Rapport Technique: Système RAG avec FAISS et Mistral AI

1. Introduction

Ce rapport présente l'architecture et l'implémentation d'un système de Retrieval-Augmented Generation (RAG) utilisant FAISS pour l'indexation vectorielle et Mistral AI pour les embeddings et la génération de réponses. Le système est conçu pour gérer efficacement une base de données d'événements publics et répondre aux questions des utilisateurs en français.

1.1 Contexte du Projet

Le projet vise à créer un système de question-réponse intelligent capable de traiter des données d'événements publics en français. L'objectif principal est de fournir des réponses précises et contextuelles aux utilisateurs en s'appuyant sur une base de données vectorielle performante.

1.2 Objectifs

- Développer un système RAG performant en français
- Intégrer efficacement les données d'OpenDataSoft
- Assurer une recherche rapide et précise
- Maintenir un historique des conversations
- Fournir une base extensible pour les futures améliorations

2. Architecture du Système

2.1 Vue d'ensemble

Le système est structuré en trois modules principaux:

1. Module de gestion des documents (`documents.py`):

- Récupération des données depuis OpenDataSoft
- Nettoyage et prétraitement des textes
- Transformation en format Document Langchain

2. Module de chat et recherche vectorielle (`chat.py`):
- Gestion de la base vectorielle FAISS
- Traitement des requêtes utilisateur
- Génération des réponses via Mistral AI
- Maintenance de l'historique des conversations

3. Module de tests (`test_chat.py`):
- Tests unitaires
- Validation des fonctionnalités
- Vérification des performances

2.2 Flux de données

Le système suit un flux de données structuré:

1. Collecte des données:
- Requêtes API vers OpenDataSoft
- Pagination automatique
- Gestion des limites de rate

2. Prétraitement:
- Nettoyage HTML avec BeautifulSoup
- Normalisation du texte
- Structuration des métadonnées

3. Indexation:
- Génération des embeddings
- Construction de l'index FAISS
- Optimisation des paramètres

4. Recherche et réponse:
- Analyse de la requête
- Recherche vectorielle
- Génération de réponse contextuelle

3. Choix Techniques

3.1 Base Vectorielle: FAISS

FAISS (Facebook AI Similarity Search) a été choisi pour plusieurs raisons:

3.1.1 Caractéristiques techniques

- Index IVFFlat pour un compromis optimal entre vitesse et précision
- Dimension des vecteurs: 1024 (optimisé pour Mistral Embed)
- Configuration:
  - N_CELLS = 10 (nombre de cellules Voronoi)
  - N_PROBE = 5 (nombre de cellules à explorer)

3.1.2 Avantages

- Performances exceptionnelles sur grands volumes
- Optimisation GPU possible
- Faible empreinte mémoire
- Maintenance simple

3.2 Modèles d'IA

3.2.1 Mistral AI

Choix motivé par:
- Support natif du français
- Performances élevées
- Rapport qualité/coût optimal

3.2.2 Configuration des modèles

1. Embeddings (mistral-embed):
   - Dimension: 1024
   - Optimisé pour le français
   - Performances stables

2. Génération (mistral-medium):
   - Temperature: 0.7
   - Langue: français
   - Contexte: 8k tokens

3.3 Framework LangChain

LangChain apporte plusieurs avantages:

3.3.1 Fonctionnalités clés

- Gestion unifiée des documents
- Intégration LLM simplifiée
- Système de mémoire conversation
- Chaînes de traitement flexibles

3.3.2 Composants utilisés

- ConversationBufferMemory
- ChatPromptTemplate
- StateGraph
- FAISS Vectorstore

4. Implémentation Détaillée

4.1 Gestion des Documents

4.1.1 Nettoyage des données

```
def clean_text(text):
    text = BeautifulSoup(text, "html.parser").get_text()
    text = text.lower()
    text = re.sub(r'[^\w\s.,!?;:\'\"À-ÿ]', ' ', text)
    text = ' '.join(text.split())
    return text
```

4.1.2 Structure des documents

```
Document(
    page_content=content,
    metadata={
        "source": "opendatasoft",
        "id": row["uid"],
```

```
        "title": clean_text(row["title_fr"]),
        "description": row["description_fr"],
        # ... autres métadonnées ...
    }
)
```

4.2 Système de Recherche

4.2.1 Architecture du graphe d'états

Le système utilise un graphe d'états avec trois nœuds principaux:

1. Search Node:

```
def _search_node(self, state: State) -> State:
    results = self.vector_store.similarity_search(state["query"])
    state["results"] = results if results else []
    return state
```

2. Memory Node:

```
def _process_memory_node(self, state: State) -> State:
    self.memory.save_context(
        {"input": state["query"]},
        {"output": result_text}
    )
    return state
```

3. Results Node:

```
def _process_results_node(self, state: State) -> State:
    response = self.doc_chain.invoke(chain_input)
    state["response"] = {
        "current_answer": response,
        "chat_history": chat_history,
        "context": {"total_interactions": len(chat_history)}
    }
    return state
```

4.3 Optimisations

4.3.1 Index FAISS

```
def create_optimized_index(dimension, embeddings, documents,
n_lists=100):
    quantizer = faiss.IndexFlatL2(dimension)
    index = faiss.IndexIVFFlat(quantizer, dimension, n_lists)
    vectors = embeddings.embed_documents(texts)
    index.train(vectors)
    return index, vectors
```

4.3.2 Gestion de la mémoire

- Sauvegarde JSON des conversations
- Nettoyage périodique
- Limitation de taille

5. Résultats du POC

5.1 Tests Unitaires

Le système a été validé par une suite de tests unitaires couvrant:

5.1.1 Couverture des tests

- Création d'index FAISS
- Ajout de documents
- Traitement des requêtes
- Gestion de la mémoire
- Intégration OpenDataSoft

5.1.2 Résultats des tests

```
class TestFaissVectorStore(unittest.TestCase):
    def test_index_creation(self):
        index, vectors = create_optimized_index(...)
        self.assertIsNotNone(index)

    def test_query_processing(self):
        response = faiss_vector_store.process_query(query)
        self.assertIn("current_answer", response["response"])
```

5.2 Performances

5.2.1 Métriques clés

- Temps de réponse moyen: < 2 secondes
- Précision des réponses: satisfaisante
- Usage mémoire: optimisé
- Latence réseau: minimale

5.2.2 Limitations identifiées

- Scalabilité limitée
- Pas de haute disponibilité
- Monitoring basique

6. Conclusion

Le POC démontre la viabilité du système RAG avec FAISS et Mistral AI. Les performances et la qualité des réponses sont satisfaisantes pour une version initiale. Les recommandations proposées permettront d'évoluer vers une solution production-ready.

6.1 Points forts

- Architecture modulaire et extensible
- Performances satisfaisantes
- Support natif du français
- Base solide pour évolution

6.2 Prochaines étapes

1. Implémentation des recommandations prioritaires
2. Tests de charge
3. Déploiement pilote
4. Documentation utilisateur

7. Références

1. Documentation FAISS
2. API Mistral AI
3. LangChain Documentation
4. OpenDataSoft API Reference

Guole YANG 27/01/2025