Université de
Versailles – Saint-Quentin-en-Yvelines

Architectures orientées services

Rapport TD 3 : Embeddings et Services Web API : Récupération Augmentée pour Répondre à des Questions sur des Données Textuelles

Réalisées par :

OLIVEIRA PEREIRA Diogo

CORNEJO GUILLEN Oscar

Supervisées par : Yehia TAHER

22 novembre 2024

Sommaire

Table des matières

Rapport TD 3 : Embeddings et Services Web API : Récupération Augmentée pour Répondre à des Questions sur des Données Textuelles

Architectures orientées services

1 Architecture du système

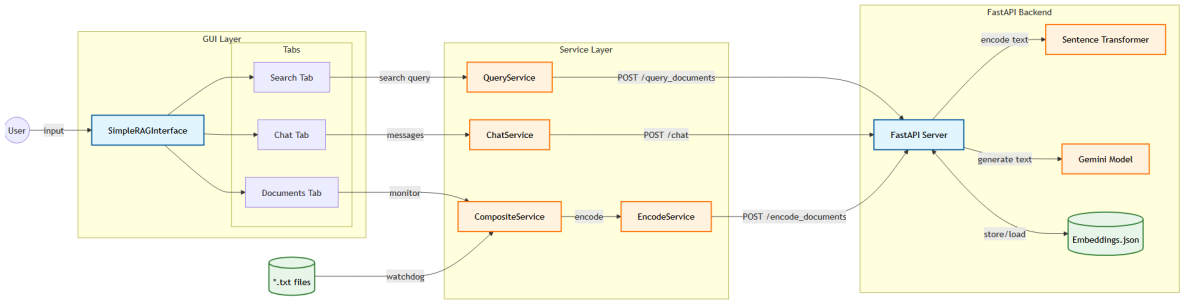

FIGURE 1 – Architecture des services

Le système suit une architecture en trois couches distinctes qui interagissent de manière hiérarchique :

1. **Couche Interface Utilisateur** : L'utilisateur interagit avec le système via une interface graphique composée de trois onglets principaux :
   — Query : pour effectuer des requêtes documentaires
   — Chat : pour interagir avec le système de manière conversationnelle
   — Documents : pour suivre l'état du traitement des documents
2. **Couche Services** : Les interactions utilisateur sont gérées par quatre services spécialisés :
   — Le Service de Recherche traite les requêtes de recherche documentaire
   — Le Service de Conversation gère le dialogue avec l'utilisateur
   — Le Service Composite coordonne le traitement des documents
   — Le Service d'Encodage transforme les documents en embeddings
3. **Couche Backend** : Le serveur FastAPI orchestre trois opérations fondamentales :
   — L'encodage des textes via Sentence Transformer
   — La génération de réponses via le modèle Gemini
   — Le stockage et la récupération des embeddings dans une base de données JSON

Le flux de données suit un parcours séquentiel : les fichiers texte sont d'abord surveillés par le Service Composite, qui les transmet au Service d'Encodage. Une fois encodés via le backend, les documents sont stockés et deviennent disponibles pour les services de recherche et de conversation. Ces derniers utilisent ces données encodées pour répondre aux requêtes utilisateur, soit par une recherche directe, soit par une génération de réponse contextuelle via le modèle Gemini.

1.1 Présentation générale

La **Récupération Augmentée** (*Retrieval-Augmented Generation*, RAG) est une approche hybride combinant la recherche d'informations et les modèles de génération de texte. Elle s'appuie sur des données structurées et non structurées pour fournir des réponses précises et contextualisées à des requêtes utilisateur. Dans ce projet, cette approche est mise en œuvre sous forme de services Web API en utilisant le framework `FastAPI`.

Le système développé intègre deux étapes principales :

— **Encodage des documents :** Les documents textuels sont transformés en vecteurs numériques appelés *embeddings*. Ces vecteurs capturent la sémantique des documents, permettant une comparaison efficace avec d'autres vecteurs. Un modèle de langage pré-entraîné tel que `paraphrase-MiniLM-L6-v2` de la bibliothèque `sentence-transformers` est utilisé pour cette tâche. Les embeddings, accompagnés des métadonnées des documents (ID et texte), sont ensuite stockés dans un fichier JSON, offrant une base pour les recherches futures.

— **Recherche de documents :** Lorsqu'un utilisateur soumet une requête textuelle, celle-ci est également encodée en un vecteur d'embedding. Le système calcule ensuite la *similarité cosinus* entre l'embedding de la requête et ceux des documents. Les documents les plus proches en termes de similarité sont renvoyés comme résultats, classés par pertinence.

L'application de cette architecture s'étend à des domaines variés, notamment :

— La recherche juridique (articles de lois, jurisprudence).

— La veille sur les réseaux sociaux (analyse des tweets).

— L'assistance automatisée (chatbot basé sur des bases de connaissances textuelles).

1.2 Objectif

Le projet vise à répondre aux défis posés par la recherche d'informations dans des ensembles de données volumineux et hétérogènes en proposant une solution performante et facilement extensible. Les objectifs spécifiques incluent :

— **Faciliter l'accès aux informations pertinentes :** Réduire le temps nécessaire pour retrouver des documents pertinents en exploitant des modèles d'apprentissage automatique.

— **Utiliser une approche modulaire :** Chaque fonctionnalité est implémentée sous forme de services indépendants, ce qui permet une maintenance et une évolutivité simplifiées.

— **Offrir une API REST performante :** Fournir une interface API standardisée, permettant une intégration facile avec d'autres systèmes ou des interfaces utilisateur.

— **Optimiser les performances :** Assurer une gestion efficace des calculs de similarité même avec un grand nombre de documents encodés, grâce à l'utilisation de bibliothèques optimisées comme `NumPy`.

En résumé, ce projet constitue une base solide pour des systèmes de récupération d'informations avancés, combinant la puissance des modèles de langage avec l'efficacité des architectures orientées services.

1.3 Architecture orientée services (SOA)

L'architecture orientée services (*Service-Oriented Architecture*, SOA) constitue une approche de conception logicielle où les fonctionnalités sont implémentées sous forme de services indépendants. Chaque service est une unité autonome qui expose des fonctionnalités spécifiques via une interface standardisée, telle qu'une API REST. Cette architecture est particulièrement adaptée aux systèmes complexes nécessitant modularité et flexibilité.

Dans le cadre de ce projet, l'architecture SOA est utilisée pour structurer les deux principaux services :

— **Service d'encodage des documents :** Un service dédié à la transformation des textes en embeddings vectoriels, avec stockage des résultats pour un usage ultérieur.

— **Service de recherche et similarité :** Un service permettant d'effectuer des recherches textuelles en calculant la similarité cosinus entre une requête utilisateur et les documents encodés.

Caractéristiques principales de l'architecture SOA appliquée :

— **Autonomie des services :** Chaque service fonctionne indépendamment, sans dépendance directe avec les autres.

— **Interopérabilité :** Les services communiquent via des requêtes HTTP standardisées, facilitant leur intégration avec des systèmes externes.

— **Réutilisabilité :** Les services peuvent être réutilisés dans d'autres projets ou contextes, réduisant ainsi les coûts de développement.

— **Extensibilité :** L'ajout de nouveaux services ou fonctionnalités est simplifié, grâce à la modularité de l'architecture.

— **Scalabilité :** Chaque service peut être déployé de manière indépendante, permettant une montée en charge facile en cas d'augmentation des utilisateurs ou des données traitées.

1.3.1 Principes architecturaux appliqués

La conception du système a suivi plusieurs principes architecturaux clés :

— **Responsabilité unique :** Chaque service a une responsabilité bien définie (encodage ou recherche), ce qui simplifie son développement et sa maintenance.

— **Cohésion forte et couplage faible :** Les services sont fortement cohésifs (focalisés sur une tâche unique) et faiblement couplés (indépendants les uns des autres).

— **Interfaces standardisées :** Les services exposent des interfaces REST claires et normalisées, permettant une interaction uniforme.

— **Orientation utilisateur :** L'architecture est conçue pour répondre efficacement aux besoins des utilisateurs finaux, notamment en termes de rapidité et de pertinence des résultats.

1.4 Modélisation de données

1.4.1 Modèle d'embedding

1.4.1.1 Modèles préentraînés

Hugging Face propose des modèles préentraînés de Sentence Transformers, à la fois originaux et issus de la communauté, pour des tâches telles que les embeddings de phrases et la recherche sémantique. Les modèles originaux, comme `all-mpnet-base-v2`, offrent une précision maximale, tandis que des modèles comme `all-MiniLM-L6-v2` privilégient un équilibre entre qualité et rapidité. Il est essentiel de prendre en compte la taille des modèles et de tester celui qui convient le mieux à nos besoins. Dans le cas de ce projet, le modèle paraphrase-MiniLM-L6-v2 a été choisi.

Model Name	Performance Sentence Embeddings (14 Datasets)	Performance Semantic Search (6 Datasets)	Avg. Performance	Speed	Model Size
all-mpnet-base-v2	69.57	57.02	63.30	2800	420 MB
multi-qa-mpnet-base-dot-v1	66.76	57.60	62.18	2800	420 MB
all-distilroberta-v1	68.73	50.94	59.84	4000	290 MB
all-MiniLM-L12-v2	68.70	50.82	59.76	7500	120 MB
multi-qa-distilbert-cos-v1	65.98	52.83	59.41	4000	250 MB
all-MiniLM-L6-v2	68.06	49.54	58.80	14200	80 MB
multi-qa-MiniLM-L6-cos-v1	64.33	51.83	58.08	14200	80 MB
paraphrase-multilingual-mpnet-base-v2	65.83	41.68	53.75	2500	970 MB
paraphrase-albert-small-v2	64.46	40.04	52.25	5000	43 MB
paraphrase-multilingual-MiniLM-L12-v2	64.25	39.19	51.72	7500	420 MB
paraphrase-MiniLM-L3-v2	62.29	39.19	50.74	19000	61 MB
distiluse-base-multilingual-cased-v1	61.30	29.87	45.59	4000	480 MB
distiluse-base-multilingual-cased-v2	60.18	27.35	43.77	4000	480 MB

FIGURE 2 – Tableau de comparaison des modèles pré-entraînés

Le modèle `paraphrase-MiniLM-L6-v2` est idéal si on cherche rapidité et efficacité. Il est léger, 5 fois plus rapide que les modèles plus volumineux, tout en maintenant une bonne qualité. Il convient parfaitement aux tâches générales nécessitant des ressources matérielles limitées ou un traitement en temps réel. Sa capacité à capturer le sens sémantique des phrases en fait un outil puissant pour une variété d'applications de traitement du langage naturel.

1.4.1.2 Modèle paraphrase-MiniLM-L6-v2

Le modèle 'paraphrase-MiniLM-L6-v2' est un transformateur de phrases développé par l'équipe Sentence Transformers. Il s'agit d'un modèle conçu pour mapper des phrases et des paragraphes dans un espace vectoriel dense de 384 dimensions. Cette représentation vectorielle capture le sens sémantique du texte d'entrée, ce qui rend le modèle particulièrement utile pour des tâches telles que le clustering ou la recherche sémantique.

Parmi ses caractéristiques, on peut citer les suivantes :

— **Dimensions du vecteur de sortie :** 384
— **Architecture :** Basée sur l'architecture MiniLM, qui est une version compacte des modèles de type BERT.
— **Entraînement :** Le modèle a été affiné sur un vaste ensemble de données comprenant plus d'un milliard de paires de phrases provenant de diverses sources.

Utilisation

Pour utiliser ce modèle, on l'importe via la bibliothèque sentence-transformers en Python :

```
# Initialize the sentence transformer model
embedding_model = SentenceTransformer('paraphrase-MiniLM-L6-v2')
```

Listing 1 – Modèle d'embedding pour les requêtes d'encodage et de recherche de documents

Le modèle 'paraphrase-MiniLM-L6-v2' est polyvalent et peut être utilisé pour diverses tâches de traitement du langage naturel, notamment :

1. **Recherche d'informations :** Pour trouver des documents ou passages pertinents en fonction d'une requête.
2. **Clustering de texte :** Pour regrouper des documents textuels similaires en fonction de leur contenu sémantique.
3. **Identification de paraphrases :** Pour détecter quand deux phrases expriment la même signification de manière différente.

Performance multilingue

Bien que principalement entraîné sur des données en anglais, le modèle montre une certaine capacité à traiter des textes dans d'autres langues, y compris le français. Cependant, pour des résultats optimaux en français, il est recommandé d'utiliser des modèles spécifiquement conçus pour le multilinguisme, comme le 'distiluse-base-multilingual-cased-v1'.

1.4.2 Structure des données

À partir des documents au format `.txt` faisant référence à :

— Innovation technologique
— Environnement
— Culture française
— Santé publique
— Économie numérique
— Tweets
— Lois et législation
— Données mixtes

Dans cette tâche, chaque document est encodé grâce à la requête d'encodage du document et stocké dans un fichier `.json` après avoir obtenu son *embedding* selon le texte saisi dans la requête avec les éléments suivants :

```
{
    "id": <int>, % L'identifiant unique du document
    "text": <string>, % Le texte du document, sous forme de chaîne de caractères
    "embedding": <vector<float>> % Le vecteur d'embedding, représentation numérique du texte
}
```

Où :

— **id** : Un entier servant d'identifiant unique au document. Il est utilisé pour indexer et retrouver rapidement les documents.

— **text** : Le contenu textuel du document, qui peut être un article de loi, un tweet, etc.

— **embedding** : Un vecteur de nombres flottants représentant le texte sous une forme numérique. Ce vecteur est obtenu par un modèle d'embedding et permet des comparaisons sémantiques entre les documents.

2 Identification des services

2.1 Service d'encodage des documents

Le **service d'encodage des documents** est une des trois fonctionnalités principales du système. Il permet de convertir des textes en vecteurs numériques (*embeddings*), qui capturent la sémantique des documents. Ces embeddings sont ensuite stockés dans un fichier JSON pour être réutilisés dans les recherches.

2.1.1 Description du processus

Le service est implémenté sous la route API suivante :

— **Route :** `POST /encode_documents`

— **Entrée :** Une liste de documents textuels à encoder, envoyée au format JSON.

— **Sortie :** Un message confirmant le succès de l'encodage et du stockage des embeddings.

Les étapes du traitement sont les suivantes :

1. **Réception des documents :** Les textes sont envoyés au service via une requête POST au format JSON.
2. **Encodage des documents :** Chaque document est transformé en vecteur d'embedding à l'aide d'un modèle de langage pré-entraîné, comme `paraphrase-MiniLM-L6-v2`, issu de la bibliothèque `sentence-transformers`.
3. **Stockage des embeddings :** Les embeddings générés sont enregistrés dans un fichier JSON avec les métadonnées des documents (ID et texte original).

2.1.2 Exemple de Requête et Réponse

Voici un exemple de requête envoyée au service :

```
POST /encode_documents HTTP/1.1
Content-Type: application/json

{
    "documents": [
        "Article 1 : Toute personne a droit au respect de sa vie privée...",
        "Le climat change, il est temps d'agir."
    ]
}
```

Et voici la réponse renvoyée par le service :

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "status": "success",
    "message": "Documents encoded and stored successfully"
}
```

2.1.3 Code d'implémentation

Le service est implémenté en Python à l'aide de `FastAPI` et utilise la bibliothèque `sentence-transformers` pour générer les embeddings. Voici un extrait de code montrant les étapes principales :

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from sentence_transformers import SentenceTransformer
import json

# Initialisation du modele d'embedding
model = SentenceTransformer("paraphrase-MiniLM-L6-v2")

# Initialisation de l'application FastAPI
app = FastAPI()

# Modele de donnes pour l'entree
class DocumentInput(BaseModel):
    documents: list[str]

@app.post("/encode_documents")
def encode_documents(input_data: DocumentInput):
    try:
        # Liste pour stocker les embeddings
        encoded_data = []

        # Encodage de chaque document
        for idx, text in enumerate(input_data.documents):
            embedding = model.encode(text).tolist()  # Conversion en liste pour JSON
            encoded_data.append({
                "id": idx + 1,
                "text": text,
                "embedding": embedding
            })

        # Stockage des embeddings dans un fichier JSON
        with open("document_embeddings.json", "w") as f:
            json.dump(encoded_data, f)

        # Reponse de succes
        return {
            "status": "success",
            "message": "Documents encoded and stored successfully"
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"An error occurred: {str(e)}")
```

Listing 2 – Implémentation du service d'encodage des documents

2.1.4 Détails Techniques

- **Modèle d'embedding utilisé :** Le modèle `paraphrase-MiniLM-L6-v2`, issu de la bibliothèque `sentence-transformers`, est un modèle léger et performant pour générer des embeddings textuels. Il est pré-entraîné sur de larges corpus et excelle dans les tâches de recherche sémantique.
- **Stockage des données :** Les embeddings sont stockés dans un fichier JSON, chaque entrée contenant :
  - **ID :** Identifiant unique pour chaque document.
  - **Texte :** Contenu textuel original du document.
  - **Embedding :** Vecteur numérique représentant le document.
- **Format des embeddings :** Les embeddings générés sont des vecteurs à 384 dimensions (taille de sortie du modèle utilisé).

2.1.5 Avantages de l'approche

— **Flexibilité :** Le service peut traiter n'importe quel type de texte (articles, tweets, etc.) sans modification.

— **Performance :** Le modèle `paraphrase-MiniLM-L6-v2` est optimisé pour des calculs rapides et précis, même sur des machines avec des ressources limitées.

— **Modularité :** Les embeddings sont stockés indépendamment, ce qui permet une réutilisation facile dans d'autres services (comme la recherche).

2.2 Service de recherche et similarité de documents

Le **service de recherche et similarité de documents** est une des trois fonctionnalités principales du système. Il permet de faire la recherche d'une chaîne, la convertir en valeur numérique (*embedding*) et la comparer avec les vecteurs numériques (*embeddings*) générés par l'encodage de documents. La comparaison est effectuée sur la base d'une similarité cosinus et le résultat est affiché sur la base des scores de comparaison par ordre décroissant.

2.2.1 Description du processus

Le service est implémenté sous la route API suivante :

— **Route :** `POST /search_documents`

— **Entrée :** Un objet JSON au format suivant :

```
{
    "query": "votre requête ici"
}
```

où `query` est une chaîne de caractères représentant la recherche à effectuer.

— **Sortie :** Un objet JSON contenant les 5 documents les plus pertinents sous le format suivant :

```
{
    "relevant_documents": [
        {
            "id": 1,
            "text": "Texte du document",
            "similarity_score": 0.95
        },
        ...
    ]
}
```

où chaque document pertinent est représenté par un identifiant (`id`), son texte (`text`), et un score de similarité (`similarity_score`).

Les étapes du traitement sont les suivantes :

1. **Chargement des embeddings :** Le service charge les embeddings des documents préalablement stockés. Si aucun document n'a été encodé, une exception est levée avec un code d'erreur 404.
2. **Encodage de la requête :** La chaîne de caractères de la requête est transformée en vecteur d'embedding à l'aide d'un modèle d'embedding.
3. **Calcul des similarités :** Pour chaque document, la similarité entre l'embedding de la requête et l'embedding du document est calculée à l'aide de la similarité cosinus.
4. **Tri des résultats :** Les résultats sont triés par score de similarité dans l'ordre décroissant.
5. **Retour des documents pertinents :** Les 5 documents les plus pertinents, avec leurs ID, textes et scores de similarité, sont retournés dans la réponse.

2.2.2 Exemple de Requête et Réponse

Voici un exemple de requête envoyée au service :

```
POST /search_documents HTTP/1.1
Content-Type: application/json

{
    "query": "Quel est le meilleur moyen de conserver l'énergie ?"
}
```

Et voici la réponse renvoyée par le service :

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "relevant_documents": [
        {
            "id": 1,
            "text": "L'énergie peut être conservée en utilisant des appareils efficaces.",
            "similarity_score": 0.92
        },
        {
            "id": 2,
            "text": "Il est important d'éteindre les lumières inutiles.",
            "similarity_score": 0.89
        },
        ...
    ]
}
```

2.2.3 Code d'implémentation

Le service est implémenté en Python à l'aide de `FastAPI` et utilise la bibliothèque `numpy` pour obtenir les résultats selon la similarité cosinus.

2.2.3.1 Similarité cosinus

L'extrait de code suivant montre la fonction `cosine_similarity` qui calcule la similarité cosinus entre deux vecteurs, `v1` et `v2`, représentés sous forme de tableaux NumPy (`np.ndarray`).

La similarité cosinus mesure la similarité angulaire entre deux vecteurs, comparant l'angle entre eux et non leur magnitude. Une valeur de **1** signifie que les vecteurs sont identiques en direction, **0** signifie qu'ils sont orthogonaux (sans relation), et **-1** indique qu'ils sont complètement opposés.

```python
# Fonction de similarite entre deux vecteurs (embeds)
def cosine_similarity(v1: np.ndarray, v2: np.ndarray) -> float:
    dot_product = np.dot(v1, v2)
    norm_v1 = np.linalg.norm(v1)
    norm_v2 = np.linalg.norm(v2)
    return dot_product / (norm_v1 * norm_v2)
```

Listing 3 – Implémentation du service d'encodage des documents

La ligne $dot_product = np.dot(v1, v2)$ représente le **produit scalaire** (ou produit intérieur) des deux vecteurs `v1` et `v2`. Le produit scalaire est une mesure de l'alignement des vecteurs. Si le produit scalaire est élevé, cela signifie que les vecteurs sont fortement alignés dans la même direction.

Mathématiquement, la formule du produit scalaire est :

$$\text{produit scalaire} = v_1 \cdot v_2 = \sum_{i=1}^{n} v_{1i} \times v_{2i}$$

Où v_{1i} et v_{2i} sont les éléments correspondants des vecteurs `v1` et `v2`.

Dans les lignes $norm_v1 = np.linalg.norm(v1)$ et $norm_v2 = np.linalg.norm(v2)$ on calcule les **normes** (ou longueurs) des vecteurs `v1` et `v2` respectivement. La norme d'un vecteur est la racine carrée de la somme des carrés de ses composants.

Mathématiquement, la norme d'un vecteur v est :

$$\|v\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

Cela mesure la longueur du vecteur. En termes de similarité cosinus, on normalise les vecteurs en les divisant par leurs normes, afin que la comparaison soit indépendante de la magnitude et se concentre uniquement sur la direction.

Enfin, la ligne $returndot_product/(norm_v1 * norm_v2)$ renvoie la similarité cosinus, qui est le quotient du produit scalaire des vecteurs et du produit de leurs normes. Cela garantit que le résultat est compris entre -1 et 1, selon la formule de la similarité cosinus :

$$\text{similarité cosinus} = \frac{v_1 \cdot v_2}{\|v_1\| \times \|v_2\|}$$

— Si les vecteurs sont dans la même direction (c'est-à-dire que l'angle entre eux est de 0), la similarité cosinus sera 1.

— Si les vecteurs sont orthogonaux (l'angle entre eux est de 90 degrés), la similarité sera 0.

— Si les vecteurs sont opposés (l'angle entre eux est de 180 degrés), la similarité sera -1.

2.2.3.2 Mise en œuvre du service

De cette manière, le service a comme valeur d'entrée la chaîne à rechercher à partir de laquelle son *embedding* est obtenue et il est comparée au `embedding.json` généré à partir des documents avec une similarité cosinus, obtenant un `score` pour que les résultats soient affichés en fonction de le `score` par ordre décroissant. Voici un extrait de code montrant les étapes principales :

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from sentence_transformers import SentenceTransformer
import json

# Initialisation du modele d'embedding
model = SentenceTransformer("paraphrase-MiniLM-L6-v2")

# Initialisation de l'application FastAPI
app = FastAPI()

# Modele de donnes pour l'entree
class SearchRequest(BaseModel):
    query: str

class SearchResult(BaseModel):
    id: int
    text: str
    similarity_score: float

class SearchResponse(BaseModel):
    relevant_documents: List[SearchResult]

@app.post("/search_documents", response_model=SearchResponse)
async def search_documents(request: SearchRequest):
    """
    Search for relevant documents based on a query.
    """
    try:
        # Load stored embeddings
        documents_data = load_embeddings()
        if not documents_data:
            raise HTTPException(status_code=404, detail="No documents have been encoded yet")

        # Encode the query
```

```
        query_embedding = embedding_model.encode(request.query)

        # Calculate similarities and sort documents
        results = []
        for doc in documents_data:
            similarity = cosine_similarity(query_embedding, doc['embedding'])
            results.append(SearchResult(
                id=doc['id'],
                text=doc['text'],
                similarity_score=float(similarity)
            ))

        # Sort by similarity score in descending order
        results.sort(key=lambda x: x.similarity_score, reverse=True)

        # Return top 5 most relevant documents
        return SearchResponse(relevant_documents=results[:5])

    except Exception as e:
        logger.error(f"Error searching documents: {str(e)}")
        raise HTTPException(status_code=500, detail=str(e))
```

Listing 4 – Implémentation du service de recherche et similarité de documents

2.2.4 Détails Techniques

- **Input de la requête :** La requête est envoyée sous forme d'objet JSON contenant la chaîne de recherche, permettant une interface simple et flexible pour l'utilisateur.
- **Calcul de similarité :** La similarité entre la requête et les documents est calculée à l'aide de la fonction de similarité cosinus, implémentée avec `numpy`, garantissant des calculs rapides et précis.
- **Gestion des erreurs :** Le service inclut une gestion des erreurs robuste qui renvoie des messages d'erreur appropriés en cas de problèmes, assurant une bonne expérience utilisateur.
- **Limite des résultats :** Pour optimiser la performance, le service retourne uniquement les 5 documents les plus pertinents, ce qui réduit le temps de réponse et la charge de traitement.

2.2.5 Avantages de l'approche

- **Rapidité :** Le service offre des réponses rapides grâce à des algorithmes optimisés, notamment en utilisant `numpy` pour des calculs vectoriels efficaces.
- **Flexibilité :** La méthode peut traiter différents types de requêtes sans nécessiter de modifications dans la structure des données.
- **Précision :** L'utilisation de la similarité cosinus permet d'obtenir des résultats hautement pertinents, améliorant ainsi la satisfaction de l'utilisateur.
- **Robustesse :** La gestion des erreurs permet au système de fonctionner de manière fiable même en cas de données manquantes ou incorrectes.

2.3 Service de Chatbot

Le service de chatbot est conçu pour permettre à un utilisateur d'interagir avec le système de manière conversationnelle. Il utilise un modèle de génération, ici **Google Gemini**, associé à des résultats pertinents issus d'une recherche documentaire basée sur des embeddings. Ce service applique une approche de génération augmentée par récupération (Retrieval-Augmented Generation, RAG) en exploitant des documents encodés et une recherche sémantique préalable.

2.3.1 Description du processus

Le service est implémenté sous la route API suivante :

- **Route :** `POST /chat`
- **Entrée :** Une liste de messages représentant une conversation. Le dernier message de l'utilisateur est utilisé comme requête principale.
- **Sortie :** Une réponse générée par le modèle Gemini, accompagnée des sources documentaires utilisées pour formuler la réponse.

Les étapes principales du traitement sont :

1. **Extraction des messages utilisateur :** À partir de la liste des messages, le dernier message envoyé par l'utilisateur est identifié comme la requête principale.
2. **Recherche documentaire :** La requête est utilisée pour rechercher les documents pertinents à l'aide de similarité cosinus sur les embeddings existants.
3. **Génération de réponse :**
   — Les documents les plus pertinents sont utilisés pour construire un contexte.
   — Ce contexte est intégré dans une invite (prompt) pour le modèle Gemini, qui génère une réponse basée uniquement sur ce contexte.
4. **Retour des résultats :** La réponse du modèle et les documents sources utilisés sont renvoyés au client.

2.3.2 Exemple de Requête et Réponse

Exemple de requête :

```
POST /chat HTTP/1.1
Content-Type: application/json

{
    "messages": [
        {"role": "user", "content": "Peux-tu expliquer les avantages des embeddings ?"}
    ],
    "temperature": 0.7,
    "max_relevant_chunks": 3
}
```

Exemple de réponse :

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "response": "Les embeddings permettent de représenter les textes sous forme de vecteurs
    numériques. Ils facilitent les comparaisons sémantiques,
    comme la recherche de similarité ou le clustering de documents.",
    "sources": [
        {
            "text": "Les embeddings sont des vecteurs qui capturent le sens sémantique des textes.",
            "similarity": 0.92
        },
        {
            "text": "Ils permettent d'effectuer des recherches rapides et précises dans de grands cor
            "similarity": 0.88
        }
    ]
}
```

2.3.3 Code d'implémentation

Le service est implémenté en Python en utilisant `FastAPI` et le modèle Google Gemini. Voici une implémentation détaillée :

```python
@app.post("/chat", response_model=ChatResponse)
async def chat(request: ChatRequest):
    """
    Service de chatbot utilisant la generation augmentee par recuperation (RAG)
    et le modele Google Gemini.
    """
```

```
    try:
        # Extraction du message utilisateur
        user_message = next(
            (msg.content for msg in reversed(request.messages) if msg.role == "user"),
            None
        )

        if not user_message:
            raise HTTPException(
                status_code=400,
                detail="Aucun message utilisateur trouve dans la conversation"
            )

        # Recherche des documents pertinents
        search_results = await search_documents(SearchRequest(query=user_message))
        relevant_docs = [
            {
                'text': doc.text,
                'similarity': doc.similarity_score
            }
            for doc in search_results.relevant_documents[:request.max_relevant_chunks]
        ]

        if not relevant_docs:
            return ChatResponse(
                response="Je n'ai pas trouve de documents pertinents pour repondre a votre question. "
                         "Pourriez-vous reformuler ou poser une autre question ?",
                sources=[]
            )

        # Construction du contexte pour le modele Gemini
        context = "Contexte pertinent :\n\n"
        for i, doc in enumerate(relevant_docs, 1):
            context += f"{i}. {doc['text']}\n"

        prompt = f"""{context}\n
En utilisant uniquement le contexte ci-dessus, reponds a la question suivante.
Si tu ne peux pas repondre completement avec le contexte donne, dis-le honnetement.

Question: {user_message}

Reponse:"""

        # Generation de reponse avec le modele Gemini
        generation_config = {
            "temperature": request.temperature,
            "top_p": 1,
            "top_k": 1,
            "max_output_tokens": 2048,
        }
        safety_settings = [
            {"category": "HARM_CATEGORY_HARASSMENT", "threshold": "BLOCK_MEDIUM_AND_ABOVE"},
            {"category": "HARM_CATEGORY_HATE_SPEECH", "threshold": "BLOCK_MEDIUM_AND_ABOVE"},
            {"category": "HARM_CATEGORY_SEXUALLY_EXPLICIT", "threshold": "BLOCK_MEDIUM_AND_ABOVE"},
            {"category": "HARM_CATEGORY_DANGEROUS_CONTENT", "threshold": "BLOCK_MEDIUM_AND_ABOVE"},
        ]

        response = model.generate_content(
            prompt,
            generation_config=generation_config,
            safety_settings=safety_settings,
        )

        return ChatResponse(
            response=response.text,
            sources=relevant_docs
        )

    except Exception as e:
        logger.error(f"Erreur dans le service de chatbot : {str(e)}")
        raise HTTPException(status_code=500, detail=str(e))
```

Listing 5 – Implémentation du service de recherche et similarité de documents

2.3.4 Détails Techniques

— **Modèle de génération :** Le service utilise le modèle Google Gemini pour produire des réponses textuelles basées sur un contexte.

— **Recherche documentaire :** Les documents pertinents sont identifiés à l'aide de similarité

cosinus entre les embeddings de la requête utilisateur et ceux des documents stockés.

- **Contexte :** Les documents les plus pertinents (limités par `max_relevant_chunks`) sont concaténés pour former un contexte utilisé dans l'invite (prompt) du modèle de génération.
- **Paramètres de génération :**
  - **Temperature :** Contrôle la créativité des réponses.
  - **Top-p / Top-k :** Détermine la probabilité cumulative et le nombre maximum de candidats pour chaque token généré.
  - **Max Output Tokens :** Limite la longueur de la réponse générée.

3 Interface Utilisateur

L'interface utilisateur a été développée pour offrir une interaction simple et intuitive avec les services API. Elle repose sur une application graphique construite avec la bibliothèque `Tkinter`. Elle est divisée en trois onglets principaux :

- **Recherche :** Cet onglet permet à l'utilisateur de soumettre une requête textuelle et d'obtenir une liste des documents les plus pertinents, classés par score de similarité. Les résultats sont présentés sous forme de texte, avec des détails sur le score et le contenu de chaque document.
- **Chat :** Dans cet onglet, l'utilisateur peut interagir de manière conversationnelle avec le système. Les réponses générées par le système sont affichées dans une zone de discussion, accompagnées des sources pertinentes lorsqu'elles sont disponibles.
- **Documents :** Cet onglet affiche une vue d'ensemble des documents encodés, avec des statistiques sur le traitement des données :
  - Nombre total de documents encodés.
  - Nombre de documents encodés avec succès.
  - Nombre de documents échoués.
  - Taux de succès global.

Cette interface masque la complexité des calculs sous-jacents et fournit une expérience utilisateur fluide et accessible, même pour des utilisateurs non techniques.

Les figures ci-dessous présentent des captures d'écran de l'interface utilisateur, illustrant les fonctionnalités principales :

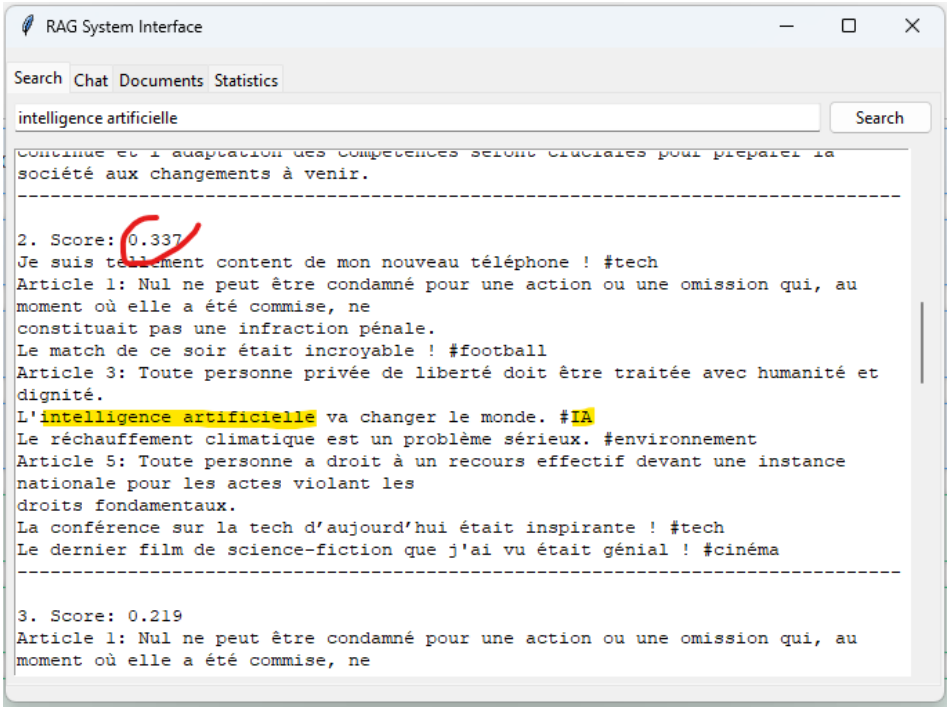

FIGURE 3 – **Onglet Recherche** : Affichage des résultats d'une requête textuelle soumise par l'utilisateur. Les documents correspondants sont classés par score de similarité décroissant. Les métadonnées affichées incluent le score et un extrait du contenu pour chaque document.

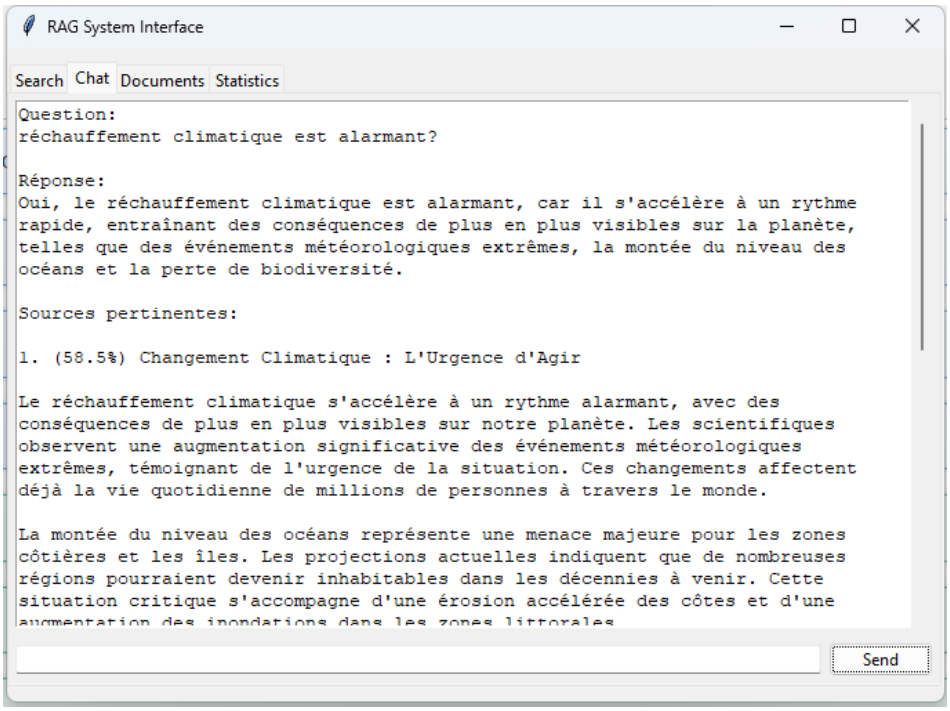

FIGURE 4 – **Onglet Chat** : Exemple d'interaction conversationnelle avec le système. Les réponses générées incluent des informations contextuelles, et lorsque possible, les sources pertinentes sont ajoutées pour justifier les réponses fournies.

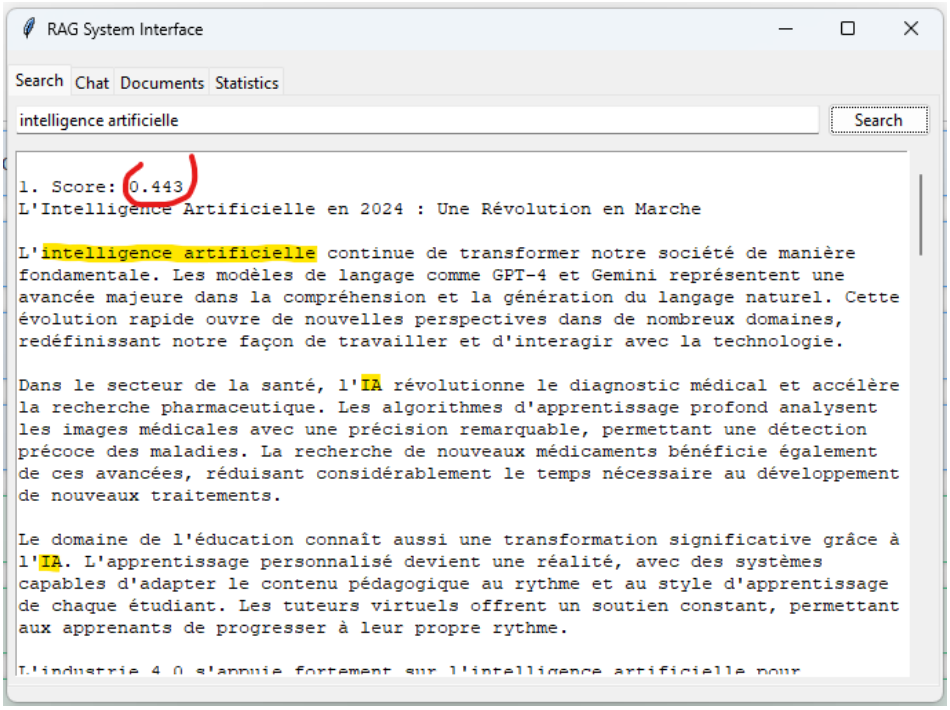

FIGURE 5 – **Onglet Documents** : Vue d'ensemble des statistiques liées au traitement des documents encodés. Les indicateurs affichés incluent le nombre total de documents encodés, le nombre de succès, les échecs, et le taux de réussite global.

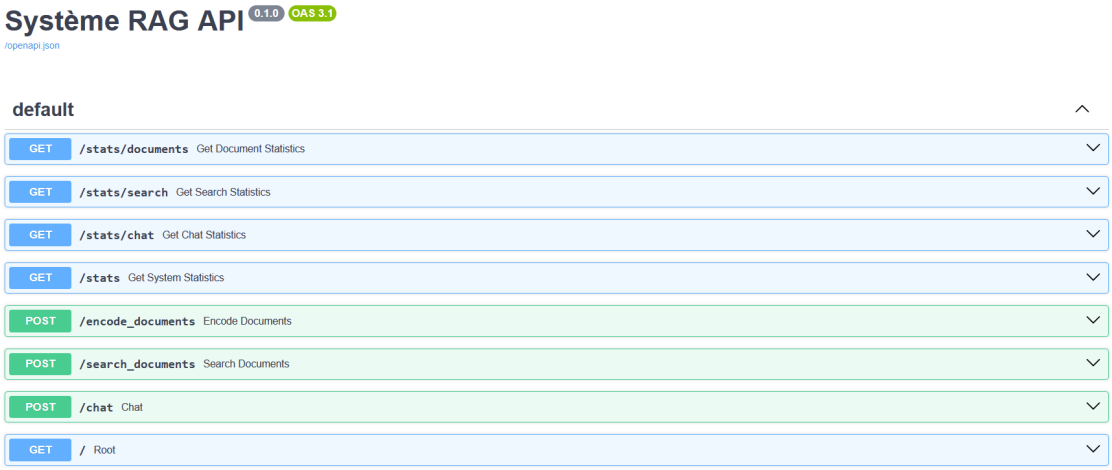

FIGURE 6 – **Documentation des Endpoints API** : Cette capture d'écran montre l'ensemble des endpoints disponibles dans le système. Elle présente les routes existantes, leurs descriptions, ainsi que les méthodes HTTP associées (GET, POST, etc.), facilitant l'intégration du système avec d'autres applications.

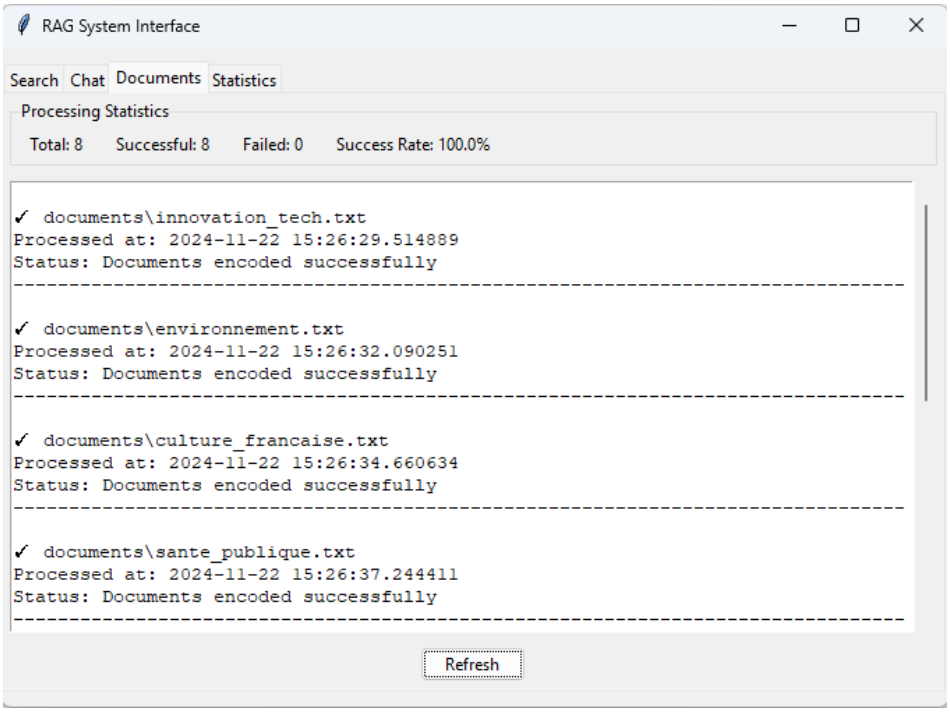

FIGURE 7 – **Page de confirmation dans l'onglet Documents** : Cette capture d'écran montre un encodage réussi. L'interface confirme le bon traitement des documents et met en avant les statistiques liées à l'encodage.

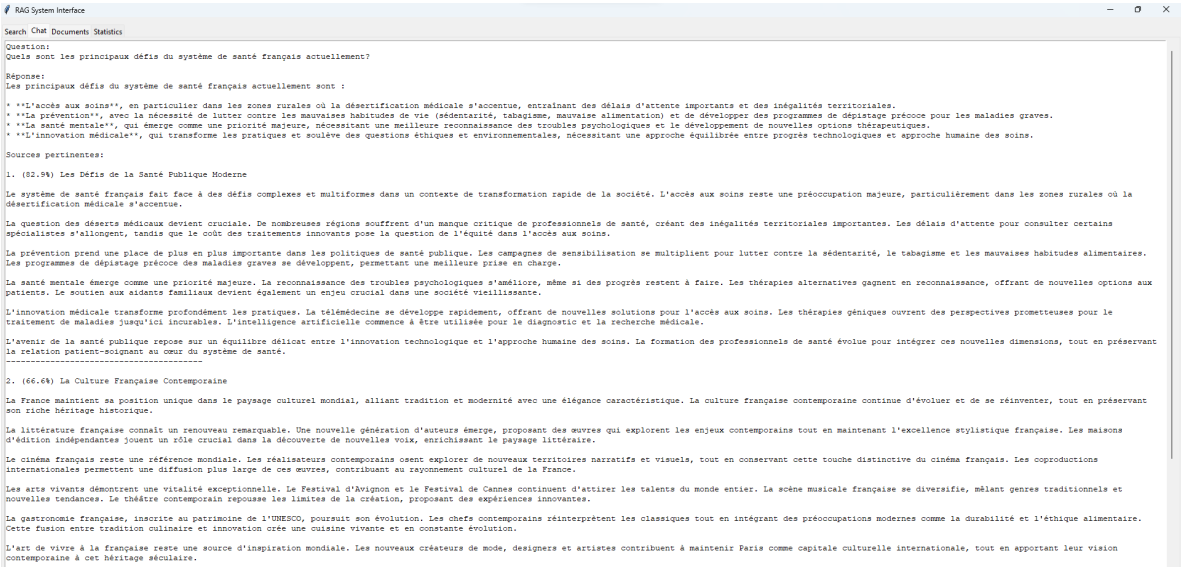

FIGURE 8 – **Onglet Chat** : Exemple d'interaction conversationnelle où l'utilisateur pose une question ("Quels sont les principaux défis du système de santé français actuellement ?"). Le système fournit une réponse complète et structurée, mettant en évidence les informations principales, et liste également les sources pertinentes avec leur score de pertinence. Cette fonctionnalité est particulièrement utile pour naviguer dans de grandes quantités de données textuelles et obtenir des réponses précises et contextualisées.