UNIVERSITAT DE BARCELONA

FUNDAMENTAL PRINCIPLES OF DATA SCIENCE
MASTER'S THESIS

LLM Adaptation Techniques. Evaluating RAG Strategies.

Author:
Francesc Josep Castanyer Bibiloni

Supervisor:
Eloi Puertas i Prats

A thesis submitted in partial fulfillment of the requirements for the degree of MSc in Fundamental Principles of Data Science

in the

Facultat de Matemàtiques i Informàtica

January 17, 2025

UNIVERSITAT DE BARCELONA

Abstract

Facultat de Matemàtiques i Informàtica

MSc

LLM Adaptation Techniques. Evaluating RAG Strategies.

by Francesc Josep Castanyer Bibiloni

This thesis explores the application of Retrieval-Augmented Generation (RAG) systems to optimize question answering tasks, addressing limitations of Large Language Models (LLMs) in scalability, efficiency, and domain adaptability. A theoretical foundation is established, highlighting RAG's role in integrating external knowledge to enhance language models.

A RAG pipeline is implemented and evaluated through experiments analyzing embedding models, similarity metrics, retrieval parameters (k), and re-ranking using cross-encoders. Results demonstrate that re-ranking improves retrieval accuracy, even with noisy, large-scale datasets, and highlight trade-offs between retrieval scope and generative performance.

This study underscores RAG's potential as a scalable alternative to fine-tuning, enabling efficient adaptation to dynamic datasets. Future research could explore advanced RAG variants and hybrid methods for broader applications.

The corresponding code notebook can be found on the following GitHub repository, https://github.com/XiscoCasta/LLM-adaptation-techniques.-Evaluating-RAG-models.

Acknowledgements

First of all, I would like to express my deepest gratitude to my thesis supervisor, Dr. Eloi Puertas i Prats, for his invaluable guidance and support. His continuous feedback and suggestions have greatly contributed to improving this work.

I am also profoundly grateful to my family, who have always supported me and believed in me throughout this journey. I would like to thank my friends from my hometown, who have always been there for me, and Arturo and Diego, who have accompanied me along my path in mathematics.

Finally, I wish to thank my classmates for their help and collaboration. In particular, I want to mention Jon and Sergio for their ongoing cooperation and support during the entire master's program, and Jokin and Ana for the countless hours we spent in the library, where much of this thesis took shape.

Contents

Chapter 1

Introduction

The field of Natural Language Processing (NLP) has witnessed significant advancements in recent years, with Large Language Models (LLMs) emerging as powerful tools for language understanding and generation. Built on the Transformer architecture, LLMs can process vast amounts of text data, capturing complex linguistic patterns and long-range dependencies. This has enabled them to excel at diverse tasks, such as text summarization, question answering, and content generation, transforming various domains including education, healthcare, and business (Brown et al., 2020).

Despite their remarkable capabilities, LLMs face several limitations that restrict their applicability, particularly in specialized or dynamic contexts. One key challenge lies in their reliance on large-scale, generic pretraining, which equips them with broad language understanding but often lacks the precision required for domain-specific tasks. Additionally, the computational demands of training and fine-tuning these models make them less accessible for smaller organizations or resource-constrained environments. Furthermore, adapting LLMs to evolving datasets or tasks can require significant effort, as retraining entire models is costly and time-consuming.

To address these challenges, researchers have developed various adaptation techniques that enhance the performance of LLMs in a more efficient and targeted manner. Among these methods, Retrieval-Augmented Generation (RAG) stands out as a promising approach. RAG systems augment LLMs by integrating external knowledge sources into their response generation process, enabling them to retrieve and utilize relevant information dynamically. This reduces the need for extensive fine-tuning and allows models to stay current with new or domain-specific knowledge. Other techniques, such as prompt engineering and partial fine-tuning, also provide flexible ways to adapt LLMs without the resource-intensive processes associated with full retraining.

This thesis focuses on the practical implementation and evaluation of RAG systems for question answering tasks. Specifically, it investigates how the retrieval and generation components of a RAG system can be optimized to balance computational efficiency and accuracy. Through a series of experiments, we evaluate the impact of various parameters, such as the choice of embedding models, similarity metrics, and the number of retrieved documents (k), on the system's overall performance. Additionally, we explore the role of advanced techniques like re-ranking with cross-encoders to refine retrieval accuracy and reduce noise in the generation process.

The contributions of this thesis are as follows:

1. A detailed implementation of a naive RAG pipeline, including preprocessing, embedding, retrieval, and generation steps.
2. An evaluation of retriever performance across different embedding models, similarity metrics, and k values on both controlled datasets (e.g., SQuAD) and real-world datasets (e.g., TriviaQA).
3. The integration of a re-ranking mechanism using cross-encoders to improve retrieval quality, demonstrating its effectiveness in both small and large-scale datasets.
4. An analysis of the scalability and practical implications of RAG systems when applied to noisy, large-scale data.

The remainder of this thesis is organized as follows: Chapter 2 provides a theoretical background on LLMs, adaptation techniques, and the RAG framework. Chapter 3 describes the experimental setup and presents the results of various RAG configurations. Finally, Chapter 4 discusses the findings, limitations, and future directions for research in this domain.

By addressing the general limitations of LLMs through efficient adaptation techniques, this thesis contributes to the ongoing efforts to make these models more practical, scalable, and applicable to a wider range of real-world tasks.

Chapter 2

Background

The aim of this chapter is to introduce the concepts that are crucial to understand the whole project. It begins from the basics of what an LLM is, progressing to more advanced procedures and techniques that can be applied to them, in order to understand the experimental section.

2.1 What is an LLM?

Large Language Models (LLMs) are a type of AI that are trained to understand, generate, and interact with human language in a way that is both coherent and contextually relevant. These models are 'large' not only in their size, spanning billions of parameters, but also in the vast amount of data they are trained on. This training involves the analysis of a wide array of text sources, from books and articles to websites and social media posts (Czerny, 2024).

These models are based on the Transformer architecture, due to Vaswani et al., 2017. This has been the base of all the advanced LLM models to date. It is based on the concept of attention mechanisms, which allows the model to weight the importance of the different parts of the input data, and use it to generate the outputs. A key innovation here is the self-attention mechanism, which enables the model to capture relationships between all the words in a sentence, regardless of their position. This contrasts with previous models, which could struggle with long sentences, as they worked sequentially.

2.1.1 Structure of LLMs.

As mentioned, Large Language Models (LLMs) are based on the Transformer architecture. At its core, the architecture consists of two main components: the encoder and the decoder. These components work in tandem to model relationships between tokens within and across sequences. The encoder-decoder structure can be seen in Figure 2.1.

The Transformer Architecture

The encoder is responsible for analyzing the input sequence and converting it into a continuous representation. This representation captures both the individual meanings of tokens and their contextual relationships within the sequence. By leveraging self-attention mechanisms, the encoder allows the model to focus on relevant parts of the input, regardless of their position in

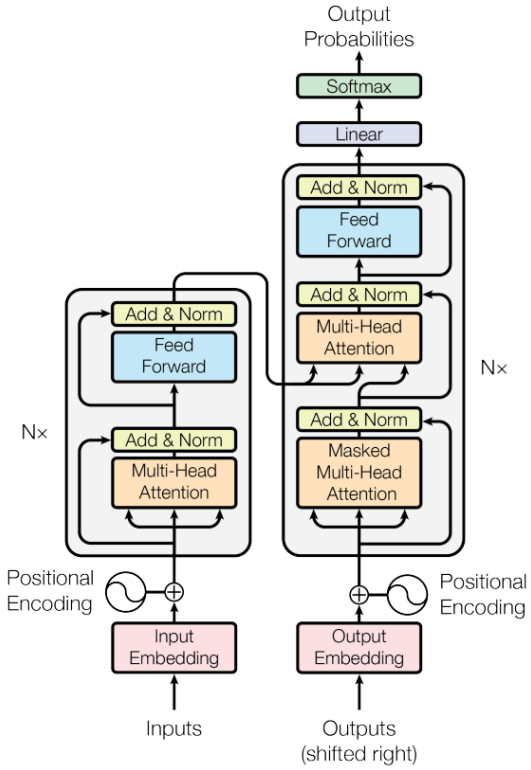

FIGURE 2.1: Transformer architecture with encoder and decoder components (Vaswani et al., 2017)

.

the sequence. As mentioned, this is particularly important for understanding long-range dependencies and complex sentence structures.

The decoder uses the encoded representation as a foundation to generate the output sequence, one token at a time. It relies on two key mechanisms: self-attention, which ensures that previously generated tokens are considered when predicting the next token, and encoder-decoder attention, which aligns the generated output with the context provided by the encoder. To maintain the causal structure of text generation, the decoder employs masking, preventing it from accessing future tokens during prediction.

The self-attention mechanism, central to both the encoder and decoder, assigns weights to different tokens in a sequence based on their relevance to the current token being processed. This is achieved by comparing query, key, and value representations derived from the input tokens. The result is a context-aware representation that enables the model to generate coherent and contextually appropriate outputs.

Components of the Transformer

The Transformer architecture consists of multiple neural network layers working together to achieve the results desired. A brief description of each type of layer within the architecture is provided below and illustrated in Figure 2.1.

- **Embedding Layer with Positional Encoding**: It converts input tokens into continuous vectors and injects positional information, since the Transformer does not inherently process sequences in order.
- **Multi-Head Self-Attention Layer**: It allows the model to focus on different positions within the input sequence to capture contextual relationships.
- **Feed-Forward Neural Network Layer**: It applies non-linear transformations to enhance the model's expressive power after attention mechanisms.
- **Layer Normalization and Residual Connections**: They improve training stability and help in training deeper networks by facilitating better gradient flow.
- **Output Layer**: It projects the final representations back to the token space to generate probabilities over the vocabulary for language modeling tasks.

Although subsequent research has introduced variations to the Transformer architecture, these core components remain essential to understanding how LLMs function.

Encoders and Embedding Models

Encoders play a crucial role in the Transformer architecture, and will also feature prominently in subsequent sections of this document, in the form of embedding models.

Encoders are responsible for transforming input sequences into continuous vector representations, known as embeddings. These embeddings capture both the semantic meaning of individual tokens and their contextual relationships within a sequence. In the Transformer architecture, the encoder achieves this by leveraging the self-attention mechanism, which allows it to assign importance to different tokens based on their relevance to the current token being processed. This process ensures that long-range dependencies and nuanced relationships within the input sequence are effectively captured.

Embedding models, on the other hand, are specialized implementations of encoders that are often pre-trained to generate high-quality vector representations for a wide range of tasks. For example, Sentence-BERT (SBERT), as introduced in Reimers and Gurevych, 2019, adapts the Transformer architecture to efficiently produce embeddings for sentence-level semantic similarity tasks. These models map text into a shared high-dimensional vector space, where semantically similar tokens or phrases are positioned closer together. For instance, the words "king" and "queen" may have embeddings that are nearby in this space, reflecting their semantic similarity, while being distant from unrelated words like "apple.

The concepts of encoders and embedding models introduced here will reappear in later sections, particularly in the context of adaptation techniques, where their utility in retrieval and context generation becomes apparent.

2.1.2 Training LLMs

The training of LLMs involves processing really large amounts of textual data, to learn patterns and structures in human language. This data comes from diverse sources, as books, articles, websites, and social media.

The training process is primarily unsupervised, where the model learns by predicting the next word in a sentence based on the previous context. This training is computationally intensive, requiring specialized hardware, and often takes weeks or months to complete. During training, the model's billions of parameters are optimized using techniques like stochastic gradient descent, enabling it to generalize language patterns effectively. Additionally, regularization methods like dropout are used to prevent overfitting, and adaptive optimizers, such as Adam, are applied to improve convergence speed and stability (Brown et al., 2020).

While this training enables the model to develop a broad understanding of language, it does not inherently prepare it to perform specific tasks. In many cases, these limitations can be addressed by using well-crafted prompts that provide clear instructions on the task at hand. This approach, known as prompt engineering, allows users to adapt the model's behavior without altering its underlying structure. However, when prompts alone are insufficient, further adaptation techniques, such as fine-tuning or Retrieval-Augmented Generation (RAG), are commonly employed to specialize the model for specific applications.

These adaptation techniques will be explained in more detail in subsequent sections.

2.1.3 Applications and Limitations of LLMs

Large Language Models have revolutionized various domains by enabling complex language understanding and generation tasks. Their versatility comes from their ability to model and leverage context, making them indispensable in areas such as natural language processing, machine learning and artificial intelligence.

Applications

The range of tasks LLMs can perform is vast, spanning areas such as text generation, machine translation or question answering (Brown et al., 2020). They excel at producing coherent and contextually relevant outputs, summarizing information or rewriting text to suit specific needs. In machine translation, their capacity to recognize linguistic patterns has significantly improved

the quality of translations. Additionally, LLMs are essential to customer service automation, where they power chatbots and virtual assistants to provide seamless interactions. Another notable application is in content moderation and analysis, where they help identify and filter inappropriate or harmful content.

Limitations

While LLMs are remarkably versatile, they are not without their challenges. One significant drawback is their lack of domain expertise. Although they can generate general responses, their outputs often lack the depth required for specialized fields unless fine-tuned with domain-specific data. Furthermore, LLMs are prone to hallucinations, where they produce factually incorrect or misleading information.

Scalability also presents a challenge. Training and deploying these models demand substantial computational resources, making them less accessible to smaller organizations. Additionally, the ethical implications of deploying LLMs are a growing concern, as biases in training data can result in prejudiced outputs, and the potential misuse of these models for harmful purposes raises questions about their broader societal impact (Brown et al., 2020).

To overcome or mitigate these limitations, adaptation techniques are necessary. Methods like fine-tuning enable the model to specialize in a particular domain or task by training it on domain/task-specific data. Prompt engineering allows the user to guide the model's behavior by carefully crafting input instructions, leveraging the model's inherent language capabilities without modifying its internal structure. Additionally, Retrieval-Augmented Generation (RAG) enhances accuracy by integrating external knowledge retrieval into the generation process, reducing hallucinations and improving factual correctness. These adaptation methods will be explored in detail in subsequent sections.

2.2 Adaptation Techniques

This section outlines key adaptation techniques aimed at enhancing a model's performance on specific tasks. These techniques include prompt engineering, fine-tuning, and Retrieval-Augmented Generation (RAG).

Prompt engineering is a straightforward method that doesn't require extra implementation. It involves creating clear and specific instructions to guide the model's behavior, using its existing abilities without making any changes to its structure or how it works. While the concept may look simple, crafting effective prompts can become quite complex, especially for advanced or nuanced tasks.

Fine-tuning is a more advanced approach that changes the model's behavior to make it perform better on specific tasks. It requires further training the model with data focused on a particular domain or task. While it is an

effective and widely used technique, fine-tuning demands a lot of computational resources because it involves retraining either specific parts of the model or the entire model.

RAG is a method specifically designed to enhance a model's performance by providing access to external information. Its purpose is to retrieve relevant data from knowledge bases or document repositories during the generation process, enabling the model to incorporate additional context or factual information into its outputs. Despite its focused purpose, RAG can be applied across a wide range of tasks, making it particularly valuable for applications like question answering, summarization or fact-checking, where access to up-to-date or domain-specific information is crucial.

The experimental phase of this project centers on utilizing RAG systems for question answering tasks, which significantly benefit from the additional context provided by RAG. This approach was selected over other techniques because it has a strong impact on these tasks while needing fewer computational resources than the resource-intensive fine-tuning.

The next subsections will explain these adaptation techniques in greater detail.

2.2.1 Prompt Engineering with Few-Shot Learning

Prompt engineering with few-shot learning is a naive yet effective approach to adapt LLMs to specific tasks without altering the model's parameters or structure. In this method, the user provides a few examples (few-shots) of the desired input-output behavior within the prompt itself. This helps the model understand the task and generate appropriate responses based on the given examples.

How it Works

Few-shots learning involves including a small number of demonstrative examples in the prompt to guide the model's output. The prompt typically consists of:

- **Task Description**: A brief explanation of what the user wants the model to do.
- **Examples**: A few input-output pairs to demonstrate the desired behavior.
- **Query**: The new input for which the user wants the model to generate an output.

By providing these components, the model can infer the task and generate accurate responses that align with the examples.

The following is an example of a few-shot learning prompt designed for sentiment classification of movie reviews:

```
Classify the sentiment of the following movie reviews as
Positive or Negative.

Review: "I absolutely loved this movie! The plot was
gripping and the characters were well developed."
Sentiment: Positive

Review: "The film was a waste of time. The storyline was
predictable and the acting was subpar."
Sentiment: Negative

Review: "The cinematography was stunning, but the pacing
was too slow for my taste."
Sentiment:
```

In this prompt, the model is given examples of movie reviews along with their sentiment classification. The model is then expected to classify the sentiment on the new review.

Advantages

- **No Training Required**: Does not require any additional training or fine-tuning of the model.
- **Flexibility**: Can be quickly adapted to a wide range of tasks by changing the examples in the prompt.
- **Accessibility**: Easy to implement and requires minimal computational resources.

Challenges and Considerations

- **Maximum input size**: The model's maximum input size limits the number of examples that can be provided.
- **Inconsistent Performance**: The model may not always generalize well from the few examples provided.
- **Prompt Sensitivity**: Small changes in the prompt wording or examples can lead to significant variations in output.

Few-shot learning through prompt engineering is a practical starting point for adapting LLMs to specific tasks. However, for more complex or specialized applications, more advanced techniques like fine-tuning or RAG may be necessary.

2.2.2 Fine-Tuning

Fine-tuning is a powerful technique for adapting LLMs to specific tasks by updating the model's parameters based on task-specific data. Unlike prompt

engineering, which requires no modification to the model, fine-tuning involves training the model on a smaller dataset tailored to the task, enabling it to better understand and solve specialized problems.

As explained previously, the general training phase of an LLM equips it with a broad understanding of language patterns and structures. Fine-tuning builds on this foundation by tailoring the model's knowledge to a specific context, such as domain-specific terminology, tone or skills. Applications of fine-tuning include (among a wide variety):

- **Specialized text processing**: It helps the model learn to handle technical documents, legal contracts, or medical records where specific terminology is critical
- **Customer service or support**: It trains the model to communicate in a specific tone and style suited to a particular business or industry.
- **Task-specific optimization**: It allows the model to enhance its performance on different tasks like sentiment analysis, summarization or entity extraction, among others.

How it Works

Three general methods of fine-tuning can be identified, each suited for different situations regarding computational resources, model size and task requirements:

- **Full model fine-tuning**: This approach involves adjusting all parameters of the model to fully specialize on the new tasks. While it can yield highly effective results, it is computationally expensive and also has the risk to overfit in small datasets.
- **Layer-wise**: In this method, only specific layers of the model (typically those closer to the output layer) are fine-tuned. This method is great when one has limited computational resources, because it allows for domain adaptation while not having to train the whole model.
- **Parameter-efficient tuning**: These techniques aim to modify only a small subset of parameters, by adding new parameters as adapters, or adjusting low-rank matrices to determine a subset to modify. These methods are particularly useful for larger models, as they reduce memory usage while still allowing the learning.

Advantages

Fine-tuning provides exceptional flexibility, making it one of the most effective techniques for customizing LLMs to specific tasks or domains. It can be applied to a wide range of use cases, from handling domain-specific terminology in fields like healthcare and law to optimizing performance in tasks like sentiment analysis, summarization, or question answering.

This adaptability extends across scales, working well with both small, focused datasets and large, complex ones. Fine-tuning also allows precise control over a model's behavior, enabling developers to align outputs with specific tones, styles, or functional requirements. This ability to finely tune model behavior makes it a critical tool for developers aiming to build solutions that align with both technical and business goals.

Its flexibility, wide range of use cases, and ability to control the model's behavior precisely in each scenario make fine-tuning one of the most widely used techniques in the field of machine learning.

Challenges and Considerations

While fine-tuning is a powerful technique, it comes with several limitations that must be considered, particularly regarding its resource requirements and data preparation needs.

One of the most significant challenges of fine-tuning is the substantial computational resources required to train the entire model. This becomes particularly evident when dealing with large-scale LLMs, as fine-tuning involves updating billions of parameters. The process often necessitates high-performance hardware such as GPUs or TPUs, making it both time-consuming and expensive, especially for organizations with limited access to computational infrastructure.

Another important limitation is the reliance on pre-prepared datasets. Fine-tuning typically follows a supervised learning approach, meaning the model requires labeled data specific to the target task or domain. Preparing such datasets can be labor-intensive, requiring careful curation, labeling, and preprocessing to ensure quality and relevance. Moreover, the availability of such datasets may be limited in highly specialized fields, further complicating the fine-tuning process.

Additionally, fine-tuning carries a risk of overfitting, particularly when working with small or narrowly focused datasets. Overfitting can lead to a model that performs well on training data but struggles to generalize to unseen examples, reducing its practical utility.

While these limitations can be challenging, they highlight the need to carefully consider whether fine-tuning is practical and worth the cost before using it. In some cases, less resource-demanding techniques may be enough to achieve the desired results.

2.2.3 Retrieval-Augmented Generation (RAG)

The following section is primarily based on information from Gao et al., 2024 and Yu et al., 2024.

Retrieval-Augmented Generation (RAG) is a technique designed to improve the capabilities of LLMs by incorporating information from external sources. This allows the model to access relevant and up-to-date information at the time of generating the output, addressing problems like "hallucination"

that is, when the model makes up the information on the output because it does not have the knowledge required by the input.

By using RAG, LLMs can retrieve document chunks relevant to the user query by calculating semantic similarity, and integrate this information to generate the response.

Its applications include a wide range of areas where the model benefits from having additional information, such as question answering, fact checking or text generation among others.

How it Works

This section outlines the structure of a naive RAG system and highlights modifications introduced in more advanced models to enhance performance. The basic schema can be seen in figure 2.2. It consists on three processes, which are the following:

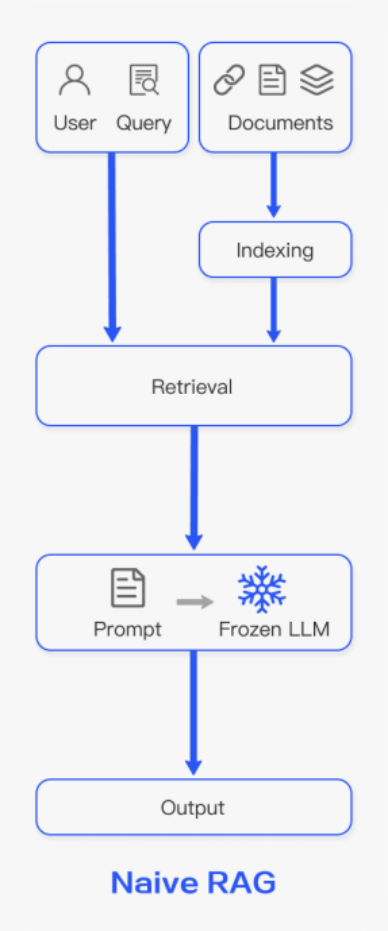

FIGURE 2.2: Naive RAG schema (modified from Gao et al., 2024).

1. **Indexing**: In this step, the documents are split into chunks and embedded into a vector space, which are stored in an indexed database. This indexing is done in a way that enables efficient similarity searches, which allows the model to retrieve information rapidly.

2. **Retrieval**: The model performs a similarity search comparing the query to all indexed documents, and retrieves to top k. This retrieval process provides the model with the additional information needed in the next step.

3. **Generation**: Once the documents are retrieved, they are combined with the query and introduced to a frozen LLM (meaning it isn't modified in this process). Then, the LLM generates a response based on the query and on the retrieved documents, producing an output based on the context retrieved.

On this basis, Advanced RAG models may introduce introduce several key modifications, including:

- **Query expansion and transformation**: Advanced RAG models incorporate mechanisms to transform the initial query through expansion or rephrasing, in order to improve the chance of retrieving relevant information.

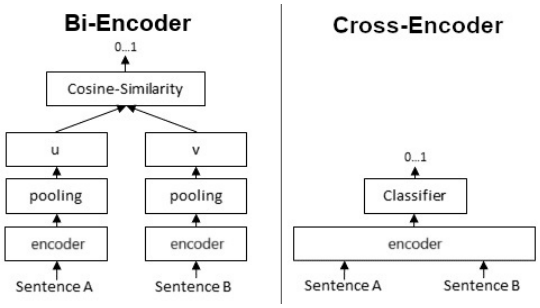

FIGURE 2.3: Comparison between Bi-encoders and Cross-encoders (modified from Transformers, n.d.).

- **Re-ranking and filtering**: After retrieving the relevant documents, an advanced RAG re-ranks them using a more computationally expensive procedure which grants better results (it only has to re-rank a small amount of already retrieved documents), ensuring the order of the retrieved documents is more accurate in terms of relevance. Then, it filters out the less relevant ones which are ranked at the bottom.

- **Refined prompting**: The retrieved context is structured and formatted with optimized prompts, allowing the LLM to produce more coherent and contextually appropriate responses.

Further building upon advanced RAG, modular RAG introduces additional flexibility, implementing independent specialized modules, such as a memory module for retaining context across interactions, or a routing module for directing queries to the most suitable components based on the task. Each module in modular RAG can be customized or replaced, making the system highly adaptable and customizable for different applications.

Role of Encoders in RAG

Encoders play a crucial role in RAG systems, as the indexing step relies on embedding models to represent document chunks in a vector embedding space. A similarity search is then performed between the query and the indexed documents. This process uses a bi-encoder, which individually embeds both the documents and the query, and computes the similarity measure between their respective vectors. This procedure is illustrated in Figure 2.3.

Another important role of encoders in RAG systems is re-ranking. For this task, a cross-encoder is used. The cross-encoder directly computes the similarity measure by processing both the query and the document together as input, as shown in Figure 2.3. This approach generally yields better results because the cross-encoder is able to compare the query and the document directly when scoring their similarity, enabling more accurate ranking. However, cross-encoders cannot be used for the initial retrieval step because they are much more computationally demanding. Instead, after a small set of documents has been retrieved, the cross-encoder can be applied to re-rank them and order the results more accurately.

Advantages

RAG systems offer several advantages that make them an attractive choice for many tasks, particularly when scalability, flexibility, and efficiency are critical. These benefits include:

- **Scalability and Dynamic Updates**
  RAG systems are designed for scalability, enabling them to handle increasing volumes of data and expand their knowledge base effortlessly. By dynamically incorporating new information through simple updates to the indexed database, they eliminate the need for retraining the underlying language model. This capability makes RAG systems ideal for applications that require frequent updates or operate in rapidly evolving domains.

- **No Need for Preprocessed or Curated Datasets**
  Unlike fine-tuning methods that often rely on labeled or curated datasets, RAG systems can process raw, unstructured data directly. This reduces setup time and makes them accessible for a wide range of use cases.

- **Efficient Use of Computational Resources**
  RAG systems optimize computational efficiency by separating retrieval and generation processes. Retrieval focuses only on relevant data, while the frozen LLM generates responses, reducing resource demands compared to full fine-tuning.

Challenges and Considerations

While RAG systems offer numerous advantages, they also come with specific challenges that need to be addressed to ensure optimal performance. Key considerations include:

1. **Quality and Coverage of Indexed Data**
   The effectiveness of a RAG system hinges on the quality and coverage of the indexed dataset. If the necessary information to perform well on the task is absent or inadequately represented, the system will struggle to produce accurate outputs. Ensuring comprehensive and relevant coverage of the knowledge domain is critical.

2. **Noise in Retrieved Results**
   Despite advancements in embedding models and retrieval algorithms, irrelevant or low-quality documents may still be retrieved, introducing noise into the generation process. This can confuse the language model and degrade the accuracy or coherence of its outputs. This will be addressed by optimizing the number of documents retrieved, or even reranking and filtering retrieved documents.

3. **Dependence on Effective Retrieval Models**
   The retrieval component of a RAG system is crucial for providing relevant and contextually appropriate information to the generative model. Suboptimal retrieval models or poorly tuned similarity metrics can lead to irrelevant or insufficiently detailed inputs, undermining the system's performance.

With the necessary context established and having reviewed the fundamental concepts and relevant adaptation techniques, we are now ready to explore the implementation of the RAG system, examining the details of its configuration and functionality for specific tasks.

Chapter 3

Experimentation

In this chapter, we build a RAG system and experiment with different configurations and datasets to evaluate the system's performance under various conditions.

All experiments are conducted on a system with the following specifications:

- **Processor**: 13th Gen Intel® Core™ i7-1355U @ 1.70 GHz
- **RAM**: 16 GB

For tasks involving large datasets, such as processing embeddings for Wikipedia passages, we utilized GPU acceleration provided by Google Colab's T4 GPU environment.

3.1 Building a Naive RAG: Step-by-Step Explanation

This section outlines the process of constructing a general naive RAG system step by step, as depicted in Figure 2.2. Subsequently, different databases and configurations for the RAG system are tested, and various characteristics of the advanced RAG system introduced earlier are explored.

We employ the Python libraries `datasets` and `transformers` from Hugging Face's well-known framework (HuggingFaceTeam, 2024), designed for managing models and datasets. For each component of the RAG pipeline, the specific packages utilized are explained.

The pipeline consists of three main steps: indexing, retrieval, and generation. In this context, indexing and retrieval are executed jointly.

3.1.1 Indexing and Retrieval

Dataset Preparation and Chunking

To begin, we use the `datasets` library to download and access the datasets required for our experiments.

The first task involves preprocessing the documents within the dataset. Depending on the structure and length of the original documents, they may need to be divided into smaller chunks to enable effective embedding into a vector space. This step is crucial: if the chunks are too small, critical context

and information might be lost. Conversely, overly large chunks may exceed computational limits from the retriever, the generative model or both. To accomplish this, we utilize the `RecursiveCharacterTextSplitter` function from LangChain (LangChainTeam, 2024), which allows to split the text taking into account the length of the text after being tokenized, but returns plain non-tokenized text.

The chunk length and maximum overlap are also important hyperparameters to consider in each experiment. Their values will be specified at the beginning and assumed to remain unchanged unless otherwise stated.

Document Embedding, storage and Retrieval

Once the text preprocessing is complete, the next step is to embed the text chunks into a vector space and store these embeddings. The embedding model will be also a parameter on the experiment. Model selection will be based on the MTEB leaderboard (Muennighoff et al., 2022), which compares the performance of different models for different tasks, taking into account the size of the model. We will be using small models to deal with our low computational resources.

After generating the embeddings, they are stored in a vector database to enable retrieval based on a similarity metric. For this purpose, we employ the FAISS library (Johnson, Douze, and Jégou, 2017), which is designed for efficient similarity searches and clustering of dense vectors. FAISS provides a direct command to perform similarity-based retrieval, specifying a similarity metric and the number of documents to retrieve, k.

3.1.2 Generation

Finally, after retrieving the documents, the remaining step is to input them to a generative model alongside with the initial query, and generate a response from the model.

The `transformers` library is used to load and apply the models, combining the retrieved documents with the query to create the input for the model. The choice of model is a parameter in every experiment. Due to limited computational resources, the Google FLAN-T5 small model (Chung et al., 2022) is utilized for all experiments. This model is a compact and efficient version of the FLAN-T5 series, fine-tuned on a variety of instruction-following tasks. Its small size enables faster inference while maintaining good performance, which is crucial for this thesis as it involves numerous evaluations, each involving 1000 or 2067 generations per test case.

3.2 First Experiments

Initial experiments focus on retriever parameters, including embedding models, k values, and distance metrics. A test RAG model is also used to examine generation behavior with varying numbers of input contexts.

3.2.1 Experiment: Distance and Embedding Model

As stated, the first experiments target the retriever by analyzing the impact of various embedding models and distance metrics. For this purpose, the `SQuAD` database (Rajpurkar et al., 2016), a reading comprehension dataset based on Wikipedia articles, is utilized. From this dataset, a subset of 2067 unique question-context pairs is extracted. The objective is to measure how often the retriever successfully identifies the correct context paired with a given question among the retrieved documents. So, the evaluation formula is

$$\text{Score} = \frac{\text{Number of correct contexts retrieved}}{\text{Total number of retrievals}} \tag{3.1}$$

In this case, as the contexts for the questions are pre-processed chunks, we will not be using the text splitter to break them in smaller chunks. For this reason, chunk size is not a parameter of the experiment.

The embedding model selection is guided by the MTEB leaderboard, which ranks models based on retrieval performance. We experiment with the top three models in the small category (less than 100 million parameters). Each of these models has 33 million parameters:

- *NoInstruct small Embedding v0*
- *Snowflake's Arctic-embed-s*
- *Bge small retail finetuned*

Additionally, we evaluate the impact of different distance measures, which include the following:

- **Cosine Distance**: Measures the angular similarity between two vectors. It is calculated as:
$$d_{\text{cosine}}(x, y) = 1 - \frac{x \cdot y}{\|x\| \|y\|} \tag{3.2}$$
where x and y are vectors, $\cdot$ denotes the dot product, and $\|\cdot\|$ is the Euclidean norm.
- **Euclidean Distance**: Measures the straight-line distance between two points in a vector space. It is given by:
$$d_{\text{euclidean}}(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{3.3}$$
where $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$ are vectors in an n-dimensional space.
- **Dot Product Distance**: Measures the negative of the dot product between two vectors. It is defined as:
$$d_{\text{dot}}(x, y) = -(x \cdot y) \tag{3.4}$$

where x and y are vectors, and $\cdot$ represents the dot product. The higher the dot product, the more similar the vectors.

By evaluating these similarity measures, our objective is to identify the method that performs best in retrieving documents relevant to the given query context. Additionally, by experimenting with different k values, we aim to determine whether certain similarity metrics excel at identifying exact matches, while others are more effective at capturing relevant context within the top k retrieved results. The results can be seen on Table 3.1.

Evaluation Results

TABLE 3.1: Performance of Different Embedding Models (Values Represent the Score Metric).

Model	$k = 1$	$k = 2$	$k = 3$	$k = 5$	$k = 10$	$k = 20$
NoInstruct small Embedding v0	0.7213	**0.8302**	**0.8761**	**0.9231**	**0.9594**	**0.9816**
Snowflake's Arctic-embed-s	0.5839	0.7073	0.7692	0.8234	0.8936	0.9361
Bge small retail finetuned	**0.7286**	0.8287	0.8742	0.9202	0.9550	0.9811

In Table 3.1, the distance metric used is not specified, as all tested metrics yielded the same results (refer to the code for more details). For the sake of consistency and simplicity, Euclidean distance will be utilized in all subsequent experiments, as it serves as the default metric in FAISS.

Regarding the models, the table highlights performance variations depending on the k value chosen. This is particularly evident when comparing the top two models, *NoInstruct small Embedding v0* and *Bge small retail finetuned*, which consistently outperform *Snowflake's Arctic-embed-s* across all metrics. Notably, *Bge small retail finetuned* performs best when $k = 1$, while *NoInstruct small Embedding v0* outperforms the others when $k > 1$. These findings will guide our model selection in future experiments, prioritizing the best performers.

Furthermore, the table shows that in 98% of cases, the correct context is included among the top 20 retrieved documents. This result demonstrates the importance of high k values during retrieval to maximize the likelihood of retrieving relevant contexts. By applying an effective re-ranking strategy, we can further refine the selection, ensuring the correct context is prioritized. This motivates the implementation of re-ranking techniques in later experiments to enhance retrieval quality and overall performance.

3.2.2 Experiment: Value of k

In this section, we explore the optimal value of k for the database used in our first RAG model. The database utilized by the model consists of the contexts corresponding to the evaluation questions. This setup ensures that the necessary information to answer each query is present in the dataset, eliminating the possibility of having unanswered questions.

As stated previously, the Euclidean distance is used, along with the embedding model that demonstrated the best performance for each specific k value. For this experiment, the value of k is limited to $k \in \{1, 2, 3\}$, as the generating model imposes constraints on input length, and larger values of k would exceed these limitations. This is because the contexts weren't split into smaller chunks, so with $k = 4$ the maximum length of the input for the generative model is exceeded.

The process for combining the query with the retrieved context is straightforward. The model receives the following input query without additional formatting:

"question: *original query*, context: *retrieved context*"

This simple structure allows the model to focus directly on the question and its associated retrieved context without unnecessary distractions from additional preprocessing steps.

Based on insights from the previous experiment, increasing k is expected to improve model performance by raising the likelihood of retrieving the correct information needed to answer the question. However, this relationship is not entirely realistic in practical scenarios. While larger k values improve the probability of retrieving relevant information, they also introduce additional noise by including unnecessary data. This can difficult the model's ability to filter out irrelevant details and focus on generating accurate responses. Consequently, performance is anticipated to improve with higher k values only up to a certain threshold, beyond which the gains may plateau or even diminish.

For the evaluation process, we define a custom metric similar to the Exact Match Score. The Exact Match Score considers an answer valid only when it fully matches the ground truth and computes the percentage of such matches. In contrast, our custom Match Score adds flexibility by tokenizing both the model's responses and the ground truth answers using the model's tokenizer.

A response is considered valid if there is at least an 80% match between the tokenized model response and the tokenized ground truths. This 80% threshold provides a balance between strictness and flexibility: it allows minor variations in phrasing while avoiding the acceptance of incomplete answers, which can happen because responses are often short.

The Match Score is computed as:

$$\text{Match Score} = \frac{\text{Number of valid responses}}{\text{Total number of responses}} \tag{3.5}$$

Below are some examples of acceptable answers under this evaluation method:

- Model response: *'an arbitrary graph'*, ground truths from the dataset: `['arbitrary graph', 'arbitrary', 'arbitrary']`.
- Model response: *'the Qur'an'*, ground truths from the dataset: `['Qur'an', 'Qur'an', 'Qur'an']`

- Model response: *'31 July 2013'*, ground truths from the dataset: `['July 2013', 'In July 2013', 'July 2013']`

Evaluation Results

We evaluated the performance of the RAG model for different values of k using the metrics outlined above. These models are also compared to a baseline model (model without any context input), and to the generative model when given the exact correct context. Results are shown in Table 3.2.

TABLE 3.2: Match Score on `SQuAD` questions with Contexts Dataset.

System / Configuration	k	Match Score
Baseline Model (without contexts)	-	0.0179 (37 / 2067)
RAG System with Contexts Dataset	1	0.5235 (1082 / 2067)
	2	**0.5636 (1165 / 2067)**
	3	0.5627 (1163 / 2067)
Model with Correct Context	-	0.6957 (1438 / 2067)

From these results, we observe a steady improvement in exact match performance when increasing k from 1 to 2, suggesting that providing additional context improves the model's ability to generate correct answers. However, when k is further increased to 3, the exact match score slightly decreases compared to $k = 2$. This behavior supports our earlier hypothesis that while increasing k may provide more relevant context, it also introduces additional noise, making it harder for the model to focus on the most critical information.

While $k = 2$ achieves the best balance between providing sufficient context and avoiding excessive noise, there is still room for improvement.

After these experiments, the gap between the best RAG model's performance (0.5636) and the model provided with the correct context (0.6957) highlights limitations in the retrieval process. To address this, we will focus on re-ranking and filtering strategies in future experiments to enhance the model's ability to retrieve and prioritize the most relevant contexts.

3.3 Real Case Experiment

In previous experiments, we were using small, pre-prepared chunks of text containing the necessary information to answer all the questions. This section presents an experiment designed to simulate a scenario closer to real-world applications.

The `TriviaQA` dataset (Joshi et al., 2017) is utilized in this experiment, with a subset of 1000 questions selected to ensure they can be answered using information from Wikipedia. We conduct two related experiments: the first uses a RAG database consisting only of the Wikipedia passages where

the answers can be found, and the second adds a large number of additional Wikipedia passages without verifying if they contain any relevant information. This approach allows us to evaluate whether adding extra documents to the dataset introduces noise and negatively impacts the model's performance.

3.3.1 Experiment: Specific Wikipedia Passages

As described in the introduction to this section, this experiment uses the `TriviaQA` questions dataset. For the RAG system, we use a dataset of Wikipedia passages where the answer to each question is guaranteed to appear (a total of 1537 passages). Since these are real Wikipedia passages, they must be preprocessed and divided into chunks, as outlined in Section 3.1.

Specifically, we use the `RecursiveCharacterTextSplitter` function from LangChain to divide the text into manageable chunks. This function ensures that each chunk's tokenized text does not exceed a specified limit. The splitting is done coherently, using a prioritized list of separators to preserve as much context as possible. The list of separators, in order of priority, is as follows:

```
MARKDOWN_SEPARATORS = [
    "\n\n",  # Paragraph breaks
    "\n",    # Line breaks
    ". ",    # Sentence endings
    "? ",    # Question endings
    "! ",    # Exclamation endings
    " ",     # Fallback to spaces
    ""       # Catch-all
]
```

This means the splitter first tries to keep paragraphs intact. If the text is still too long, it keeps lines together, and so on. The chunk size is set to a maximum of 128 tokens to comply with the input length constraints of the generative model, and a chunk overlap of 10%. While larger chunks would preserve more context, computational limitations necessitate this smaller size. Additionally, the title of each passage is prepended to the start of each chunk to provide the retriever with extra context.

After chunking the text, we create the vector database using the embedding model and the FAISS library, as described earlier. This process is computationally intensive, as the final dataset contains $111,999$ chunks, each requiring embedding using the model and storage in the vector database.

After setting up the retriever, it is connected to the generative model to produce responses. The evaluation process utilizes the custom Match Score described earlier, and system performance is tested across various values of k.

We expect lower performance compared to previous experiments, as this dataset consists of real Wikipedia passages divided into chunks rather than

small, preprocessed contexts. The system's performance is compared against the baseline model, which does not use any additional documents or context.

Evaluation Results

We evaluated the performance of the RAG system on the `TriviaQA` dataset using different values of k. These results are compared against a baseline system (without any context input). The results are shown in Table 3.3.

TABLE 3.3: Match Score for RAG Systems on `TriviaQA` Questions.

System / Configuration	k	Match Score
Baseline Model (without contexts)	–	0.070 (70 / 1000)
RAG System with Wikipedia Contexts	1	0.384 (384 / 1000)
	2	0.457 (457 / 1000)
	3	0.476 (476 / 1000)
	4	**0.487 (487 / 1000)**
	5	0.478 (478 / 1000)
	6	0.475 (475 / 1000)

Similar to the previous experiment on `SQuAD`, we observe an improvement in the Match Score as k increases, up to a certain point. In this case, the best performance is achieved at $k = 4$ with a Match Score of 0.487. Beyond this, the score begins to decline slightly as k increases further.

This behavior mirrors the pattern observed earlier, where increasing k initially provides more relevant context but eventually introduces additional noise, reducing the model's ability to focus on critical information. However, in this experiment, the optimal k value is higher compared to the `SQuAD` experiment ($k = 2$). This is likely due to the shorter chunks of text resulting from our preprocessing step, which necessitates the inclusion of more chunks to capture sufficient context for answering questions accurately.

These results reinforce again the need to carefully balance k to maximize performance while minimizing noise.

3.3.2 Experiment: Additional Wikipedia Passages

In this experiment, we combine the 1537 Wikipedia passages provided as context in the `TriviaQA` dataset with extra passages from an additional dataset, the `wikipedia.20220301.simple` dataset from HuggingFace, 2022. This additional dataset is a preprocessed collection of 205,328 Wikipedia entries in English, provided by HuggingFace. While some of the added passages might help answer a few questions, most of them are unrelated and introduce noise into the retrieval process. This setup creates a more realistic scenario for testing the system.

We preprocess the passages in the same way as the previous experiment and merge all the chunks into a single FAISS vector database as the retriever. The final database contains 768,281 chunks.

With this larger dataset, we expect that if the retriever is not robust enough, the added noise will reduce the system's performance. The results from this setup are compared to the best-performing model from the previous experiment, with evaluations conducted across various k values. The outcomes are presented in Table 3.4.

Evaluation Results

TABLE 3.4: Match Score for RAG Systems on `TriviaQA` Questions.

System / Configuration	k	Match Score
Baseline Model (without contexts)	–	0.070 (70 / 1000)
RAG System with Wikipedia Contexts	4	0.487 (487 / 1000)
RAG System with Additional Contexts	1	0.399 (399 / 1000)
	2	0.463 (463 / 1000)
	3	0.487 (487 / 1000)
	4	**0.500 (500 / 1000)**
	5	0.496 (496 / 1000)
	6	0.493 (493 / 1000)

The results show steady improvement as k increases, with the highest Match Score of 0.500 at $k = 4$. This pattern is similar to the one in the previous experiment, where increasing k initially improves performance but then slightly reduces it as k becomes too large.

When comparing both real-case experiments, this last experiment shows slightly better results overall. For example, the Match Score improves from 0.487 in the previous setup to 0.500 here, both at $k = 4$. This suggests that the retriever is strong enough to manage the extra passages and still retrieve relevant contexts. So, we conclude that adding large volumes of potentially unrelated text does not necessarily degrade performance, always when being capable to deal with computational costs.

3.4 Re-Ranking and Filtering Experiments

For the final experiment, we implemented a re-ranking and filtering mechanism for the retrieved documents to improve the performance of our retriever.

To achieve this, we used a cross-encoder model to compute new similarity scores and re-rank the retrieved documents. Unlike the bi-encoder used in the initial retrieval—which embeds the query and documents separately into a vector space and computes their similarity—the cross-encoder takes both the query and the document as inputs and directly computes their similarity. This approach is explained in detail in Section 2.2.3.

The cross-encoder used is `cross-encoder/ms-marco-MiniLM-L-12-v2`, a cross-encoder fine-tuned specifically for retrieving and re-ranking documents based on queries.

3.4.1 Experiment: Re-Ranking Retrieval

The retrieval strategy begins by retrieving $k_{\text{retriever}} = 20$ documents using the bi-encoder-based retriever. Each of these documents is then paired with the query and scored using the cross-encoder. These new similarity scores are used to re-rank the documents, and the top k_{rerank} documents are selected for the final generation step. The choice of $k_{\text{retriever}} = 20$ is based on the results from Table 3.1, which show that 98% of the time, the correct context is found among the top 20 retrieved documents. By re-ranking the correct context to a higher position and filtering out less relevant ones, we aim to improve model performance.

This experiment is closely aligned with the previous one that assessed the retrievers (Table 3.1). In this case, the re-ranking retriever's ability to retrieve the correct context is evaluated with different values of k_{rerank}. A higher performance with lower k_{rerank} would indicate that the cross-encoder effectively prioritizes the most relevant retrieved contexts. The Score metric from the earlier experiment is also applied here

$$\text{Score} = \frac{\text{Number of correct contexts retrieved}}{\text{Total number of retrievals}}. \tag{3.6}$$

Results are presented in Table 3.5.

Evaluation Results

TABLE 3.5: Performance of NoInstruct Small Embedding v0 with and without Cross-Encoder (Score Metric).

Configuration	k = 1	k = 2	k = 3	k = 4
NoInstruct Small Embedding v0 (Bi-Encoder)	0.7213	0.8302	0.8761	0.9231
NoInstruct Small Embedding v0 + ms-marco-MiniLM-L-12-v2 (Cross-Encoder)	**0.9163**	**0.9584**	**0.9719**	**0.9739**

The results clearly demonstrate the superior performance of the cross-encoder-enhanced retriever compared to the bi-encoder-only approach across all values of k. For $k = 1$, the cross-encoder achieves a significantly higher score (0.9163 vs. 0.7213), and this trend persists at higher k-values. However, performance plateaus at $k = 4$, indicating diminishing returns as additional contexts are included.

Since the model already achieves strong performance at $k = 1$, increasing k often does not provide additional necessary information but instead introduces noise, outweighing the cases where it adds useful context. Therefore, we anticipate better performance with lower k-values in the subsequent experiments with RAG systems using re-ranking.

3.4.2 Re-Ranking on Different Questions Datasets

The RAG systems with re-ranking, as detailed in this section, are implemented next. The experiments conducted throughout the thesis are repeated, utilizing $k_{retriever} = 20$ while testing various values of k_{rerank}.

Experiment: SQuAD questions

In the first place, the experiment with the `SQuAD` questions is repeated, using as the RAG dataset the contexts of the questions. We compare the results with the ones obtained in previous experiments. Results are shown in Table 3.6

TABLE 3.6: Match Score for Different Configurations on `SQuAD` Questions.

System / Configuration	k	Match Score
Baseline Model (without contexts)	-	0.0179 (37 / 2067)
Best Simple RAG System	2	0.5636 (1165 / 2067)
RAG System with Re-Rank	1	**0.6454 (1334 / 2067)**
	2	0.6188 (1279 / 2067)
	3	0.6091 (1259 / 2067)
Model with Correct Context	-	0.6957 (1438 / 2067)

The results in Table 3.6 confirm the behavior anticipated in previous experiment: smaller k_{rerank} values provide better performance as the cross-encoder effectively prioritizes the most relevant context while avoiding noise introduced by less relevant ones. The best performance is achieved with $k_{\text{rerank}} = 1$, achieving a Match Score of 0.6454. While performance slightly decreases for larger k_{rerank} values, it remains competitive. Importantly, the gap between the best-performing re-ranked configuration ($k_{\text{rerank}} = 1$) and the model provided with the correct context (0.6957) has significantly narrowed, highlighting the effectiveness of re-ranking in improving the retrieval process.

Experiment: TriviaQA questions with Wikipedia contexts

The next experiment is the same as in 3.3, using the `TriviaQA` questions dataset, and using as the RAG's database only the Wikipedia passages directly related to the questions. Again, we compare the results obtained with re-ranking to the previous best results otained. These are presented in Table 3.7

The results in Table 3.7 demonstrate a significant improvement in Match Score when re-ranking is applied to the RAG system with Wikipedia contexts. The best performance is achieved with $k = 2$ and $k = 3$, both reaching a Match Score of 0.554, notably higher than the best non-re-ranked RAG configuration (0.487). This confirms the effectiveness of the cross-encoder in prioritizing relevant contexts and filtering noise. Similar to previous experiments, the system performs better with lower k-values. However, as seen in

TABLE 3.7: Match Score for RAG Systems with Re-Ranking on `TriviaQA` Questions.

System / Configuration	k	Match Score
Baseline System (without contexts)	–	0.070 (70 / 1000)
Best RAG System with Wikipedia Contexts	4	0.487 (487 / 1000)
Best RAG System with Additional Contexts	4	0.500 (500 / 1000)
RAG System with Wikipedia Contexts and Re-Ranking	1	0.547 (547 / 1000)
	2	**0.554 (554 / 1000)**
	3	**0.554 (554 / 1000)**

earlier comparisons between the `SQuAD` and `TriviaQA` datasets, the optimal k is slightly higher for `TriviaQA` ($k = 2, 3$, not $k = 1$), likely because the text is shorter after preprocessing.

Experiment: TriviaQA questions with additional contexts

Finally, we repeat the last experiment on the `TriviaQA` questions dataset, using as the RAG's database all the additional Wikipedia passages, as in 3.4. Results are shown in Table 3.8.

TABLE 3.8: Match Score for RAG Systems with Re-Ranking on `TriviaQA` Questions.

System / Configuration	k	Match Score
Baseline System (without contexts)	–	0.070 (70 / 1000)
Best RAG System with Wikipedia Contexts	4	0.487 (487 / 1000)
Best RAG System with Additional Contexts	4	0.500 (500 / 1000)
RAG System with Wikipedia Contexts and Re-Ranking	2	0.554 (554 / 1000)
RAG System with Additional Contexts and Re-Ranking	1	0.553 (553 / 1000)
	2	**0.563 (563 / 1000)**
	3	0.558 (558 / 1000)

In Table 3.8, we observe consistent gains in performance with re-ranking, even when using the larger, noisier dataset. The highest Match Score (0.563) is achieved with $k = 2$, surpassing the best non-re-ranked system with additional contexts (0.500). However, as k increases to 3, performance slightly decreases (0.558), which further reinforces the idea that the inclusion of additional contexts introduces noise to the generator.

Overall, these results highlight the robustness of re-ranking strategies, both when handling narrower and larger, more complex datasets, and demonstrate their ability to outperform simpler configurations. While a Match Score of 0.563 may seem modest, it is important to note that the generative model paired with the correct preprocessed context achieved a maximum Match Score of only 0.7. Therefore, the results from these experiments reflect a strong performance of the retrieval system with re-ranking.

Chapter 4

Conclusions

In this thesis, we explored the theoretical foundations of Large Language Models (LLMs) and adaptation techniques, applying it to a practical implementation of Retrieved-Augmented Generation (RAG) systems for question answering tasks. We emphasized the challenge of balancing computational efficiency and the generation of high-quality responses using advanced models. The key findings of this study are summarized below:

1. **Effective retrieval is essential**.
   Our experiments showed that the choice of the embedding model and the retrieval process plays a critical role in improving the model's performance. We observed how a large value of k (the number of retrieved documents) almost guarantees that the correct passages are retrieved. This is a crucial step in the RAG pipeline. On small, controlled datasets, we observed near-perfect retrieval in top 20 documents, while in larger noisier datasets, retrieval quality remained high based on the results.

2. **Trade-offs in k value**.
   Increasing k raises the chances of retrieving relevant documents containing the answer. However, this doesn't necessarily boost the model's performance, as adding too many documents to the generative model may introduce noise, compromising the model's ability to answer correctly, even when the right context is among the retrieved documents. The optimal k value depends on the context. Smaller k values worked better with preprocessed, concise chunks, while larger k values worked better for longer or less structured texts, closer to real-world scenarios.

3. **Re-ranking with cross-encoders**.
   Incorporating a cross-encoder to re-rank retrieved documents significantly enhanced performance in all experiments. This underscores the importance of the retrieval process and the trade-offs in k values. The cross-encoder allows the system to rank the most relevant passages higher, even with smaller k_{rerank} values. Its superior ability to assess query-document similarity outperformed the bi-encoder retriever alone.

4. **Scalability and practical implications**.
   Our experiments showed that adding large volumes of potentially unrelated text does not necessarily degrade performance, provided the

retriever system is robust enough. This result encourages the development of systems with vast knowledge bases, as they can handle large-scale information without sacrificing accuracy.

4.1 Limitations and Future Work

While the findings are promising, several limitations were identified that open opportunities for future work:

- **Model size and computational constraints**.
  We relied on a small generative model (`FLAN-T5 small`) to handle repeated evaluations efficiently. Although practical for this study, larger models may yield better results by processing more extensive and complex contexts. Future work could incorporate larger models to fully explore the real potential of RAG systems.

- **Domain-Specific adaptations**.
  The datasets used in this study focused on generic language and vocabulary. In specialized domains such as medicine or law, RAG systems might require additional techniques like partial fine-tuning or carefully designed prompts to achieve better performance. Further studies could explore hybrids methods that combine the strengths of fine-tuning and RAG.

- **Advanced RAG variants**.
  Techniques such as query expansion, dynamic re-ranking, or even modular RAG systems, could further boost performance. Testing these enhancements across diverse scenarios would help optimize RAG pipelines for specific tasks and real-world applications.

We believe that these proposals provide a strong foundation for further research and practical advancements in the field.

Bibliography

Brown, Tom B. et al. (2020). *Language Models are Few-Shot Learners*. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

Czerny, Thomas (2024). *An Intro to Large Language Models (LLMs)*. https://medium.com/@thomasczerny/an-intro-to-large-language-models-llms-41f0c802b900. Accessed: November 2024.

Vaswani, Ashish et al. (2017). "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Reimers, Nils and Iryna Gurevych (Nov. 2019). "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. URL: https://arxiv.org/abs/1908.10084.

Gao, Yunfan et al. (2024). *Retrieval-Augmented Generation for Large Language Models: A Survey*. arXiv: 2312.10997 [cs.CL]. URL: https://arxiv.org/abs/2312.10997.

Yu, Hao et al. (2024). *Evaluation of Retrieval-Augmented Generation: A Survey*. arXiv: 2405.07437 [cs.CL]. URL: https://arxiv.org/abs/2405.07437.

Transformers, Sentence (n.d.). *Cross-Encoders: SentenceTransformers Documentation*. https://www.sbert.net/examples/applications/cross-encoder/README.html. Accessed: December 2024.

HuggingFaceTeam (2024). *Hugging Face: State-of-the-art Machine Learning for Everyone*. https://huggingface.co/. Accessed: November 2024.

LangChainTeam (2024). *LangChain: A Framework for Developing Applications Powered by Language Models*. https://python.langchain.com/. Accessed: November 2024.

Muennighoff, Niklas et al. (2022). "MTEB: Massive Text Embedding Benchmark". In: *arXiv preprint arXiv:2210.07316*. DOI: 10.48550/ARXIV.2210.07316. URL: https://arxiv.org/abs/2210.07316.

Johnson, Jeff, Matthijs Douze, and Hervé Jégou (2017). *FAISS: A library for efficient similarity search and clustering of dense vectors*. https://faiss.ai/. Accessed: November 2024.

Chung, Hyung Won et al. (2022). *Scaling Instruction-Finetuned Language Models*. arXiv: 2210.11416 [cs.LG]. URL: https://arxiv.org/abs/2210.11416.

Rajpurkar, Pranav et al. (2016). *SQuAD: 100,000+ Questions for Machine Comprehension of Text*. arXiv: 1606.05250 [cs.CL]. URL: https://arxiv.org/abs/1606.05250.

Joshi, Mandar et al. (2017). *TriviaQA: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension*. arXiv: 1705.03551 [cs.CL]. URL: https://arxiv.org/abs/1705.03551.

HuggingFace (2022). *Wikipedia Dataset*. Accessed: November 2024. URL: https://huggingface.co/datasets/wikipedia.