A Detailed Methodology for a RAG Pipeline for Fact-Checking Marketing Claims

Rahul Manikandan

April 12, 2025

Contents

1 Introduction

This document details the methodology implemented in a Retrieval-Augmented Generation (RAG) pipeline designed to fact-check marketing claims against clinical document evidence. The pipeline leverages LangChain modules to load and preprocess clinical PDFs, splits the documents into manageable chunks, computes semantic embeddings, builds a FAISS vector index, and finally uses a retrieval chain to extract supporting evidence. In this report, we compare the RAG approach to a basic TF-IDF–based matching system, examine model size considerations, downloading times, and discuss some observed results such as truncated sentences in the JSON output.

2 Data Inputs and Setup

2.1 Data Sources

- **Marketing Claims:** Stored in a JSON file (e.g., `Flublok_Claims.json`) where each claim is a short textual assertion about a vaccine.
- **Clinical Documents:** A set of clinical PDFs stored in `data/Clinical Files/` that contain detailed studies, trial data, and regulatory information.

2.2 Project Directory Organization

The repository structure for the RAG pipeline is organized as follows:

```
solstice-fact-check/
          data/
                Clinical Files/              % Clinical PDFs
                Flublok_Claims.json          % Marketing claims JSON
          notebooks/
                main1.ipynb                  % Notebook for basic TF-IDF matchin
                main_rag.ipynb               % Notebook for the RAG pipeline exp
          src/
                __init__.py
                preprocess.py                % Loading and cleaning PDFs and cla
                matcher.py                   % Basic matching logic (for TF-IDF
                utils.py                     % Helper functions (e.g., logger, J
                rag/                         % Modules for the RAG pipeline
                     __init__.py
                     chunking.py            % Splitting text into overlapping ch
                     embedding.py           % Generating embeddings via Gemini c
                     retrieval.py           % Building a FAISS index and similar
                     generation.py          % (Optional) Generation module to sy
          requirements.txt                 % All required libraries and framewor
          results/
                ...                          % Output JSON files with matching e
          README.md
```

2.3 Environment and Dependencies

The key dependencies are listed in `requirements.txt`. For instance, the file may include:

```
pymupdf==1.22.3
scikit-learn==1.2.2
numpy==1.24.2
faiss-cpu==1.7.3
jupyter==1.0.0
langchain==0.0.148
sentence-transformers==2.2.2
transformers==4.31.0
google-genai==0.2.0
```

These libraries support PDF extraction, text splitting, embedding generation, vector search, and LLM inference via LangChain.

3 Methodology

3.1 Preprocessing and Chunking

Preprocessing: Clinical PDFs are processed using PyMuPDF in `preprocess.py` to extract text. The process includes cleaning methods (removing extraneous headers, URLs, and whitespace).

Chunking: Due to the large size of clinical documents, the text is split into smaller overlapping chunks using a function in `src/rag/chunking.py`. For example, each chunk may consist of 1000 characters with a 200-character overlap. This ensures that important context isn't lost at the boundaries.

3.2 Embedding Generation

Embeddings: Numerical embedding vectors are generated for each text chunk (and for each marketing claim) using the `HuggingFaceEmbeddings` class. In our RAG pipeline, the embedding model `sentence-transformers/all-MiniLM-L6-v2` is used.

- **Model Size:** This model is relatively lightweight (around 100–200 MB) compared to larger models, which leads to faster downloading and inference times. However, even with a smaller model, generating hundreds of embeddings can be time-consuming.
- **Downloading Time:** The first run may download the model from Hugging Face, which can take a few minutes depending on your network speed. Subsequent runs will use the locally cached version.

3.3 Building the Vector Store

FAISS Index: With the computed embeddings for all clinical chunks, a FAISS index is built (see `retrieval.py`). FAISS quickly performs similarity searches, enabling retrieval of the top-K related chunks for a given claim embedding.

3.4 Retrieval and Matching

RetrievalQA Chain: For each marketing claim, its embedding is computed and the FAISS index is queried to retrieve the most relevant clinical chunks. A RetrievalQA chain in LangChain uses a prompt template to instruct the LLM to extract evidence directly from the retrieved context. For example, the prompt template might say:

```
prompt_template = """
You are a fact-checking assistant. Your task is to find the sentences, tables
in the context that best support the following claim.

Claim: {question}

Context:
{context}

Identify and return only the exact text from the context that supports the cl
"""
```

Input Keys: The chain is configured to expect input with key `"question"` (instead of `"query"`), so when making calls, the proper key must be provided.

3.5 Output JSON and Evidence Truncation

The final output is an aggregated JSON where each claim is associated with a list of matching evidence:

- **Document Name:** Indicates the source file.
- **Matching Text:** A snippet (typically the first 200 characters) from the chunk where relevant evidence is found.
- **Truncated Sentences:** Often, the snippet may not show a complete sentence due to its fixed-length truncation. This is a trade-off between readability and brevity, ensuring that each snippet is short yet indicative of the supporting content.

In comparison, your basic approach used TF-IDF similarity which generally returned lower similarity scores. In contrast, the RAG pipeline with semantic embeddings tends to provide richer, contextually aligned evidence even if some snippets are truncated.

3.6 Computational Considerations

- **Processing Time:** Generating embeddings, building the FAISS index, and performing retrieval for multiple claims can be time intensive—especially if run on a CPU.
- **Hardware Constraints:** Model size and available memory (e.g., GPU vs. CPU) greatly affect inference speed. Larger models or low-memory environments may result in slower processing.

4 Results and Comparison

4.1 RAG Pipeline Output JSON

The RAG pipeline produced a JSON output where, for each claim, multiple supporting sources are listed with snippets. For example:

- **Claim:** "Flublok ensures identical antigenic match with WHO- and FDA-selected flu strains."
  Matches: Snippets from `FlublokPI.pdf` and `Treanor et al. (2011).pdf` detailing percentages and evaluation results.

Some output snippets are truncated (only 200 characters), which is expected to maintain brevity. If further completeness is needed, the truncation parameter can be adjusted.

4.2 Comparison with Basic Approach

- **Basic Approach:** Utilized TF-IDF and cosine similarity. It generally returned lower similarity scores because TF-IDF is based on word frequency and does not capture semantic meaning in depth.
- **RAG Pipeline:** Uses semantic embeddings (e.g., via `sentence-transformers/all-MiniLM-L6-v2`) which capture contextual meaning and provide more robust retrieval. As a result, the scores and retrieved evidence better reflect the nuanced relationship between the claims and clinical evidence, even if snippet outputs might occasionally be truncated.

5 Conclusion

In summary, the RAG pipeline enhances the fact-checking process by:

1. Extracting and processing clinical text from PDFs.
2. Splitting documents into overlapping chunks for context preservation.
3. Leveraging semantic embeddings to accurately capture document meaning.
4. Building a FAISS index that allows efficient similarity search.
5. Using a RetrievalQA chain with a custom prompt to retrieve evidence that supports each marketing claim.

Key challenges include managing large model sizes (which affect download and inference time) and handling truncated outputs due to fixed snippet lengths. Compared to the basic approach, the RAG pipeline provides richer, more context-aware evidence, reflected in higher and more meaningful similarity scores, though some post-processing (e.g., snippet extraction) may require further tuning.

6 Appendix: Relevant Python Notebook Code Overview

Cell-by-Cell Summary

1. **Environment Setup:** Importing necessary packages and adding the project root to `sys.path`.
2. **Data Loading:** Loading clinical documents using LangChain's `DirectoryLoader` and marketing claims from JSON.
3. **Document Chunking:** Splitting PDF text into overlapping chunks using a `RecursiveCharacterTex`
4. **Embedding Generation:** Generating embeddings for the chunks with `HuggingFaceEmbeddings`.
5. **Vector Store Construction:** Building a FAISS index from chunk embeddings for efficient similarity search.
6. **Retrieval Chain Setup:** Configuring a RetrievalQA chain with a custom prompt template that instructs the LLM to extract supporting evidence.
7. **Claim Processing:** Iterating through each claim, computing its embedding, retrieving the top matching chunks, and aggregating the results.
8. **Result Saving:** Saving the aggregated output as a JSON file.

Future Improvements

Possible directions for enhancing the pipeline include:

- Exploring alternative embedding models (e.g., larger or specialized models) to improve retrieval quality.
- Adjusting chunk size and overlap parameters for optimal context preservation.
- Implementing post-retrieval summarization or answer synthesis through an LLM (using `generation.py`).
- Fine-tuning prompt templates to better guide the model in extracting complete sentences.

Acknowledgements

This work builds on methods developed for fact-checking in clinical contexts and leverages open-source tools including LangChain, FAISS, and HuggingFace Transformers.