INTRODUCTION ET VUE D'ENSEMBLE

1.1 CONTEXTE ET OBJECTIFS

Ce projet est conçu pour le groupe IMA dans le but de simplifier et d'accélérer la recherche d'informations au sein des contrats d'assurance, une tâche souvent longue et fastidieuse. Trouver des clauses spécifiques ou vérifier des conditions dans des documents contractuels demande généralement du temps et de la rigueur.

L'objectif principal est de développer un chatbot capable d'interroger ces documents pour fournir rapidement des réponses précises aux utilisateurs. L'approche choisie est la **RAG** (Retrieval-Augmented Generation), une technique avancée combinant deux étapes essentielles :

1. **Récupération d'informations (Retrieval)** : Le système effectue une recherche dans une base de documents pour trouver les passages les plus pertinents en fonction de la requête de l'utilisateur.
2. **Génération augmentée (Augmented Generation)** : Un modèle de génération de texte (comme GPT) utilise ces passages pour fournir une réponse pertinente et contextualisée.

Grâce à cette approche, le chatbot sera capable de trouver des informations précises et de les restituer sous une forme compréhensible et adaptée à la question posée. Pour des raisons de confidentialité, les documents utilisés dans ce projet seront des contrats d'exemple et non des contrats réels.

OBJECTIFS DU SYSTEME :

- Fournir des réponses précises aux questions sur les contrats d'assurance en utilisant les documents sources pour garantir la fiabilité des informations.
- Assurer la cohérence des réponses en maintenant une correspondance stricte avec le contenu des contrats afin d'éviter toute interprétation erronée.
- Proposer une interface utilisateur intuitive pour une recherche rapide et efficace des informations.
- Optimiser le traitement des documents afin d'accélérer la récupération et l'analyse des données.

1.2 GROUPE DE TRAVAIL ET METHODOLOGIE

Ce projet a été proposé aux étudiants de 3^{e} année du BUT Science des données de l'Université de Poitiers, sur le campus de Niort. Notre groupe est composé de 3 membres :

- **Dorine BARBEY**
- **Juliette ROSSIGNOL**
- **Thibault DAGUIN**

Dans un temps restreint (4 demi-journées bloquées dans l'agenda, en plus du temps personnel chez nous le soir) nous avons souhaité que notre travail soit collaboratif et facilement réutilisable.

C'est dans cet esprit que vous pourrez consulter et cloner notre projet à partir d'un repository GitHub public (https://github.com/julietterssgnl/RAG-IUT/) et que sur notre application vous aurez la possibilité d'utiliser votre propre clé API 'Google AI' afin que le projet ne dépende pas de nos clés pour assurer son bon fonctionnement.

1.3 BIBLIOTHEQUES ET OUTILS UTILISES

Ce projet repose sur une série de bibliothèques et d'outils spécialisés pour l'indexation des documents, la génération d'embeddings, le stockage des vecteurs, ainsi que la récupération et la génération de réponses. Chaque bibliothèque joue un rôle spécifique dans l'optimisation du processus.

1.3.1 BEAUTIFULSOUP (BS4) – Indexation des Documents

- **Rôle principal** : BeautifulSoup est une bibliothèque Python pour le parsing et la manipulation de documents HTML et XML. Elle est utilisée pour extraire des informations de documents structurés, tout en éliminant les balises HTML inutiles.
- **Dans ce projet** :
  - **Chargement des documents** : Permet de charger des documents HTML dans un format lisible pour le programme.
  - **Extraction de contenu** : Extrait le texte pertinent tout en éliminant les balises inutiles.
  - **Segmentation** : Divise le texte en sections logiques (paragraphe, titres, etc.) pour faciliter l'analyse.

1.3.2 SENTENCE TRANSFORMERS – Embedding des Textes

- **Rôle principal** : Sentence Transformers est une bibliothèque pour la génération d'embeddings de phrases et de textes. Elle repose sur Hugging Face Transformers et permet d'utiliser des modèles pré-entraînés pour la compréhension sémantique des textes.
- **Dans ce projet** :
  - **Conversion en vecteurs** : Transforme des segments de texte (phrases, paragraphes) en vecteurs numériques représentant leur signification.
  - **Modèle utilisé** : Le modèle "HIT-TMG/KaLM-embedding-multilingual-mini-instruct-v1.5", un modèle multilingue, est utilisé pour traiter des textes en plusieurs langues.

1.3.3 CHROMADB – Stockage et Recherche des Vecteurs

- **Rôle principal** : ChromaDB est une base de données optimisée pour le stockage et la gestion de vecteurs, permettant une recherche sémantique efficace.
- **Dans ce projet** :
  - **Stockage des embeddings** : Permet de stocker les vecteurs dans une base de données spécialisée.
  - **Recherche rapide** : Lors d'une requête, ChromaDB compare les embeddings et récupère rapidement les passages pertinents.

1.3.4 GOOGLE AI STUDIO – Récupération et Génération de Réponses

- **Rôle principal** : Google AI Studio permet d'utiliser des modèles de langage avancés pour la génération de texte, souvent basés sur des architectures comme GPT.
- **Dans ce projet** :
  - **Récupération sémantique** : Utilise les passages extraits de la base ChromaDB pour comprendre le contexte.
  - **Génération de réponses** : Le modèle génère des réponses cohérentes en utilisant les passages récupérés, en garantissant leur fidélité au contenu des documents sources.

ARCHITECTURE DU SYSTEME

Le système repose sur une architecture modulaire, où chaque composant joue un rôle spécifique dans le traitement des contrats d'assurance. Ces modules sont interconnectés pour assurer un flux de données fluide et une réponse rapide aux requêtes des utilisateurs.

2.1 STRUCTURE DES MODULES

L'architecture du projet est organisée comme suit :

```
RAG-IUT/
├── .gitignore
├── README.md
├── requirements.txt
├── app.py
├── feedback.db
├── setup.docx
├── structure.txt
├── démo_chatbot.mp4
│
├── chroma_db/
│   ├── chroma.sqlite3
│   ├── <collection_id>/
│       ├── data_level0.bin
│       ├── header.bin
│       ├── length.bin
│       ├── link_lists.bin
│
├── documents/
│   ├── accident_automobile.html
│   ├── dégât_des_eaux.html
│   ├── incendie_domestique.html
│   ├── vol.html
│
├── pages/
│   ├── 1_A_propos.py
│   ├── 2_Statistiques.py
│
└── rag/
    ├── chat/
    │   ├── chatbot.py
    │
    ├── feedback/
    │   ├── feedback_manager.py
    │
    ├── indexing/
    │   ├── document_loader.py
    │   ├── text_splitter.py
    │   ├── vectorstore.py
```

MODULES DE TRAITEMENT DES DREFOCUMENTS (RAG/INDEXING/)

- **document_loader.py** : Charge et parse les documents HTML en extrayant le contenu pertinent.
- **text_splitter.py** : Divise les documents en unités plus petites pour faciliter leur traitement.
- **vectorstore.py** : Convertit les segments de texte en vecteurs et les stocke dans une base de données pour des recherches rapides.
- **chatbot.py** : Interagit avec l'API Google AI pour effectuer des recherches sémantiques et générer des réponses.
- **app.py** : Fournit l'interface utilisateur avec Streamlit pour l'interaction avec le chatbot.

2.2 FLUX DE DONNEES

Le système suit un processus structuré en plusieurs étapes :

1. **Chargement initial des documents HTML** : Les fichiers sont extraits et transformés en objets exploitables.
2. **Traitement et segmentation du texte** : Le contenu est divisé en unités plus petites pour une gestion optimale.
3. **Création des embeddings et stockage** : Les segments sont vectorisés et indexés dans la base de données.
4. **Interface utilisateur pour les requêtes** : L'utilisateur interagit avec le chatbot via l'interface Streamlit.
5. **Génération des réponses** : Une recherche sémantique est effectuée et une réponse est générée en utilisant les passages pertinents.

3. ANALYSE DETAILLEE DES COMPOSANTS

3.1 INTERFACE UTILISATEUR (APP.PY)

L'interface utilisateur est gérée via **Streamlit**, une bibliothèque Python qui permet de créer des applications web interactives de manière simple.

Architecture de l'application :

- **Gestion des sessions Streamlit** pour suivre les interactions de l'utilisateur et garantir la persistance des données pendant la session.
- Un **système d'initialisation des composants** est mis en place pour charger les documents et préparer les autres modules à fonctionner dès que l'utilisateur lance l'application.
- **Stockage temporaire des données de session,** permettant de récupérer les informations nécessaires lors de l'interaction avec le chatbot.

Éléments d'interface :

- **Des composants Streamlit** sont utilisés pour l'affichage, tels que les boutons pour soumettre les questions et les affichages de réponses.
- **Des indicateurs de chargement** sont présents pour informer l'utilisateur que le chatbot est en train de traiter sa demande.
- **Un système de boutons pour le feedback** est mis en place pour recueillir des évaluations sur la qualité des réponses fournies.

Gestion des erreurs :

- La gestion des erreurs inclut la **détection des erreurs de base de données** liées à ChromaDB ou à SQLite.
- Un **mécanisme de réinitialisation** permet de résoudre les erreurs sans nécessiter de redémarrage complet de l'application.
- Les messages d'erreur sont **descriptifs**, permettant aux utilisateurs et aux développeurs de comprendre rapidement l'origine du problème.

3.2 MODULE D'INDEXATION (RAG/INDEXING/)

3.2.1 DOCUMENTLOADER (DOCUMENT_LOADER.PY)

Le **DocumentLoader** gère le chargement et l'analyse des documents HTML.

Structure technique :

- Utilisation des fonctions os.listdir() et os.path.join() pour la gestion des chemins de fichiers, permettant de parcourir le dossier contenant les documents.
- **BeautifulSoup** est utilisé pour le parsing HTML avec l'encodage UTF-8 afin d'extraire le texte pertinent.
- Méthode de nettoyage des balises **script** et **style** via la fonction decompose() pour éviter que ces éléments n'affectent le contenu analysé.

Traitement des données :

- Extraction du texte via la méthode get_text(), en utilisant un espace comme séparateur pour garantir que le texte est proprement formaté.
- Organisation des données en dictionnaires avec deux clés principales : page_content (le texte du document) et metadata (les informations source comme l'URL ou le nom du fichier).
- Conservation des informations source dans les métadonnées pour garantir la traçabilité du contenu traité.

3.2.2 TEXTSPLITTER (TEXT_SPLITTER.PY)

Le **TextSplitter** effectue la segmentation des documents en unités plus petites afin de faciliter leur traitement ultérieur par le modèle d'embedding.

Paramètres de configuration :

- La taille des chunks est définie à **1000 caractères** par défaut pour assurer une bonne granularité tout en évitant des segments trop longs.
- Le chevauchement entre les segments est de **200 caractères**, ce qui permet de maintenir le contexte tout en découpant le texte de manière cohérente.

Fonctionnement :

- Le **TextSplitter** parcourt les documents séquentiellement, en divisant le texte en sections selon des motifs de titre ou d'autres sections clés à l'aide d'expressions régulières.
- Le découpage utilise des titres comme points de segmentation, ce qui permet de garder une certaine logique structurelle.
- Chaque segment est ensuite stocké dans un format **JSON** avec une structure uniforme comprenant le texte et les métadonnées associées.

3.2.3 VECTORSTORE (VECTORSTORE.PY)

Le **VectorStore** est responsable de la vectorisation des documents et du stockage des vecteurs dans une base de données optimisée pour les recherches sémantiques.

Technologies employées :

- Le modèle **KaLM-embedding-multilingual-mini-instruct-v1.5** est utilisé pour générer des embeddings multilingues. Ce choix permet de traiter des documents en plusieurs langues tout en garantissant une bonne qualité de vectorisation.
- La base de données **ChromaDB** est utilisée en mode persistant pour stocker les vecteurs de manière efficace. Elle permet de gérer de grandes quantités de données tout en offrant une recherche rapide et pertinente.
- Une **collection unique** est créée pour le stockage des vecteurs, facilitant l'organisation et la gestion des données.

Processus de traitement :

- Vérification initiale de l'état de la collection dans ChromaDB, en s'assurant que la base de données est prête à recevoir de nouveaux vecteurs.
- Les textes sont convertis en vecteurs via le modèle d'embedding, qui génère une représentation numérique des segments de texte.
- Une **recherche par similarité** est effectuée pour retrouver les passages les plus pertinents. Le paramètre **k** est configurable pour ajuster le nombre de résultats retournés lors de la recherche.

3.3 MODULE DE CHAT (RAG/CHAT/)

3.3.1 CHATBOT (CHATBOT.PY)

Le **Chatbot** constitue l'interface avec l'API Google AI pour la génération de réponses en fonction des requêtes de l'utilisateur.

Structure technique :

- L'API de **Google AI** est configurée via une clé d'API, permettant d'utiliser les services d'intelligence artificielle pour la génération de texte.
- Le modèle **Gemini Pro** est utilisé pour répondre aux questions de manière contextuelle et pertinente.
- Les **prompts** sont construits en intégrant le contexte extrait des recherches vectorielles, ce qui permet de répondre de manière précise et liée aux informations extraites des documents.

Fonctionnalités :

- Le **Chatbot** génère des réponses en français, assurant une communication fluide pour les utilisateurs francophones.
- Le contexte de la recherche vectorielle est intégré dans le prompt afin que les réponses soient basées sur les passages les plus pertinents récupérés.
- Les réponses sont renvoyées sous forme de texte, adaptées aux besoins de l'utilisateur et conformes aux informations des documents sources.

3.4 STATISTIQUES ET FEEDBACK

3.4.1 STATISTIQUES (2_STATISTIQUES.PY)

Les statistiques sont collectées pour évaluer la performance du chatbot et la satisfaction des utilisateurs.

Structure des données :

- Une base de données **SQLite** est utilisée pour le stockage des retours d'utilisateurs. Une **table unique** est créée pour stocker les informations sur les interactions et les évaluations des réponses.
- Des **requêtes SQL** sont utilisées pour agréger les données et calculer les métriques de satisfaction.

Visualisation :

- Des **graphiques Plotly** sont générés pour afficher les statistiques de manière visuelle, comme les taux de satisfaction ou le nombre de requêtes traitées.
- Les **métriques de satisfaction utilisateur** sont mesurées pour déterminer l'efficacité du chatbot.
- Les données sont actualisées en temps réel, permettant de suivre l'évolution des interactions avec le chatbot.

3.4.2 FEEDBACKMANAGER (FEEDBACK_MANAGER.PY)

Le **FeedbackManager** gère la collecte et l'analyse des retours des utilisateurs.

Structure de la base de données :

- **SQLite** est utilisé comme système de gestion de base de données.
- La **table 'feedback'** comprend plusieurs colonnes :
  - id: Identifiant unique auto-incrémenté.
  - question: Texte de la question posée par l'utilisateur.
  - response: Réponse générée par le chatbot.
  - is_helpful: Indicateur booléen de satisfaction (utile ou non).
  - timestamp: Horodatage automatique des retours.

Méthodes principales :

- **_init_db()** : Initialise la base de données, crée la table si elle n'existe pas, et gère les connexions à la base.
- **add_feedback()** : Enregistre les nouveaux retours utilisateurs, en incluant la question, la réponse et l'indicateur de satisfaction.
- **get_statistics()** : Calcule les métriques de satisfaction (positifs et négatifs) et retourne ces données sous forme de tuple.

Implémentation technique :

- La gestion des connexions se fait via un **context manager** (with), garantissant une gestion efficace des ressources.
- Les **requêtes paramétrées** sont utilisées pour éviter les injections SQL et garantir la sécurité des données.
- Des **valeurs par défaut** sont définies pour les statistiques en l'absence de données, assurant ainsi la stabilité du système même en cas de faible volume de retours.

SCHEMA D'ARCHITECTURE