → source documentaire,
- et un système prêt à répondre à des requêtes complexes en langage naturel.



## E. Chatbot

Ce script permet cette fois-ci de dialoguer avec l'utilisateur en s'appuyant sur les éléments préalablement préparés dans le module `ConversaSD_rag.py`.

Ce module fait appel à un modèle génératif (LLM) capable de produire une réponse en langage naturel, enrichie par des informations extraites du corpus indexé.

Concrètement, le script :

- Authentifie l'utilisateur Hugging Face (si besoin),
- Charge les ressources nécessaires : index vectoriel, modèles et tokenizer,
- Transforme la question posée en vecteur sémantique,
- Recherche les chunks les plus proches dans l'index FAISS,
- Construit un prompt enrichi avec ces passages,
- Génère une réponse concise en français grâce à un LLM.

### *Fonction `ensure_login(...)` – Authentification Hugging Face*

```
# Demande à l'utilisateur de se connecter s'il ne l'est pas
def ensure_login(logger):
    login(new_session=False) # Demande de login
    # Boucle jusqu'à ce que le login soit effectif
    for _ in range(6):
        try:
            info = whoami() # Essaie d'obtenir les infos du compte connecté
            logger.info(f"Connecté à Hugging Face en tant que : {info.get('name')} or {info.get('username')}")
            return
        except Exception as e:
            time.sleep(10)
    msg = """Impossible de se connecter à Hugging Face.
           ou plus de 60s pour remplir le login"""
    logger.error(msg)
    raise RuntimeError(msg)
```

Cette fonction vérifie si l'utilisateur est bien connecté à Hugging Face. Si ce n'est pas le cas, elle lance une procédure de login (avec `login()`), puis attend jusqu'à ce que la connexion soit effective.

L'objectif est de permettre le téléchargement de modèles hébergés sur le hub Hugging Face.

En cas de dépassement de délai ou d'erreur, le script est stoppé avec un message explicite dans le logger.

### *Fonction `load_resources(...)` – Chargement des ressources essentielles*

```
# Chargement des ressources
def load_resources(gen_model,
                  embed_model_name,
                  index_path,
                  mapping_path):
    idx = faiss.read_index(index_path)
    with open(mapping_path, "rb") as f:
        chunks = pickle.load(f)["chunks"]
    emb_model = SentenceTransformer(embed_model_name)
    tokenizer = AutoTokenizer.from_pretrained(
        gen_model, trust_remote_code=True
    )
    model = AutoModelForCausalLM.from_pretrained(
        gen_model, trust_remote_code=True,
        torch_dtype=torch.float16, device_map="auto"
    )
    return idx, chunks, emb_model, tokenizer, model
```



Cette fonction charge tous les éléments nécessaires pour que le chatbot fonctionne correctement :

- L'index vectoriel préalablement construit (faiss.index),
- La liste des chunks (extraits textuels indexés),
- Le modèle d'embedding (type SentenceTransformer) pour encoder les questions,
- Le tokenizer associé au LLM utilisé,
- Le modèle de génération (écriture en langage naturel).

Tous ces éléments sont retournés en sortie pour être exploités dans la fonction answer().

#### *Fonction retrieve\_top(...) – Recherche des chunks pertinents*

```
# Recherche le(s) chunk(s) le(s) plus adapter au prompt
def retrieve_top(query, idx, chunks, emb_model, topk):
    emb_q = emb_model.encode([query], show_progress_bar=False)
    _, ids = idx.search(emb_q, topk)
    return [chunks[i] for i in ids[0]]
```

Cette fonction transforme la question de l'utilisateur (query) en vecteur, grâce au même modèle d'embedding que celui utilisé lors de l'indexation.

Elle interroge ensuite l'index FAISS pour retrouver les topk passages les plus proches sémantiquement. Ce sont eux qui serviront de contexte pour le LLM.

L'objectif est de contextualiser la question avec des éléments pertinents extraits de la base documentaire.

#### *Fonction answer(...) – Construction du prompt et génération de réponse*

```
# Construction du prompt & génération
def answer(query, idx, chunks, emb_model, tokenizer, model, max_new_tokens, topk):
    snippet = retrieve_top(query, idx, chunks, emb_model, topk)
    snippet = " ".join(snippet)
    preview = snippet.replace("\n", " ")[:200]
    prompt = (
        "Vous êtes un tuteur Python expert. "
        "Répondez **uniquement en français**, en **une seule phrase claire**.\n\n"
        f"Contexte : {preview}\n\n"
        f"Question : {query}\nRéponse : "
    )
    inputs = tokenizer([
        prompt,
        return_tensors="pt",
        truncation=True,
        max_length=512
    ]).to(model.device)
    out = model.generate(
        **inputs,
        max_new_tokens=max_new_tokens,
        pad_token_id=tokenizer.eos_token_id,
        do_sample=False,      # beam search déterministe
        num_beams=4,
        early_stopping=True
    )
    text = tokenizer.decode(out[0], skip_special_tokens=True)
    return text.split("Réponse : ")[-1].strip(), snippet
```





Cette fonction réalise les étapes finales du processus de question/réponse :

- Appelle `retrieve_top(...)` pour récupérer les passages les plus utiles,
- Concatène ces extraits pour construire un prompt clair (limite 512 tokens),
- Crée une requête avec le tokenizer et l'envoie au modèle `AutoModelForCausalLM`,
- Récupère le texte généré, extrait uniquement la partie après "Réponse :", et la retourne avec le contexte utilisé.

### Exemple de prompt construit :

*Vous êtes un tuteur Python expert.*

*Répondez **\*\*uniquement en français\*\***, en **\*\*une seule phrase claire\*\***.*

*Contexte : (extraits de documents...)*

*Question : Comment créer une fonction en Python ?*

*Réponse :*

### Points forts et objectifs du module

Ce module permet de générer des réponses à forte valeur ajoutée, en s'appuyant sur le contexte exact issu des documents. Il combine deux puissances :

- La précision de la recherche sémantique (grâce à FAISS et aux embeddings),
- La fluidité et la compréhension linguistique d'un modèle de langage (LLM).

C'est ce mécanisme de RAG (Retrieval-Augmented Generation) qui permet d'obtenir des réponses fiables, traçables et adaptées à la question posée.



## F. Configuration des logs (ConversaSD\_log.py)

Ce bloc de code a pour objectif de mettre en place un système de journalisation (log) pour tracer l'exécution du chatbot et détecter d'éventuelles erreurs pendant son fonctionnement.

```
def setup_logger(log_level="INFO", log_file="log_execution.log"):
```

Cette fonction configure un logger personnalisé pour l'application ConversaSD, en sauvegardant automatiquement les événements dans un fichier texte, tout en les affichant en temps réel dans la console. Voici les étapes clefs de la fonction :

1. Création du dossier de log
2. Création du fichier log
3. Initialisation du logger
4. Formatage des logs
5. Double sortie des logs
6. Retour

Maintenant si on regarde partie par partie :

Initialisation :

```
%%writefile /kaggle/working/ConversaSD_log.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# -----
# Created By : Gabriel BREGAND gabriel.bregand@gmail.com
#             Zouhour KAMTAOUI Zouhour.Kamtaoui@gmail.com
#             Cordelia GUY guy.cordelia@gmail.com
#             Najoua AJOUIRJA najoua.ajouirja@gmail.com
# Created Date: June 2025
# Modif Date v1 : June 2025
# version ='1'
# -----
"""
Création des log pour faire un suivi du programme et des erreur
"""
# -----
```

*%%writefile /kaggle/working/ConversaSD\_log.py*

Ceci est une commande Jupyter/Kaggle, qui indique que tout le code suivant sera enregistré dans un fichier situé à l'adresse /kaggle/working/ConversaSD\_log.py.

*#!/usr/bin/env python3*

*# -\*- coding: utf-8 -\*-*

Il permet d'indiquer au système d'exploitation que ce fichier doit être exécuté avec Python 3 et être codé en UTF-8.



```
# -----
# Created By : Gabriel BREGAND ...
# version ='1'
# ...
Création des log pour faire un suivi du programme et des erreurs
"""
```

Donne des informations sur le code

Import de packages :

```
import logging
import os
from datetime import datetime
```

**logging** permet de gérer les logs générés pendant l'exécution du programme. Il entre autre de suivre le comportement du programme, détecter les erreurs, et faciliter la phase de test ou de maintenance.

**os** permet d'effectuer des opérations liées au système de fichiers. Assure que les logs sont organisés dans un répertoire dédié, quel que soit l'environnement d'exécution (Kaggle, local, serveur...).

**Datetime** module permet de travailler avec les dates et heures en Python. Il garantit que chaque exécution du programme dispose de son propre fichier log distinct, facilitant la traçabilité et le débogage.

Définition de la fonction :

Cette fonction a pour but de configurer un logger personnalisé nommé "ConversaSD".

Elle permet d'enregistrer les messages de log :

- dans un fichier texte horodaté stocké dans un répertoire logs/,
- et en temps réel dans la console.

Cela permet un suivi structuré de l'exécution du programme et facilite la détection d'erreurs ou le débogage.

```
def setup_logger(log_level="INFO", log_file="log_execution.log"):
    # Crée le dossier "logs" dans le dossier du script s'il n'existe pas
    base_dir = os.path.dirname(os.path.abspath(__file__))
    log_dir = os.path.join(base_dir, "logs")
    os.makedirs(log_dir, exist_ok=True)

    # Ajoute la date au fichier log
    dated_log_file = os.path.join(log_dir, f"{datetime.now():%Y%m%d_%H%M%S}_{log_file}")

    logger = logging.getLogger("ConversaSD")
    logger.setLevel(log_level.upper())
    logger.handlers = [] # Reset pour éviter les doublons

    # Format
    formatter = logging.Formatter("%(asctime)s - %(levelname)s - %(message)s")

    # Handler fichier
    file_handler = logging.FileHandler(dated_log_file, encoding="utf-8")
    file_handler.setFormatter(formatter)
    logger.addHandler(file_handler)

    # Handler console
    console_handler = logging.StreamHandler()
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)

    return logger
```



```
def setup_logger(log_level="INFO", log_file="log_execution.log"):
```

Déclaration de la fonction `setup_logger` qui accepte deux paramètres :

- `log_level` (par défaut "INFO") : permet de choisir le niveau de log.
- `log_file` : nom de base du fichier log.

```
base_dir = os.path.dirname(os.path.abspath(__file__))
```

```
log_dir = os.path.join(base_dir, "logs")
```

```
os.makedirs(log_dir, exist_ok=True)
```

Ce bloc de code détermine le répertoire dans lequel les fichiers de logs seront stockés. Il commence par récupérer le chemin absolu du fichier courant, puis crée un sous-dossier `logs/` à cet emplacement. La fonction `os.makedirs(..., exist_ok=True)` s'assure que le dossier est créé s'il n'existe pas encore, sans provoquer d'erreur si c'est déjà le cas. Cela garantit une bonne organisation des fichiers de journalisation dans un emplacement centralisé.

```
dated_log_file = os.path.join(log_dir, f"{datetime.now():%Y%m%d_%H%M%S}_{log_file}")
```

Cette ligne construit le nom complet du fichier de log, en y ajoutant un horodatage basé sur la date et l'heure du moment d'exécution.

```
logger = logging.getLogger("ConversaSD")
```

```
logger.setLevel(log_level.upper())
```

```
logger.handlers = [] # Reset pour éviter les doublons
```

Ici, un logger nommé "ConversaSD" est initialisé à l'aide du module `logging`. Le niveau de log (INFO, DEBUG, ERROR...) est défini selon le paramètre fourni à la fonction. Avant de lui ajouter des gestionnaires (handlers), la liste existante est vidée pour éviter que les messages ne soient enregistrés plusieurs fois si la fonction est appelée plusieurs fois.

```
formatter = logging.Formatter("%(asctime)s - %(levelname)s - %(message)s")
```

Ce formatter définit le modèle de présentation des messages de log. Chaque message comportera la date et l'heure d'émission, le niveau de gravité (par exemple INFO ou ERROR), ainsi que le texte du message.

```
file_handler = logging.FileHandler(dated_log_file, encoding="utf-8")
```

```
file_handler.setFormatter(formatter)
```

```
logger.addHandler(file_handler)
```

On crée ici un outil qui va écrire tous les messages de log dans le fichier dont le nom contient la date et l'heure. On lui applique le format défini plus tôt, pour que chaque message soit bien présenté avec la date, le niveau d'alerte et le texte. Enfin, on relie cet outil au logger principal pour qu'il fonctionne automatiquement dès que le programme démarre.



```
console_handler = logging.StreamHandler()
console_handler.setFormatter(formatter)
logger.addHandler(console_handler)
```

En complément, un second gestionnaire est configuré pour afficher les messages en direct dans la console. Il utilise le même format que celui utilisé pour les fichiers afin d'assurer une cohérence visuelle. Cette double sortie fichier et console permet à la fois de consulter les logs en temps réel et de garder un historique exploitable.

```
return logger
```

Enfin, la fonction retourne le logger entièrement configuré.



## G. Main

Le fichier main joue un rôle central dans l'organisation du projet ConversaSD. Il permet de coordonner l'exécution globale du pipeline, en orchestrant les différents modules spécialisés (clearfil, rag, chatbot, etc.).

Créer un fichier main permet de rassembler toutes les étapes du projet en un seul point d'entrée :

- Prétraitement des fichiers
- Constitution du corpus
- Indexation sémantique
- Authentification
- Lancement du chatbot RAG

Cela permet de lancer **l'ensemble du processus** par une simple commande :

```
python ConversaSD_main.py
```

Chaque étape du pipeline est isolée dans un module spécifique :

Module	Rôle
ConversaSD_clearfil.py	Nettoyage, décompression, conversion des fichiers
ConversaSD_rag.py	Indexation vectorielle des documents
ConversaSD_chatbot.py	Réponses aux questions via RAG
ConversaSD_parametre.py	Centralisation des paramètres
ConversaSD_log.py	Configuration du journal des logs

La main agit comme chef d'orchestre, appelant ces modules dans un ordre logique.

Un main bien structuré :

- Permet d'ajouter facilement de nouvelles fonctionnalités (ex. : options avancées d'analyse ou de visualisation)
- Favorise le débogage modulaire : chaque module peut être testé indépendamment
- Sépare la logique métier de l'interface d'exécution

Le fichier ConversaSD\_main.py n'est pas un simple script. C'est le point d'entrée du programme, pensé pour gérer, configurer, et exécuter le pipeline complet ConversaSD de manière modulaire, réutilisable et maintenable.

### 1. Imports et préparation

Le script commence par importer les bibliothèques nécessaires :



```

#import
import logging
import os
import sys
import argparse
import importlib
import nltk
from llama_index.core import Document
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core import Settings
from sentence_transformers import SentenceTransformer
from transformers import AutoTokenizer, AutoModelForCausalLM
import faiss
from pathlib import Path
import pickle
from huggingface_hub import login
import torch
# link
from ConversaSD_log import setup_logger
import ConversaSD_parametre as parametre
import ConversaSD_clearfil as clearfil
import ConversaSD_rag as rag
import ConversaSD_chatbot as chatbot

```

Modules standards :

- os, sys, argparse : gestion système, chemins, arguments CLI
- pickle : sérialisation d'objets Python
- Path : manipulation de chemins

Modules NLP et IA :

- transformers : chargement des modèles génératifs HF
- sentence\_transformers : embeddings de phrases
- faiss : indexation rapide des vecteurs
- torch : gestion du modèle sur GPU

Modules internes :

- ConversaSD\_clearfil : nettoyage, conversion fichiers
- ConversaSD\_rag : découpe et vectorisation
- ConversaSD\_chatbot : génération de réponse

Cette étape prépare tous les composants nécessaires pour le pipeline complet.

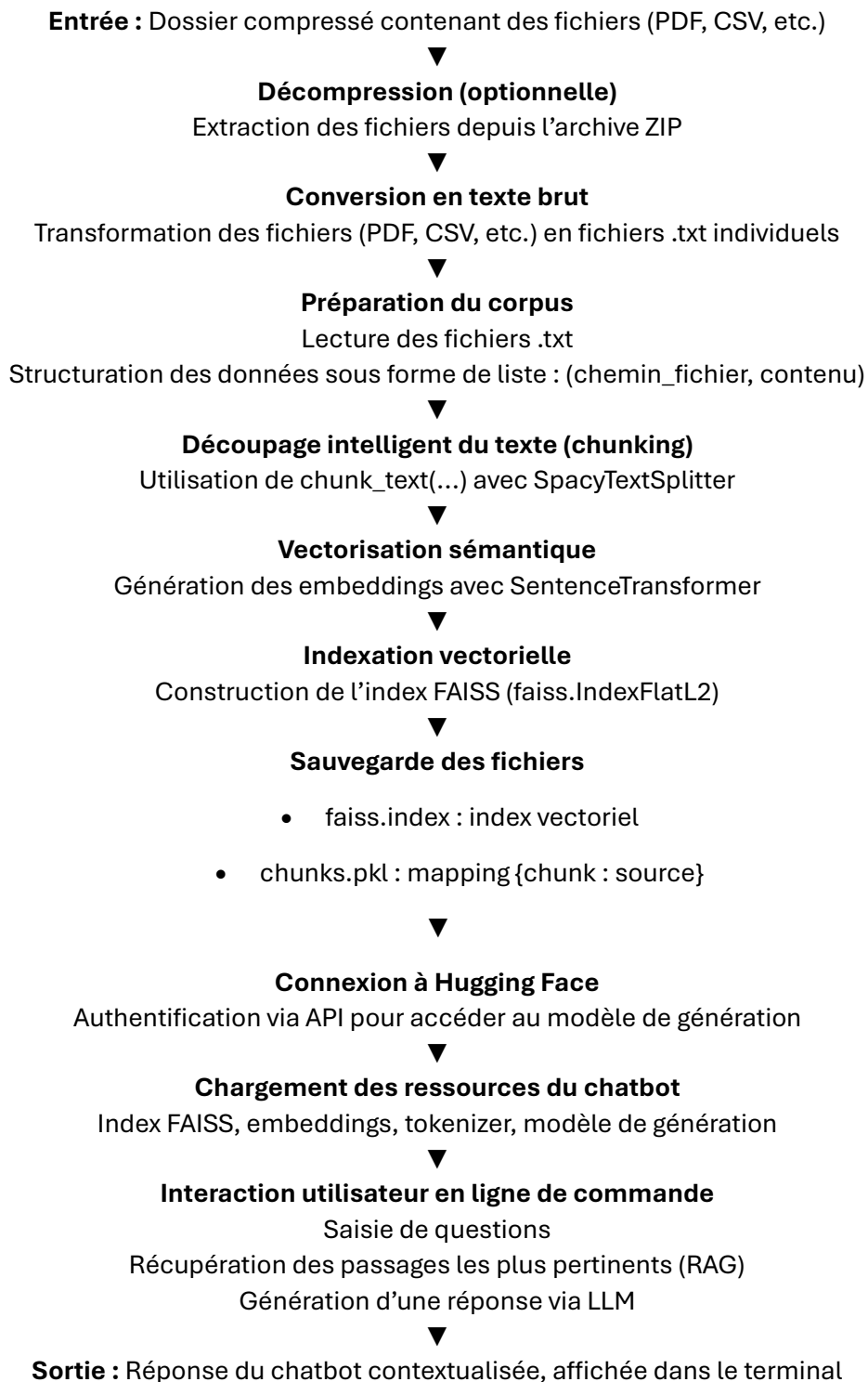
### 1. Objectif général

Le script principal ConversaSD\_main.py orchestre toutes les étapes de la chaîne RAG (Retrieval-Augmented Generation) :

1. Extraction des fichiers
2. Conversion en texte brut
3. Découpage + Embedding + Indexation
4. Authentification Hugging Face
5. Chargement du modèle
6. Chatbot interactif



## 2. Architecture globale





### 3. Description des composants

#### Fonction `get_config()` – Gestion des paramètres

```
def get_config():
    # Création du parser pour les arguments en ligne de commande
    parser = argparse.ArgumentParser(description="Script de ...")

    # Ajout des arguments avec types et options valides pour --log-level
    parser.add_argument("-ll", "--log-level",
                        type=str,
                        default="debug",
                        choices=["debug", "info", "warning", "error", "critical"],
                        help="Niveau de log à utiliser (debug, info, warning, error, critical)")
    parser.add_argument("-lf", "--log-file",
                        type=str,
                        default="log_execution.log",
                        help="Nom du fichier log")
    parser.add_argument("-uz", "--unzipfil",
                        type=bool,
                        default=False,
                        help="Extraire du zip contenu dans 'data' les fichier pdf, csv, ...")
    parser.add_argument("-et", "--extractetxt",
                        type=bool,
                        default=True,
                        help="convertie les fichier pdf, csv, ... en txt")
    parser.add_argument("-cs", "--chunk_size",
                        type=int,
                        default=parametre.chunk_size_default,
                        help="Nombre de token par chunk")
    parser.add_argument("-co", "--chunk_overlap",
                        type=int,
                        default=parametre.chunk_overlap_default,
                        help="Nombre tokens pouvant etre partagés entre 2 chunk consécutifs")
    parser.add_argument("-tk", "--topk",
                        type=int,
                        default=parametre.topk_default,
                        help="Nombre de chunk fourni au chatbot")
    parser.add_argument("-mt", "--max_tokens",
                        type=int,
                        default=parametre.max_tokens_default,
                        help="Nombre de token par reponse du chatboot")
```

```
# Analyse des arguments
args, _ = parser.parse_known_args()

# Retourner la configuration sous forme de dictionnaire
config = {
    "log_level": args.log_level,
    "log_file": args.log_file,
    "unzipfil": args.unzipfil,
    "extractetxt": args.extractetxt,
    "chunk_size": args.chunk_size,
    "chunk_overlap": args.chunk_overlap,
    "topk": args.topk,
    "max_tokens": args.max_tokens,
}

return config
```

Cette fonction récupère les paramètres de configuration passés en ligne de commande.

Paramètre	Rôle
--log-level	Niveau de log (debug, info, ...)
--log-file	Nom du fichier log
--unzipfil	Décompression des fichiers ZIP
--extractetxt	Conversion des fichiers .pdf, .csv, ... en .txt
--chunk_size	Nombre de tokens par chunk
--chunk_overlap	Tokens partagés entre 2 chunks
--topk	Nombre de chunks transmis au modèle
--max_tokens	Longueur maximale de la réponse

Elle retourne un dictionnaire Python prêt à être utilisé dans le pipeline.



## Fonction run(config, logger) – Lancement complet du pipeline

```
def run(config, logger):
    logger.info(f"Début du programme paramétrage : {config}")
    if config["unzipfil"]:
        try:
            clearfil.unzip(parametre.zip_file, parametre.extract_folder, logger)
            logger.info(f"Extraction du fichier {parametre.zip_file} terminée avec succès.")
        except Exception as e:
            # Gestion des exceptions et affichage de l'erreur
            # SUR KAGGLE unzip auto importer le fichier selon la doc
            if parametre.emplacement_code == "/kaggle/" :
                print("KAGGLE unzip auto")
                pass
            else :
                msg = "Une erreur s'est produite lors de l'extraction du fichier ZIP : {str(e)}"
                logger.error(msg)
                raise FileNotFoundError(msg)

    if config["extractetxt"]:
        clearfil.converte_fil_to_individual_texts(parametre.output_txt_file, parametre.full_paths, logger)

    base_folder = Path(parametre.output_txt_file)

    corpus = []

    for file_path in base_folder.glob("*.txt"):
        filename = file_path.stem # nom fichier sans extension
        parts = filename.split("-", 1) # split en 2 parties max sur le premier "-"

        if len(parts) == 2: #un peut bricole avec la combinaison de code final TODO best si temp
            new_path = f"{parts[0]}/{parts[1]}"
        else:
            new_path = parts[0]

        with file_path.open(encoding="utf-8") as f:
            text = f.read()

        corpus.append( (new_path, text) )
```

Cette fonction exécute toutes les étapes de traitement : extraction, conversion, indexation et lancement du chatbot.

Elle appelle notamment :

```
if config["unzipfil"]:
    try:
        clearfil.unzip(parametre.zip_file, parametre.extract_folder, logger)
```

a) clearfil.unzip : décompression/ extraction conditionnelle : cette étape appelle la fonction unzip() qui extrait les documents du .zip. Elle est prévue pour fonctionner localement et sur Kaggle (détection automatique).

```
if config["extractetxt"]:
    clearfil.converte_fil_to_individual_texts(parametre.output_txt_file, parametre.full_paths, logger)
```

b) clearfil.converte\_fil\_to\_individual\_texts : Cette fonction convertit et transforme les fichiers extraits (PDF, CSV) en fichiers .txt.

```
corpus = []

for file_path in base_folder.glob("*.txt"):
    filename = file_path.stem # nom fichier sans extension
    parts = filename.split("-", 1) # split en 2 parties max sur le premier "-"

    if len(parts) == 2: #un peut bricole avec la combinaison de code final TODO best si temp
        new_path = f"{parts[0]}/{parts[1]}"
    else:
        new_path = parts[0]

    with file_path.open(encoding="utf-8") as f:
        text = f.read()

    corpus.append( (new_path, text) )
```



c) corpus : cette étape permet de construire le corpus. Elle parcourt le dossier contenant les .txt et construit une **liste de tuples** (chemin, contenu) à passer à l'index

*Fonction rag.build\_index : création de l'index vectoriel (Indexation sémantique)*

Cette fonction regroupe :

```
rag.build_index(logger=logger,
                corpus = corpus,
                embed_model_name= parametre.embed_model_name,
                index_path = parametre.index_path,
                mapping_path = parametre.mapping_path)

#login(new_session=False)
# Appelle la fonction de login sécurisé
chatbot.ensure_login(logger=logger)

idx, chunks, emb_model, tokenizer, model = chatbot.load_resources(parametre.gen_model,
                                                                    parametre.embed_model_name,
                                                                    parametre.index_path,
                                                                    parametre.mapping_path)

print("Chatbot RAG prêt. (tapez 'exit' pour quitter)\n")
logger.info("lancement du chat bot")
while True:
    q = input("Votre question : ").strip()
    if not q or q.lower() in ("exit", "quit"):
        print("Au revoir !")
        logger.info("exit chatbot")
        break
    logger.info(f"question posé : {q}")
    try:
        resp, used_chunk = chatbot.answer(q, idx, chunks, emb_model, tokenizer, model, config["max_tokens"], config["topk"])
        logger.debug("\n Chunk utilisé :\n")
        logger.debug(used_chunk)
        print(f"\n Réponse : {resp}\n")
    except Exception as e:
        msg = "Error chat bot"
        logger.error(msg)
        raise RuntimeError(msg)

if __name__ == "__main__":
    importlib.reload(parametre) #POUR PERMETTRE A KAGGLE DE PAS GARDER EN MEMOIRE UN IMPORT MODIF !
    importlib.reload(chatbot) #POUR PERMETTRE A KAGGLE DE PAS GARDER EN MEMOIRE UN IMPORT MODIF !
    importlib.reload(clearfil)
    config = get_config()
    logger = setup_logger(log_level=config["log_level"], log_file=config["log_file"])
    run(config, logger)
```

- Découpage linguistique (SpacyTextSplitter)
- Vectorisation via SentenceTransformer
- Indexation avec FAISS
- Sauvegarde dans deux fichiers
- faiss.index (vecteurs encodés)
- chunks.pkl (métadonnées des documents)

## Authentification HuggingFace

```
#login(new_session=False)
# Appelle la fonction de login sécurisé
chatbot.ensure_login(logger=logger)
```

chatbot.ensure\_login : Cette étape invite l'utilisateur à se connecter et s'authentifier à Hugging Face si besoin. Elle utilise une boucle de vérification jusqu'à obtention de l'identité.



## Chargement des ressources

```
idx, chunks, emb_model, tokenizer, model = chatbot.load_resources(parametre.gen_model,
                                                                    parametre.embed_model_name,
                                                                    parametre.index_path,
                                                                    parametre.mapping_path)
```

chatbot.load\_resources : Cette fonction charge les composants du chatbot :

- L'index FAISS (faiss.read\_index)
- Le mapping des chunks (pickle.load)
- Le modèle de transformation (SentenceTransformer)
- Le modèle génératif (AutoModelForCausalLM)
- Le tokenizer associé

Tous ces éléments sont nécessaires pour faire fonctionner le chatbot.

## Boucle d'interaction utilisateur

```
print("Chatbot RAG prêt. (tapez 'exit' pour quitter)\n")
logger.info("lancement du chat bot")
while True:
    q = input("Votre question : ").strip()
    if not q or q.lower() in ("exit", "quit"):
        print("Au revoir !")
        logger.info("exit chatbot")
        break
    logger.info(f"question posé : {q}")
    try:
        resp, used_chunk = chatbot.answer(q, idx, chunks, emb_model, tokenizer, model, config["max_tokens"], config["topk"])
        logger.debug("\n Chunk utilisé :\n")
        logger.debug(used_chunk)
        print(f"\n Réponse : {resp}\n")
    except Exception as e:
        msg = "Error chat bot"
        logger.error(msg)
        raise RuntimeError(msg)
```

Il s'agit d'une boucle interactive qui permet à l'utilisateur de poser des questions et générer des réponses.

- L'utilisateur pose une question.
- Le modèle :
- Vectorise la question
- Cherche les chunks les plus proches (via FAISS)
- Génère une réponse via le modèle HF
- La réponse est affichée dans le terminal.
- Exit ou quit permet de quitter la session.

## Mécanisme de rechargement (Kaggle)

```
if __name__ == "__main__":
    importlib.reload(parametre) #POUR PERMETTRE A KAGGLE DE PAS GARDER EN MEMOIRE UN IMPORT MODIF !
    importlib.reload(chatbot) #POUR PERMETTRE A KAGGLE DE PAS GARDER EN MEMOIRE UN IMPORT MODIF !
    importlib.reload(clearfil)
    config = get_config()
    logger = setup_logger(log_level=config["log_level"], log_file=config["log_file"])
    run(config, logger)
```



Sur Kaggle, certains modules restent en mémoire.  
Ces appels reload(...) forcent leur rechargement à chaque exécution.

#### 4. Résultats attendus

Après exécution :

- Index sémantique généré (fichier .index)
- Mapping sauvegardé (chunks.pkl)
- Conversion complète du corpus
- Lancement du chatbot interactif

#### 5. Exemple de lancement

```
python ConversaSD_main.py --chunk_size 128 --topk 3 --max_tokens 100
```

#### **Terminal :**

Votre question : parle moi de str

2025-06-23 12:55:07,686 - INFO - question posé : parle moi de str

2025-06-23 12:55:11,307 - DEBUG -

Chunk utilisé :

2025-06-23 12:55:11,308 - DEBUG - Pour rappel, n est premier s'il admet exactement deux diviseurs, 1 et lui-même.

Réponse : str est un type de données en Python qui représente une chaîne de caractères.

Chatbot RAG prêt. (tapez 'exit' pour quitter)

Votre question : exit

2025-06-23 15:19:52,457 - INFO - exit chatbot

Au revoir !

