Project TLDR - Capstone Report - 2024 Autumn

Project Title: Project TLDR: Standalone Desktop application for Question-Answering and Summary using resource efficient LLMs

Student: Manu Hegde

Project Committee:
Prof. Erika Parsons, Committee Chair
Prof. Michael Stiber, Committee Member
Prof. Shane Steinert-Threlkeld, Committee Member

Table of contents:

I. Overview

This project aims to develop a standalone desktop application that enables ChatGPT-like Question-Answering and Summarization on top of a corpus of documents on the user's device. The application shall embody a resource efficient implementation of a chosen LLM (Large Language Model), targeting Apple's M1/M2 hardware platform. The primary intended user base for this application is students and researchers in academia.

The motivation behind this project stems from the growing need for tools that can efficiently process and interpret large volumes of academic and research material. Current solutions often require significant manual interventions or involve sharing data with third-party servers, raising concerns about privacy and data security. By creating a resource-efficient, standalone application, this project aims to provide students and researchers with a tool that offers convenience, confidentiality, and enhanced productivity. The project considers recent advancements in natural language processing (NLP), particularly the use of large language models (LLMs) for tasks like summarization and question-answering. The application will utilize techniques like weight quantization and low-rank adaptation to optimize LLM performance on Apple's M1/M2 architecture, including the use of Apple Metal GPU and Apple Neural Engine (ANE) for hardware acceleration. The project will incorporate retrieval-augmented generation to yield contextually relevant outputs derived from text corpus provided by the user, thereby ensuring credibility of the information.

The expected contributions of this project include the development of a desktop application with a graphical user interface, capable of processing and summarizing large text corpora locally, without requiring an internet connection. The application will also explore the potential for running complex NLP models in resource-constrained environments, offering insights into optimizing LLMs for specific hardware platforms. Ultimately, this project aims to provide a valuable tool for researchers and students, enhancing their ability to interact with and understand extensive collections of academic materials.

In conclusion, the project aims to create a standalone desktop application that uses a Large Language Model for document-based Q&A, with minimal and predictable resource usage for smooth multitasking on the user's device.

II. Literature Review

A. Ingredients of RAG (Retrieval Augmented Generation) Application:

A Retrieval Augmented Generation system contains four major components: ***Document loader***, ***Text embedder***, ***Context retriever*** and ***Language model***. As illustrated in Figure 1.1, the documents are embedded and stored in a vector database which is queried to obtain the relevant context for the user's query and passed along to the LLM. LLM hence able to generate outputs based on a particular source of data, improving the credibility and usefulness of the output.

The RAG process could be divided into two phases: Embedding, Retrieval. The Embedding phase (as illustrated by Figure 1.2), consists of loading, chunking and embedding the document. On the other hand, the Retrieval phase deals with vector similarity search to fetch relevant context vectors in their original text form and pass them along with user query to the Large Language Model, as seen in Figure 1.3.

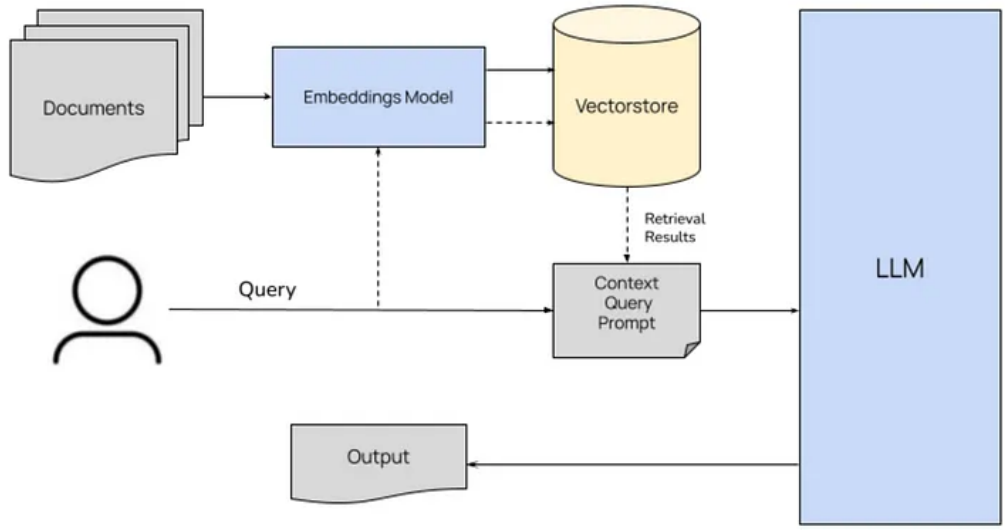

Figure 1.1 – Architecture of a RAG Application [1]

a) Embedding phase: In this phase we build our text corpus that can later be used as the knowledge base to answer user's queries. Although the concept of text embeddings has been since the introduction of Word2Vec embeddings [3] in 2013, their use for storing and retrieving contextual information is a recent innovation. This came about with the introduction of 'In context learning' technique as mentioned in the GPT3 paper by OpenAI in 2020 [4]. This technique particularly works well for decoder-only transformers due to their purely auto-regressive nature. Due to this, one can obtain relevant answers on a task or topic with just a few or even just one example or reference [5].

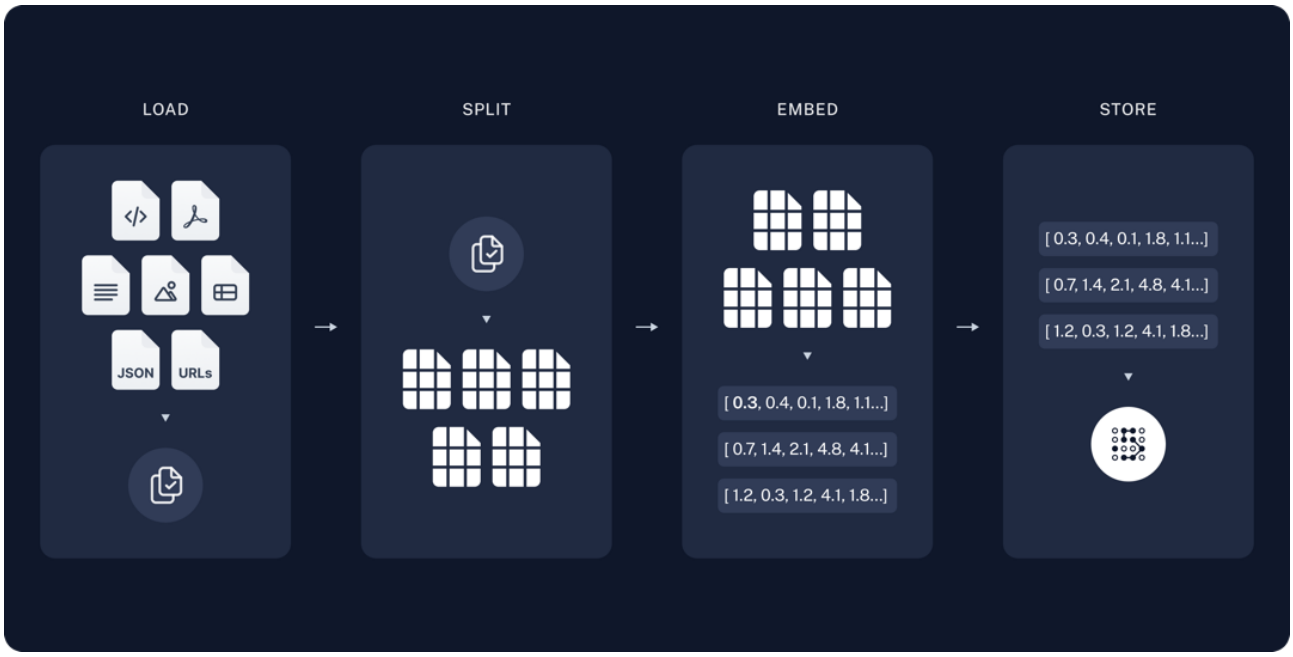

Figure 1.2: Illustration of the Embedding Phase [2]

The embedding phase consists of the following steps:

i. **Document processing:** In this phase we load the document from its source format (txt, pdf, docx). This could be done in either text-only mode or multi-modal mode. Where the latter allows for the preservation and usage of diagrams and images present in the source document.

The ***Document Loader*** component oversees this phase and performs *loading* and *chunking*. The latter step is handled by text-splitters [6] which split the data based on preset configuration.

Why do we need chunking? Quick answer, LLM Context-Length and limited GPU RAM.
LLMs have a limited context length, i.e. the amount of text that can be processed in one shot. This ranges from 2K (i.e. 2000) for GPT3, 4K for Llama2 up to 128K for Llama 3 [7]. Although a context size like 128K, to leveraging large context sizes also requires large amounts of GPU memory to store the model weights and attention.
This need for memory increases further when techniques like kv-caching (key & value caching of attention heads) and speculative decoding are implemented to improve output generation speed.

Hence, it is important to ensure only relevant and limited text is passed on as context for a user query. To do so, the document needs to be split into chunks.

Although splitting the text is straightforward, determining the split locations is crucial and should be considered carefully due to the following reasons.

- If the text chunk is too large, there could be a loss of knowledge due to 'Lost in the middle' problem [8] i.e. only the information at the begging and end of chunk gets used effectively.
- If the text chunk is too small, it could result high redundancy of data, since chunks often maintain overlapping content to preserve context. Additionally, too many chunks also result in greater latency of the system.

Hence, determining the chunk size is extremely crucial to ensure effectiveness of the RAG System. Chunking can be done by following techniques:
1. Length based chunking: Chunking the text based on a fixed pre-determined size. The size can be expressed in terms of token-level or character level. A token, yielded after sub-word tokenization of input text using byte-pair-encoding [9] can range from a character to sub-word to even an entire phrase (common multi word sequences).

2. Semantic structure-based chunking: Splitting the test based on the structure of the document i.e. converting paragraphs or blocks of text into a single chunk.

3. Context aware splitting: Splitting using textual context i.e. retaining an entire sub-context or sub-topic inside a single chunk.

Note: In case of token length-based chunking or context aware chunking, the embedding step would precede splitting.

ii. Text Embedding: In this phase the **Text Embedder** tokenizes and obtains embeddings for the text. This embedding need not be same as the one used by the LLM since these embeddings are used only to perform vector search. Once all the chunks relevant to a user query are obtained, their corresponding text value is returned. We do not pass on the embedding values to the LLM. Models like BERT / DistilBERT are often used for this purpose since they are faster to run and generate smaller embeddings (ex: 512 or 768) unlike LLMs like LLaMa [10] or GPT3 [4] which use embeddings with size between 1024-12,288. While smaller embeddings are less accurate, they are efficient for the purpose of high-level context retrieval.

b) Retrieval Phase: This phase happens every time a user inputs a question or task to the model. As Illustrated in Figure 1.3, this phase utilizes the vector database created during the embedding phase and retrieves all content relevant to the user's input prompt and passes it on to the LLM.

The phase involves the following steps:

i. **Context Retrieval:** This step is carried out by the ***Context Retriever*** in the following sequence.
   a. Embed the user's input using the same language model used during the embedding phase
   b. Perform similarity search on the vector database using the embedded user input

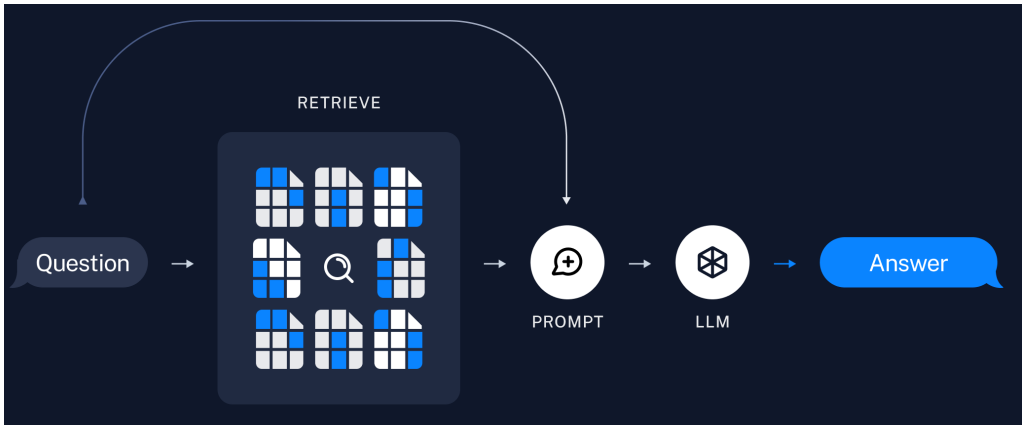

Figure 1.3: Illustration of the Retrieval Phase [2]

The Similarity Search on the database is the most crucial step since it determines the relevance and quality of the output generated. The similarity in the embeddings space is attributable to neighborliness. Points closer to each other are similar than those which are distant. Finding neighbors brings about a classic trade-off between resource available, latency and output quality. While performing complete scan of the database could obtain the best result, it is extremely costly to do so [11].

Hence, to balance the trade-offs, we may use one of the following algorithms:

1. kNN: k nearest neighbors is one of the most popular and simplest algorithms to find points in vector space that are closest to a given point. But this involves recursive iterations over vector space [12].

Approximate Nearest Neighbor algorithms:

2. Locality-Sensitive Hashing (LSH): It leverages multiple cryptographic, 1-way hash functions to hash vectors into hash buckets. The search query is then subjected to the same hashing process and the contents of the hash bucket it maps to are returned as its neighbors [12].

3. k-d Trees: k-d trees are binary search trees, and they can used by partitioning each embedding dimension into binary partitions (like decision tree algorithms used by random forests). This tree is built through repeat splitting until all vectors are accounted for. During query phase, the query is split on the same logic and the closest sub-tree results as its neighbors [12].

4. Hierarchical Navigable Small World (HNSW) Graphs: It constructs a multi-layered hierarchical graph where each edge of a node is a similar neighbor. The different layers represent the levels of granularity. The search starts with the upper most layer and gradually moves down to more granular layers. Hence, it can execute a faster search mechanism where the search could be terminated early if has very low similarity with its neighbors in the upper layers [12].

5. ScaNN (Scalable Nearest Neighbors): Implements a multistage scaling, pruning, partitioning and refinement of graph of embedded points. It also uses asymmetric vector quantization to reduce the memory footprint of vectors. Invented and used by Google to handle billions of searches [13]

All the above algorithms need a mechanism to calculate distance between two vectors, which could be done in one of the following ways:

1. Dot Product
2. Cosine Similarity
3. Euclidean distance (L1/L2 norm)

Cosine similarity is in general the most preferred way to perform vector similarity for embeddings since its scale invariant and interpretable (as a function of the cosine value of angle between the vectors), despite weaknesses like its reliance on robust regularization [14].

The vector search mechanism should also for incremental addition or deletion of documents of chunks/documents, it is often recommended to use an existing framework to handle the process in an efficient manner. One of the most popular libraries for the purpose is FAISS by Meta [15] which also supports CUDA GPU based vector search. While FAISS is a pioneer in the space, many more alternatives have emerged since the advent of ChatGPT in 2023 [16].

ii. Output Generation:

In this phase, the retrieved context and the user input, both in their plain text form are passed onto the Large Language Model as illustrated in Figure 1.3. The LLM then processes over the portion of input that fits within its context window and yields an output token. The output token is then concatenated with the input and passed to the LLM again. This process repeats until the LLM generates an '<EOS>' (end of sentence/output) token.

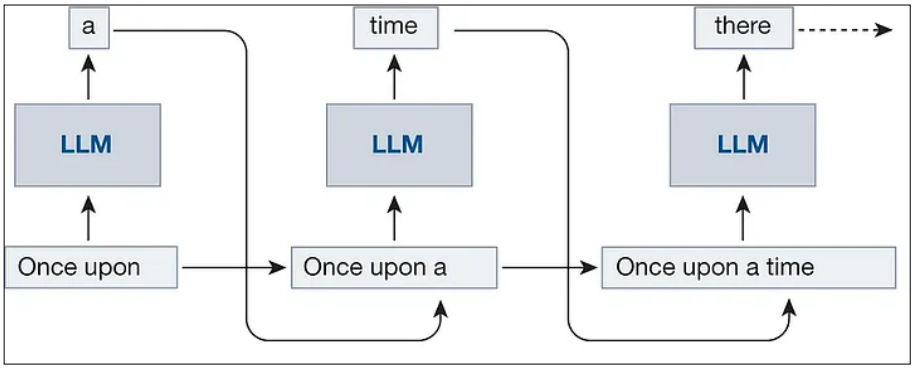

Figure 1.4: Auto regressive process of LLM output generation

c) Evaluation of RAG

Evaluation of the output quality of a RAG application needs to be performed in terms of relevance to the user's query and the content present in the vector database. The resultant metrics are designed to measure different performance of different components of the system since obtaining a single score can lead to difficulty in attribution and optimization.

The evaluation is performed using classical machine learning metrics like precision, recall, F1 score and NLP metrics like ROUGE. The evaluation in performed for multiple factors of the output as in Figure 1.5:

- Faithfulness, Output relevance & Semantic Similarity: Evaluate output quality with respect to inputs.
- Context Recall, Context Precision and: Evaluate the context retrieval and it's usage by the LLM.

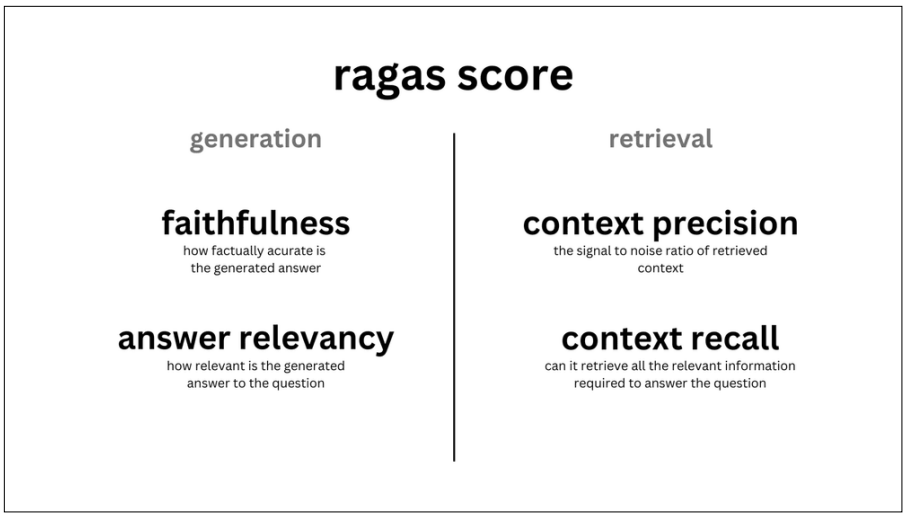

Figure 1.5: RAG application scores [18]

B. Apple M1 Architecture
    a) Chip Layout and Capabilities
    b) Accelerating LLM execution
B. Text generation using Large Language Models
    a) Process of text generation
    b) Performance optimization techniques
        - KV caching
        - Speculative Decoding
        - Lora & Quantization
    c) Scopes for optimization

II. Current Status
    A. Chosen repo, model & quantization
    B. Performance characteristics

III. Next Steps
    A. FAISS integration and optimization (metal/npu support?)
    B. Llama.cpp optimization (add NPU support?)

References:

01. Glenn. (2024). Mastering Retrieval-Augmented Generation (RAG) Architecture
02. LangChain. (2023). Building a RAG App
03. Mikolov et. al. (2013). Efficient Estimation of Word Representations in Vector Space
04. Brown et. al. (2020). Language Models are Few-Shot Learners
05. Bashir. (2023). In-Context Learning, In Context
06. LangChain. (2023). Text Splitters
07. AGI-Sphere. (2023). Context Length in LLMs
08. Liu et. al. (2023). Long in the Middle: How Language Models Use Long Contexts
09. Sennrich et. al. (2015). Neural Machine Translation of Rare Words with Subword Units
10. Touvron et. al. (2023). LLaMA: Open and Efficient Foundation Language Models
11. Sugawara et. al. (2016). On Approximately Searching for Similar Word Embeddings
12. Labelbox. (2023). Vector similarity search techniques
13. Guo et. al. (2020). Accelerating Large-Scale Inference with Anisotropic Vector Quantization
14. Steck et. al. (2024). Is Cosine-Similarity of Embeddings Really About Similarity?
15. CurrentsLab. (2023). Vector Search Techniques and Engines
16. Johnson et. al. (2017). Billion-scale similarity search with GPUs
17. Nabi. (2024). All You Need To Know About LLM Text Generation
18. Cardenas. (2023). Overview of RAG Evaluation